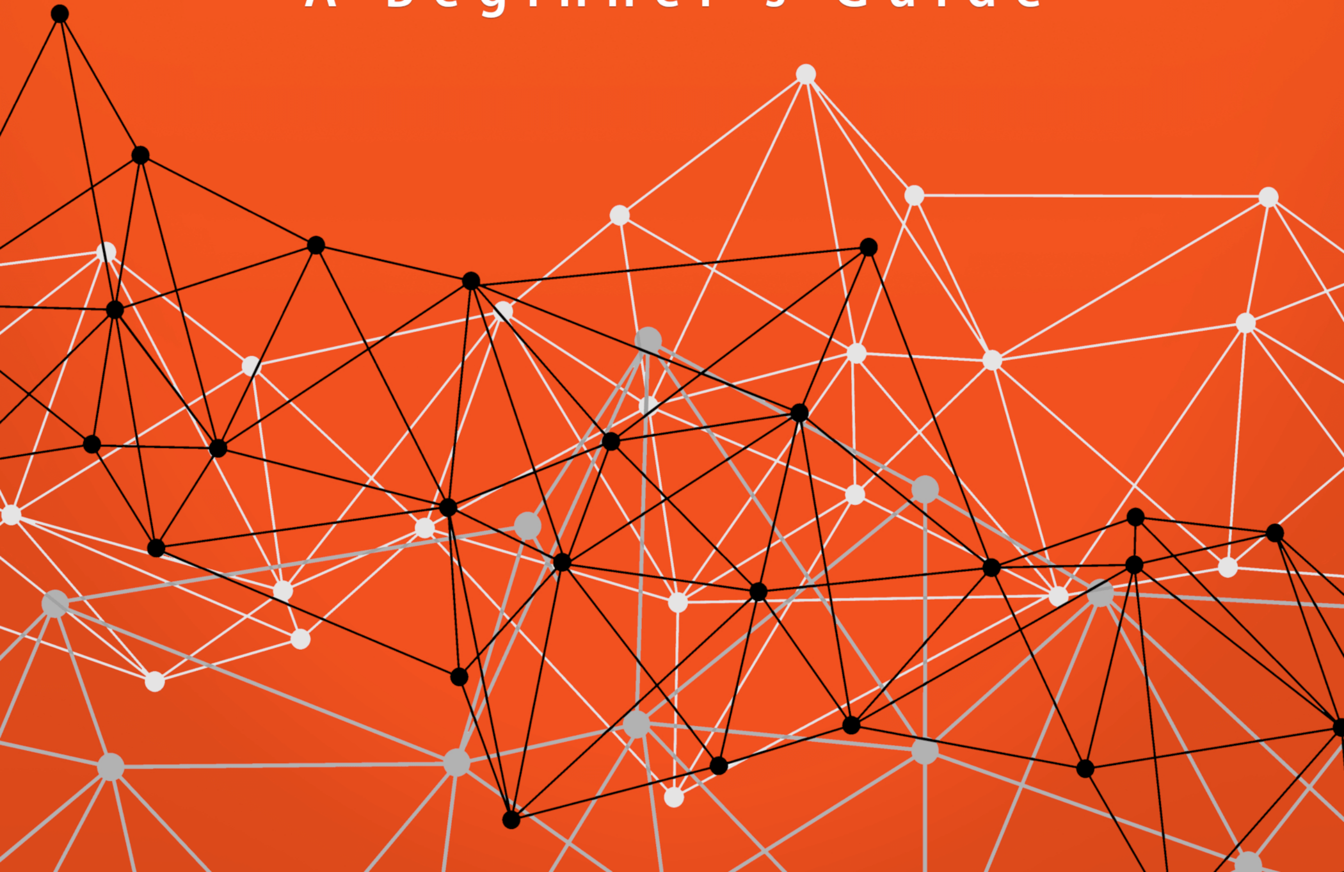
Daniel Voigt Godoy

Deep Learning with PyTorch Step-by-Step

A Beginner's Guide



Deep Learning with PyTorch Step-by-Step

A Beginner's Guide

Daniel Voigt Godoy

Version 1.0, 2021-05-18

Deep Learning with PyTorch Step-by-Step: A Beginner's Guide

by Daniel Voigt Godoy

May 2021: First Edition

Revision History for the First Edition:

- 2021-05-18: v1.0

This book is for sale at http://leanpub.com/pytorch. For more information, please send an email to contact@dvgodoy.com

"*What I cannot create, I do not understand.*"

Richard P. Feynman

Table of Contents

Preface

If you're reading this, I probably don't need to tell you that Deep Learning is amazing and PyTorch is cool, right?

But I will tell you, briefly, how this book came to be. In 2016, I started teaching a class on machine learning with Apache Spark and, a couple of years later, another class on the fundamentals of machine learning.

At some point, I tried to find a blog post that would visually explain, in a clear and concise manner, the concepts behind binary cross-entropy so that I could show it to my students. Since I could not find any that would fit my purpose, I decide to write one myself. Although I thought of it as a fairly basic topic, it turned out to be my most popular blog post! My readers have welcomed the simple, straightforward, and conversational way I used to explain the topic.

Then, in 2019, I used the same approach for writing another blog post: "*Understanding PyTorch with an example: a step-by-step tutorial*". Once again, I was amazed by the reaction from the readers!

It was their positive feedback that motivated me to write this book to help beginners start their journey into Deep Learning and PyTorch. I hope you enjoy reading this book as much as I enjoyed writing it.

Acknowledgements

First and foremost, I'd like to thank YOU, my reader, for making this book possible. If it weren't for the amazing feedback I got from the thousands of readers of my blog post about PyTorch, I would have never mustered the strength to start *and finish* such a major undertaking as writing a 1,000-page book!

I'd like to thank my good friends Jesús Martínez-Blanco, (who managed to read absolutely *everything* that I wrote), Jakub Cieslik, Hannah Berscheid, Mihail Vieru, Ramona Theresa Steck, Mehdi Belayet Lincon, and António Góis, for helping me out and dedicating a good chunk of their time to reading, proofing, and suggesting improvements to my drafts. I'm forever grateful for your support! I'd also like to thank my friend José Luis Lopez Pino for the initial push I needed to actually *start* writing this book.

Many thanks to my friends José Quesada and David Anderson, for taking me as a student at the Data Science Retreat in 2016 and, later on, for inviting me to be a teacher there. That was the starting point of my career both as a data scientist and a teacher.

I'd also like to thank the PyTorch developers for developing such an amazing framework, and the teams from Leanpub and Towards Data Science for making it incredibly easy for content creators like me to share their work with the community.

Finally, I'd like to thank my wife, Jerusa, for always being supportive throughout the entire writing of this book, and for taking the time to read *every* single page in it :-)

About the Author

Daniel is a data scientist, developer, writer, and teacher. He has been teaching machine learning and distributed computing technologies at Data Science Retreat, the longest-running Berlin-based bootcamp, since 2016, helping more than 150 students advance their careers.

Daniel is also the main contributor of two Python packages: HandySpark and DeepReplay.

His professional background includes 20 years of experience working for companies in several industries: banking, government, fintech, retail, and mobility.

Frequently Asked Questions (FAQ)

Why PyTorch?

First, coding in PyTorch is **fun** :-) Really, there is something to it that makes it very enjoyable to write code in… Some say it is because it is very **pythonic**, or maybe there is something else, who knows? I hope that, by the end of this book, you feel like that too!

Second, maybe there are even some *unexpected benefits* to your health - check Andrej Karpathy's tweet[1] about it!

Jokes aside, PyTorch is the **fastest-growing**[2] framework for developing Deep Learning models and it has a **huge ecosystem**[3]. That is, there are many *tools* and *libraries* developed on top of PyTorch. It is the **preferred framework**[4] in academia already and it is making its way in the industry.

Several companies are already powered by PyTorch[5], to name a few:

- **Facebook**: the company is the original developer of PyTorch, released in October 2016
- **Tesla**: watch Andrej Karpathy (AI Director at Tesla) speak about "*how Tesla is using PyTorch to develop full self-driving capabilities for its vehicles*" in this video[6]
- **OpenAI**: in January 2020, OpenAI decided to standardize its deep learning framework on PyTorch (source[7])
- **fastai**: fastai is a library[8] built on top of PyTorch to simplify model training and it is used in its Practical Deep Learning for Coders[9] course. The fastai library is deeply connected to PyTorch and "*you can't become really proficient at using fastai if you don't know PyTorch well, too*"[10]
- **Uber**: the company is a significant contributor to PyTorch's ecosystem, having developed libraries like Pyro[11] (probabilistic programming) and Horovod[12] (a distributed training framework)

- **Airbnb**: PyTorch sits at the core of the company's dialog assistant for customer service (source[13])

This book **aims to get you started with PyTorch** while giving you a **solid understanding of how it works**.

Why *this* book?

There are so many PyTorch books and tutorials around, and its documentation is quite complete and extensive. So, **why** should you read ***this*** book?

First, this is **not** a typical book: most tutorials *start* with some nice and pretty *image classification problem* to illustrate how to use PyTorch. It may seem cool, but I believe it **distracts** you from the **main goal**: **how PyTorch works**? In this book, I present a **structured**, **incremental**, and **from first principles** approach to learn PyTorch.

Second, this is **not** a **formal book** in any way: I am writing this book **as if I were having a conversation with you**, the reader. I will ask you **questions** (and give you answers shortly afterward), and I will also make (silly) **jokes**.

My job here is to make you **understand** the topic, so I will **avoid fancy mathematical notation** as much as possible and **spell it out in plain English**.

In this book, I will **guide** you through the **development** of many models in PyTorch, showing you why PyTorch makes it much **easier** and more **intuitive** to build models in Python: *autograd*, *dynamic computation graph*, *model classes*, and much, much more.

We will build, **step-by-step**, not only the models themselves but also your **understanding** as I show you both the **reasoning** behind the code and **how to avoid** some **common pitfalls** and **errors** along the way.

There is yet another advantage of **focusing on the basics**: this book is likely to have a **longer shelf life**. It is fairly common for technical books, especially those focusing on cutting-edge technology, to become outdated really fast. Hopefully, this is not going to be the case here, since the **underlying mechanics are not changing, neither are the concepts**. It is expected that some syntax changes over time, but I do not see backward compatibility breaking changes coming anytime soon.

> **One more thing**: if you hadn't noticed already, I **really** like to make use of **visual cues**, that is, **bold** and *italic* highlights. I firmly believe this helps the reader to **grasp** the **key ideas** I am trying to convey in a sentence more easily. You can find more on that in the section "**How to read this book?**".

Who should read this book?

I wrote this book for **beginners in general** - not only PyTorch beginners. Every now and then, I will spend some time explaining some **fundamental concepts** which I believe are **essential** to have a proper **understanding of what's going on in the code**.

The best example is **gradient descent**, which most people are familiar with at some level. Maybe you know its general idea, perhaps you've seen it in Andrew Ng's Machine Learning course, or maybe you've even **computed some partial derivatives yourself**!

In real life, the **mechanics** of gradient descent will be **handled automatically by PyTorch** (uh, spoiler alert!). But, I will walk you through it anyway (unless you choose to skip Chapter 0 altogether, of course), because lots of **elements in the code**, as well as **choices of hyper-parameters** (learning rate, mini-batch size, etc.), can be much more easily understood if you know **where they come from**.

Maybe you already know some of these concepts well: if this is the case, you can simply *skip* them, since I've made these explanations as *independent* as possible from the rest of the content.

But **I want to make sure everyone is on the same page** so, if you have just heard about a given concept or if you are unsure if you had entirely understood it, these explanations are for you.

What do I need to know?

This is a book for beginners, so I am assuming as **little prior knowledge** as possible - as mentioned in the previous section, I will take the time to explain fundamental concepts whenever needed.

That being said, this is what I expect from you, the reader:

- to be able to code in **Python** (if you are familiar with Object-Oriented Programming (OOP), even better)
- to be able to work with PyData stack (**numpy**, **matplotlib**, and **pandas**) and **Jupyter notebooks**
- to be familiar with some basic concepts used in **Machine Learning**, like:
    - supervised learning: regression and classification
    - loss functions for regression and classification (mean squared error, cross-entropy, etc.)
    - training-validation-test split
    - underfitting and overfitting (bias-variance trade-off)
    - evaluation metrics (confusion matrix, accuracy, precision, recall, etc.)

Even so, I am still briefly touching **some** of the topics above, but I need to draw a line somewhere; otherwise, this book would be gigantic!

How to read this book?

Since this book is a **beginner's guide**, it is meant to be read **sequentially**, as ideas and concepts are progressively built. The same holds true for the **code** inside the book - you should be able to *reproduce* all outputs, provided you execute the chunks of code in the same order as they are introduced.

This book is **visually** different than other books: as I've mentioned already in the "**Why *this* book?**" section, I **really** like to make use of **visual cues**. Although this is not, *strictly speaking*, a **convention**, this is how you can interpret those cues:

- I use **bold** to highlight what I believe to be the **most relevant words** in a sentence or paragraph, while *italicized* words are considered *important* too (not important enough to be bold, though)
- *Variables coefficients* and *parameters* in general, are *italicized*
- `Classes` and `methods` are written in a `monospaced` font, and they link to PyTorch [14] documentation the first time they are introduced, so you can easily follow it (unlike other links in this book, links to documentation are *numerous* and thus *not* included in the footnotes)
- Every **code cell** is followed by *another* cell showing the corresponding **outputs** (if any)
- **All code** presented in the book is available at its **official repository** on GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

Code cells with **titles** are an important piece of the workflow:

Title Goes Here

```
# Whatever is being done here is going to impact OTHER code
# cells. Besides, most cells have COMMENTS explaining what
# is happening
x = [1., 2., 3.]
print(x)
```

If there is any output to the code cell, titled or not, there *will* be another code cell depicting the corresponding **output** so you can *check* if you successfully reproduced it or not.

Output

```
[1.0, 2.0, 3.0]
```

Some code cells **do not** have titles - running them does not affect the workflow:

```
# Those cells illustrate HOW TO CODE something, but they are
# NOT part of the main workflow
dummy = ['a', 'b', 'c']
print(dummy[::-1])
```

But even these cells have their outputs shown!

Output

```
['c', 'b', 'a']
```

I use asides to communicate a variety of things, according to the corresponding icon:

⚠	*WARNING* Potential **problems** or things to **look out** for.
💡	*TIP* Key aspects I really want you to **remember**.
ℹ	*INFORMATION* Important information to **pay attention** to.
⚙	*TECHNICAL* Technical aspects of **parameterization** or **inner workings of algorithms**.

QUESTION AND ANSWER

Asking myself **questions** (pretending to be you, the reader) and answering them, either in the same block or shortly after.

DISCUSSION

Really brief discussion on a concept or topic.

LATER

Important topics that will be covered in more detail later.

SILLY

Jokes, puns, memes, quotes from movies.

What's Next?

It's time to **set up** an environment for your learning journey using the **Setup Guide**.

[1] https://bit.ly/2MQoYRo

[2] https://bit.ly/37uZgLB

[3] https://pytorch.org/ecosystem/

[4] https://bit.ly/2MTN0Lh

[5] https://bit.ly/2UFHFve

[6] https://bit.ly/2XXJkyo

[7] https://openai.com/blog/openai-pytorch/

[8] https://docs.fast.ai/

[9] https://course.fast.ai/

[10] https://course.fast.ai/

[11] http://pyro.ai/

[12] https://github.com/horovod/horovod

[13] https://bit.ly/30CPhm5

[14] https://bit.ly/3cT1aH2

Setup Guide

Official Repository

This book’s official repository is on GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

It contains **one Jupyter notebook** for every **chapter** in this book. Each notebook contains **all the code shown** in its corresponding chapter, and you should be able to **run its cells in sequence** to get the **same outputs**, as shown in the book. I strongly believe that being able to **reproduce the results** brings **confidence** to the reader.

Environment

There are **three options** for you to run the Jupyter notebooks:

- Google Colab (*https://colab.research.google.com*)
- Binder (*https://mybinder.org*)
- Local Installation

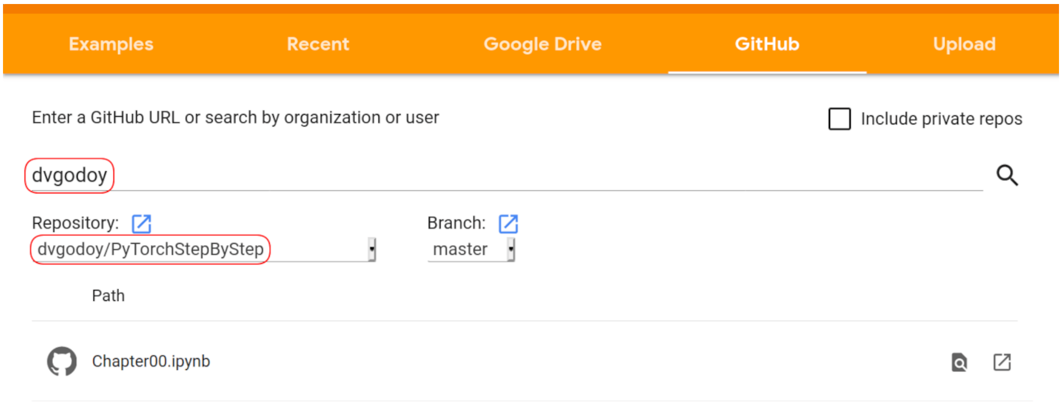
Let’s briefly explore the **pros** and **cons** of each one of those options:

Google Colab

Google Colab "*allows you to write and execute Python in your browser, with zero configuration required, free access to GPUs and easy sharing*"[15].

You can easily **load notebooks directly from GitHub** using Colab’s special URL (*https://colab.research.google.com/github/*). Just type in the GitHub’s user or organization (like mine, `dvgodoy`), and it will show you a list of all its public repositories (like this book’s - `PyTorchStepByStep`).

After choosing a repository, it will also list the available notebooks and corresponding links to open them in a new browser tab.



Enter a GitHub URL or search by organization or user

Include private repos

dvgodoy

Repository: dvgodoy/PyTorchStepByStep

Branch: master

Path

Chapter00.ipynb

Figure 0.1 - Google Colab's special URL

You also get access to a **GPU**, which is very useful to train Deep Learning models **faster**. More importantly, if you **make changes** to the notebook, Google Colab will **keep them**. The whole setup is very convenient - the only **cons** I can think of are:

- you need to be **logged in** a Google Account
- you need to (re)install Python packages that are not part of Google Colab's default configuration

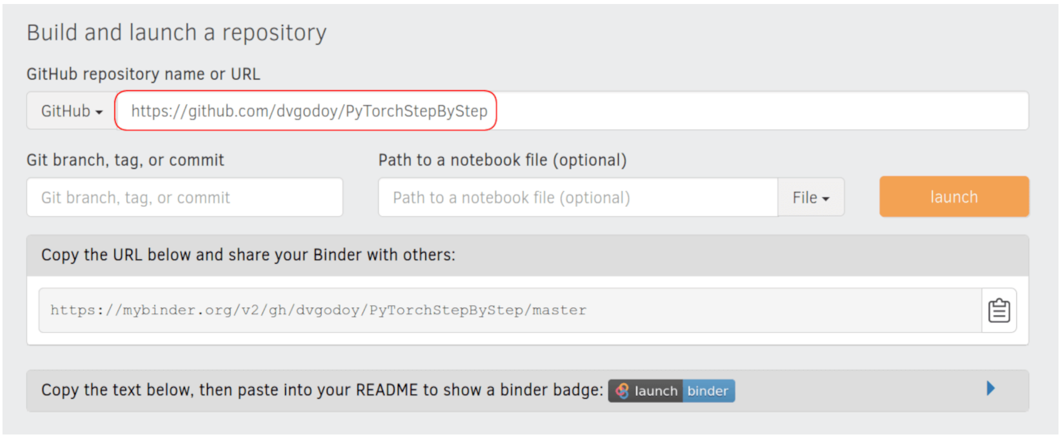
Binder

Binder "*allows you to create custom computing environments that can be shared and used by many remote users*"[16].

You can also **load notebooks directly from GitHub**, but the process is slightly different. Binder will create something like a "*virtual machine*" (technically, it is a container, but let's leave at that), clone the repository and start Jupyter. This allows you to have access to **Jupyter's Home Page** in your browser, just like the way you would if you were running it locally, but everything is running in a JupyterHub server on their end.

Just go to Binder's site (*https://mybinder.org/*) and type in the URL to the GitHub repository you want to explore (for instance, `https://github.com/dvgodoy/PyTorchStepByStep`) and click on **Launch**. It will take a couple of minutes to build the image and open Jupyter's home page.

You can also **launch Binder** for this book's repository directly using the following link: *https://mybinder.org/v2/gh/dvgodoy/PyTorchStepByStep/master*.


Build and launch a repository

GitHub repository name or URL

GitHub https://github.com/dvgodoy/PyTorchStepByStep

Git branch, tag, or commit: Git branch, tag, or commit

Path to a notebook file (optional): Path to a notebook file (optional)

Copy the URL below and share your Binder with others:

```
https://mybinder.org/v2/gh/dvgodoy/PyTorchStepByStep/master
```

Copy the text below, then paste into your README to show a binder badge: launch binder

Figure 0.2 - Binder's page

Binder is very convenient since it **does not require a prior setup** of any kind. Any Python packages needed to successfully run the environment are likely installed during launch (if provided by the author of the repository).

On the other hand, it may **take time** to start, and it **does not keep your changes** after your session expires (so, make sure you **download** any notebooks you modify).

Local Installation

This option will give you more **flexibility**, but it will require **more effort to set up**. I encourage you to try setting up your own environment. It may seem daunting at first, but you can surely accomplish it following **seven easy steps**:

Checklist

- ▢ 1. Install **Anaconda**
- ▢ 2. Create and activate a **virtual environment**
- ▢ 3. Install **PyTorch** package
- ▢ 4. Install **TensorBoard** package
- ▢ 5. Install **GraphViz** software and **TorchViz** package (**optional**)
- ▢ 6. Install **git** and **clone** the repository
- ▢ 7. Start **Jupyter** Notebook

1. Anaconda

If you don't have **Anaconda's Individual Edition**[17] installed yet, that would be a good time to do it - it is a convenient way to start - since it contains most of the Python libraries a data scientist will ever need to develop and train models.

Please follow the **installation instructions** for your OS:

- Windows (*https://docs.anaconda.com/anaconda/install/windows/*)
- macOS (*https://docs.anaconda.com/anaconda/install/mac-os/*)
- Linux (*https://docs.anaconda.com/anaconda/install/linux/*)

Make sure you choose **Python 3.X** version since Python 2 was discontinued in January 2020.

After installing Anaconda, it is time to create an **environment**.

2. Conda (Virtual) Environments

Virtual environments are a convenient way to isolate Python installations associated with different projects.

"*What is an environment?*"

It is pretty much a **replication of Python itself and some (or all) of its libraries** so, effectively, you'll end up with multiple Python installations on your computer.

You may be wondering: "*why can't I just use one single Python installation for everything?*"

With so many independently developed Python **libraries**, each having many different **versions** and each version has various **dependencies** (on other libraries), **things can go out of hand** real quick.

It is beyond the scope of this guide to debate these issues, but take my word for it (or Google it!), you'll benefit a great deal if you pick up the habit of **creating a different environment for every project you start working on**.

"*How do I create an environment?*"

First, you need to choose a **name** for your environment :-) Let's call ours `pytorchbook` (or anything else you find easy to remember). Then, you need to open a **terminal** (in Ubuntu) or **Anaconda Prompt** (in Windows or macOS) and type the following command:

```
$ conda create -n pytorchbook anaconda
```

The command above creates a conda environment named `pytorchbook` and includes **all anaconda packages** in it (time to get a coffee, it will take a while…). If you want to learn more about creating and using conda environments, please check Anaconda's Managing Environments[18] user guide.

Did it finish creating the environment? Good! It is time to **activate it**, meaning, making **that Python installation** the one to be used now. In the same terminal (or Anaconda Prompt), just type:

```
$ conda activate pytorchbook
```

Your prompt should look like this (if you're using Linux)…

```
(pytorchbook)$
```

or like this (if you're using Windows):

```
(pytorchbook)C:\>
```

Done! You are using a **brand new conda environment** now. You'll need to **activate it** every time you open a new terminal or if you're a Windows or macOS user, you can open the corresponding Anaconda Prompt (it will show up as **Anaconda Prompt (pytorchbook)**, in our case), which will have it activated from the start.

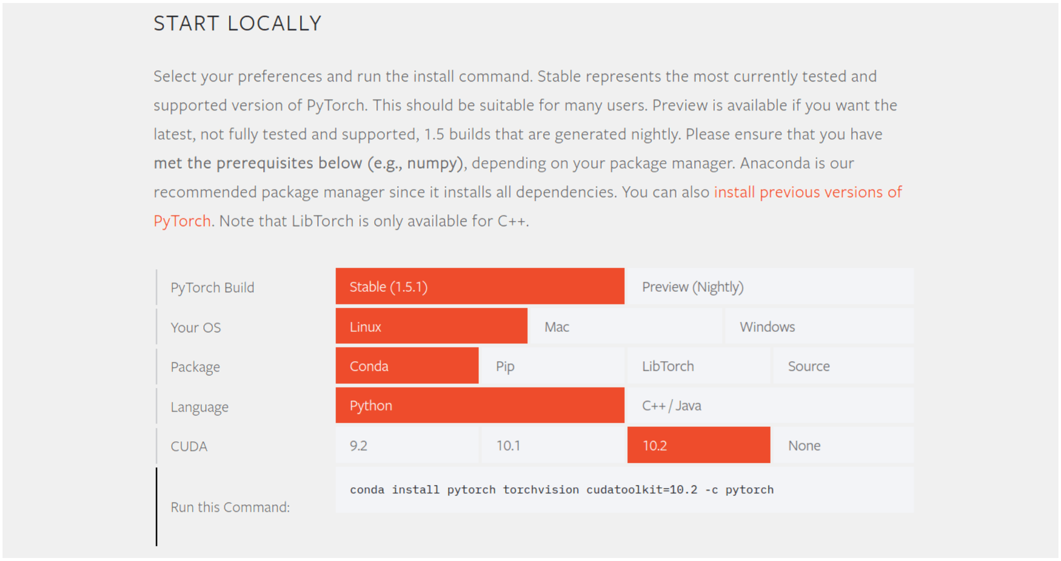
IMPORTANT: From now on, I am assuming you'll activate the `pytorchbook` environment every time you open a terminal / Anaconda Prompt. Further installation steps **must** be executed inside the environment.

3. PyTorch

PyTorch is the coolest **deep learning framework**, just in case you skipped the introduction.

It is "*an open source machine learning framework that accelerates the path from research prototyping to production deployment*"[19]. Sounds good, right? Well, I probably don't have to convince you at this point :-)

It is time to install the star of the show :-) We can go straight to the **Start Locally** (https://pytorch.org/get-started/locally/) section of its website, and it will automatically select the options that best suit your local environment and it will show you the **command to run**.


START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.5 builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	[✓ Stable (1.5.1)]	Preview (Nightly)		
Your OS	[✓ Linux]	Mac	Windows	
Package	[✓ Conda]	Pip	LibTorch	Source
Language	[✓ Python]	C++ / Java		
CUDA	9.2	10.1	[✓ 10.2]	None

Run this Command:

```
conda install pytorch torchvision cudatoolkit=10.2 -c pytorch
```

Figure 0.3 - PyTorch's Start Locally

Some of these options are given:

- PyTorch Build: always select a **Stable** version
- Package: I am assuming you're using **Conda**
- Language: obviously, **Python**

So, two options remain: **Your OS** and **CUDA**.

"*What is CUDA?*" you ask.

Using GPU / CUDA

CUDA "*is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs)*"[20].

If you have a **GPU** in your computer (likely a *GeForce* graphics card), you can leverage its power to train deep learning models **much faster** than using a CPU. In this case, you should choose a PyTorch installation that includes CUDA support.

This is not enough, though: if you haven't done so yet, you need to install up-to-date drivers, the CUDA Toolkit, and the CUDA Deep Neural Network library (cuDNN). Unfortunately, more detailed installation instructions for CUDA are outside the scope of this book.

The **advantage** of using a GPU is that it allows you to **iterate faster** and **experiment with more complex models and a more extensive range of hyper-parameters**.

In my case, I use **Linux**, and I have a **GPU** with CUDA version 10.2 installed. So I would run the following command in the **terminal** (after activating the environment):

```
(pytorchbook)$ conda install pytorch torchvision\
cudatoolkit=10.2 -c pytorch
```

Using CPU

If you **do not** have a **GPU**, you should choose **None** for CUDA.

"*Would I be able to run the code **without** a GPU*?" you ask.

Sure! The code and the examples in this book were designed to allow **all readers** to follow them promptly. Some examples may demand a bit more of computing power, but we are talking about a **couple of minutes** in a CPU, not hours. If you do not have a GPU, **don't worry** :-) Besides, you can always use Google Colab if you need to use a GPU for a while!

If I had a **Windows** computer, and **no GPU**, I would have to run the following command in the **Anaconda Prompt (pytorchbook)**:

```
(pytorchbook) C:\> conda install pytorch torchvision cpuonly\
 -c pytorch
```

Installing CUDA

CUDA: Installing drivers for a GeForce graphics card, NVIDIA's cuDNN and CUDA Toolkit can be challenging and is highly dependent on which model you own and which OS you use.

For installing GeForce's drivers, go to GeForce's website (*https://www.geforce.com/drivers*), select your OS and the model of your graphics card, and follow the installation instructions.

For installing NVIDIA's CUDA Deep Neural Network library (cuDNN), you need to register at *https://developer.nvidia.com/cudnn*.

For installing CUDA Toolkit (*https://developer.nvidia.com/cuda-toolkit*), please follow instructions for your OS and choose a local installer or executable file.

macOS: If you're a macOS user, please beware that PyTorch's binaries **DO NOT** support **CUDA**, meaning you'll need to install PyTorch **from source** if you want to use your GPU. This is a somewhat **complicated** process (as described in *https://github.com/pytorch/pytorch#from-source*) - so, if you don't feel like it, you can choose to proceed **without CUDA**, and you'll still be able to execute the code in this book promptly.

4. TensorBoard

TensorBoard is TensorFlow's **visualization toolkit**, it "*provides the visualization and tooling needed for machine learning experimentation*"[21].

TensorBoard is a powerful tool, and we can use it even if we are developing models in PyTorch. Luckily, you don't need to install the whole TensorFlow to get it, you can easily **install TensorBoard alone** using **conda**. You just need to run this command in your **terminal** or **Anaconda Prompt** (again, after activating the environment):

```
(pytorchbook)$ conda install -c conda-forge tensorboard
```

5. GraphViz and Torchviz (optional)

This step is optional, mostly because the installation of GraphViz can sometimes be *challenging* (especially on Windows). If for any reason, you do not succeed in installing it correctly, or if you decide to skip this installation step, you will still be **able to execute the code in this book** (except for a couple of cells that generate images of a model's structure in the Dynamic Computation Graph section of Chapter 1).

GraphViz is an open-source graph visualization software. It is "*a way of representing structural information as diagrams of abstract graphs and networks*"[22].

We need to install GraphViz to use **TorchViz**, a neat package that allows us to visualize a model's structure. Please check the **installation instructions** for your OS at *https://www.graphviz.org/download/*.

If you are using Windows, please use the **GraphViz's Windows Package** installer at *https://graphviz.gitlab.io/_pages/Download/windows/graphviz-2.38.msi*.

You also need to add GraphViz to the PATH (environment variable) in Windows. Most likely, you can find GraphViz executable file at `C:\ProgramFiles(x86)\Graphviz2.38\bin`. Once you find it, you need to set or change the PATH accordingly, adding GraphViz's location to it.

For more details on how to do that, please refer to How to Add to Windows PATH Environment Variable[23].

For additional information, you can also check the How to Install Graphviz

Software[24] guide.

After installing GraphViz, you can install the **torchviz**[25] package. This package is **not** part of Anaconda Distribution Repository[26] and is only available at **PyPI**[27], the Python Package Index, so we need to `pip install` it.

Once again, open a **terminal** or **Anaconda Prompt** and run this command (just once more: after activating the environment):

```
(pytorchbook)$ pip install torchviz
```

To check your GraphViz / TorchViz installation, you can try the Python code below:

```
(pytorchbook)$ python

Python 3.7.5 (default, Oct 25 2019, 15:51:11)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import torch
>>> from torchviz import make_dot
>>> v = torch.tensor(1.0, requires_grad=True)
>>> make_dot(v)
```

If everything is **working correctly**, you should see something like this:

Output

```
<graphviz.dot.Digraph object at 0x7ff540c56f50>
```

If you get an **error** of any kind (the one below is pretty common), it means there is still some **installation issue** with GraphViz.

Output

```
ExecutableNotFound: failed to execute ['dot', '-Tsvg'], make
sure the Graphviz executables are on your systems' PATH
```

6. Git

It is *way* beyond this guide's scope to introduce you to version control and its most popular tool: `git`. If you are familiar with it already, great, you can skip this section altogether!

Otherwise, I'd recommend you to learn more about it, it will **definitely** be useful for you later down the line. In the meantime, I will show you the bare minimum, so you can use `git` to **clone the repository** containing all code used in this book and get your **own**, **local copy** of it to modify and experiment with it as you please.

First, you need to install it. So, head to its downloads page (*https://git-scm.com/downloads*) and follow instructions for your OS. Once the installation is complete, please open a **new terminal** or **Anaconda Prompt** (it's OK to close the previous one). In the new terminal or Anaconda Prompt, you should be able to **run `git` commands**.

To clone this book's repository, you only need to run:

```
(pytorchbook)$ git clone https://github.com/dvgodoy/\
PyTorchStepByStep.git
```

The command above will create a `PyTorchStepByStep` folder which contains a local copy of everything available on GitHub's repository.

conda install vs pip install

Although they may seem equivalent at first sight, you should **prefer `conda install`** over `pip install` when working with Anaconda and its virtual environments.

The reason is that `conda install` is sensitive to the active virtual environment: the package will be installed only for that environment. If you use `pip install`, and `pip` itself is not installed in the active environment, it will fall back to the **global `pip`**, and you definitely **do not** want that.

Why not? Remember the problem with **dependencies** I mentioned in the virtual environment section? That's why! The `conda` installer assumes it handles all packages that are part of its repository and keeps track of the complicated network of dependencies among them (to learn more about this, check this link[28]).

To learn more about the differences between `conda` and `pip`, read Understanding Conda and Pip[29].

As a rule, first, try to `conda install` a given package and, only if it does not exist there, fall back to `pip install`, as we did with `torchviz`.

7. Jupyter

After cloning the repository, navigate to the `PyTorchStepByStep` folder and, **once inside it**, you only need to **start Jupyter** on your terminal or Anaconda Prompt:

```
(pytorchbook)$ jupyter notebook
```

This will open your browser up, and you will see **Jupyter's Home Page** containing the repository's notebooks and code.

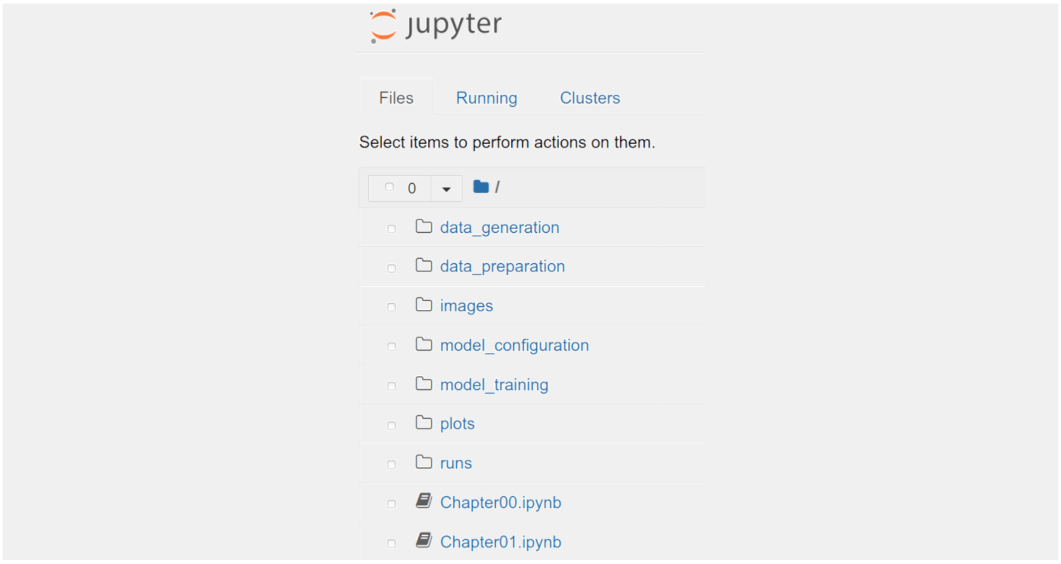


Figure 0.4 - Running Jupyter

Moving On

Regardless of which one of the three environments you chose, now you are ready to move on and tackle the development of your first PyTorch model, **step by step**!

[15] https://colab.research.google.com/notebooks/intro.ipynb

[16] https://mybinder.readthedocs.io/en/latest/

[17] https://www.anaconda.com/products/individual

[18] https://bit.ly/2MVk0CM

[19] https://pytorch.org/

[20] https://developer.nvidia.com/cuda-zone

[21] https://www.tensorflow.org/tensorboard

[22] https://www.graphviz.org/

[23] https://bit.ly/3fIwYA5

[24] https://bit.ly/30Ayct3

[25] https://github.com/szagoruyko/pytorchviz

[26] https://docs.anaconda.com/anaconda/packages/pkg-docs/

[27] https://pypi.org/

[28] https://bit.ly/37onBTt

[29] https://bit.ly/2AAh8J5

Part I
Fundamentals

Chapter 0
Visualizing Gradient Descent

Spoilers

In this chapter, we will:

- define a **simple linear regression model**
- walk through **every step of gradient descent**: initializing parameters, forward pass, computing errors and loss, computing gradients, and updating parameters
- understand **gradients** using **equations**, **code**, and **geometry**
- understand the difference between **batch**, **mini-batch** and **stochastic** gradient descent
- visualize the **effects on the loss** of using different **learning rates**
- understand the importance of **standardizing/scaling features**
- and much, much more!

There is **no** actual PyTorch code in this chapter… it is *Numpy* all along because our focus here is to understand, inside and out, how gradient descent works. PyTorch will be introduced in the next chapter.

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 0[30] is part of the official "**Deep Learning with PyTorch Step-by-Step**" repository on GitHub. You can also run it directly in **Google Colab**[31].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 0's notebook. If not, just click on `Chapter00.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
```

Visualizing Gradient Descent

According to Wikipedia[32]: "***Gradient descent*** *is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.*"

But I would go with: "**Gradient descent** is an iterative technique commonly used in Machine Learning and Deep Learning to try to find the best possible set of parameters / coefficients for a given model, data points and loss function, starting from an initial, and usually random, guess."

"*Why **visualizing** gradient descent?*"

I believe *the way gradient descent is usually explained lacks intuition*. Students and beginners are left with a *bunch of equations* and *rules of thumb* - **this is not the way one should learn such a fundamental topic**.

If you **really understand** how gradient descent works, you will also understand how the **characteristics of your data** and your **choice of hyper-parameters** (mini-batch size and learning rate, for instance) have an **impact** on how **well** and how **fast** the model training is going to be.

By *really understanding*, I do not mean working through the equations manually: this does not develop intuition either. I mean **visualizing** the effects of different settings, I mean **telling a story** to illustrate the concept. That's how you **develop intuition**.

That being said, I'll cover the **five basic steps** you'd need to go through to use gradient descent. I'll show you the corresponding *Numpy* code while explaining lots of **fundamental concepts** along the way.

But first, we need some **data** to work with. Instead of using some *external dataset*, let's:

- define which **model** we want to train to better understand gradient descent
- **generate synthetic data** for that model

Model

The model must be **simple** and **familiar**, so you can focus on the **inner workings** of gradient descent.

So, I will stick with a model as simple as it can be: a **linear regression with a single feature x**!

$$y = b + wx + \epsilon$$

Equation 0.1 - Simple Linear Regression model

In this model, we use a **feature (*x*)** to try to predict the value of a **label (*y*)**. There are three elements in our model:

- **parameter *b***, the *bias* (or *intercept*), which tells us the expected average value of *y* when *x* is zero
- **parameter *w***, the *weight* (or *slope*), which tells us how much *y* increases, on average, if we increase *x* by one unit
- and that **last term** (why does it *always* have to be a Greek letter?), *epsilon*, which is there to account for the inherent **noise**, that is, the **error** we cannot get rid of

We can also conceive the very same model structure in a less abstract way:

salary = minimum wage + increase per year * years of experience + noise

And to make it even more concrete, let’s say that the **minimum wage** is **$1,000** (whatever the currency or time frame, this is not important). So, if you have **no experience**, your salary is going to be the **minimum wage** (parameter ***b***).

Also, let’s say that, **on average**, you get a **$2,000 increase** (parameter ***w***) for every year of experience you have. So, if you have **two years of experience**, you are expected to earn a salary of **$5,000**. But your actual salary is **$5,600** (lucky you!). Since the model cannot account for those **extra $600**, your extra money is, technically speaking, **noise**.

Data Generation

We know our model already. In order to generate **synthetic data** for it, we need to pick values for its **parameters**. I chose ***b* = 1** and ***w* = 2** (as in thousands of $) from the example above.

First, let’s generate our **feature (*x*)**: we use *Numpy*'s `rand` method to randomly generate 100 (*N*) points between 0 and 1.

Then, we plug our **feature (*x*)** and our **parameters *b* and *w*** into our **equation** to compute our **labels (*y*)**. But we need to add some **Gaussian noise**[33] **(*epsilon*)** as well;

otherwise, our synthetic dataset would be a perfectly straight line. We can generate noise using *Numpy*'s `randn` method, which draws samples from a normal distribution (of mean 0 and variance 1), and then multiplying it by a **factor** to adjust for the **level of noise**. Since I don't want to add so much noise, I picked 0.1 as my factor.

Synthetic Data Generation

Data Generation

```
1 true_b = 1
2 true_w = 2
3 N = 100
4
5 # Data Generation
6 np.random.seed(42)
7 x = np.random.rand(N, 1)
8 epsilon = (.1 * np.random.randn(N, 1))
9 y = true_b + true_w * x + epsilon
```

Did you notice the `np.random.seed(42)` at line 6? This line of code is actually more important than it looks. It guarantees that, every time we run this code, the **same random numbers will be generated**.

"*Wait; what?! Aren't the numbers supposed to be **random**? How could they possibly be the **same** numbers?*" you ask, perhaps even a bit annoyed by this.

(Not So) Random Numbers

Well, you know, random numbers are not **quite** random… they are really **pseudo-random**, which means, *Numpy*'s number generator spits out a **sequence of numbers** that **looks like it's random**. But it is not, really.

The **good** thing about this behavior is that we can tell the generator to **start a particular sequence of pseudo-random numbers**. To some extent, it works as if we tell the generator: "*please generate sequence #42*", and it will spill out a sequence of numbers. That number, 42, which works like the *index* of the sequence, is called a **seed**. Every time we give it the **same seed**, it generates the **same numbers**.

This means we have the **best of both worlds**: on the one hand, we do **generate** a sequence of numbers that, for all intents and purposes, is **considered to be random**; on the other hand, we have the **power to reproduce any given sequence**. I cannot stress enough how convenient that is for **debugging** purposes :-)

Moreover, you can guarantee that **other people will be able to reproduce your results**. Imagine how annoying it would be to run the code in this book and get different outputs every time, having to wonder if there is anything wrong with it… But since I've set a seed, you and I can achieve the very same outputs, even if it involved generating random data!

Next, let's **split** our synthetic data into **train** and **validation** sets, shuffling the array of indexes and using the first 80 shuffled points for training.

"*Why do you need to* ***shuffle*** *randomly generated data points? Aren't they random enough?*"

Yes, they **are** random enough, and shuffling them is indeed redundant in this example. But it is best practice to **always shuffle** your data points before training a model to improve the performance of gradient descent.

There is one **exception** to the "always shuffle" rule, though: **time series** problems, where shuffling can lead to data leakage. We'll get back to this when we introduce recurrent neural networks (much) later down the line.

Train-Validation-Test Split

It is beyond the scope of this book to explain the reasoning behind the **train-validation-test split**, but there are two points I'd like to make:

1. The split should **always** be the **first thing** you do - no preprocessing, no transformations; **nothing happens before the split** - that's why we do this **immediately after the synthetic data generation**
2. In this chapter we will use **only the training set** - so I did not bother to create a **test set**, but I performed a split nonetheless to **highlight point #1** :-)

Train-Validation Split

```
# Shuffles the indices
idx = np.arange(N)
np.random.shuffle(idx)

# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

"*Why didn't you use* `train_test_split` *from Scikit-Learn?*" you may be asking…

That’s a fair point. Later on, we will refer to the **indices of the data points** belonging to either train or validation sets, instead of the points themselves. So, I thought of using them from the very start.

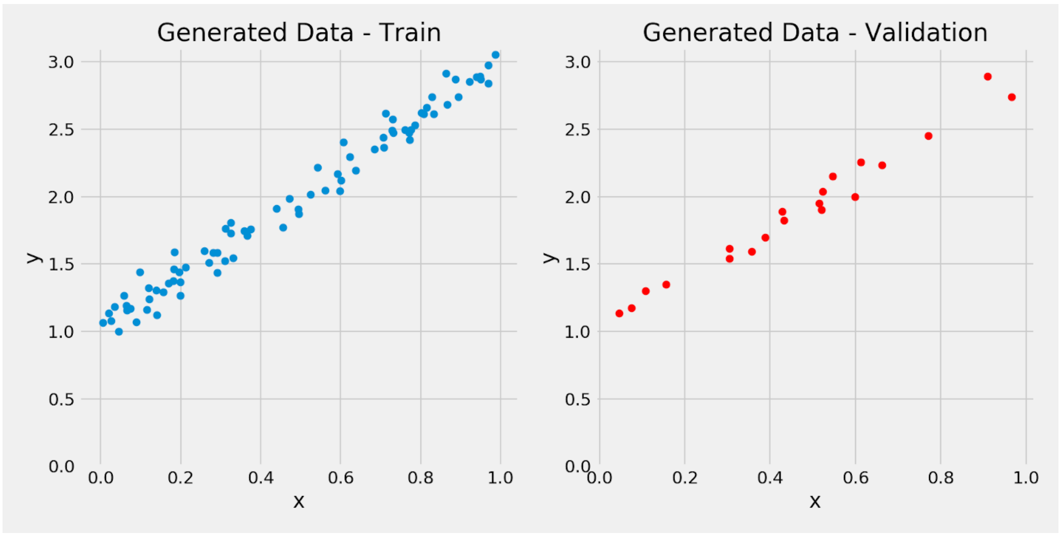


Figure 0.1 - Synthetic data: Train and Validation sets

We **know** that ***b* = 1**, ***w* = 2**, but now let’s see **how close** we can get to the true values by using **gradient descent** and the 80 points in the **training set** (for training, ***N* = 80**).

Step 0 - Random Initialization

In our example, we already **know** the **true values** of the **parameters**, but this will obviously never happen in real life: if we *knew* the true values, why even bother to train a model to find them?!

OK, given that **we’ll never know** the **true values** of the parameters, we need to set **initial values** for them. How do we choose them? It turns out; a **random guess** is as good as any other.

Even though the initialization is **random**, there are some clever **initialization schemes** that should be used when training more complex models. We'll get back to them (much) later.

For training a model, you need to **randomly initialize the parameters/weights** (we have only two, ***b*** and ***w***).

Random Initialization

```
# Step 0 - Initializes parameters "b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)
w = np.random.randn(1)

print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
```

Step 1 - Compute Model's Predictions

This is the **forward pass** - it simply *computes the model's predictions using the current values of the parameters/weights*. At the very beginning, we will be producing **really bad predictions**, as we started with **random values from Step 0**.

Step 1

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train
```

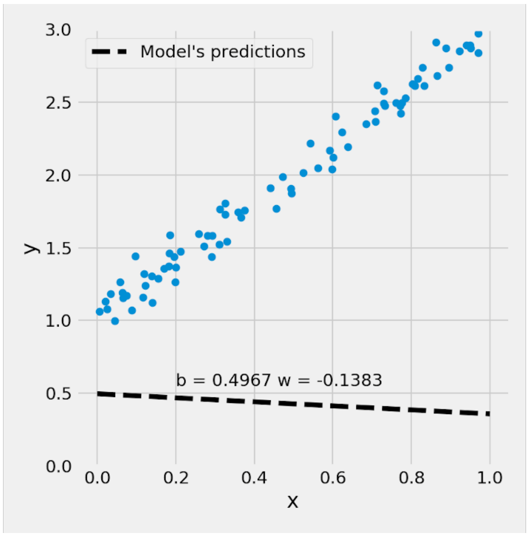


Figure 0.2 - Model's predictions (with random parameters)

Step 2 - Compute the Loss

There is a subtle but fundamental difference between **error** and **loss**.

The **error** is the difference between the **actual value (label)** and the **predicted value** computed for a single data point. So, for a given *i*-th point (from our dataset of *N* points), its error is:

$$error_i = \hat{y}_i - y_i$$

Equation 0.2 - Error

The error of the **first point** in our dataset (*i* = 0) can be represented like this:

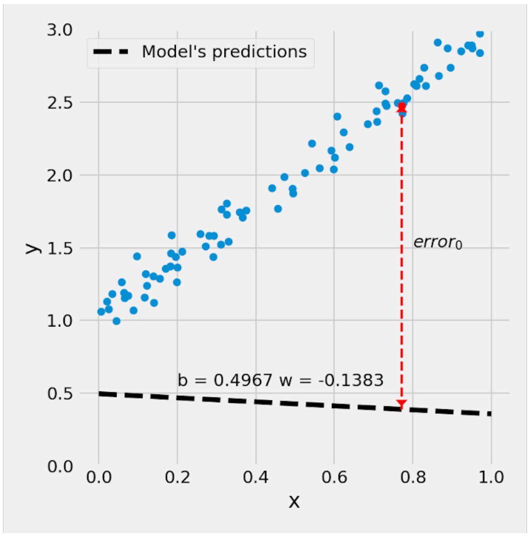


Figure 0.3 - Prediction error (for one data point)

The **loss**, on the other hand, is some sort of **aggregation of errors for a set of data points**.

It seems rather obvious to compute the loss for **all** (*N*) data points, right? Well, yes and no. Although it will surely yield a **more stable path** from the initial **random parameters** to the parameters that **minimize the loss**, it will also surely be **slow**.

This means one *needs to sacrifice (a bit) the stability for the sake of speed*. This is easily accomplished by randomly choosing (*without replacement*) a subset of ***n*** out of ***N*** data points each time we compute the loss.

Batch, **Mini-batch**, *and* **Stochastic Gradient Descent**

- if we use **all points** in the training set (*n* = **N**) to compute the loss, we are performing a **batch** gradient descent
- if we were to use a **single point** (*n* = **1**) each time, it would be a **stochastic** gradient descent
- anything else (*n*) **in-between 1 and N** characterizes a **mini-batch** gradient descent

For a regression problem, the **loss** is given by the **Mean Squared Error (MSE)**, that is, the average of all squared errors, that is, the average of all squared differences between **labels** (*y*) and **predictions** (*b* + *wx*).

$$
\begin{aligned}
MSE &= \frac{1}{n}\sum_{i=1}^{n} {error_i}^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} (\hat{y_i} - y_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} (b + wx_i - y_i)^2
\end{aligned}
$$

Equation 0.3 - Loss: Mean Squared Error (MSE)

In the code below, we are using **all data points** of the training set to compute the **loss**, so ***n* = *N* = 80**, meaning we are indeed performing **batch gradient descent**.

Step 2

```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train)

# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()

print(loss)
```

Output

```
2.7421577700550976
```

Loss Surface

We have just computed the **loss** (2.74) corresponding to our **randomly initialized parameters** (*b* = 0.49 and *w* = -0.13). What if we did the same for **ALL** possible values of ***b*** and ***w***? Well, not *all* possible values, but *all combinations of evenly spaced values in a given range*, like:

```
# Reminder:
# true_b = 1
# true_w = 2

# we have to split the ranges in 100 evenly spaced intervals each
b_range = np.linspace(true_b - 3, true_b + 3, 101)
w_range = np.linspace(true_w - 3, true_w + 3, 101)
# meshgrid is a handy function that generates a grid of b and w
# values for all combinations
bs, ws = np.meshgrid(b_range, w_range)
bs.shape, ws.shape
```

Output

```
((101, 101), (101, 101))
```

The result of the `meshgrid` operation was two (101, 101) matrices representing the values of each parameter inside a grid. How does one of these matrices look like?

```
bs
```

Output

```
array([[-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       ...,
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ]])
```

Sure, we're somewhat *cheating* here, since we *know* the **true** values of *b* and *w*, so we can choose the **perfect ranges** for the parameters. But it is for educational purposes only :-)

Next, we could use those values to compute the corresponding **predictions**, **errors**, and **losses**. Let's start taking a **single data point** from the training set and computing the predictions for every combination in our grid:

```
dummy_x = x_train[0]
dummy_yhat = bs + ws * dummy_x
dummy_yhat.shape
```

Output

```
(101, 101)
```

Thanks to its broadcasting capabilities, *Numpy* is able to understand we want to multiply the **same *x* value** by **every entry** in the **ws matrix**. This operation resulted in a **grid of predictions** for that **single data point**. Now we need to do this for **every one of our 80 data points** in the training set.

We can use *Numpy*'s `apply_along_axis` to accomplish this:

☺ Look ma, no loops!

```
all_predictions = np.apply_along_axis(
    func1d=lambda x: bs + ws * x,
    axis=1,
    arr=x_train,
)
all_predictions.shape
```

Output

```
(80, 101, 101)
```

Cool! We got **80 matrices** of shape (101, 101), **one matrix for each data point**, each matrix containing a **grid of predictions**.

The **errors** are the difference between the predictions and labels, but we cannot perform this operation right away - we need to work a bit on our **labels (*y*)**, so they have the proper **shape** for it (broadcasting is good, but not *that* good):

```
all_labels = y_train.reshape(-1, 1, 1)
all_labels.shape
```

Output

```
(80, 1, 1)
```

Our **labels** turned out to be **80 matrices of shape (1, 1)** - the most boring kind of matrix - but that is enough for broadcasting to work its magic. We can compute the **errors** now:

```
all_errors = (all_predictions - all_labels)
all_errors.shape
```

Output

```
(80, 101, 101)
```

Each prediction has its own error, so we got **80 matrices** of shape (101, 101), again, one matrix for each data point, each matrix containing a **grid of errors**.

The only step missing is to compute the **mean squared error**. First, we take the square of all errors. Then we **average the squares over all data points**. Since our data points are in the **first dimension**, we use `axis=0` to compute this average:

```
all_losses = (all_errors ** 2).mean(axis=0)
all_losses.shape
```

Output

```
(101, 101)
```

The result is a **grid of losses**, a matrix of shape (101, 101), **each loss** corresponding to a **different combination of the parameters *b* and *w***.

These losses are our **loss surface**, which can be visualized in a 3D plot, where the vertical axis (*z*) represents the loss values. If we **connect** the combinations of *b* and *w* that yield the **same loss value**, we'll get an **ellipse**. Then, we can draw this ellipse in the original *b* x *w* plane (in blue, for a loss value of 3). This is, in a nutshell, what a **contour plot** does. From now on, we'll always use the contour plot, instead of the corresponding 3D version.

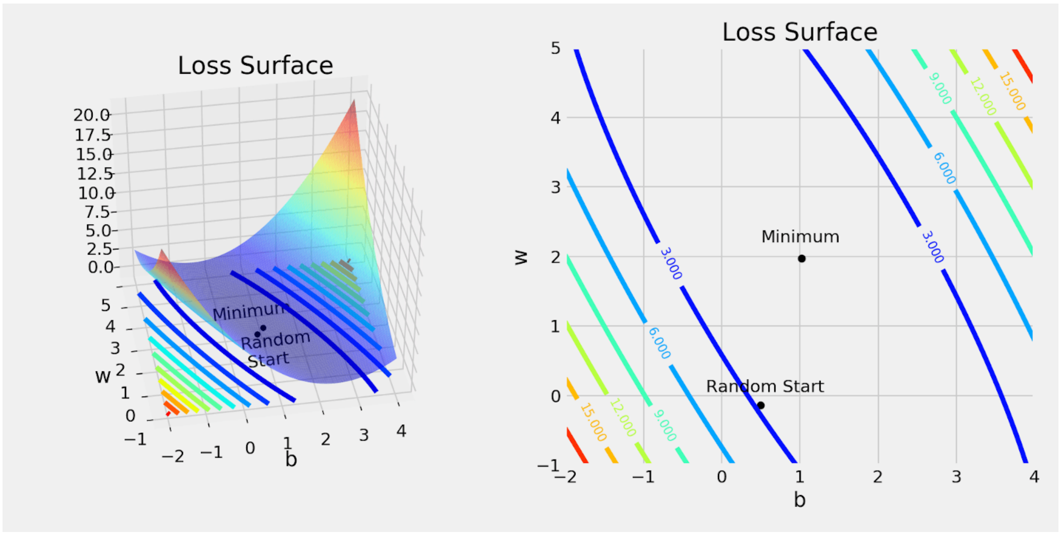


Figure 0.4 - Loss surface

In the center of the plot, where parameters (*b*, *w*) have values close to (1, 2), the loss is at its **minimum** value. This is the point we're trying to reach using gradient descent.

In the bottom, slightly to the left, there is the **random start** point, corresponding to our randomly initialized parameters.

This is one of the nice things about tackling a simple problem like a linear regression with a single feature: we have only **two parameters**, and thus **we can compute and visualize the loss surface**.

Unfortunately, for the absolute majority of problems, **computing the loss surface is not going to be feasible**: we have to rely on gradient descent's ability to reach a point of minimum, even if it starts at some random point.

Cross-Sections

Another nice thing is that we can cut a **cross-section** in the loss surface to check what the **loss** looks like if **the other parameter were held constant**.

Let's start by making ***b* =0.52** (the value from `b_range` that is closest to our initial random value for *b*, 0.4967) - we cut a cross-section *vertically* (the red dashed line) on our loss surface (left plot), and we get the resulting plot on the right:

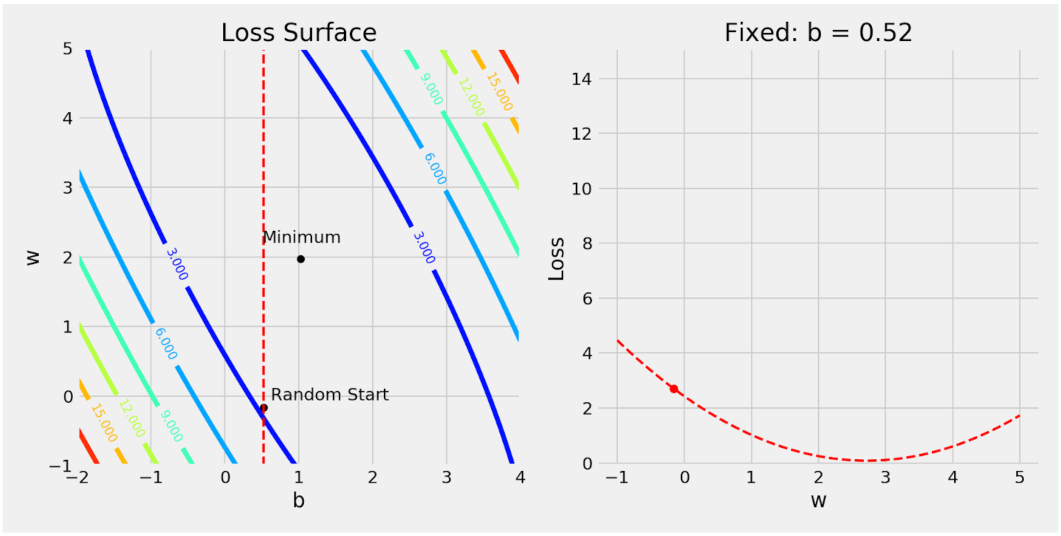


Figure 0.5 - Vertical cross-section - parameter **b** *is fixed*

What does this cross-section tell us? It tells us that, **if we keep *b* constant** (at 0.52), the **loss**, seen from the **perspective of parameter *w***, can be minimized if ***w* gets increased** (up to some value between 2 and 3).

Sure, **different values of *b*** produce **different cross-section loss curves for *w***. And those curves will depend on the **shape of the loss surface** (more on that later, in the "**Learning Rate**" section).

OK, so far, so good... what about the *other* cross-section? Let's cut it *horizontally* now, making ***w* = -0.16** (the value from `w_range` that is closest to our initial random value for *b*, -0.1382). The resulting plot is on the right:

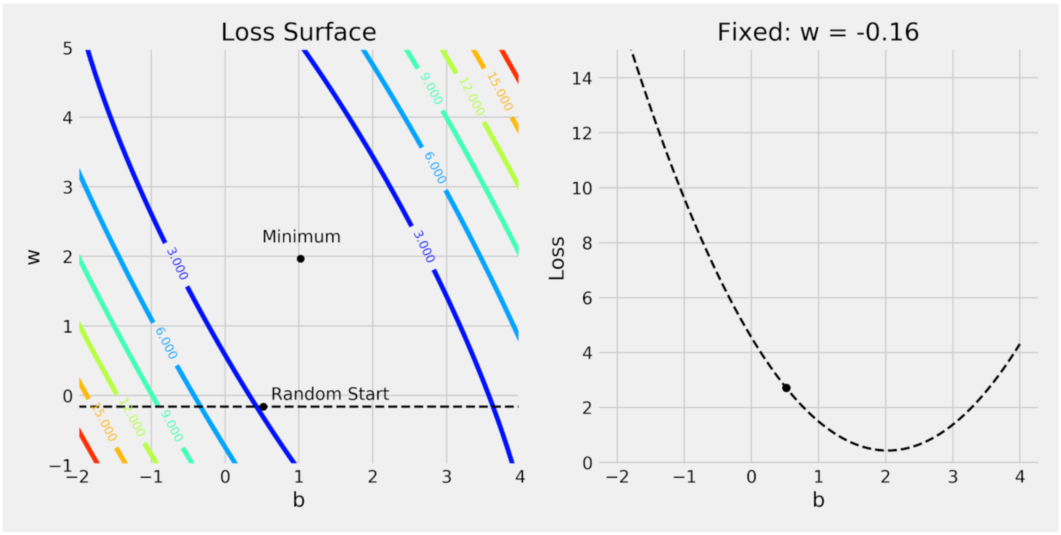


Figure 0.6 - Horizontal cross-section - parameter **w** *is fixed*

Now, **if we keep** ***w*** **constant** (at -0.16), the **loss**, seen from the **perspective of parameter b**, can be minimized if ***b*** **gets increased** (up to some value close to 2).

In general, the purpose of this cross-section is to get the **effect on the loss** of **changing a single parameter**, while keeping **everything else constant**. This is, in a nutshell, a **gradient** :-)

Now I have a question for you: **which one** of the two dashed curves, *red* (*w* changes, *b* is constant) or *black* (*b* changes, *w* is constant) yields the **biggest changes in loss** when we modify the changing parameter?

The answer is coming right up in the next section!

Step 3 - Compute the Gradients

A **gradient** is a **partial derivative** — *why* **partial**? Because one computes it *with respect to* (w.r.t.) a **single parameter**. We have two parameters, ***b*** and ***w***, so we must compute two partial derivatives.

A **derivative** tells you *how much* **a given quantity changes** when you *slightly* **vary** some **other quantity**. In our case, how much does our ***MSE* loss** change when we vary **each one of our two parameters separately**?

Gradient = **how much** the **loss** changes if **ONE parameter** changes **a little bit**!

The *right-most* part of the equations below is what you usually see in implementations of gradient descent for simple linear regression. In the **intermediate step**, I show you **all elements** that pop-up from the application of the chain rule[34], so you know how the final expression came to be.

$$
\begin{aligned}
\frac{\partial MSE}{\partial b} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial b} &= \frac{1}{n} \sum_{i=1}^{n} 2(b + wx_i - y_i) \\
&= 2\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \\
\frac{\partial MSE}{\partial w} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w} &= \frac{1}{n} \sum_{i=1}^{n} 2(b + wx_i - y_i) \cdot x_i \\
&= 2\frac{1}{n} \sum_{i=1}^{n} x_i(\hat{y}_i - y_i)
\end{aligned}
$$

Equation 0.4 - Computing gradients w.r.t coefficients **b** *and* **w** *using* **n** *points*

Just to be clear: we will always **use our "*regular*" error computed at the beginning of Step 2**. The loss surface is surely eye candy, but, as I mentioned before, it is only feasible to use it for educational purposes.

Step 3

```python
# Step 3 - Computes gradients for both "b" and "w" parameters
b_grad = 2 * error.mean()
w_grad = 2 * (x_train * error).mean()
print(b_grad, w_grad)
```

Output

```
-3.044811379650508 -1.8337537171510832
```

Visualizing Gradients

Since the **gradient for *b*** is **bigger** (in absolute value, 3.04) than the gradient for *w* (in absolute value, 1.83), the answer for the question I posed you in the "**Cross-Sections**" section is: the **black** curve (*b* changes, *w* is constant) yields the biggest changes in loss.

"Why is that?"

To answer that, let’s first put both cross-section plots side-by-side, so we can more easily compare them: what is the **main difference** between them?

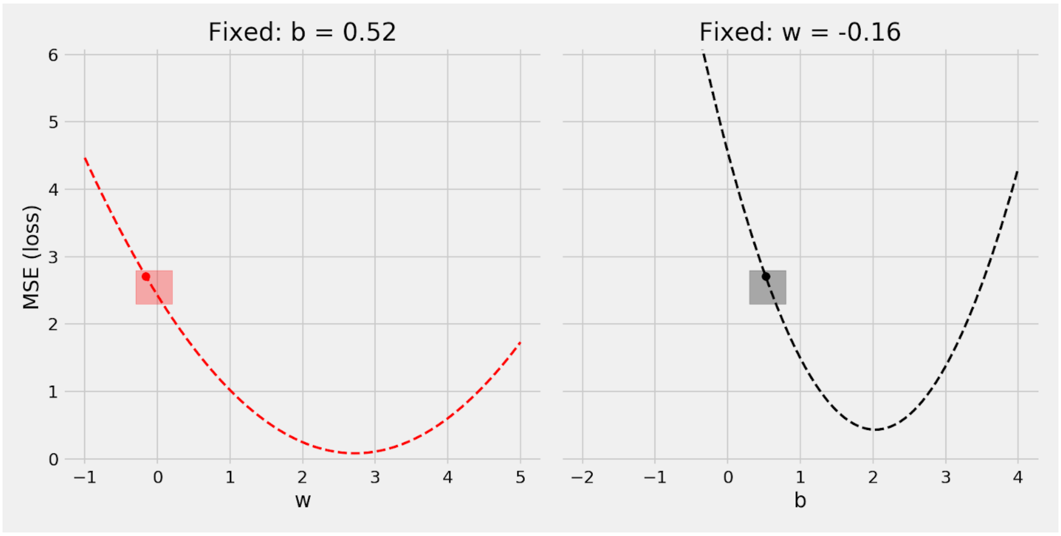


Figure 0.7 - Cross-sections of the loss surface

The curve on the right is **steeper**. That's your answer! **Steeper curves** have **bigger gradients**.

Cool! That's the intuition… now, let's get a bit more *geometrical*. So, I am **zooming in** the regions given by the *red* and *black* squares of Figure 0.7.

From the "**Cross-Sections**" section, we already know that to *minimize the loss*, both *b* and *w* needed to be **increased**. So, keeping the spirit of using gradients, let's **increase each parameter a *little bit*** (always keeping the other one fixed!). By the way, in this example, *a little bit* equals 0.12 (for convenience sake, so it results in a nicer plot).

What effect do these increases have on the loss? Let's check it out:

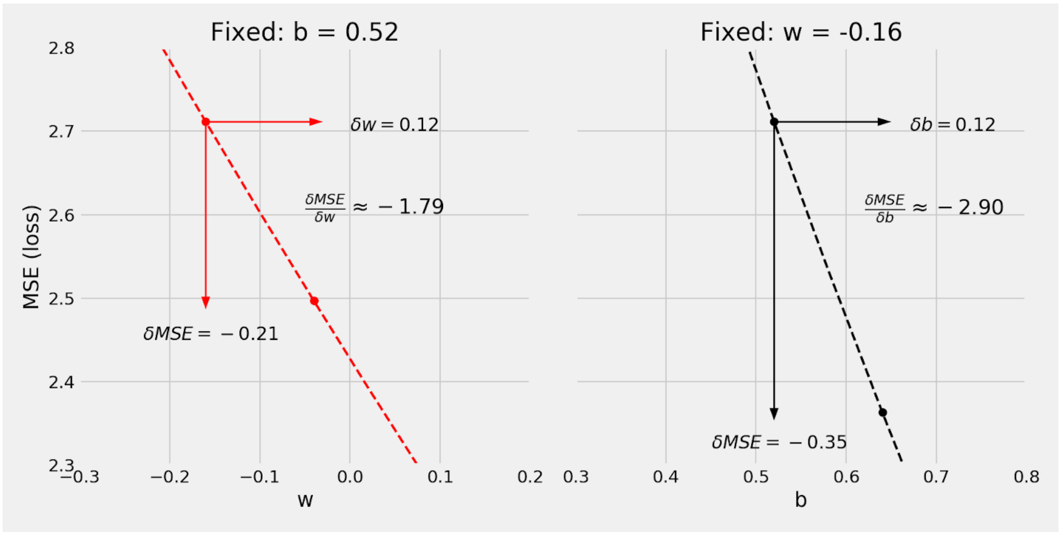


Figure 0.8 - Computing (approximate) gradients, geometrically

On the left plot, **increasing *w* by 0.12** yields a **loss reduction of 0.21**. The geometrically computed and roughly approximate gradient is given by the ratio between the two values: **-1.79**. How does this result compare to the actual value of the gradient (-1.83)? It is actually not bad for a crude approximation… Could it be better? Sure, if we make the **increase in *w* smaller and smaller** (like 0.01, instead of 0.12), we'll get **better and better** approximations… in the limit, as the **increase approaches zero**, we'll arrive at the **precise value of the gradient**. Well, that's the definition of a derivative!

The same reasoning goes for the plot on the right: **increasing *b* by the same 0.12** yields a **bigger loss reduction of 0.35**. Bigger loss reduction, bigger ratio, bigger gradient - and bigger error, too, since the geometric approximation (-2.90) is farther away from the actual value (-3.04).

Time for another question: **which curve**, red or black, you like **best** to **reduce the loss**? It should be the **black one**, right? Well, yes, but it is not as straightforward as we'd like it to be. We'll dig deeper into this in the "**Learning Rate**" section.

Backpropagation

Now that you’ve learned about *computing the gradient of the loss function w.r.t. to each parameter using the chain rule*, let me show you how *Wikipedia* describes **backpropagation** (highlights are mine):

> The backpropagation algorithm works by **computing the gradient of the loss function with respect to each weight by the chain rule**, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule;
>
> …
>
> The term *backpropagation* strictly refers only to the algorithm for computing the gradient, not how the gradient is used; but the term is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent.

Does it seem familiar? That’s it; **backpropagation** is nothing more than **"chained" gradient descent**. That’s, in a nutshell, how a neural network is trained: it uses backpropagation, starting at its **last layer** and working its way back, to update the weights through all the layers.

In our example, we have a **single layer**, even a **single neuron**, so there is no need to backpropagate anything (more on that in the next chapter).

Step 4 - Update the Parameters

In the final step, we **use the gradients to update** the parameters. Since we are trying to **minimize** our **losses**, we **reverse the sign** of the gradient for the update.

There is still another (hyper-)parameter to consider: the **learning rate**, denoted by the *Greek letter* ***eta*** (that looks like the letter ***n***), which is the **multiplicative factor** that we need to apply to the gradient for the parameter update.

$$b = b - \eta \frac{\partial MSE}{\partial b}$$
$$w = w - \eta \frac{\partial MSE}{\partial w}$$

Equation 0.5 - Updating coefficients **b** *and* **w** *using computed gradients and a learning rate*

We can also interpret this a bit differently: **each parameter** is going to have its value **updated by a constant value *eta*** (the learning rate), but this constant is going to be **weighted by how much that parameter contributes to minimizing the loss** (its gradient).

Honestly, I believe this way of thinking about the parameter update makes more sense: first, you decide on a *learning rate* that specifies your **step size**, while the *gradients* tell you the **relative impact** (on the loss) of taking a step for each parameter. Then you take a given **number of steps** that's **proportional** to that **relative impact**: **more impact**, **more steps**.

"*How to **choose** a learning rate?*"

That is a topic on its own and beyond the scope of this section as well. We'll get back to it later on…

In our example, let's start with a value of **0.1** for the learning rate (which is a relatively *big value*, as far as learning rates are concerned!).

Step 4

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1
print(b, w)

# Step 4 - Updates parameters using gradients and the
# learning rate
b = b - lr * b_grad
w = w - lr * w_grad

print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
[0.80119529] [0.04511107]
```

What's the **impact of one update** on our model? Let's visually check its predictions:

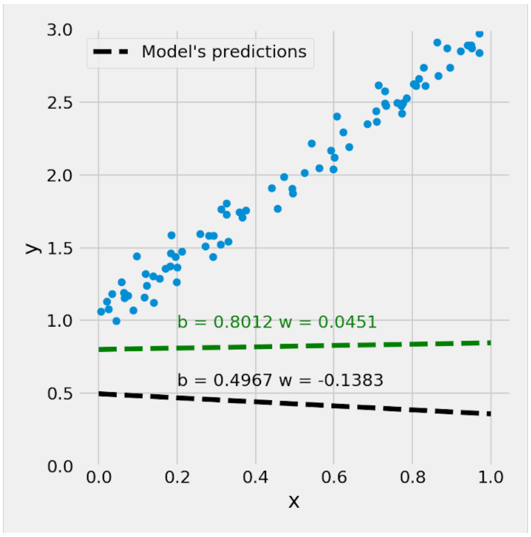


Figure 0.9 - Updated model's predictions

It looks better… at least it started pointing in the right direction :-)

Learning Rate

The **learning rate** is the most important hyper-parameter - there is a gigantic amount of material on how to *choose* a learning rate, how to *modify* the learning rate during the training, and how the *wrong* learning rate can completely ruin the model training.

Maybe you've seen this famous graph[35](from Stanford's CS231n class) that shows how a learning rate that is **too big** or **too small** affects the **loss** during training. Most people will see it (or have seen it) at some point in time. This is pretty much general knowledge, but I think it needs to be **thoroughly explained and visually demonstrated** to be *truly* understood. So, let's start!

I will tell you a little story (trying to build an analogy here, please bear with me!): imagine you are coming back from hiking in the mountains and you want to get back home as quickly as possible. At some point in your path, you can either choose to *go ahead* or to *make a right turn*.

The path *ahead* is almost *flat*, while the path to your *right* is kinda *steep*. The **steepness** is the **gradient**. If you take a single step one way or the other, it will lead to different outcomes (you'll descend more if you take one step to the right instead of going ahead).

But, here is the thing: you know that the path to your *right* is getting you home **faster**, so you don't take just one step, but **multiple steps** in that direction: **the steeper the path, the more steps you take**! Remember, "*more impact, more steps*"! You just cannot resist the urge to take that many steps; your behavior seems to be completely determined by the landscape. This analogy is getting weird, I know…

But, you still have **one choice**: you **can adjust the size of your step**. You can choose to take steps of any size, from *tiny steps* to *long strides*. That's your **learning rate**.

OK, let's see where this little story brought us so far… that's how you'll move, in a nutshell:

updated location = previous location + step size * number of steps

Now, compare it to what we did with the parameters:

updated value = previous value - learning rate * gradient

You got the point, right? I hope so because the analogy completely falls apart now… at this point, after moving in one direction (say, the *right turn* we talked about), you'd have to stop and move in the *other* direction (for just a fraction of a step, because the path was almost *flat*, remember?). And so on and so forth… Well, I don't think anyone has ever returned from hiking in such an orthogonal zig-zag path…

Anyway, let's explore further the **only choice** you have: the size of your *step*, I mean, the **learning rate**.

"*Choose your learning rate wisely.*"

Grail Knight

Small Learning Rate

It makes sense to start with *baby steps*, right? This means using a **small learning rate**. Small learning rates are **safe(r)**, as expected. If you were to take *tiny steps* while returning home from your hiking, you'd be more likely to arrive there safe and sound - but it would take a **lot of time**. The same holds true for training models: small learning rates will likely get you to (some) minimum point, **eventually**. Unfortunately, time is money, especially when you're paying for GPU time in the cloud… so, there is an *incentive* to try **bigger learning rates**.

How does this reasoning apply to our model? From computing our (geometric) gradients, we know we need to take a given **number of steps**: **1.79** (parameter *w*) and **2.90** (parameter *b*), respectively. Let's set our **step size to 0.2** (small-ish). It means we **move 0.36 for *w*** and **0.58 for *b***.

IMPORTANT: in real life, a learning rate of 0.2 is usually considered *BIG* - but in our very simple linear regression example, it still qualifies as small-ish.

Where does this movement lead us? As you can see in the plots below (as shown by the **new dots** to the right of the original ones), in both cases, the movement took us closer to the minimum - more so on the right because the curve is **steeper**.

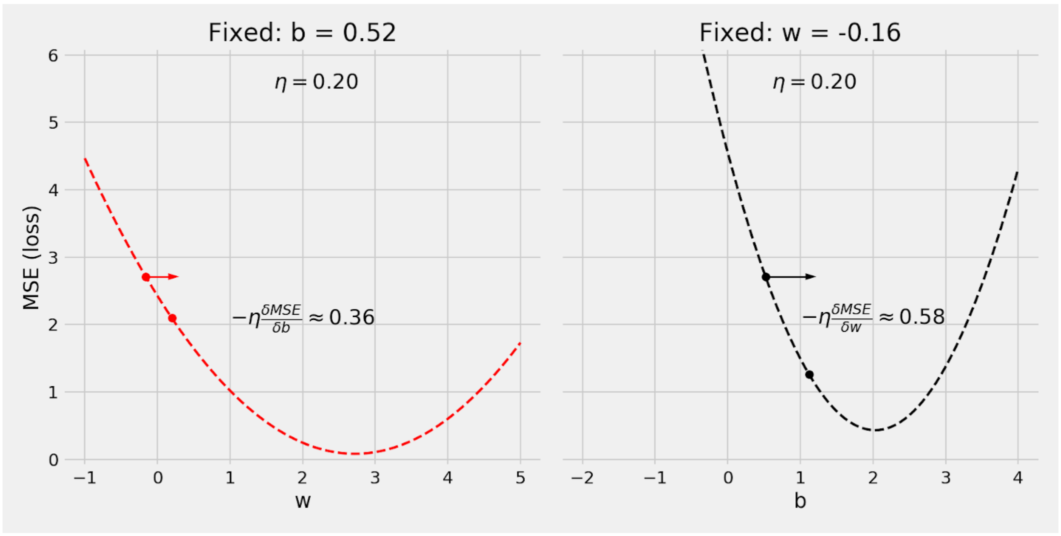


Figure 0.10 - Using a small-ish learning rate

Big Learning Rate

What would have happened if we had used a **big** learning rate instead, say, a **step size of 0.8**? As we can see in the plots below, we start to, literally, **run into trouble**…

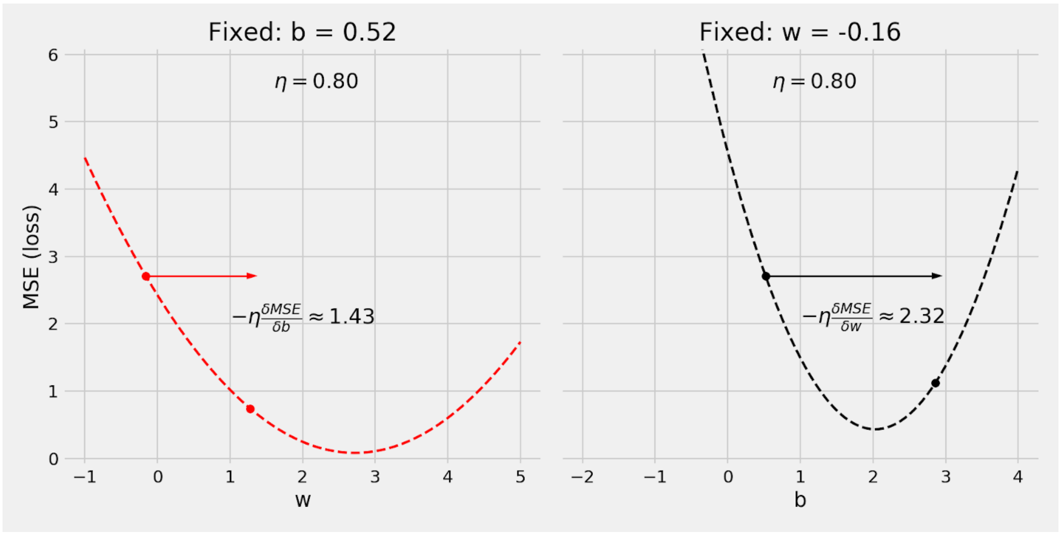


Figure 0.11 - Using a BIG learning rate

Even though everything is still OK on the left plot, the right plot shows us a completely different picture: **we ended up on the other side of the curve**. That is ***not*** good… you'd be going **back and forth**, alternately hitting both sides of the curve.

*"Well, even so, I may **still** reach the minimum, why is it so bad?"*

In our simple example, yes, you'd eventually reach the minimum because the **curve is nice and round**.

But, in real problems, the "curve" has some really **weird shape** that allows for **bizarre outcomes**, such as going back and forth **without ever approaching the minimum**.

In our analogy, you **moved so fast** that you **fell down** and hit the **other side of the valley**, then kept going down like a ***ping-pong***. Hard to believe, I know, but you definitely don't want that…

Very Big Learning Rate

Wait, it may get **worse** than that… let's use a **really big learning rate**, say, a **step size of 1.1**!

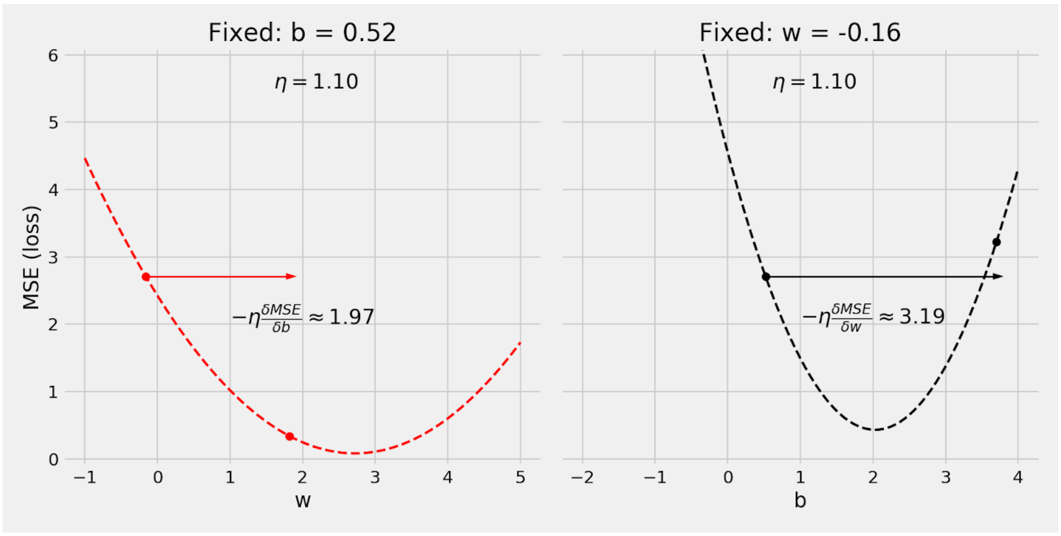


Figure 0.12 - Using a REALLY BIG learning rate

"He chose… poorly."

Grail Knight

Ok, that ***is*** bad… on the right plot, not only we ended up on the *other side of the curve* again, but we actually **climbed up**. This means **our loss increased**, instead of decreasing! How is that even possible? *You're moving so fast downhill that you end up climbing it back up*?! Unfortunately, the analogy cannot help us anymore. We need to think about this particular case in a different way…

First, notice that everything is *fine* on the left plot. The *enormous learning rate* **did not cause any issues** because the left curve is **less steep** than the one on the right. In other words, the curve on the left **can take bigger learning rates** than the curve on the right.

What can we learn from it?

Too big, for a **learning rate**, is a relative concept: it depends on **how steep** the curve is or, in other words, it depends on **how big the gradient is**.

We do have *many curves*, **many gradients**: one for each parameter. But we only have **one single learning rate** to choose (sorry, that's the way it is!).

It means that the **size of the learning rate is limited by the steepest curve**. All other curves must follow suit, meaning, they'd be using a *suboptimal* learning rate, given their shapes.

The reasonable conclusion is: it is **best** if all the **curves are equally steep**, so the **learning rate** is closer to optimal for all of them!

"Bad" Feature

How do we achieve *equally steep curves*? I'm on it! First, let's take a look at a *slightly modified* example, which I am calling the "bad" dataset:

- I **multiplied our feature (*x*) by 10**, so it is in the range [0, 10] now, and renamed it `bad_x`
- but since I **do not want the labels (*y*) to change**, I **divided the original `true_w` parameter by 10** and renamed it `bad_w` - this way, both `bad_w * bad_x` and `w * x` yield the same results

```
true_b = 1
true_w = 2
N = 100

# Data Generation
np.random.seed(42)

# We divide w by 10
bad_w = true_w / 10
# And multiply x by 10
bad_x = np.random.rand(N, 1) * 10

# So, the net effect on y is zero - it is still
# the same as before
y = true_b + bad_w * bad_x + (.1 * np.random.randn(N, 1))
```

Then I performed the same split as before for both, *original* and *bad*, datasets and plot the training sets side by side, as you can see below:

```
# Generates train and validation sets
# It uses the same train_idx and val_idx as before,
# but it applies to bad_x
bad_x_train, y_train = bad_x[train_idx], y[train_idx]
bad_x_val, y_val = bad_x[val_idx], y[val_idx]
```

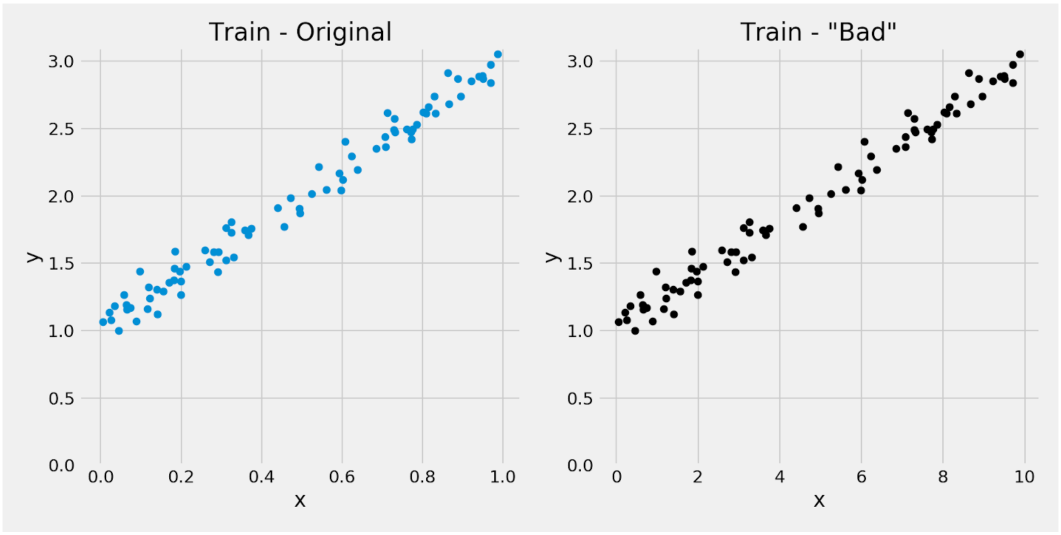


Figure 0.13 - Same data, different scales for feature **x**

The **only** difference between the two plots is the **scale of feature x**. Its range was [0, 1], now it is [0, 10]. The label *y* hasn't changed, and I did not touch `true_b`.

Does this simple **scaling** have any meaningful impact on our gradient descent? Well, if it hadn't, I wouldn't be asking it, right? Let's compute a new **loss surface** and compare to the one we had before:

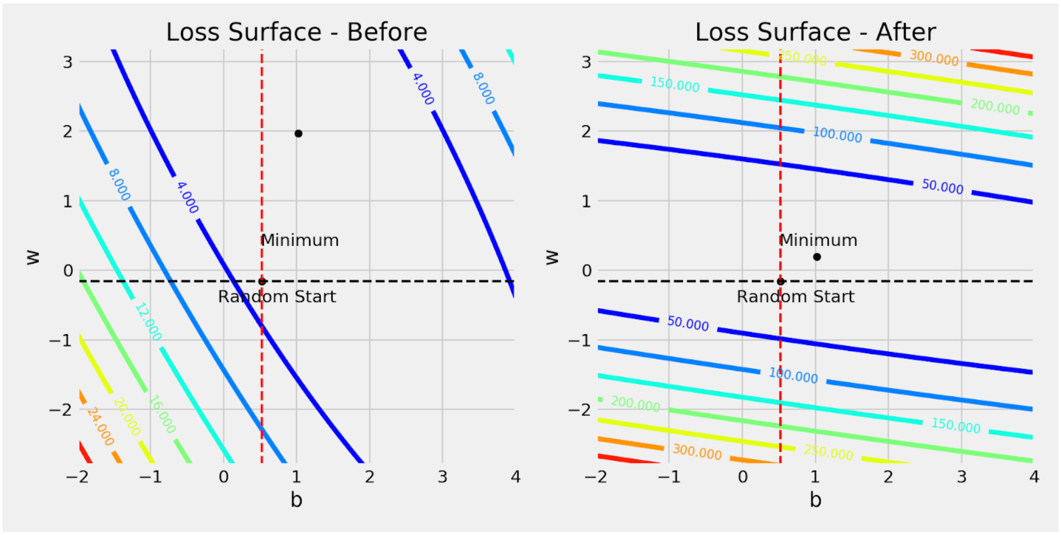


Figure 0.14 - Loss surface - before and after scaling feature **x** *(Obs.: left plot looks a bit different than Figure 0.6 because it is centered at the "after" minimum)*

Look at the **contour values** of Figure 0.14: the *dark blue* line was *3.0*, and now it is *50.0*! For the same range of parameter values, **loss values are much bigger**.

Let’s look at the *cross-sections* before and after we multiplied feature *x* by 10:

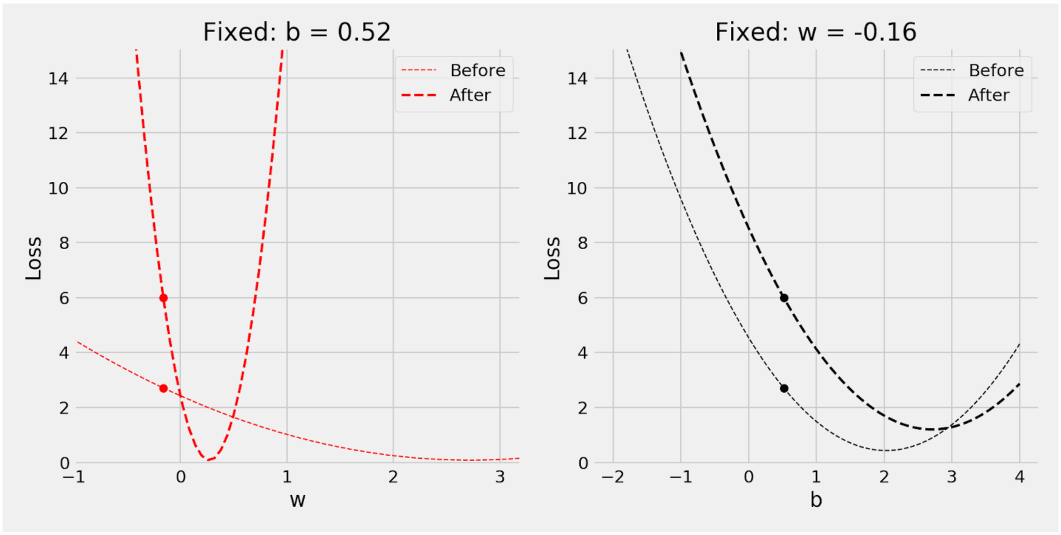


Figure 0.15 - Comparing cross-sections: before and after

What happened here? The **red curve** got much **steeper** (bigger gradient), and thus we must use a **smaller learning rate** to safely descend along it.

More importantly, the **difference** in **steepness** between the red and the black curves **increased**.

This is exactly what **WE NEED TO AVOID!**

Do you remember why?

Because the **size of the learning rate is limited by the steepest curve**!

How can we fix it? Well, we *ruined* it by **scaling it 10x bigger**... perhaps we can make it better if we **scale it in a different way**.

Scaling / Standardizing / Normalizing

Different how? There is this *beautiful* thing called the `StandardScaler`, which transforms a **feature** in such a way that it ends up with **zero mean** and **unit standard deviation**.

How does it achieve that? First, it computes the *mean* and the *standard deviation* of a given **feature (*x*)** using the training set (***N*** points):

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma(X) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{X})^2}$$

Equation 0.6 - Computing mean and standard deviation

Then it uses both values to **scale** the feature:

$$\text{scaled } x_i = \frac{x_i - \overline{X}}{\sigma(X)}$$

Equation 0.7 - Standardizing

If we were to recompute the mean and the standard deviation of the scaled feature, we would get 0 and 1, respectively. This preprocessing step is commonly referred to as *normalization*, although, technically, it should always be referred to as *standardization*.

Zero Mean and Unit Standard Deviation

Let's start with the **unit standard deviation**, that is, scaling the feature values such that its **standard deviation** equals **one**. This is one the **most important preprocessing steps**, not only for the sake of improving the performance of **gradient descent** but for other techniques such as **Principal Component Analysis (PCA)** as well. The **goal** is to have **all numerical features** in a **similar scale**, so the results are not affected by the original **range** of each feature.

Think of two common features in a model: *age* and *salary*. While *age* usually varies between 0 and 110, salaries can go from the low hundreds (say, 500) to several thousand (say, 9,000). If we compute the corresponding standard deviations, we may get values like 25 and 2,000, respectively. Thus, we need to **standardize** both features to have them on **equal footing**.

And then the **zero mean**, that is, **centering** the feature at **zero**. **Deeper neural networks** may suffer from a very serious condition called **vanishing gradients**. Since the gradients are used to update the parameters, smaller and smaller (that is, vanishing) gradients mean smaller and smaller updates, up to the point of a standstill: the network simply stops learning. One way to help the network to fight this condition is to **center its inputs**, the features, at **zero**. We'll get back to this later on while discussing *activation functions*.

IMPORTANT: preprocessing steps like the `StandardScaler` **MUST** be performed **AFTER** the train-validation-test split; otherwise, you'll be **leaking** information from the validation and/or test sets to your model!

After using the **training set only** to fit the `StandardScaler`, you should use its `transform` method to apply the preprocessing step to **all datasets**: training, validation, and test.

The code below will illustrate this well.

```
scaler = StandardScaler(with_mean=True, with_std=True)
# We use the TRAIN set ONLY to fit the scaler
scaler.fit(x_train)

# Now we can use the already fit scaler to TRANSFORM
# both TRAIN and VALIDATION sets
scaled_x_train = scaler.transform(x_train)
scaled_x_val = scaler.transform(x_val)
```

Notice that we are **not** regenerating the data - we are using the **original feature *x*** as input for the `StandardScaler` and transforming it into a **scaled *x***. The labels (*y*) are left untouched.

Let's plot the three of them, *original*, *"bad"* and *scaled*, side-by-side to illustrate the differences:

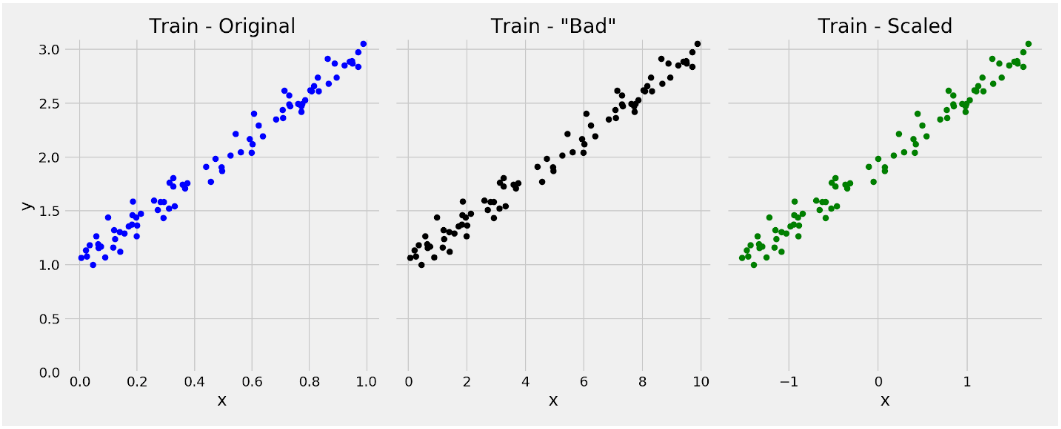


Figure 0.16 - Same data, three different scales for feature **x**

Once again, the **only** difference between the plots is the **scale of feature x**. Its range was originally [0, 1], then we made it into [0, 10], and now the `StandardScaler` made it into [-1.5, 1.5].

OK, time to check the **loss surface**: to illustrate the differences, I am plotting the three of them side-by-side: *original*, *"bad"* and *scaled*… it looks like this:

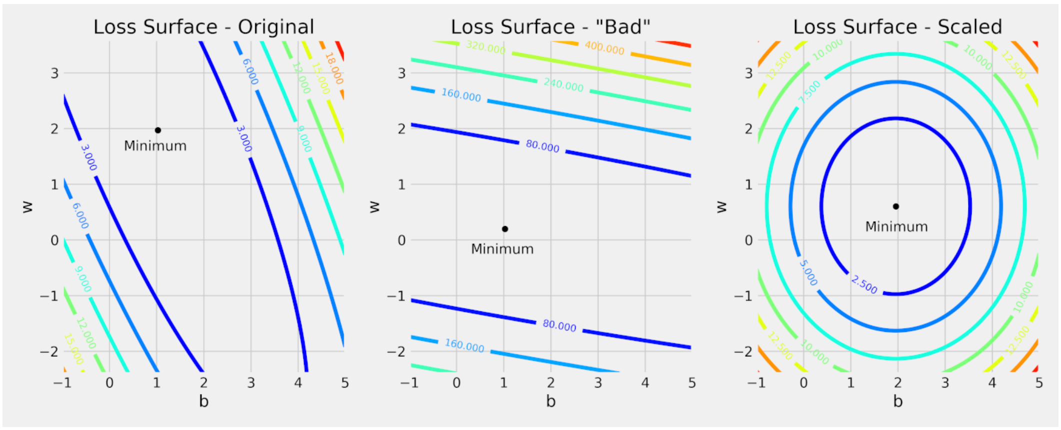


Figure 0.17 - Loss surfaces for different scales for feature **x** *(Obs.: left and center plots look a bit different than Figure 0.14 because they are centered at the "scaled" minimum)*

BEAUTIFUL, isn't it? The textbook definition of a **bowl** :-)

In practice, this is the **best surface** one could hope for: the **cross-sections** are going to be **similarly steep**, and a **good learning rate** for one of them is also good for the other.

Sure, in the real world, you'll never get a *pretty bowl* like that. But our conclusion still holds:

1. **Always standardize (scale) your features.**
2. **DO NOT EVER FORGET #1 :-)**

Step 5 - Rinse and Repeat!

Now we use the **updated parameters** to go back to **Step 1** and restart the process.

Definition of Epoch

An **epoch is complete whenever every point in the training set (N) has already been used in all steps: forward pass, computing loss, computing gradients, and updating parameters**.

During **one epoch**, we perform at least **one update**, but no more than **N updates**.

The number of **updates** (*N*/*n*) will depend on the type of gradient descent being used:

- for **batch** ($n = N$) gradient descent, this is trivial, as it uses all points for computing the loss — **one epoch** is the same as **one update**
- for **stochastic** ($n = 1$) gradient descent, **one epoch** means **N** updates, since every individual data point is used to perform an update
- for **mini-batch** (of size *n*), **one epoch** has **N/n updates**, since a mini-batch of *n* data points is used to perform an update

Repeating this process over and over for **many epochs** is, in a nutshell, **training** a model.

What happens if we run it over **1,000 epochs**?

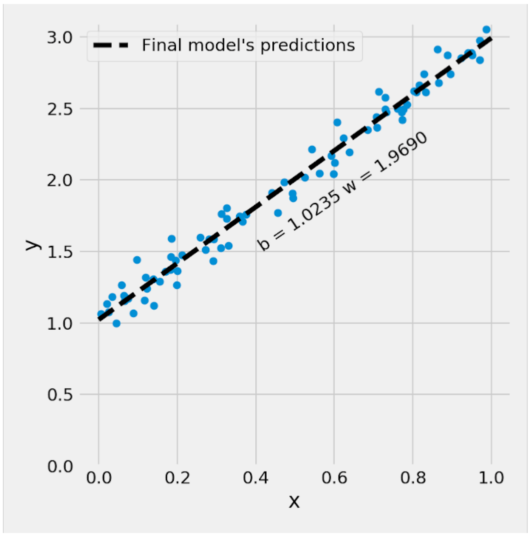


Figure 0.18 - Final model's predictions

In the next chapter, we'll put all these steps together and run it for 1,000 epochs, so we'll get to the parameters depicted in the figure above b = 1.0235 and w = 1.9690.

"Why 1,000 epochs?"

No particular reason, but this is a fairly simple model, and we can afford to run it over a large number of epochs. In more complex models, though, a couple of dozen epochs may be enough. We'll discuss this a bit more in Chapter 1.

The Path of Gradient Descent

In Step 3, we have seen the **loss surface** and both Random Start and Minimum points.

Which **path** is gradient descent going to take to go from **random start** to a **minimum**? **How long** will it take? Will it actually **reach the minimum**?

The answers to all these questions depend on many things, like the *learning rate*, the *shape of the loss surface*, and the **number of points** we use to compute the loss.

Depending on whether we use **batch**, **mini-batch**, or **stochastic** gradient descent, the path is going to be more or less **smooth**, and it is likely to reach the minimum in more or less **time**.

To illustrate the differences, I've generated paths over 100 **epochs** using either 80 data points (*batch*), 16 data points (*mini-batch*) or a single data point (*stochastic*) for computing the loss, as shown in the figure below:

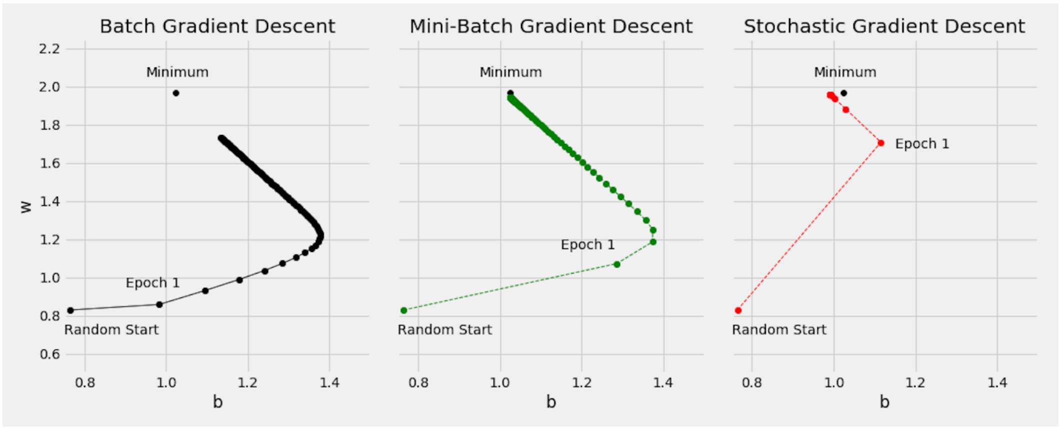


Figure 0.19 - The paths of Gradient Descent (Obs.: Random Start is different from Figure 0.4)

You can see that the resulting parameters at the end of **Epoch 1** differ greatly from one another. This is a direct consequence of the **number of updates** happening during **one epoch**, according to the batch size. In our example, for 100 epochs:

- 80 data points (*batch*): 1 update / epoch, totaling **100 updates**
- 16 data points (*mini-batch*): 5 updates / epoch, totaling **500 updates**
- 1 data point (*stochastic*): 80 updates / epoch, totaling **8.000 updates**

So, for both *center* and *right* plots, the **path between random start and Epoch 1** contains **multiple updates** which are not depicted in the plot (otherwise it would be *very* cluttered) - that's why the line connecting two epochs are **dashed**, instead of solid. In reality, there would be **zig-zagging lines** connecting every two epochs.

There are two things to notice:

- it should be no surprise that **mini-batch** gradient descent is able to get **closer to the minimum point** (using the same number of epochs) since it benefits from a *larger number of updates* than batch gradient descent
- **stochastic** gradient descent path is somewhat weird: it gets quite close to the **minimum point** at the end of Epoch 1 already, but then it seems to **fail to actually reach it**. But this is *expected* since it uses a *single data point for each update*, it will never stabilize, forever **hovering** in the neighborhood of the **minimum point**

Clearly, there is a **trade-off** here: either we have a **stable and smooth** trajectory, or we **move faster towards the minimum**.

Recap

This finishes our journey through the inner workings of **gradient descent**. By now, I hope you are able to develop better **intuition** about the many different aspects involved in the process.

In time, with practice, you'll observe the behaviors described here in your own models. Make sure to try plenty of different combinations: mini-batch sizes, learning rates, etc. This way, not only your models will learn, but so will you :-)

This is a (not so) short recap of everything we covered in this chapter:

- defining a **simple linear regression model**
- generating **synthetic data** for it
- performing a **train-validation split** on our dataset
- **randomly initializing the parameters** of our model
- performing a **forward pass**, that is, *making predictions* using our model
- computing the **errors** associated with our **predictions**

- *aggregating* the errors into a **loss** (mean squared error)
- learning that the **numbers of points** used to compute the **loss** defines the kind of gradient descent we're using: **batch** (all), **mini-batch** or **stochastic** (1)
- visualizing an example of a **loss surface** and using its **cross-sections** to get the **loss curves** for individual parameters
- learning that a **gradient is a partial derivative** and it represents **how much the loss changes if one parameter changes a little bit**
- computing the **gradients** for our model's parameters using **equations**, **code**, and **geometry**
- learning that **larger gradients** correspond to **steeper loss curves**
- learning that **backpropagation** is nothing more than "**chained**" gradient descent
- using the **gradients** and a **learning rate** to **update the parameters**
- comparing the **effects on the loss** of using **small**, **big** and **very big learning rates**
- learning that **loss curves** for all parameters should be, ideally, **similarly steep**
- visualizing the effects of using a **feature with a larger range**, making the loss curve for the corresponding parameter **much steeper**
- using Scikit Learn's `StandardScaler` to bring a feature to a reasonable range and thus making the **loss surface more bowl-shaped** and its cross-sections **similarly steep**
- learning that **preprocessing steps** like scaling should be applied **after the train-validation split** to prevent **leakage**
- figuring that performing **all steps** (forward pass, loss, gradients and parameter update) makes **one epoch**
- visualizing the **path of gradient descent** over many epochs and realizing it is heavily **dependent on the kind of gradient descent** used: batch, mini-batch or stochastic
- learning that there is a **trade-off** between the *stable and smooth path* of batch

gradient descent and the *fast and somewhat chaotic path* of stochastic gradient descent, making the use of **mini-batch gradient descent a good compromise** between the other two

You are now **ready** to put it all together and actually **train a model using PyTorch**!

[30] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter00.ipynb

[31] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter00.ipynb

[32] https://en.wikipedia.org/wiki/Gradient_descent

[33] https://en.wikipedia.org/wiki/Gaussian_noise

[34] https://en.wikipedia.org/wiki/Chain_rule

[35] https://bit.ly/2BxCxTO

Chapter 1
A Simple Regression Problem

Spoilers

In this chapter, we will:

- briefly **review** the steps of gradient descent (*optional*)
- use gradient descent to implement a **linear regression** in **Numpy**
- create **tensors in PyTorch** (**finally!**)
- understand the difference between **CPU** and **GPU tensors**
- understand PyTorch's main feature, **autograd**, to perform automatic differentiation
- visualize the **Dynamic Computation Graph**
- create a **loss function**
- define an **optimizer**
- implement our **own model class**
- implement **nested** and **sequential** models, using PyTorch's layers
- organize our code into three parts: **data preparation**, **model configuration** and **model training**

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 1[36] is part of the official "**Deep Learning with PyTorch Step-by-Step**" repository on GitHub. You can also run it directly in **Google Colab**[37].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 1's notebook. If not, just click on `Chapter01.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from sklearn.linear_model import LinearRegression

import torch
import torch.optim as optim
import torch.nn as nn
from torchviz import make_dot
```

A Simple Regression Problem

Most tutorials start with some nice and *pretty image classification problem* to illustrate how to use PyTorch. It may seem cool, but I believe it **distracts** you from the **main goal**: **how PyTorch works**?

For this reason, in this first example, I will stick with a **simple** and **familiar** problem: a **linear regression with a single feature** ***x***! It doesn't get much simpler than that…

$$y = b + wx + \epsilon$$

Equation 1.1 - Simple Linear Regression model

It is also possible to think of it as the **simplest neural network** possible: **one** input, **one** output, and **no** activation function (that is, **linear**).

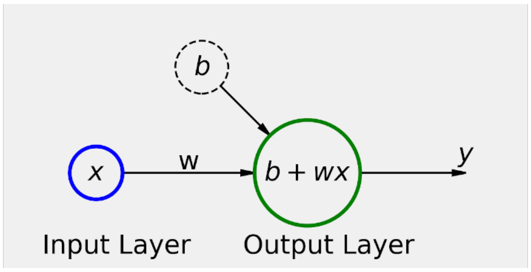


Figure 1.1 - The simplest of all neural networks

If you have read **Chapter 0**, you can either choose to skip to the "**Linear Regression in Numpy**" section or to use the **next two sections as a review**.

Data Generation

Let's start **generating** some synthetic data: we start with a vector of 100 (*N*) points for our **feature** (*x*) and create our **labels** (*y*) using ***b* = 1**, ***w* = 2**, and some **Gaussian noise**[38] (***epsilon***).

Synthetic Data Generation

Data Generation

```
true_b = 1
true_w = 2
N = 100

# Data Generation
np.random.seed(42)
x = np.random.rand(N, 1)
epsilon = (.1 * np.random.randn(N, 1))
y = true_b + true_w * x + epsilon
```

Next, let's **split** our synthetic data into **train** and **validation** sets, shuffling the array of indexes and using the first 80 shuffled points for training.

Notebook Cell 1.1 - *Splitting synthetic dataset into train and validation sets for linear regression*

```
# Shuffles the indices
idx = np.arange(N)
np.random.shuffle(idx)

# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

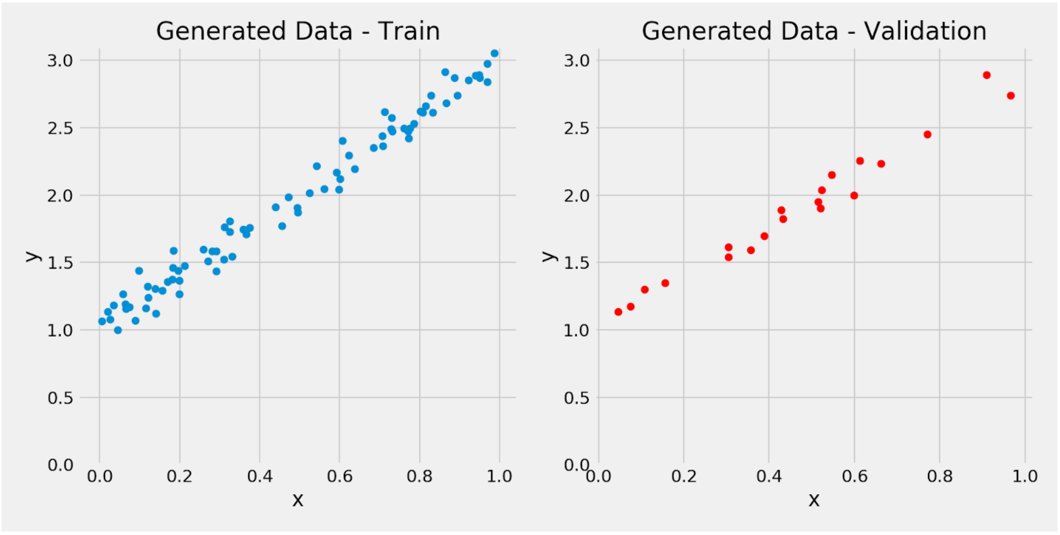


Figure 1.2 - Synthetic data: Train and Validation sets

We **know** that ***b*** **= 1**, ***w*** **= 2**, but now let's see **how close** we can get to the true values by using **gradient descent** and the 80 points in the **training set** (for training, *N* = 80).

Gradient Descent

I'll cover the **five basic steps** you'd need to go through to use gradient descent and the corresponding *Numpy* code.

Step 0 - Random Initialization

For training a model, you need to **randomly initialize the parameters/weights** (we have only two, ***b*** and ***w***).

Step 0

```
# Step 0 - Initializes parameters "b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)
w = np.random.randn(1)

print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
```

Step 1 - Compute Model's Predictions

This is the **forward pass** - it simply *computes the model's predictions using the current values of the parameters/weights*. At the very beginning, we will be producing **really bad predictions**, as we started with **random values from Step 0**.

Step 1

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train
```

Step 2 - Compute the Loss

For a regression problem, the **loss** is given by the **Mean Squared Error (MSE)**, that is, the average of all squared errors, that is, the average of all squared differences between **labels** (y) and **predictions** ($b + wx$).

In the code below, we are using **all data points** of the training set to compute the **loss**, so $n = N = 80$, meaning we are performing **batch gradient descent**.

Step 2

```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train)

# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()

print(loss)
```

Output

```
2.7421577700550976
```

Batch, **Mini-batch**, *and* **Stochastic Gradient Descent**

- if we use **all points** in the training set (n = **N**) to compute the loss, we are performing a **batch** gradient descent
- if we were to use a **single point** (n = **1**) each time, it would be a **stochastic** gradient descent
- anything else (n) **in-between 1 and N** characterizes a **mini-batch** gradient descent

Step 3 - Compute the Gradients

A **gradient** is a **partial derivative** — *why* **partial**? Because one computes it *with respect to* (w.r.t.) a **single parameter**. We have two parameters, ***b*** and ***w***, so we must compute two partial derivatives.

A **derivative** tells you *how much* **a given quantity changes** when you *slightly vary* some **other quantity**. In our case, how much does our ***MSE*** **loss** change when we vary **each one of our two parameters separately**?

Gradient = **how much** the **loss** changes if **ONE parameter** changes **a little bit**!

Step 3

```
# Step 3 - Computes gradients for both "b" and "w" parameters
b_grad = 2 * error.mean()
w_grad = 2 * (x_train * error).mean()
print(b_grad, w_grad)
```

Output

```
-3.044811379650508 -1.8337537171510832
```

Step 4 - Update the Parameters

In the final step, we **use the gradients to update** the parameters. Since we are trying to **minimize** our **losses**, we **reverse the sign** of the gradient for the update.

There is still another (hyper-)parameter to consider: the **learning rate**, denoted by the *Greek letter* ***eta*** (that looks like the letter ***n***), which is the **multiplicative factor** that we need to apply to the gradient for the parameter update.

"*How to* ***choose*** *a learning rate?*"

That is a topic on its own and beyond the scope of this section as well. We'll get back to it later on…

In our example, let's start with a value of **0.1** for the learning rate (which is a relatively *big value*, as far as learning rates are concerned!).

Step 4

```
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1
print(b, w)

# Step 4 - Updates parameters using gradients and
# the learning rate
b = b - lr * b_grad
w = w - lr * w_grad

print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
[0.80119529] [0.04511107]
```

Step 5 - Rinse and Repeat!

Now we use the **updated parameters** to go back to **Step 1** and restart the process.

Definition of Epoch

An **epoch is complete whenever every point in the training set (N) has already been used in all steps: forward pass, computing loss, computing gradients, and updating parameters.**

During **one epoch**, we perform at least **one update**, but no more than **N updates**.

The number of **updates** (*N*/*n*) will depend on the type of gradient descent being used:

- for **batch** (*n* = *N*) gradient descent, this is trivial, as it uses all points for computing the loss — **one epoch** is the same as **one update**
- for **stochastic** (*n* = 1) gradient descent, **one epoch** means **N** updates, since every individual data point is used to perform an update
- for **mini-batch** (of size *n*), **one epoch** has **N/n updates**, since a mini-batch of *n* data points is used to perform an update

Repeating this process over and over for **many epochs** is, in a nutshell, **training** a model.

Linear Regression in Numpy

It's time to implement our linear regression model using gradient descent and ***Numpy* only**.

"*Wait a minute… I thought this book was about PyTorch!*" Yes, it is, but this serves **two purposes**: *first*, to introduce the **structure** of our task, which will remain largely the same and, *second*, to show you the main **pain points** so you can fully appreciate how much PyTorch makes your life easier :-)

For training a model, there is a first **initialization step** (line numbers refer to **Notebook Cell 1.2** code below):

- Random initialization of parameters/weights (we have only two, *b* and *w*) — lines 3 and 4

- Initialization of hyper-parameters (in our case, only *learning rate* and *number of epochs*) — lines 9 and 11

Make sure to *always initialize your random seed* to ensure the **reproducibility** of your results. As usual, the random seed is 42[39], the **(second) least random**[40] of all random seeds one could possibly choose.

For each epoch, there are **four training steps** (line numbers refer to **Notebook Cell 1.2** code below):

- Compute model's predictions — this is the **forward pass** — line 15
- Compute the loss, using *predictions* and *labels* and the appropriate **loss function** for the task at hand — lines 20 and 22
- Compute the **gradients** for every parameter — lines 25 and 26
- **Update** the parameters — lines 30 and 31

For now, we will be using **batch** gradient descent only, meaning, we'll use **all data points** for each one of the four steps above. It also means that going **once** through all of the steps is already **one epoch**. Then, if we want to train our model over 1,000 epochs, we just need to add a **single loop**.

In Chapter 2, we'll introduce **mini-batch** gradient descent, and then we'll have to include a second *inner loop*.

"*Do we need to run it for 1,000 epochs? Shouldn't it* ***stop*** *automatically after getting close enough to the minimum loss?*"

Good question: we **don't** need to run it for 1,000 epochs. There are ways of **stopping** it earlier, once the progress is considered negligible (for instance, if the loss was barely reduced). These are called, most appropriately, **early stopping** methods. For now, since our model is a very simple one, we can afford to train it for 1,000 epochs.

Notebook Cell 1.2 - *Implementing gradient descent for linear regression using Numpy*

```
# Step 0 - Initializes parameters "b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)                                          ①
w = np.random.randn(1)                                          ①

print(b, w)

# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1                                                        ②
# Defines number of epochs
n_epochs = 1000                                                 ②

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train                                      ③

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train)                                    ④
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()                                  ④

    # Step 3 - Computes gradients for both "b" and "w" parameters
    b_grad = 2 * error.mean()                                   ⑤
    w_grad = 2 * (x_train * error).mean()                       ⑤

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    b = b - lr * b_grad                                         ⑥
    w = w - lr * w_grad                                         ⑥

print(b, w)
```

① Step 0: Random initialization of parameters/weights

② Initialization of hyper-parameters

③ Step 1: Forward pass

④ Step 2: Computing loss

⑤ Step 3: Computing gradients

⑥ Step 4: Updating parameters

Output

```
# b and w after initialization
[0.49671415] [-0.1382643]
# b and w after our gradient descent
[1.02354094] [1.96896411]
```

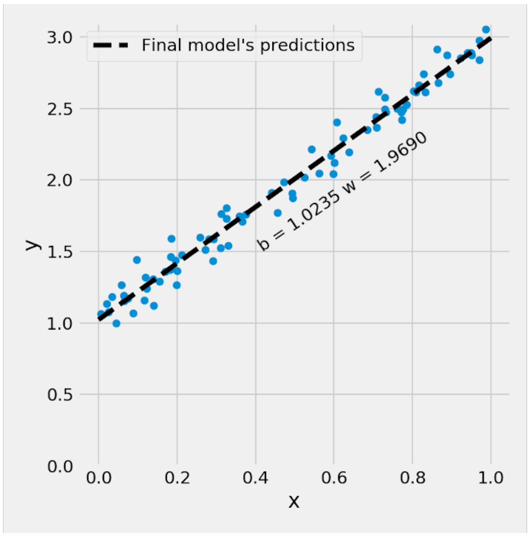


Figure 1.3 - Fully trained model's predictions

Just to make sure we haven't done any mistakes in our code, we can use Scikit-Learn's Linear Regression to fit the model and compare the coefficients.

```
# Sanity Check: do we get the same results as our
# gradient descent?
linr = LinearRegression()
linr.fit(x_train, y_train)
print(linr.intercept_, linr.coef_[0])
```

Output

```
# intercept and coef from Scikit-Learn
[1.02354075] [1.96896447]
```

They **match** up to 6 decimal places — we have a *fully working implementation of linear regression* using *Numpy*.

Time to **TORCH** it :-)

PyTorch

First, we need to cover a **few basic concepts** that may throw you off-balance if you don't grasp them well enough before going full-force on modeling.

In Deep Learning, we see **tensors** everywhere. Well, Google's framework is called *TensorFlow* for a reason! *What is a tensor, anyway*?

Tensor

In *Numpy*, you may have an **array** that has **three dimensions**, right? That is, technically speaking, a **tensor**.

A **scalar** (a single number) has **zero** dimensions, a **vector has one** dimension, a **matrix has two** dimensions and a **tensor has three or more** dimensions. That's it!

But, to keep things simple, it is commonplace to call vectors and matrices tensors as well — so, from now on, **everything is either a scalar or a tensor**.

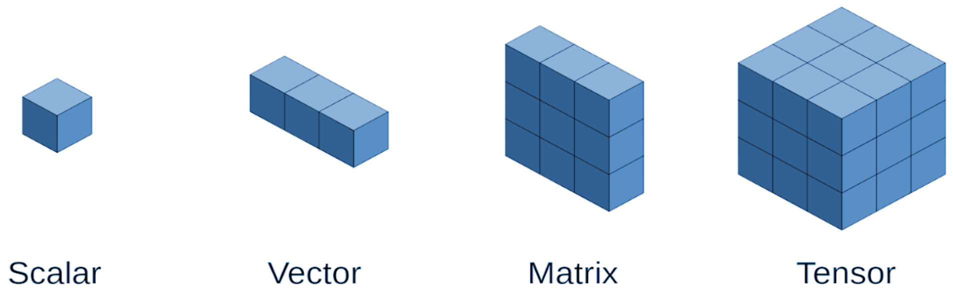


Figure 1.4 - Tensors are just higher-dimensional matrices - make sure to check this version out :-)

You can create **tensors** in PyTorch pretty much the same way you create **arrays** in *Numpy*. Using `tensor()` you can create either a scalar or a tensor.

PyTorch's tensors have equivalent functions as its *Numpy* counterparts, like: `ones()`, `zeros()`, `rand()`, `randn()` and many more. In the example below, we create one of each: scalar, vector, matrix and tensor or, saying it differently, one scalar and three tensors.

```
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)

print(scalar)
print(vector)
print(matrix)
print(tensor)
```

Output

```
tensor(3.1416)
tensor([1, 2, 3])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[[-1.0658, -0.5675, -1.2903, -0.1136],
         [ 1.0344,  2.1910,  0.7926, -0.7065],
         [ 0.4552, -0.6728,  1.8786, -0.3248]],

        [[-0.7738,  1.3831,  1.4861, -0.7254],
         [ 0.1989, -1.0139,  1.5881, -1.2295],
         [-0.5338, -0.5548,  1.5385, -1.2971]]])
```

You can get the shape of a tensor using its `size()` method or its `shape` attribute.

```
print(tensor.size(), tensor.shape)
```

Output

```
torch.Size([2, 3, 4]) torch.Size([2, 3, 4])
```

All tensors have shapes, but scalars have "empty" shapes, since they are **dimensionless** (or zero dimensions, if you prefer):

```
print(scalar.size(), scalar.shape)
```

Output

```
torch.Size([]) torch.Size([])
```

You can also reshape a tensor using its `view()` (*preferred*) or `reshape()` methods.

Beware: the `view()` method only returns a **tensor** with the desired shape that **shares the underlying data** with the **original tensor** - it **DOES NOT create a new, independent, tensor**!

The `reshape()` method **may** or **may not** create a copy! The reasons behind this apparently weird behavior are beyond the scope of this section - but this behavior is the reason why **`view()` is preferred** :-)

```
# We get a tensor with a different shape but it still is
# the SAME tensor
same_matrix = matrix.view(1, 6)
# If we change one of its elements...
same_matrix[0, 1] = 2.
# It changes both variables: matrix and same_matrix
print(matrix)
print(same_matrix)
```

Output

```
tensor([[1., 2., 1.],
        [1., 1., 1.]])
tensor([[1., 2., 1., 1., 1., 1.]])
```

If you want to copy all data for real, that is, **duplicate the data** in memory, you may use either its `new_tensor()` or `clone()` methods.

```
# We can use "new_tensor" method to REALLY copy it  into a new one
different_matrix = matrix.new_tensor(matrix.view(1, 6))
# Now, if we change one of its elements...
different_matrix[0, 1] = 3.
# The original tensor (matrix) is left untouched!
# But we get a "warning" from PyTorch telling us
# to use "clone()" instead!
print(matrix)
print(different_matrix)
```

Output

```
tensor([[1., 2., 1.],
        [1., 1., 1.]])
tensor([[1., 3., 1., 1., 1., 1.]])
```

Output

```
UserWarning: To copy construct from a tensor, it is
recommended to use sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True),
rather than tensor.new_tensor(sourceTensor).
 """Entry point for launching an IPython kernel.
```

It seems that PyTorch prefers that we use `clone()` - together with `detach()` - instead of `new_tensor()`... Both ways accomplish **exactly the same result**, but the code below is deemed cleaner and more readable.

```
# Lets follow PyTorch's suggestion and use "clone" method
another_matrix = matrix.view(1, 6).clone().detach()
# Again, if we change one of its elements...
another_matrix[0, 1] = 4.
# The original tensor (matrix) is left untouched!
print(matrix)
print(another_matrix)
```

Output

```
tensor([[1., 2., 1.],
        [1., 1., 1.]])
tensor([[1., 4., 1., 1., 1., 1.]])
```

You're probably asking yourself: "*but, what about the* `detach()` *method, what does it do?*"

It *removes the tensor from the computation graph*, which probably raises more questions than it answers, right? Don't worry, we'll get back to it later in this chapter.

Loading Data, Devices and CUDA

It is time to start converting our *Numpy* code to PyTorch: we'll start with the **training data**, that is, our `x_train` and `y_train` arrays.

"How do we go from Numpy's arrays to PyTorch's tensors?"

That's what `as_tensor()` is good for (which works like `from_numpy()`).

This operation **preserves the type** of the array:

```
x_train_tensor = torch.as_tensor(x_train)
x_train.dtype, x_train_tensor.dtype
```

Output

```
(dtype('float64'), torch.float64)
```

You can also easily **cast** it to a different type, like a *lower precision* (32-bit) float which will occupy *less space in memory*, using `float()`:

```
float_tensor = x_train_tensor.float()
float_tensor.dtype
```

Output

```
torch.float32
```

IMPORTANT: both `as_tensor()` and `from_numpy()` return a tensor that **shares the underlying data** with the original Numpy array. Similarly to what happened when we used `view()` in the last section, if you **modify the original Numpy array**, you're modifying the corresponding **PyTorch tensor too**, and vice-versa.

```
dummy_array = np.array([1, 2, 3])
dummy_tensor = torch.as_tensor(dummy_array)
# Modifies the numpy array
dummy_array[1] = 0
# Tensor gets modified too...
dummy_tensor
```

Output

```
tensor([1, 0, 3])
```

"What do I need `as_tensor()` *for? Why can't I just use* `torch.tensor()`*?"*

Well, you could… just keep in mind that, `torch.tensor()` always **makes a copy of the data**, instead of sharing the underlying data with the Numpy array.

You can also perform the **opposite** operation, namely, transforming a PyTorch tensor back to a *Numpy* array. That's what `numpy()` is good for:

```
dummy_tensor.numpy()
```

Output

```
array([1, 0, 3])
```

So far, we have only created **CPU tensors**. What does it mean? It means the **data** in the tensor is **stored** in the computer's **main memory** and any operations performed on it are going to be **handled by its CPU** (the **C**entral **P**rocessing **Unit**, for instance, an Intel® Core™ i7 Processor). So, although the data is, technically speaking, in the memory, we're still calling this kind of tensor a **CPU tensor**.

"Is there any other kind of tensor?"

Yes, there is also a **GPU tensor**. A **GPU** (which stands for **G**raphics **P**rocessing **U**nit) is the **processor** of a **graphics card**. These tensors **store their data** in the **graphics card's memory** and operations on top of them are performed by the **GPU**. For more information on the differences between CPUs and GPUs, please refer to this link[41].

If you have a graphics card from NVIDIA, you can use the **power of its GPU** to

speed up model training. PyTorch supports the use of these GPUs for model training using **CUDA** (**C**ompute **U**nified **D**evice **A**rchitecture), which needs to be previously installed and configured (please refer to the **Setup Guide** for more information on this).

If you **do** have a **GPU** (and you managed to install CUDA), we're getting to the part where you get to use it with PyTorch. But, even if you **do not** have a **GPU**, you should stick around in this section anyway… why? First, you can use a **free GPU from Google Colab** and, second, you should always make your code *GPU-ready*, that is, it should **automatically run in a GPU, if one is available**.

"How do I know if a GPU is available?"

PyTorch got your back once more — you can use `cuda.is_available()` to find out if you have a GPU at your disposal and set your device accordingly. So, it is good practice to figure this out at the top of your code:

Defining your device

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

So, if you don't have a GPU, your `device` is called `cpu`. If you do have a GPU, your device is called `cuda` or `cuda:0`. Why isn't it called `gpu`, then? Don't ask me… The important thing is, your code will be able to always use the appropriate device.

"*Why* `cuda:0`*? Are there others, like* `cuda:1`, `cuda:2` *and so on?*"

There may be if you are lucky enough to have *multiple GPUs* in your computer. Since this is usually *not* the case, I am assuming you have either **one GPU** or **none**. So, when we tell PyTorch to send a tensor to `cuda` without any numbering, it will send to the current CUDA device, which is the device #0 by default.

If you are using someone else's computer and you don't know how many GPUs it has, or which model they are, you can figure it out using `cuda.device_count()` and `cuda.get_device_name()`:

```
n_cudas = torch.cuda.device_count()
for i in range(n_cudas):
    print(torch.cuda.get_device_name(i))
```

Output

```
GeForce GTX 1060 6GB
```

In my case, I have only *one GPU*, and it is a *GeForce GTX 1060* model with 6 GB RAM.

There is only one thing left to do: turn our tensor into a **GPU tensor**. That's what `to()` is good for. It sends a tensor to the specified **device**.

```
gpu_tensor = torch.as_tensor(x_train).to(device)
gpu_tensor[0]
```

Output - GPU

```
tensor([0.7713], device='cuda:0', dtype=torch.float64)
```

Output - CPU

```
tensor([0.7713], dtype=torch.float64)
```

In this case, there is no `device` information in the printed output because PyTorch simply assumes the default (`cpu`).

"*Should I use* `to(device)`, *even if I am using CPU only?*"

Yes, you should, because there is **no cost** in doing so. If you have only a CPU, your tensor is already a CPU tensor, so nothing will happen. But if you share your code with others on GitHub, whoever has a GPU will benefit from it.

Let's put it all together now and make our **training data ready for PyTorch**.

Notebook Cell 1.3 - *Loading data: turning Numpy arrays into PyTorch tensors*

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

So, we defined a device, converted both *Numpy* arrays into PyTorch tensors, cast them to floats, and sent them to the device. Let's take a look at the types:

```
# Here we can see the difference - notice that .type() is more
# useful since it also tells us WHERE the tensor is (device)
print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

Output - GPU

```
<class 'numpy.ndarray'> <class 'torch.Tensor'>
torch.cuda.FloatTensor
```

Output - CPU

```
<class 'numpy.ndarray'> <class 'torch.Tensor'>
torch.FloatTensor
```

If you compare the **types** of both variables, you'll get what you'd expect: `numpy.ndarray` for the first one and `torch.Tensor` for the second one.

But where does the `x_train_tensor` "live"? Is it a CPU or a GPU tensor? You can't say... but if you use PyTorch's `type()`, it will reveal its **location** —

`torch.cuda.FloatTensor` — a GPU tensor in this case (assuming the output using a GPU, of course).

There is one more thing to be aware of when using GPU tensors… remember `numpy()`? What if we want to turn a GPU tensor back into a *Numpy* array? We'll get an **error**:

```
back_to_numpy = x_train_tensor.numpy()
```

Output

```
TypeError: can't convert CUDA tensor to numpy. Use
Tensor.cpu() to copy the tensor to host memory first.
```

Unfortunately, *Numpy* **cannot** handle GPU tensors… you need to make them CPU tensors first using `cpu()`:

```
back_to_numpy = x_train_tensor.cpu().numpy()
```

So, to avoid this error, use *first* `cpu()` and *then* `numpy()`, even if you are using a CPU. It follows the same principle of `to(device)`: you may share your code with others who may be using a GPU.

Creating Parameters

What distinguishes a *tensor* used for *training data (or validation, or test)* — like the ones we've just created — from a **tensor** used as a (*trainable*) **parameter/weight**?

The latter requires the **computation of its gradients**, so we can **update** their values (the parameters' values, that is). That's what the `requires_grad=True` argument is good for. It tells PyTorch to compute gradients for us.

A tensor for a **learnable parameter** requires a **gradient**!

You may be tempted to create a simple tensor for a parameter and, later on, send it to your chosen device, as we did with our data, right? *Not so fast…*

In the next few pages, I will present you **four** chunks of code showing different attempts at creating parameters.

The first three attempts are shown to *build up* to a solution. The first one only works well if you're never using a GPU. The second one doesn't work at all. The third one works, but it is too verbose.

The **recommended** way of creating parameters is the **last**: **Notebook Cell 1.4**.

The first chunk of code below creates two tensors for our parameters, including gradients, and all. But they are **CPU** tensors, by default.

```
# FIRST
# Initializes parameters "b" and "w" randomly, ALMOST as we
# did in Numpy since we want to apply gradient descent on
# these parameters we need to set REQUIRES_GRAD = TRUE
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, dtype=torch.float)
w = torch.randn(1, requires_grad=True, dtype=torch.float)
print(b, w)
```

Output

```
tensor([0.3367], requires_grad=True)
tensor([0.1288], requires_grad=True)
```

Never forget to set the **seed** to ensure reproducibility, just like we did before while using Numpy. PyTorch's equivalent is `torch.manual_seed()`.

*"If I use the **same seed** in PyTorch as I used in Numpy (or, to put it differently, if I use 42 everywhere), will I get the **same numbers**?"*

Unfortunately, **NO**.

You'll get the **same numbers** for the **same seed** in the **same package**. PyTorch generates a number sequence that is different from the one generated by *Numpy*, even if you use the same seed in both.

I am assuming you'd like to use your **GPU** (or the one from Google Colab), right? So we need to **send those tensors to the device**. We can try the **naive** approach, the one that worked well for sending the training data to the device. That's our second (and failed) attempt:

```
# SECOND
# But what if we want to run it on a GPU? We could just
# send them to device, right?
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, dtype=torch.float).to(device)
w = torch.randn(1, requires_grad=True, dtype=torch.float).to(device)
print(b, w)
# Sorry, but NO! The to(device) "shadows" the gradient...
```

Output

```
tensor([0.3367], device='cuda:0', grad_fn=<CopyBackwards>)
tensor([0.1288], device='cuda:0', grad_fn=<CopyBackwards>)
```

We succeeded in sending them to another device, but we **"lost"** the **gradients** somehow, since there is no more `requires_grad=True`, (don't bother the weird `grad_fn`). Clearly, we need to do better…

In the third chunk, we **first** send our tensors to the **device** and **then** use `requires_grad_()` method to set its `requires_grad` attribute to `True` in place.

In PyTorch, every method that **ends** with an **underscore** (_), like the `requires_grad_()` method above, makes changes **in-place**, meaning, they will **modify** the underlying variable.

```
# THIRD
# We can either create regular tensors and send them to
# the device (as we did with our data)
torch.manual_seed(42)
b = torch.randn(1, dtype=torch.float).to(device)
w = torch.randn(1, dtype=torch.float).to(device)
# and THEN set them as requiring gradients...
b.requires_grad_()
w.requires_grad_()
print(b, w)
```

Output

```
tensor([0.3367], device='cuda:0', requires_grad=True)
 tensor([0.1288], device='cuda:0', requires_grad=True)
```

This approach worked fine; we managed to end up with gradient-requiring **GPU tensors** for our parameters *b* and *w*. It seems a lot of work, though… Can we do better still?

Yes, we **can** do better: we can **assign** tensors to a **device** at the moment of their **creation**.

Notebook Cell 1.4 - *Actually creating variables for the coefficients :-)*

```
# FINAL
# We can specify the device at the moment of creation
# RECOMMENDED!

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
print(b, w)
```

Output

```
tensor([0.1940], device='cuda:0', requires_grad=True)
tensor([0.1391], device='cuda:0', requires_grad=True)
```

Much easier, right?

Always **assign** tensors to a **device** at the moment of their **creation** to avoid unexpected behaviors!

If you **do not** have a **GPU**, your outputs are going to be slightly different:

Output - CPU

```
tensor([0.3367], requires_grad=True)
tensor([0.1288], requires_grad=True)
```

*"Why are they different, even if I am using the **same seed**?"*

Similarly to what happens with using the same seed in **different packages** (Numpy and PyTorch), we also get **different sequences of random numbers** if PyTorch generates them in **different devices** (CPU and GPU).

Now that we know how to create tensors that require gradients, let's see how PyTorch handles them — that's the role of the…

Autograd

Autograd is PyTorch's *automatic differentiation package*. Thanks to it, we **don't need to worry** about *partial derivatives*, *chain rule*, or anything like it.

backward

So, how do we tell PyTorch to do its thing and **compute all gradients**? That's the role of the `backward()` method. It will compute gradients for *all (requiring gradient) tensors* involved in the computation of a given variable.

Do you remember the **starting point** for **computing the gradients**? It was the **loss**, as we computed its partial derivatives w.r.t. our parameters. Hence, we need to invoke the `backward()` method from the corresponding Python variable: `loss.backward()`.

Notebook Cell 1.5 - *Autograd in action!*

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor

# Step 2 - Computes the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train_tensor)
# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()

# Step 3 - Computes gradients for both "b" and "w" parameters
# No more manual computation of gradients!
# b_grad = 2 * error.mean()
# w_grad = 2 * (x_tensor * error).mean()
loss.backward() ①
```

① New "Step 3 - Computing Gradients" using `backward`

Which tensors are going to be handled by the `backward()` method applied to the `loss`?

- `b`
- `w`
- `yhat`
- `error`

We have set `requires_grad=True` to both `b` and `w`, so they are obviously included in the list. We use them both to compute `yhat`, so it will also make it to the list. Then we use `yhat` to compute the `error`, which is also added to the list.

Do you see the pattern here? If a tensor in the list is used to compute another tensor, the latter will also be included in the list. Tracking these dependencies is exactly what the **dynamic computation graph** is doing, as we'll see shortly.

What about `x_train_tensor` and `y_train_tensor`? They are involved in the computation too… but we created them as **not** gradient-requiring tensors, so `backward()` does not care about them.

```
print(error.requires_grad, yhat.requires_grad, \
      b.requires_grad, w.requires_grad)
print(y_train_tensor.requires_grad, x_train_tensor.requires_grad)
```

Output

```
True True True True
False False
```

grad

What about the **actual values** of the **gradients**? We can inspect them by looking at the `grad` **attribute** of a tensor.

```
print(b.grad, w.grad)
```

Output

```
tensor([-3.3881], device='cuda:0')
tensor([-1.9439], device='cuda:0')
```

If you check the method's documentation, it clearly states that **gradients are accumulated**. What does it mean? It means that, if we run **Notebook Cell 1.5**'s code (Steps 1 to 3) twice and check the `grad` attribute afterward, we will end up with:

Output

```
tensor([-6.7762], device='cuda:0')
tensor([-3.8878], device='cuda:0')
```

If you **do not** have a **GPU**, your outputs are going to be slightly different:

Output

```
tensor([-3.1125]) tensor([-1.8156])
```

Output

```
tensor([-6.2250]) tensor([-3.6313])
```

These gradients' values are exactly **twice** as much as they were before, as expected!

OK, but that is actually a **problem**: we need to use the gradients corresponding to the **current** loss to perform the parameter update. We should **NOT** use **accumulated gradients**.

*"If **accumulating gradients** is a **problem**, why does PyTorch do it by default?"*

It turns out; this behavior can be useful to circumvent hardware limitations.

During the training of large models, the necessary number of data points in a mini-batch may be **too big to fit in memory** (of the graphics card). How to solve this, other than buying more expensive hardware?

One can **split a mini-batch** into "*sub-mini-batches*" (horrible name, I know, don't quote me on this!), compute the gradients for those "sub" and **accumulate** them to achieve the same result of computing the gradients on the **full** mini-batch.

Sounds confusing? No worries, this is fairly advanced already and somewhat outside of the scope of this book, but I thought this particular behavior of PyTorch needed to be explained.

Luckily, this is easy to solve…

zero_

Every time we use the **gradients** to **update** the parameters, we need to **zero the gradients afterward**. And that's what `zero_()` is good for.

```
# This code will be placed _after_ Step 4
# (updating the parameters)
b.grad.zero_(), w.grad.zero_()
```

Output

```
(tensor([0.], device='cuda:0'),
tensor([0.], device='cuda:0'))
```

What does the **underscore** (_) at the **end of the method's name** mean? Do you remember? If not, go back to the previous section and find out.

So, let's **ditch** the **manual computation of gradients** and use both `backward()` and `zero_()` methods instead.

That's it? Well, pretty much… but there is always a **catch**, and this time it has to do with the **update** of the **parameters**…

Updating Parameters

> *"One does not simply update parameters…"*
>
> Boromir

Unfortunately, our *Numpy*'s code for updating parameters is not enough… why not?! Let's try it out, simply copying and pasting it (this is the *first attempt*), changing it slightly (*second attempt*), and then asking PyTorch to **back off** (yes, it is PyTorch's fault!).

Notebook Cell 1.6 - *Updating Parameters*

```
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train_tensor

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train_tensor)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w"
    # parameters. No more manual computation of gradients!
    # b_grad = 2 * error.mean()
    # w_grad = 2 * (x_tensor * error).mean()
    # We just tell PyTorch to work its way BACKWARDS
    # from the specified loss!
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate. But not so fast...
    # FIRST ATTEMPT - just using the same code as before
    # AttributeError: 'NoneType' object has no attribute 'zero_'
    # b = b - lr * b.grad                               ①
    # w = w - lr * w.grad                               ①
    # print(b)                                          ①

    # SECOND ATTEMPT - using in-place Python assignment
    # RuntimeError: a leaf Variable that requires grad
    # has been used in an in-place operation.
    # b -= lr * b.grad                                  ②
    # w -= lr * w.grad                                  ②

    # THIRD ATTEMPT - NO_GRAD for the win!
    # We need to use NO_GRAD to keep the update out of
    # the gradient computation. Why is that? It boils
    # down to the DYNAMIC GRAPH that PyTorch uses...
    with torch.no_grad():                               ③
        b -= lr * b.grad                                ③
        w -= lr * w.grad                                ③

    # PyTorch is "clingy" to its computed gradients, we
    # need to tell it to let it go...
    b.grad.zero_()                                      ④
    w.grad.zero_()                                      ④

print(b, w)
```

① First Attempt: leads to an `AttributeError`

② Second Attempt: leads to an `RuntimeError`

③ Third Attempt: `no_grad` solves the problem!

④ `zero_` prevents gradient accumulation

In the **first attempt**, if we use the same update structure as in our *Numpy* code, we'll get the weird **error** below… but we can get a *hint* of what's going on by looking at the tensor itself — once again, we **"lost"** the **gradient** while reassigning the update results to our parameters. Thus, the `grad` attribute turns out to be `None`, and it raises the error…

Output - First Attempt - Keeping the same code

```
tensor([0.7518], device='cuda:0', grad_fn=<SubBackward0>)
AttributeError: 'NoneType' object has no attribute 'zero_'
```

We then change it slightly, using a familiar **in-place Python assignment** in our **second attempt**. And, once again, PyTorch complains about it and raises an **error**.

Output - Second Attempt - In-place assignment

```
RuntimeError: a leaf Variable that requires grad has been used in
an in-place operation.
```

Why?! It turns out to be a case of **"too much of a good thing"**. The culprit is PyTorch's ability to build a **dynamic computation graph** from every **Python operation** that involves any **gradient-computing tensor** or **its dependencies**.

We'll go deeper into the inner workings of the dynamic computation graph in the next section.

Time for our **third attempt**…

no_grad

So, how do we tell PyTorch to **"back off"** and let us **update our parameters** without messing up with its *fancy dynamic computation graph*? That's what `torch.no_grad()` is good for. It allows us to **perform regular Python operations on tensors**, **without affecting PyTorch's computation graph**.

Finally, we managed to successfully run our model and get the **resulting parameters**. Surely enough, they **match** the ones we got in our *Numpy*-only implementation.

Output - Third Attempt - NO_GRAD for the win!

```
# THIRD ATTEMPT - NO_GRAD for the win!
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Remember:

> *"One does not simply update parameters... **without** `no_grad`"*
>
> Boromir

It was true for going into *Mordor*, and it is also true for updating parameters.

It turns out, `no_grad` has another use case other than allowing us to update parameters - we'll get back to it in Chapter 2 when dealing with a model's evaluation.

Dynamic Computation Graph

> *"Unfortunately, no one can be told what the dynamic computation graph is. You have to see it for yourself."*
>
> Morpheus

How great was "*The Matrix*"? Right? Right? But, jokes aside, I want **you** to **see the graph for yourself** too!

The PyTorchViz package and its `make_dot(variable)` method allow us to easily visualize a graph associated with a given Python variable involved in the gradient computation.

If you chose a "Local Installation" in the Setup Guide and skipped or had issues with Step 5 ("Install GraphViz software and TorchViz package"), you will get an **error** when trying to visualize the graphs using `make_dot`.

So, let's stick with the **bare minimum**: two (*gradient computing*) **tensors** for our parameters, predictions, errors, and loss - these are Steps 0, 1, and 2.

```
# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor
# Step 2 - Computes the loss
error = (yhat - y_train_tensor)
loss = (error ** 2).mean()

# We can try plotting the graph for any python variable:
# yhat, error, loss...
make_dot(yhat)
```

Running the code above will show us the **graph** below:

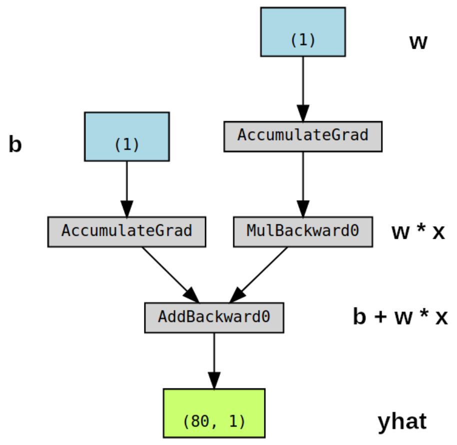


Figure 1.5 - Computation graph generated for `yhat` *- Obs.: the corresponding variable names were inserted manually*

Let's take a closer look at its components:

- **blue boxes** (`(1)`s): these boxes correspond to the **tensors** we use as **parameters**, the ones we're asking PyTorch to **compute gradients** for
- **gray boxes** (`MulBackward0` and `AddBackward0`): a **Python operation** that involves a **gradient-computing tensor** or **its dependencies**
- **green box** (`(80, 1)`): the tensor used as the **starting point for the computation** of gradients (assuming the `backward()` method is called from the **variable used** to **visualize** the graph) — they are computed from the **bottom-up** in a graph

Now, take a closer look at the **gray box** at the bottom of the graph: **two arrows** are pointing to it since it is **adding** up **two variables**, `b`, and `w*x`. Seems obvious, right?

Then, look at the **gray box** (`MulBackward0`) of the same graph: it is performing a **multiplication**, namely, `w*x`. But there is only one arrow pointing to it! The arrow comes from the **blue box** that corresponds to our **parameter *w***.

"***Why** don't we have a box for our **data** (`x`)?*"

The answer is: we **do not compute gradients** for it!

So, even though there are *more* tensors involved in the operations performed by the computation graph, it **only** shows **gradient-computing tensors** and **its dependencies**.

What would happen to the computation graph if we set `requires_grad` to `False` for our **parameter *b***?

```
b_nograd = torch.randn(1, requires_grad=False, \
                       dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)

yhat = b_nograd + w * x_train_tensor

make_dot(yhat)
```

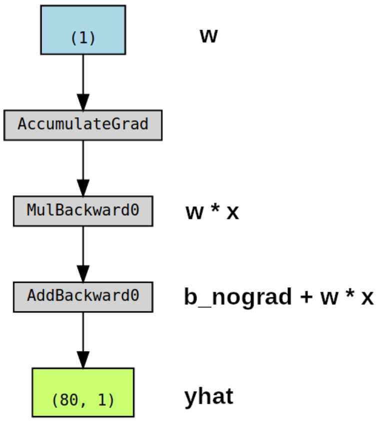


Figure 1.6 - now parameter "b" does NOT have its gradient computed anymore, but it is STILL used in computation

Unsurprisingly, the **blue box** corresponding to the **parameter *b*** is no more!

Simple enough: **no gradients, no graph**!

The **best** thing about the *dynamic computation graph* is the fact that you can make it

as complex as you want it. You can even use *control flow statements* (e.g., if statements) to **control the flow of the gradients**.

Figure 1.7 below shows an example of this. And yes, I do know that the computation itself is *complete* **nonsense**…

```
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)

yhat = b + w * x_train_tensor
error = yhat - y_train_tensor
loss = (error ** 2).mean()

# this makes no sense!!
if loss > 0:
    yhat2 = w * x_train_tensor
    error2 = yhat2 - y_train_tensor

# neither does this :-)
loss += error2.mean()

make_dot(loss)
```

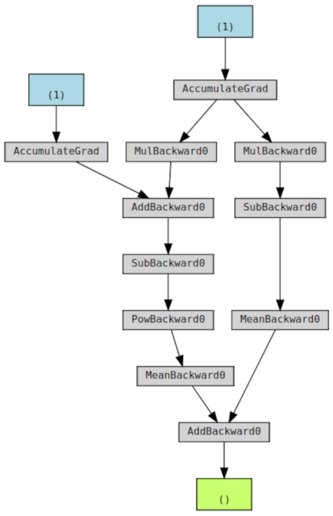


Figure 1.7 - Complex (and nonsensical!) computation graph just to make a point

Even though the computation is nonsensical, you can clearly see the **effect** of adding a **control flow statement** like `if loss > 0`: it **branches** the computation graph in two parts. The **right branch** performs the computation **inside the if statement**, which gets added to the result of the left branch in the end. Cool, right?

Even though we are not building more complex models like that in this book, this small example illustrates very well PyTorch's capabilities and how easily they can be implemented in code.

Optimizer

So far, we've been **manually** updating the parameters using the computed gradients. That's probably fine for **two parameters**… but what if we had a **whole lot of them**?! We need to use one of PyTorch's optimizers, like SGD, RMSprop, or Adam.

There are **many** optimizers: **SGD** is the most basic of them, and **Adam** is one of the most popular.

Different optimizers use different mechanics for **updating the parameters**, but they all achieve the same goal through, literally, **different paths**.

To see what I mean by this, check out this animated GIF[42] developed by Alec Radford[43], available at Stanford's CS231n: Convolutional Neural Networks for Visual Recognition[44] course. The animation shows a **loss surface**, just like the ones we computed in Chapter 0, and the **paths** traversed by some optimizers to achieve the **minimum** (represented by a star).

Remember, the **choice of mini-batch size** influenced the **path of gradient descent**, and so does the **choice of an optimizer**.

step / zero_grad

An optimizer takes the **parameters** we want to update, the **learning rate** we want to use (and possibly many other hyper-parameters as well!), and **performs the updates** through its `step()` method.

```
# Defines an SGD optimizer to update the parameters
optimizer = optim.SGD([b, w], lr=lr)
```

Besides, we also don't need to *zero the gradients* one by one anymore. We just invoke the optimizer's `zero_grad()` method, and that's it!

In the code below, we create a *Stochastic Gradient Descent* (SGD) optimizer to update our parameters ***b*** and ***w***.

Don't be fooled by the **optimizer**'s name: if we use **all training data** at once for the update — as we are actually doing in the code — the optimizer is performing a **batch** gradient descent, despite its name.

Notebook Cell 1.7 - *PyTorch's optimizer in action — no more manual update of parameters!*

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([b, w], lr=lr)                ①

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train_tensor

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train_tensor)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate. No more manual update!
    # with torch.no_grad():
    #     b -= lr * b.grad
    #     w -= lr * w.grad
    optimizer.step()                                    ②

    # No more telling Pytorch to let gradients go!
    # b.grad.zero_()
    # w.grad.zero_()
    optimizer.zero_grad()                               ③

print(b, w)
```

① Defining an optimizer

② New "Step 4 - Updating Parameters" using the optimizer

③ New "gradient zeroing" using the optimizer

Let's inspect our two parameters just to make sure everything is still working fine:

Output

```
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Cool! We've *optimized* the **optimization** process :-) What's left?

Loss

We now tackle the **loss computation**. As expected, PyTorch got us covered once again. There are many loss functions to choose from, depending on the task at hand. Since ours is a regression, we are using the **Mean Squared Error** (MSE) as loss, and thus we need PyTorch's `nn.MSELoss`):

```
# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
loss_fn
```

Output

```
MSELoss()
```

Notice that `nn.MSELoss` **is NOT the loss function itself**: we do not pass *predictions* and *labels* to it! Instead, as you can see, it **returns another function**, which we called `loss_fn`: *that* is the **actual loss function**. So, we can pass a prediction and a label to it and get the corresponding loss value:

```
# This is a random example to illustrate the loss function
predictions = torch.tensor(0.5, 1.0)
labels = torch.tensor(2.0, 1.3)
loss_fn(predictions, labels)
```

Output

```
tensor(1.1700)
```

Moreover, you can also specify a **reduction method** to be applied, that is, **how do you want to aggregate the errors for individual points** — you can average them (`reduction=`“mean”) or simply sum them up (`reduction=`“sum”). In our example, we use the typical `mean` reduction to compute **MSE**. If we had used `sum` as reduction, we would actually be computing **SSE** (Sum of Squared Errors).

Technically speaking, `nn.MSELoss` is a **higher-order function**.

If you're not familiar with the concept, I will explain it briefly in Chapter 2.

We then **use** the created loss function in the code below, at line 29, to compute the loss, given our **predictions** and our **labels**:

Notebook Cell 1.8 - *PyTorch's loss in action: no more manual loss computation!*

```
 1 # Sets learning rate - this is "eta" ~ the "n" like
 2 # Greek letter
 3 lr = 0.1
 4 
 5 # Step 0 - Initializes parameters "b" and "w" randomly
 6 torch.manual_seed(42)
 7 b = torch.randn(1, requires_grad=True, \
 8                 dtype=torch.float, device=device)
 9 w = torch.randn(1, requires_grad=True, \
10                 dtype=torch.float, device=device)
11 
12 # Defines a SGD optimizer to update the parameters
13 optimizer = optim.SGD([b, w], lr=lr)
14 
15 # Defines a MSE loss function
16 loss_fn = nn.MSELoss(reduction='mean')        ①
17 
18 # Defines number of epochs
19 n_epochs = 1000
20 
21 for epoch in range(n_epochs):
22     # Step 1 - Computes model's predicted output - forward pass
23     yhat = b + w * x_train_tensor
24 
25     # Step 2 - Computes the loss
26     # No more manual loss!
```

```
    # error = (yhat - y_train_tensor)
    # loss = (error ** 2).mean()
    loss = loss_fn(yhat, y_train_tensor)          ②

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()

print(b, w)
```

① Defining a loss function

② New "Step 2 - Computing Loss" using `loss_fn`

Output

```
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Let's take a look at the **loss value** at the end of training…

```
loss
```

Output

```
tensor(0.0080, device='cuda:0', grad_fn=<MeanBackward0>)
```

What if we wanted to have it as a *Numpy* array? I guess we could just use `numpy()` again, right? (and `cpu()` as well, since our *loss* is in the `cuda` device…)

```
loss.cpu().numpy()
```

Output

```
RuntimeError              Traceback (most recent call last)
<ipython-input-43-58c76a7bac74> in <module>
----> 1 loss.cpu().numpy()

RuntimeError: Can't call numpy() on Variable that requires
grad. Use var.detach().numpy() instead.
```

What happened here? Unlike our *data tensors*, the **loss tensor** is actually computing gradients - to use `numpy()`, we need to `detach()` the tensor from the computation graph first:

```
loss.detach().cpu().numpy()
```

Output

```
array(0.00804466, dtype=float32)
```

This seems like **a lot of work**; there must be an easier way! And there is one indeed: we can use `item()`, for **tensors with a single element** or `tolist()` otherwise (it still returns a scalar if there is only *one* element, though).

```
print(loss.item(), loss.tolist())
```

Output

```
0.008044655434787273 0.008044655434787273
```

At this point, there's only one piece of code left to change: the **predictions**. It is then time to introduce PyTorch's way of implementing a…

Model

In PyTorch, a **model** is represented by a regular **Python class** that inherits from the `Module` class.

> **IMPORTANT**: Are you comfortable with **object-oriented programming (OOP)** concepts like *classes*, *constructors*, *methods*, *instances*, and **attributes**?
>
> If you're *unsure* about any of these terms, I'd **strongly recommend** you to follow tutorials like Real Python's Objected-Oriented Programming (OOP) in Python 3[45] and Supercharge Your Classes With Python super()[46] before proceeding.
>
> Having a good understanding of **OOP is key** to benefit the most from PyTorch's capabilities.

So, assuming you're already comfortable with OOP, let's dive into developing a **model** in PyTorch.

The most fundamental methods a **model class** needs to implement are:

- `__init__(self)`: **it defines the parts that make up the model** — in our case, two *parameters*, ***b*** and ***w***.

> You are **not** limited to defining **parameters**, though… **models can contain other models as their attributes** as well, so you can easily nest them. We'll see an example of this shortly as well.
>
> Besides, **do not forget** to include **`super().__init__()`** to execute the `__init__()` method of the **parent class** (`nn.Module`) before your own.

- `forward(self, x)`: it performs the **actual computation**, that is, it **outputs a prediction**, given the input **x**.

It may seem weird but, whenever using your model to make predictions, you should **NOT call the `forward(x)`** method!

You should **call the whole model instead**, as in **`model(x)`**, to perform a forward pass and output predictions.

The reason is, the call to the whole model involves *extra steps*, namely, handling **forward** and **backward hooks**. If you don't use **hooks** (and we don't use any right now), both calls are equivalent.

Hooks are a very useful mechanism that allows retrieving intermediate values in deeper models. We'll get to them eventually.

Let's build a proper (yet simple) model for our regression task. It should look like this:

Notebook Cell 1.9 - *Building our "Manual" model, creating parameter by parameter!*

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "b" and "w" real parameters of the model,
        # we need to wrap them with nn.Parameter
        self.b = nn.Parameter(torch.randn(1,
                                          requires_grad=True,
                                          dtype=torch.float))
        self.w = nn.Parameter(torch.randn(1,
                                          requires_grad=True,
                                          dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.b + self.w * x
```

Parameters

In the `__init__` method, we define our **two parameters**, ***b*** and ***w***, using the `Parameter()` class, to tell PyTorch that these **tensors**, which are **attributes** of the `ManualLinearRegression` **class**, should be considered **parameters of the model** the class represents.

Why should we care about that? By doing so, we can use our model's `parameters()` method to retrieve **an iterator over all model's parameters**, including parameters of **nested models**. Then we can use it to feed our *optimizer* (instead of building a list of parameters ourselves!).

```
torch.manual_seed(42)
# Creates a "dummy" instance of our ManualLinearRegression model
dummy = ManualLinearRegression()
list(dummy.parameters())
```

Output

```
[Parameter containing:
 tensor([0.3367], requires_grad=True), Parameter containing:
 tensor([0.1288], requires_grad=True)]
```

state_dict

Moreover, we can get the **current values of all parameters** using our model's `state_dict()` method.

```
dummy.state_dict()
```

Output

```
OrderedDict([('b', tensor([0.3367])), ('w', tensor([0.1288]))])
```

The `state_dict()` of a given model is simply a Python dictionary that **maps each attribute/parameter to its corresponding tensor**. But only **learnable** parameters are included, as its purpose is to keep track of parameters that are going to be updated by the **optimizer**.

By the way, the **optimizer** itself has a `state_dict()` too, which contains its internal state, as well as other hyper-parameters. Let's take a quick look at it:

```
optimizer.state_dict()
```

Output

```
{'state': {},
 'param_groups': [{'lr': 0.1,
   'momentum': 0,
   'dampening': 0,
   'weight_decay': 0,
   'nesterov': False,
   'params': [140535747664704, 140535747688560]}]}
```

"What do we need this for?"

It turns out; *state dictionaries* can also be used for **checkpointing** a model, as we will see in Chapter 2.

Device

IMPORTANT: we need to **send our model to the same device where the data is**. If our data is made of GPU tensors, our model must “live” inside the GPU as well.

If we were to send our dummy model to a device, it would look like this:

```
torch.manual_seed(42)
# Creates a "dummy" instance of our ManualLinearRegression model
# and sends it to the device
dummy = ManualLinearRegression().to(device)
```

Forward Pass

The **forward pass** is the moment when the model **makes predictions**.

Remember: you should make predictions calling **`model(x)`**

DO NOT call `model.forward(x)`!

Otherwise, your model's *hooks* will not work (if you have them).

We can use all these handy methods to change our code, which should be looking like this:

Notebook Cell 1.10 - *PyTorch's model in action: no more manual prediction/forward step!*

```
# Sets learning rate - this is "eta" ~ the "n" like
# Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = ManualLinearRegression().to(device)        ①

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    model.train() # What is this?!?                ②

    # Step 1 - Computes model's predicted output - forward pass
    # No more manual prediction!
    yhat = model(x_train_tensor)                   ③
```

```
26
27     # Step 2 - Computes the loss
28     loss = loss_fn(yhat, y_train_tensor)
29
30     # Step 3 - Computes gradients for both "b" and "w" parameters
31     loss.backward()
32
33     # Step 4 - Updates parameters using gradients and
34     # the learning rate
35     optimizer.step()
36     optimizer.zero_grad()
37
38 # We can also inspect its parameters using its state_dict
39 print(model.state_dict())
```

① Instantiating a model

② What **IS** this?!?

③ New "Step 1 - Forward Pass" using a model

Now, the printed statements will look like this — final values for parameters ***b*** and ***w*** are still the same, so everything is OK :-)

Output

```
OrderedDict([('b', tensor([1.0235], device='cuda:0')),
('w', tensor([1.9690], device='cuda:0'))])
```

train

I hope you noticed one particular statement in the code (line 21), to which I assigned a comment **"What is this?!?"** — `model.train()`.

In PyTorch, models have a `train()` method which, somewhat disappointingly, **does NOT perform a training step**. Its only purpose is to **set the model to training mode**.

Why is this important? Some models may use mechanisms like `Dropout`, for instance, which have **distinct behaviors during training and evaluation** phases.

It is good practice to call `model.train()` in the training loop. It is also possible to set a model to evaluation mode, but this is a topic for the next chapter.

Nested Models

In our model, we *manually* created *two parameters* to perform a linear regression. What if, instead of defining individual parameters, we use PyTorch's `Linear` model?

We are implementing a *single feature linear regression*, *one input* and *one output*, so the corresponding linear model would look like this:

```
linear = nn.Linear(1, 1)
linear
```

Output

```
Linear(in_features=1, out_features=1, bias=True)
```

Do we still have our ***b*** and ***w*** parameters? Sure we do:

```
linear.state_dict()
```

Output

```
OrderedDict([('weight', tensor([[-0.2191]])),
('bias', tensor([0.2018]))])
```

So, our former parameter *b* is the **bias**, and our former parameter *w* is the **weight** (your values will be different since I haven't set up a random seed for this example).

Now, let's use PyTorch's `Linear` model as an **attribute** of our own, thus creating a **nested model**.

You are **not** limited to defining parameters, though… **models can contain other models as their attributes** as well, so you can easily nest them. We'll see an example of this shortly.

Even though this clearly is a contrived example, since we are pretty much *wrapping the underlying model without adding anything useful* (or, at all!) to it, it illustrates the concept well.

Notebook Cell 1.11 - *Building a model using PyTorch's Linear model*

```
class MyLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear model
        # with a single input and a single output
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        # Now it only takes a call
        self.linear(x)
```

In the `__init__` method, we created an **attribute** that contains our **nested** `Linear` **model**.

In the `forward()` method, we **call the nested model itself** to perform the forward pass (**notice, we are *not* *calling*** `self.linear.forward(x)`!).

Now, if we call the `parameters()` method of this model, **PyTorch will figure the parameters of its attributes recursively**.

```
torch.manual_seed(42)
dummy = MyLinearRegression().to(device)
list(dummy.parameters())
```

Output

```
[Parameter containing:
 tensor([[0.7645]], device='cuda:0', requires_grad=True),
 Parameter containing:
 tensor([0.8300], device='cuda:0', requires_grad=True)]
```

You can also add extra `Linear` attributes and, even if you don't use them at all in the forward pass, they will **still** be listed under `parameters()`.

If you prefer, you can also use `state_dict()` to get the parameter values together with their names:

```
dummy.state_dict()
```

Output

```
OrderedDict([('linear.weight',
              tensor([[0.7645]], device='cuda:0')),
             ('linear.bias',
              tensor([0.8300], device='cuda:0'))])
```

Notice that both *bias* and *weight* got a **prefix** with the **attribute name**: *linear*, from `self.linear` in the `__init__` method.

Sequential Models

Our model was simple enough… You may be thinking: "*why even bother to build a class for it?!*" Well, you have a point…

For **straightforward models**, that use **a series of built-in PyTorch models** (like `Linear`), where the output of one is sequentially fed as an input to the next, we can use a, er… `Sequential` model :-)

In our case, we would build a Sequential model with a single argument, that is, the `Linear` model we used to train our linear regression. The model would look like this:

Notebook Cell 1.12 - *Building a model using PyTorch's Sequential model*

```
torch.manual_seed(42)
# Alternatively, you can use a Sequential model
model = nn.Sequential(nn.Linear(1, 1)).to(device)

model.state_dict()
```

Output

```
OrderedDict([('0.weight', tensor([[0.7645]], device='cuda:0')),
             ('0.bias', tensor([0.8300], device='cuda:0'))])
```

Simple enough, right?

We've been talking about **models inside other models**… this may get confusing real quick, so let's follow convention and call any *internal* model a **layer**.

Layers

A `Linear` model can be seen as a **layer** in a neural network.

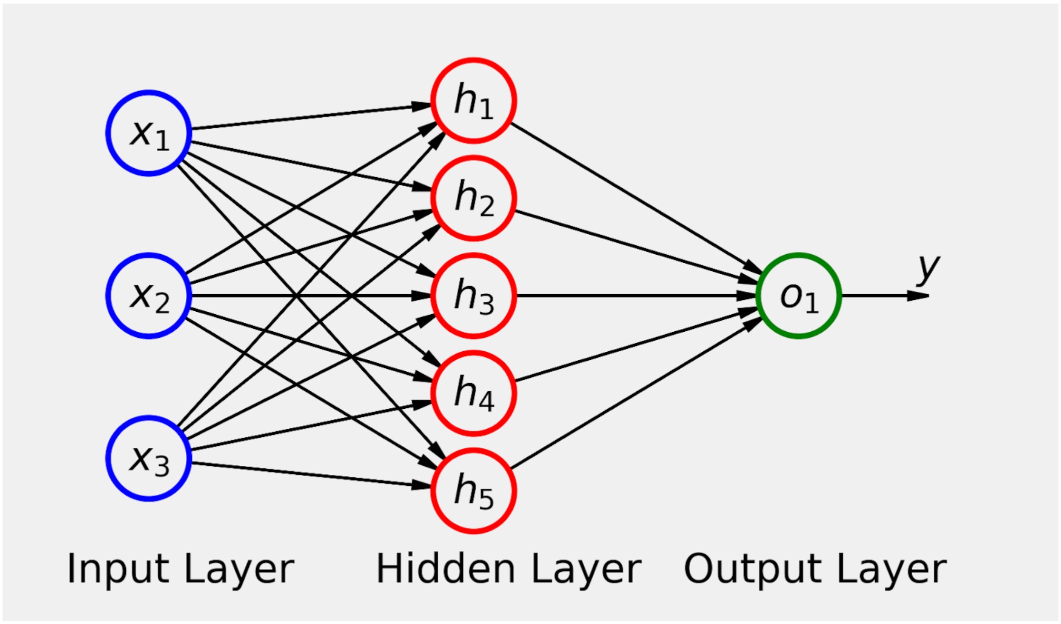


Figure 1.8 - Layers of a neural network

In the example above, the **hidden layer** would be `nn.Linear(3, 5)` (since it takes 3 inputs - from the input layer - and generates 5 outputs), and the **output layer** would be `nn.Linear(5, 1)` (since it takes 5 inputs - the outputs from the hidden layer - and generates a single output).

If we use `Sequential` to build it, it looks like this:

```
torch.manual_seed(42)
# Building the model from the figure above
model = nn.Sequential(nn.Linear(3, 5), nn.Linear(5, 1)).to(device)

model.state_dict()
```

Output

```
OrderedDict([
 ('0.weight',
   tensor([[ 0.4414,  0.4792, -0.1353],
           [ 0.5304, -0.1265,  0.1165],
           [-0.2811,  0.3391,  0.5090],
           [-0.4236,  0.5018,  0.1081],
           [ 0.4266,  0.0782,  0.2784]],
           device='cuda:0')),
 ('0.bias',
   tensor([-0.0815,  0.4451,  0.0853, -0.2695,  0.1472],
          device='cuda:0')),
 ('1.weight',
   tensor([[-0.2060, -0.0524, -0.1816,  0.2967, -0.3530]],
          device='cuda:0')),
 ('1.bias',
   tensor([-0.2062], device='cuda:0'))])
```

Since this sequential model **does not have attribute names**, `state_dict()` uses **numeric prefixes**.

You can also use a model's `add_module()` method to be able to **name** the layers:

```
torch.manual_seed(42)
# Building the model from the figure above
model = nn.Sequential()
model.add_module('layer1', nn.Linear(3, 5))
model.add_module('layer2', nn.Linear(5, 1))
model.to(device)
```

Output

```
Sequential(
  (layer1): Linear(in_features=3, out_features=5, bias=True)
  (layer2): Linear(in_features=5, out_features=1, bias=True)
)
```

There are **MANY** different layers that can be used in PyTorch:

- Convolution Layers
- Pooling Layers
- Padding Layers
- Non-linear Activations
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers (embeddings)
- Vision Layers
- DataParallel Layers (multi-GPU)
- Flatten Layer

So far, we have just used a `Linear` layer. In the chapters ahead, we'll use many others, like Convolution, Pooling, Padding, Flatten, Dropout, and Non-linear Activations.

Putting It All Together

We've covered a lot of ground so far, from coding a **linear regression in Numpy**

using gradient descent to transforming it into a **PyTorch model**, step-by-step.

It is time to put it all together and organize our code so far into **three** fundamental parts, namely:

- **data preparation** (*not* data generation!)
- **model configuration**
- **model training**

Let's tackle these three parts, in order:

Data Preparation

There isn't much data preparation at this point, to be honest. After generating our data points in **Notebook Cell 1.1**, the only preparation step performed so far was transforming *Numpy* arrays into PyTorch tensors, as in **Notebook Cell 1.3**, which is reproduced below:

Define - Data Preparation V0

```
%%writefile data_preparation/v0.py

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

Run - Data Preparation V0"

```
%run -i data_preparation/v0.py
```

This part will get much more interesting in the next chapter when we get to use

Dataset and **DataLoader** classes :-)

Jupyter's *magic* commands

You probably noticed the somewhat unusual `%%writefile` and `%run` commands above. These are built-in magic commands[47]. A magic is a kind of a shortcut that extends a notebook's capabilities.

We are using the following two magics to better organize our code:

- `%%writefile`[48]: as its name says, it writes the contents of the cell to a file, but it **does not run it**, so we need to use yet another magic…
- `%run`[49]: it runs the named file inside the notebook as a program - but **independently from the rest of the notebook**, so we need to use the `-i` option to make all variables available, both from the notebook and the file (technically speaking, the file is executed in IPython's namespace).

In a nutshell, a cell containing one of our three fundamental parts will be written to a versioned file inside the folder corresponding to that part.

In the example above, we write the cell to the **`data_preparation`** folder, name it **`v0.py`** and then execute it using the `%run -i` magic.

"*What's the purpose of saving cells to these files?*"

We know we have to run the **full sequence** to **train a model**: data preparation, model configuration, and model training. In Chapter 2, we'll *gradually improve* each one of these parts, *versioning* them inside each corresponding folder. So, **saving them to files** allows us to **run** a full sequence using **different versions without having to duplicate code**.

Let's say we start improving **model configuration** first (and we will do exactly that in Chapter 2), but the other two parts are still the same… how do we run the full sequence? We use **magic**, just like that:

```
%run -i data_preparation/v0.py
%run -i model_configuration/v1.py
%run -i model_training/v0.py
```

Since we're using the `-i` option, it works exactly as if we had copied the code from the files into a cell and executed it.

Model Configuration

We have seen plenty of this part: from defining parameters ***b*** and ***w*** manually, then wrapping them up using the `Module` class, to using **layers** in a `Sequential` model. We have also defined a **loss function** and an **optimizer** for our particular **linear regression** model.

For the purpose of organizing our code, we'll include the following elements in the **model configuration** part:

- a **model**
- a **loss function** (which needs to be chosen according to your model)
- an **optimizer** (although some people may disagree with this choice, it makes it easier for further organizing the code…)

Most of the corresponding code can be found in **Notebook Cell 1.10**, lines 1-15, but we'll replace the *ManualLinearRegression* model with the `Sequential` model from **Notebook Cell 1.12**:

Define - Model Configuration V0

```
%%writefile model_configuration/v0.py

# This is redundant now, but it won't be when we introduce
# Datasets...
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

Run - Model Configuration V0

```
%run -i model_configuration/v0.py
```

Model Training

This is the last part, where the *actual training* takes place. It loops over the **gradient descent steps** we've seen at the beginning of this chapter:

- Step 1: compute **model's predictions**
- Step 2: compute the **loss**
- Step 3: compute the **gradients**

- Step 4: update the **parameters**

This sequence is repeated over and over until the **number of epochs** is reached. The corresponding code for this part also comes from **Notebook Cell 1.10**, lines 17-36.

"*What happened to the random initialization step?*"

Since we are not manually creating parameters anymore, the initialization is handled **inside each layer** during model creation.

Define - Model Training V0

```
%%writefile model_training/v0.py

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Sets model to TRAIN mode
    model.train()

    # Step 1 - Computes model's predicted output - forward pass
    yhat = model(x_train_tensor)

    # Step 2 - Computes the loss
    loss = loss_fn(yhat, y_train_tensor)

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
```

Run - Model Training V0

```
%run -i model_training/v0.py
```

One last check to make sure we have everything right:

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9690]], device='cuda:0')),
('0.bias', tensor([1.0235], device='cuda:0'))])
```

Now, take a close, hard look at the code **inside the training loop**.

Ready? I have a question for you then…

"*Would this code **change** if we were using a **different optimizer**, or **loss**, or even **model**?*"

Before I give you the answer… let me address something else that may be in your mind: "*What is the point of all this?*"

Well, in the next chapter we'll get fancier using more of PyTorch's classes (like **Dataset** and **DataLoader**) to further refine our **data preparation step** and we'll also try to **reduce boilerplate code** to a minimum - so, splitting our code into three logical parts will allow us to better handle these improvements.

And here is the **answer**: **NO**, the code inside the loop **would not change**.

I guess you figured out which **boilerplate** I was referring to, right?

Recap

First of all, **congratulations** are in order: you have successfully implemented a **fully-functioning model** and **training loop** in PyTorch!

We have covered a lot of ground in this first chapter:

- implementing a linear regression in *Numpy* using **gradient descent**
- creating **tensors** in PyTorch, sending them to a **device** and making **parameters** out of them
- understanding PyTorch's main feature, **autograd**, to perform automatic differentiation using its associated properties and methods, like `backward`, `grad`, `zero_`, and `no_grad`
- visualizing the **Dynamic Computation Graph** associated with a sequence of operations
- creating an **optimizer** to simultaneously update multiple parameters, using its `step` and `zero_grad` methods
- creating a **loss function** using a PyTorch's higher-order function (more on that topic in the next chapter)
- understanding PyTorch's `Module` class and creating your own **models**, implementing `__init__` and `forward` methods and making use of its built-in `parameters` and `state_dict` methods
- transforming the original *Numpy* implementation into a **PyTorch** one using the elements above
- realizing the importance of including `model.train()` inside the **training loop** (never forget that!)
- implementing **nested** and **sequential** models using PyTorch's **layers**
- putting it all together into neatly organized code divided into **three distinct parts**: data preparation, model configuration, and model training

You are now ready for the next chapter: we'll see **more** of PyTorch's capabilities, and we'll **further develop our training loop**, so it can be used for different problems and models. You'll be building your *own, small draft of a library* for training deep learning models.

[36] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter01.ipynb

[37] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter01.ipynb

[38] https://en.wikipedia.org/wiki/Gaussian_noise

[39] https://bit.ly/2XZXjnk

[40] https://bit.ly/3fjCSHR

[41] https://bit.ly/2Y0lhPn

[42] https://bit.ly/2UDXDWM

[43] https://twitter.com/alecrad

[44] http://cs231n.stanford.edu/

[45] https://realpython.com/python3-object-oriented-programming/

[46] https://realpython.com/python-super/

[47] https://ipython.readthedocs.io/en/stable/interactive/magics.html

[48] https://bit.ly/30GH0vO

[49] https://bit.ly/3g1eQCm

Chapter 2
Rethinking the Training Loop

Spoilers

In this chapter, we will:

- build a **function** to perform **training steps**
- implement our **own dataset class**
- use **data loaders** to **generate mini-batches**
- build a **function** to perform **mini-batch gradient descent**
- **evaluate** our model
- integrate **TensorBoard** to monitor model training
- **save/checkpoint** our model to disk
- **load** our model from disk to **resume training** or to **deploy**

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 2[50] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[51].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 2's

notebook. If not, just click on `Chapter02.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```python
import numpy as np
from sklearn.linear_model import LinearRegression

import torch
import torch.optim as optim
import torch.nn as nn
from torch.utils.data import Dataset, TensorDataset, DataLoader
from torch.utils.data.dataset import random_split
from torch.utils.tensorboard import SummaryWriter

import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('fivethirtyeight')
```

Rethinking the Training Loop

We finished the previous chapter with an important question:

> ? *"Would the code inside the training loop **change** if we were using a **different optimizer**, or **loss**, or even **model**?"*

The answer: **NO**.

But we have not actually elaborated on it in the previous chapter, so let's do it now.

The model training involves looping over **the four gradient descent steps** (or **one training step**, for that matter) and those are always the same, regardless of which

model, **loss** or **optimizer** we use (there may be exceptions to this, but it holds true for the scope of this book).

Let's take a look at the code once again:

Run - Data Generation & Preparation, Model Configuration

```
%run -i data_generation/simple_linear_regression.py
%run -i data_preparation/v0.py
%run -i model_configuration/v0.py
```

Run - Model Training V0

```
# %load model_training/v0.py

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Sets model to TRAIN mode
    model.train()

    # Step 1 - Computes model's predicted output - forward pass
    # No more manual prediction!
    yhat = model(x_train_tensor)

    # Step 2 - Computes the loss
    loss = loss_fn(yhat, y_train_tensor)

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
```

```python
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9690]], device='cuda:0')),
('0.bias', tensor([1.0235], device='cuda:0'))])
```

So, I guess we could say all these lines of code (7-23) **perform a training step**. For a given combination of **model**, **loss**, and **optimizer**, it takes the **features** and corresponding **labels** as arguments. Right?

How about **writing a function that takes a model, a loss, and an optimizer** and **returns another function that performs a training step**? The latter would then take the features and corresponding labels as arguments and returning the corresponding loss.

"*Wait; what?! A function that returns another function?*"

Sounds complicated, right? It is not as bad as it sounds, though… that's called a **higher-order function**, and it is very useful for reducing boilerplate.

If you're familiar with the concept of higher-order functions, feel free to skip the aside.

Higher-Order Functions

Although this is more of a *coding* topic, I believe it is necessary to have a good grasp on *how higher-order functions work* to fully benefit from Python's capabilities and make the best out of our code.

I will illustrate higher-order functions with an example so that you can gain a **working knowledge** of it, but I am not delving any deeper into the topic, as it is outside the scope of this book.

Let's say we'd like to build a **series of functions**, each performing an exponentiation to a given power. The code would look like this:

```python
def square(x):
    return x ** 2

def cube(x):
    return x ** 3

def fourth_power(x):
    return x ** 4

# and so on and so forth...
```

Well, clearly there is a **higher structure** to this:

- **every function takes a single argument *x***, which is the number we'd like to exponentiate
- **every function performs the same operation**, an exponentiation, but each function has a different exponent

One way of solving this is to **make the exponent an explicit argument**, just like the code below:

```python
def generic_exponentiation(x, exponent):
    return x ** exponent
```

That's perfectly fine, and it works quite well. But it also requires that you *specify the exponent every time* you call the function. **There must be another way**! Of course, there is, that's the purpose of this section!

We need to **build another (higher-order) function to build those functions (square, cube, etc.)** for us. The (higher-order) function is just a **function builder**. But how do we do that?

First, let's build the "*skeleton*" of the functions we are trying to generate… they all **take a single argument *x***, and **they all perform an exponentiation**, each using a different exponent.

Fine. It should look like this:

```python
def skeleton_exponentiation(x):
    return x ** exponent
```

If you try **calling this function** with any *x*, say, `skeleton_exponentiation(2)`, you'll get the following **error**:

```python
skeleton_exponentiation(2)
```

Output

```
NameError: name 'exponent' is not defined
```

This is expected: your "*skeleton*" function has **no idea what the variable `exponent` is**! And that's what the higher-order function is going to accomplish.

We **"wrap" our skeleton function with a higher-order function** (which will build the desired functions). Let's call it, rather unimaginatively, `exponentiation_builder`. What are **its arguments**, if any? Well, we're trying to **tell our skeleton function what its exponent should be**, so let's start with that :-)

```python
def exponentiation_builder(exponent):
    def skeleton_exponentiation(x):
        return x ** exponent

    return skeleton_exponentiation
```

Now I want you to take a look at the (outer) **return statement**. It is **not** returning a **value**; it is **returning the skeleton function** instead. This is a function builder after all: it should build (and return) functions.

What happens if we call this higher-order function with a given exponent, say, 2?

```python
returned_function = exponentiation_builder(2)

returned_function
```

Output

```
<function __main__.exponentiation_builder.<locals>.skeleton_
exponentiation(x)>
```

The **result** is, as expected, a **function**! What does this function do? It should square its argument… let's check it out:

```python
returned_function(5)
```

Output

```
25
```

And *voilà*! We have a function builder! We can use it to create as many exponentiation functions as we'd like:

```
square = exponentiation_builder(2)
cube = exponentiation_builder(3)
fourth_power = exponentiation_builder(4)

# and so on and so forth...
```

"*How does this apply to the training loop*" you may ask.

We'll be doing something similar to our training loop: the equivalent to the `exponent` argument of the higher-order function is the combination of **model**, **loss**, and **optimizer**. Every time we execute a training step for a different set of **features** and **labels**, which are the equivalent of the *x* argument in the skeleton function, we'll be using the same model, loss, and optimizer.

Training Step

The higher-order function that builds a training step function for us is taking, as already mentioned, the key elements of our training loop: **model**, **loss**, and **optimizer**. The actual training step function to be returned will have two arguments, namely, **features** and **labels**, and will return the corresponding **loss value**.

Apart from returning the loss value, the inner `perform_train_step()` function below is exactly the same as the code inside the loop in **Model Training V0**. The code should look like this:

Helper Function #1

```python
def make_train_step(model, loss_fn, optimizer):
    # Builds function that performs a step in the train loop
    def perform_train_step(x, y):
        # Sets model to TRAIN mode
        model.train()

        # Step 1 - Computes model's predictions - forward pass
        yhat = model(x)
        # Step 2 - Computes the loss
        loss = loss_fn(yhat, y)
        # Step 3 - Computes gradients for "b" and "w" parameters
        loss.backward()
        # Step 4 - Updates parameters using gradients and
        # the learning rate
        optimizer.step()
        optimizer.zero_grad()

        # Returns the loss
        return loss.item()

    # Returns the function that will be called inside the
    # train loop
    return perform_train_step
```

Then we need to update our **Model Configuration** code (adding line 21 in the snippet below) to call this higher-order function to build a `train_step` function. But we need to run a data preparation script first.

Run - Data Preparation V0

```
%run -i data_preparation/v0.py
```

Define - Model Configuration V1

```
%%writefile model_configuration/v1.py

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model, loss function
# and optimizer
train_step = make_train_step(model, loss_fn, optimizer) ①
```

① Creating a function that performs a training step

Run - Model Configuration V1

```
%run -i model_configuration/v1.py
```

Let's check our `train_step` function out!

```
train_step
```

Output

```
<function __main__.make_train_step.<locals>\
.perform_train_step(x, y)>
```

Looking good! Now we need to update our **Model Training** to replace the code inside the loop with a call to our newly created function.

Our code should look like this… see how **tiny** the training loop is now? Lots of **boilerplate** code are inside the `make_train_step` helper function now!

Define - Model Training V1

```
%%writefile model_training/v1.py

# Defines number of epochs
n_epochs = 1000

losses = []                                         ②

# For each epoch...
for epoch in range(n_epochs):
    # Performs one train step and returns the corresponding loss
    loss = train_step(x_train_tensor, y_train_tensor) ①
    losses.append(loss)                             ②
```

① Performing one training step

② Keeping track of the training loss

Run - Model Training V1

```
%run -i model_training/v1.py
```

Besides getting rid of boilerplate code, there is another change introduced in the code. We keep track of the **loss** value now. Every epoch, we append the last computed loss to a list.

"*Adding to a list? This does not seem very cutting-edge…*"

Indeed, it is not. But please bear with me, we'll replace it with something nicer soon enough :-)

After updating two out of three fundamental parts, our current state of development is:

- **Data Preparation V0**
- **Model Configuration V1**
- **Model Training V1**

How do we check if our changes did not introduce any bugs? We can inspect our model's `state_dict()`:

```
# Checks model's parameters
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9690]], device='cuda:0')),
('0.bias', tensor([1.0235], device='cuda:0'))])
```

Let's give our training loop a rest and focus on our **data** for a while… so far, we've simply used our *Numpy arrays* turned into **PyTorch tensors**. But we can do better; we can build a…

Dataset

In PyTorch, a **dataset** is represented by a regular **Python class** that inherits from the `Dataset` class. You can think of it as a **list of tuples**, each tuple corresponding to

one point (features, label).

The most fundamental methods it needs to implement are:

- `__init__(self)`: it takes **whatever arguments** needed to build a **list of tuples** — it may be the name of a CSV file that will be loaded and processed; it may be *two tensors*, one for features, another one for labels; or anything else, depending on the task at hand.

> There is **no need to load the whole dataset in the constructor method** (`__init__`). If your **dataset is big** (tens of thousands of image files, for instance), loading it at once would not be memory efficient. It is recommended to **load them on demand** (whenever `__get_item__` is called).

- `__get_item__(self, index)`: it allows the dataset to be **indexed** so that it can work **like a list** (`dataset[i]`) — it must **return a tuple (features, label)** corresponding to the requested data point. We can either return the **corresponding slices** of our **pre-loaded** dataset or, as mentioned above, **load them on demand** (like in this example[52]).
- `__len__(self)`: it should simply return the **size** of the whole dataset so, whenever it is sampled, its indexing is limited to the actual size.

Let's build a simple custom dataset that takes two tensors as arguments: one for the features, one for the labels. For any given index, our dataset class will return the corresponding slice of each one of those tensors. It should look like this:

Notebook Cell 2.1 - Creating a custom dataset

```python
class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

# Wait, is this a CPU tensor now? Why? Where is .to(device)?
x_train_tensor = torch.as_tensor(x_train).float()
y_train_tensor = torch.as_tensor(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

Output

```
(tensor([0.7713]), tensor([2.4745]))
```

Did you notice we built our **training tensors** out of *Numpy* arrays, but we **did not send them to a device**? So, they are **CPU** tensors now! **Why**?

We **don't want our whole training data to be loaded into GPU tensors**, as we have been doing in our example so far, because **it takes up space** in our precious **graphics card's RAM**.

TensorDataset

Once again, you may be thinking, "*why go through all this trouble to wrap a couple of* *tensors in a class?*". And, once again, you do have a point… if a dataset is nothing more than a **couple of tensors**, we can use PyTorch's `TensorDataset` class, which will do pretty much the same as our custom dataset above.

So, right now, the full-fledged **custom dataset class** may seem like a stretch, but we will use this structure repeatedly in later chapters. For now, let's enjoy the simplicity of the `TensorDataset` class :-)

Notebook Cell 2.2 - Creating a dataset from tensors

```
train_data = TensorDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

Output

```
(tensor([0.7713]), tensor([2.4745]))
```

OK, fine, but then again, **why** are we building a dataset anyway? We're doing it because we want to use a…

DataLoader

Until now, we have used the **whole training data** at every training step. It has been **batch gradient descent** all along. This is fine for our *ridiculously small dataset*, sure, but if we want to go serious about all this, we **must** use **mini-batch** gradient descent. Thus, we need mini-batches. Thus, we need to **slice** our dataset accordingly. Do you want to do it **manually**?! Me neither!

So we use PyTorch's `DataLoader` class for this job. We tell it which **dataset** to use (the one we have just built in the previous section), the desired **mini-batch size**, and if we'd like to **shuffle** it or not. That's it!

IMPORTANT: in the absolute majority of cases, you **should** set `shuffle=True` for your **training set** to improve the performance of gradient descent. There are a few exceptions, though, like time series problems, where shuffling actually leads to data leakage.

So, always ask yourself: "***do I have a reason NOT to shuffle the data?***"

"*What about the validation and test sets?*" There is **no need to shuffle** them since we are **not computing gradients** with them.

There is more to a `DataLoader` than meets the eye… it is also possible to use it together with a **sampler** to fetch mini-batches that compensate for **imbalanced classes**, for instance. Too much to handle right now, but we will eventually get there.

Our **loader** will behave like an **iterator**, so we can **loop over it** and **fetch a different mini-batch** every time.

"*How do I choose my **mini-batch size**?*"

It is typical to use **powers of two** for mini-batch sizes, like 16, 32, 64 or 128, and **32** seems to be the choice of most people, Yann LeCun[53] included.

Some more complex models may use even larger sizes, although sizes are usually constrained by hardware limitations (i.e., how many data points actually fit into memory).

In our example, we have only 80 training points, so I chose a mini-batch size of 16 to conveniently split the training set into five mini-batches.

Notebook Cell 2.3 - Building a data loader for our training data

```
train_loader = DataLoader(
    dataset=train_data,
    batch_size=16,
    shuffle=True,
)
```

To retrieve a mini-batch, one can simply run the command below — it will return a list containing two tensors, one for the features, another one for the labels:

```
next(iter(train_loader))
```

Output

```
[tensor([[0.1196],
         [0.1395],
         ...
         [0.8155],
         [0.5979]]), tensor([[1.3214],
         [1.3051],
         ...
         [2.6606],
         [2.0407]])]
```

"*Why not use a **list** instead?*"

If you call `list(train_loader)`, you'll get; as a result, a list of 5 elements, that is, all five mini-batches. Then you could take the first element of that list to obtain a single mini-batch as in the example above. It would **defeat the purpose** of using the **iterable** provided by the **DataLoader**, that is, to **iterate** over the elements (mini-batches, in that case) **one at a time**.

To learn more about it, check RealPython's material on iterables[54] and iterators[55].

How does this change our code so far? Let's check it out!

First, we need to add both **Dataset** and **DataLoader** elements into our **data preparation** part of the code. Also, notice that we **do not** send our tensors to the device just yet (just like we did in **Notebook Cell 2.1**). It should look like this:

Define - Data Preparation V1

```
%%writefile data_preparation/v1.py

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors
x_train_tensor = torch.as_tensor(x_train).float()
y_train_tensor = torch.as_tensor(y_train).float()

# Builds Dataset
train_data = TensorDataset(x_train_tensor, y_train_tensor) ①

# Builds DataLoader
train_loader = DataLoader(                                 ②
    dataset=train_data,
    batch_size=16,
    shuffle=True,
)
```

① Building a dataset of tensors

② Building a data loader that yields mini-batches of size 16

Run - Data Preparation V1

```
%run -i data_preparation/v1.py
```

Next, we need to incorporate the **mini-batch** gradient descent logic into our **model training** part of the code. But we need to run the model configuration first.

Run - Model Configuration V1

```
%run -i model_configuration/v1.py
```

Define - Model Training V2

```
%%writefile model_training/v2.py

# Defines number of epochs
n_epochs = 1000

losses = []

# For each epoch...
for epoch in range(n_epochs):
    # inner loop
    mini_batch_losses = []                                ④
    for x_batch, y_batch in train_loader:                 ①
        # the dataset "lives" in the CPU, so do our mini-batches
        # therefore, we need to send those mini-batches to the
        # device where the model "lives"
        x_batch = x_batch.to(device)                      ②
        y_batch = y_batch.to(device)                      ②

        # Performs one train step and returns the
        # corresponding loss for this mini-batch
        mini_batch_loss = train_step(x_batch, y_batch)   ③
        mini_batch_losses.append(mini_batch_loss)         ④

    # Computes average loss over all mini-batches
    # That's the epoch loss
    loss = np.mean(mini_batch_losses)                     ⑤

    losses.append(loss)
```

① Mini-batch inner loop

② Sending one mini-batch to the device

③ Performing a training step

④ Keeping track of the loss inside each mini-batch

⑤ Averaging losses of mini-batches to get epoch's loss

Run - Model Training V2

```
%run -i model_training/v2.py
```

"Wow! What happened here?!"

It seems like a lot changed… Let's take a closer look, step by step:

- we added an **inner loop** to handle the **mini-batches** produced by the `DataLoader` (line 12)
- we sent **only one mini-batch to the device**, as opposed to sending the whole training set (lines 16 and 17)

For bigger datasets, **loading data on demand** (into a **CPU** tensor) inside **Dataset's** `__get_item__` method and then **sending all data points** that belong to the same **mini-batch at once to your GPU** (device) is the way to go to make the **best use of your graphics card's RAM**.

Moreover, if you have **many GPUs** to train your model on, it is best to keep your dataset "device-agnostic" and assign the batches to different GPUs during training.

- we performed a **`train_step`** on a mini-batch (line 21) and appended the corresponding loss to a list (line 22)
- after going through all mini-batches, that is, at the end of an **epoch**, we calculated the total loss for the epoch, which is the average loss over all mini-

batches, appending the result to a list (lines 26 and 28)

After another two updates, our current state of development is:

- **Data Preparation V1**
- **Model Configuration V1**
- **Model Training V2**

Not so bad, right? So, it is time to check if our code still works well:

```
# Checks model's parameters
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9684]], device='cuda:0')),
('0.bias', tensor([1.0235], device='cuda:0'))])
```

Did you get *slightly different* values? Try running the whole pipeline again:

Full pipeline

```
%run -i data_preparation/v1.py
%run -i model_configuration/v1.py
%run -i model_training/v2.py
```

Since the `DataLoader` draws random samples, executing other code between the last two steps of the pipeline may interfere with the reproducibility of the results.

Anyway, as long as your results are less than 0.01 far from mine for both weight and bias, your code is working fine :-)

Did you notice it is taking **longer** to train now? Can you guess **why**?

ANSWER: the training time is **longer** now because the inner loop is executed **five times** for each epoch (in our example, since we are using a mini-batch of size 16 and we have 80 training data points in total, we execute the inner loop 80 / 16 = 5 times). So, in total, we are calling the `train_step` a total of **5,000 times** now! No wonder it's taking longer!

Mini-Batch Inner Loop

From now on, it is very unlikely that you'll **ever** use **(full) batch gradient descent** again, both in this book or in real life :-) So, it makes sense to, once again, organize a piece of code that's going to be used repeatedly into its own function: the **mini-batch inner loop**!

The inner loop depends on **three elements**:

- the **device** where data is being sent to
- a **data loader** to draw mini-batches from
- a **step function**, returning the corresponding loss

Taking these elements as inputs and using them to perform the inner loop, we'll end up with a function like this:

Helper Function #2

```
def mini_batch(device, data_loader, step):
    mini_batch_losses = []
    for x_batch, y_batch in data_loader:
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        mini_batch_loss = step(x_batch, y_batch)
        mini_batch_losses.append(mini_batch_loss)

    loss = np.mean(mini_batch_losses)
    return loss
```

In the last section, we realized that we were executing **five times more updates** (the `train_step` function) per epoch due to the *mini-batch inner loop*. Before, 1,000 epochs meant 1,000 updates. Now, we only need **200 epochs** to perform the same 1,000 updates.

How does our **training loop** look like now? It's **very** lean!

Run - Data Preparation V1, Model Configuration V1

```
%run -i data_preparation/v1.py
%run -i model_configuration/v1.py
```

Define - Model Training V3

```
%%writefile model_training/v3.py

# Defines number of epochs
n_epochs = 200

losses = []

for epoch in range(n_epochs):
    # inner loop
    loss = mini_batch(device, train_loader, train_step) ①
    losses.append(loss)
```

① Performing mini-batch gradient descent

Run - Model Training V3

```
%run -i model_training/v3.py
```

After updating the model training part, our current state of development is:

- **Data Preparation V1**
- **Model Configuration V1**
- **Model Training V3**

Let's inspect the model's state:

```
# Checks model's parameters
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9687]], device='cuda:0')),
('0.bias', tensor([1.0236], device='cuda:0'))])
```

So far, we've focused on the **training data** only. We built a *dataset* and a *data loader* for it. We could do the same for the **validation** data, using the **split** we performed at the beginning of this book… or we could use `random_split` instead.

Random Split

PyTorch's `random_split()` method is an easy and familiar way of performing a **training-validation split**.

So far, we've been using `x_train_tensor` and `y_train_tensor`, built out of the original split in *Numpy*, to build the **training dataset**. Now, we're going to be using the **full data** from *Numpy* (`x` and `y`), to build a PyTorch `Dataset` **first** and only then **split** the data using `random_split()`.

Then, for each subset of data, we're building a corresponding `DataLoader`, so our code looks like this:

Define - Data Preparation V2

```
%%writefile data_preparation/v2.py

torch.manual_seed(13)

# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(x).float()                            ①
y_tensor = torch.as_tensor(y).float()                            ①

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train
train_data, val_data = random_split(dataset, [n_train, n_val]) ②

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_data,
    batch_size=16,
    shuffle=True,
)
val_loader = DataLoader(dataset=val_data, batch_size=16)         ③
```

① Making tensors out of the full dataset (before split)

② Performing train-validation split in PyTorch

③ Creating a data loader for the validation set

Run - Data Preparation V2

```
%run -i data_preparation/v2.py
```

Now that we have a **data loader** for our **validation set**, it makes sense to use it for the…

Evaluation

How can we **evaluate** the model? We can compute the **validation** loss, that is, how wrong are the model's predictions for **unseen data**.

First, we need to use the **model** to compute **predictions** and then use the **loss function** to compute the loss, given our predictions and the true labels. Sounds familiar? These are pretty much the **first two steps** of the **training step function** we've built as **Helper Function #1**.

So, we can use that code as a starting point, getting rid of its steps 3 and 4, and, most importantly, we need to **use the model's `eval()` method**. The only thing it does is **setting the model to evaluation mode** (just like its `train()` counterpart did), so the model can adjust its behavior accordingly when it has to perform some operations, like `Dropout`.

"Why is setting the mode so important?"*

As mentioned above, dropout (a commonly used regularization technique used for reducing overfitting) is the main reason for it, since it requires the model to behave **differently** during training and evaluation. In a nutshell, dropout **randomly sets some weights to zero** during training.

What would happen if this behavior persisted outside of training time? You would end up with possibly **different predictions** for the **same input** since different weights would be set to zero every time you make a prediction. It would **ruin evaluation** and, if deployed, it would also **ruin the confidence of the user**.

We **don't want that**, so we use `model.eval()` to prevent it :-)

Just like `make_train_step`, our new function, `make_val_step` is a higher-order function as well. Its code looks like this:

Helper Function #3

```
def make_val_step(model, loss_fn):
    # Builds function that performs a step
    # in the validation loop
    def perform_val_step(x, y):
        # Sets model to EVAL mode
        model.eval()     ①

        # Step 1 - Computes our model's predicted output
        # forward pass
        yhat = model(x)
        # Step 2 - Computes the loss
        loss = loss_fn(yhat, y)
        # There is no need to compute Steps 3 and 4,
        # since we don't update parameters during evaluation
        return loss.item()

    return perform_val_step
```

① Setting model to evaluation mode

And then, we update our **Model Configuration** code to include the creation of the corresponding function for the **validation step**.

Define - Model Configuration V2

```
%%writefile model_configuration/v2.py

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model, loss function
# and optimizer
train_step = make_train_step(model, loss_fn, optimizer)

# Creates the val_step function for our model and loss function
val_step = make_val_step(model, loss_fn) ①
```

① Creating a function that performs a validation step

Run - Model Configuration V2

```
%run -i model_configuration/v2.py
```

Finally, we need to change the training loop to include the **evaluation of our model**. The first step is to include another inner loop to handle the *mini-batches* that come from the *validation loader*, sending them to the same *device* as our model. Then, inside that inner loop, we use the *validation step function* to compute the loss.

"*Wait, this looks oddly familiar too…*"

And indeed, it is - it is structurally the same as our **mini-batch function** (Helper Function #2). So let's use it once again!

There is just **one small, yet important** detail to consider… Remember `no_grad()`? We used it in Chapter 1 to avoid messing with PyTorch's Dynamic Computation Graph during the (manual) update of the parameters. And it is making a comeback now - we need to use it to wrap our new validation's inner loop:

`torch.no_grad()`: even though it won't make a difference in our simple model, it is a **good practice** to **wrap the validation** inner loop with this **context manager**[56] **to disable any gradient computation** that you may inadvertently trigger — **gradients belong in training**, not in validation steps.

Now, our training loop should look like this:

Define - Model Training V4

```
%%writefile model_training/v4.py

# Defines number of epochs
n_epochs = 200

losses = []
val_losses = []                                                 ③

for epoch in range(n_epochs):
    # inner loop
    loss = mini_batch(device, train_loader, train_step)
    losses.append(loss)

    # VALIDATION - no gradients in validation!
    with torch.no_grad():                                       ①
        val_loss = mini_batch(device, val_loader, val_step)     ②
        val_losses.append(val_loss)                             ③
```

① Using `no_grad` as context manager to prevent gradient computation

② Performing a validation step

③ Keeping track of validation loss

Run - Model Training V4

```
%run -i model_training/v4.py
```

After updating all parts, in sequence, our current state of development is:

- **Data Preparation V2**
- **Model Configuration V2**
- **Model Training V4**

Let's inspect the model's state:

```
# Checks model's parameters
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9419]], device='cuda:0')),
('0.bias', tensor([1.0244], device='cuda:0'))])
```

Plotting Losses

Let's take a look at both losses - **training**, and **validation**:

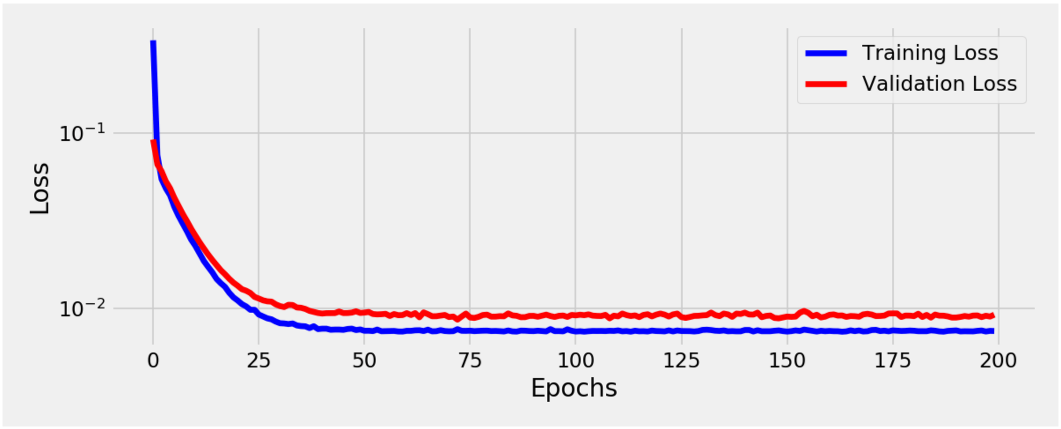


Figure 2.1 - Training and validation losses during training

Does your plot look different? Try running the whole pipeline again:

Full pipeline

```
%run -i data_preparation/v2.py
%run -i model_configuration/v2.py
%run -i model_training/v4.py
```

And then plot the resulting losses one more time.

Cool, right? But, remember in the **Training Step** function, when I mentioned that adding losses to a list was not very **cutting-edge**? Time to fix that! To better visualizing the training process, we can make use of…

TensorBoard

Yes, TensorBoard is *that* good! So good that we'll be using a tool from the competing framework, Tensorflow :-) Jokes aside, TensorBoard is a very useful tool, and PyTorch provides classes and methods for us to integrate it with our model.

Running It Inside a Notebook

This section applies to both *Google Colab* and *local installation*.

If you are using a *local installation*, you can either run Tensorboard *inside* a notebook or *separately* (check the next section for instructions).

If you chose to follow this book using *Google Colab*, you'll **need** to run TensorBoard **inside a notebook**. Luckily, this is easily accomplished using some Jupyter **magics**.

If you are using *Binder*, this Jupyter **magic will not work**, for reasons that are beyond the scope of this section. More details on how to use TensorBoard with *Binder* can be found in the corresponding section below.

First, we need to load Tensorboard's *extension* for Jupyter:

Loading extension

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

Then, we *run* Tensorboard using the newly available magic:

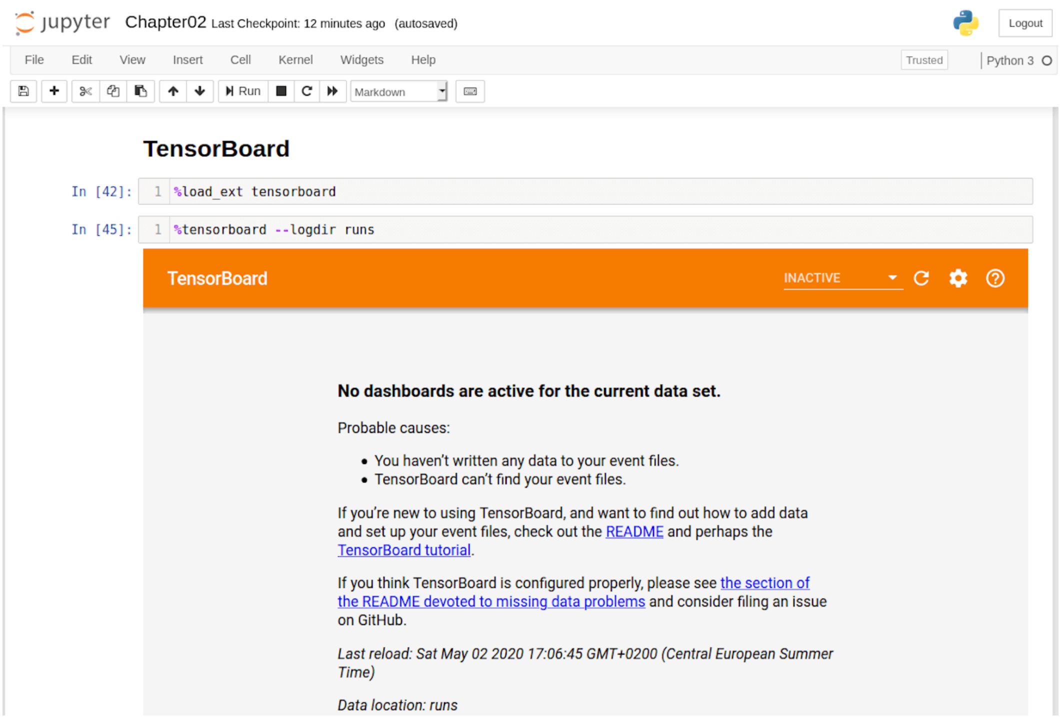
Running TensorBoard

```
%tensorboard --logdir runs
```

The magic above tells TensorBoard to *look for logs* inside the folder specified by the `logdir` argument: `runs`. So, *there must be a `runs` folder in the same location as the notebook* you're using to train the model. To make things easier for you, I created a `runs` folder in the repository, so you get it out-of-the-box already.

If you get an error **"TypeError: Function expected"**, please switch to a modern browser like Firefox or Chrome.

Your notebook will show TensorBoard inside a cell, just like this:



TensorBoard

```
In [42]: %load_ext tensorboard
```

```
In [45]: %tensorboard --logdir runs
```

No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Sat May 02 2020 17:06:45 GMT+0200 (Central European Summer Time)

Data location: runs

Figure 2.2 - TensorBoard running inside a notebook

It doesn't show you anything yet because it cannot find any data inside the `runs` folder, as we haven't sent anything there yet. It will be automatically updated when we send some data to it so, let's send some data to TensorBoard!

If you want to learn more about running TensorBoard inside a notebook, configurations options, and all, please check this official guide[57].

Running It Separately (Local Installation)

Assuming you've installed TensorBoard while following the **Setup Guide**, now you need to open a *new* terminal or Anaconda Prompt, *navigate* to the `PyTorchStepByStep` folder you cloned from GitHub, and *activate* the `pytorchbook` environment:

Activating environment

```
conda activate pytorchbook
```

Then you can run TensorBoard:

Running Tensorboard

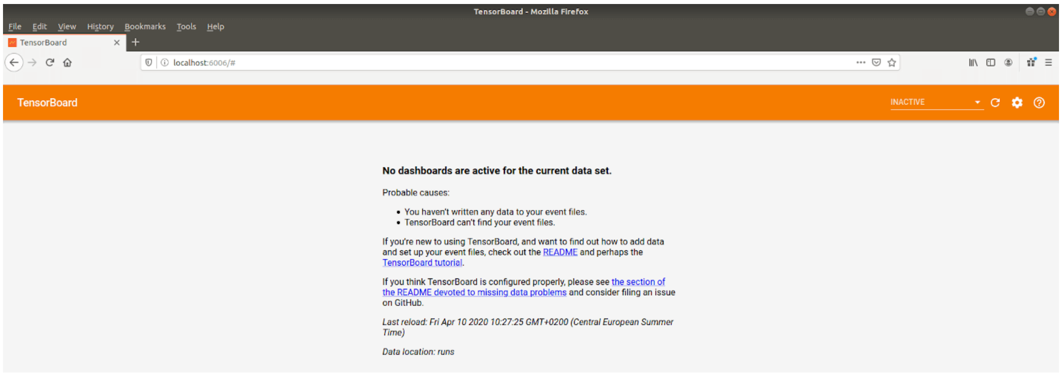
```
(pytorchbook)$ tensorboard --logdir runs
```

The above command tells TensorBoard to *look for logs* inside the folder specified by the `logdir` argument: `runs`. So, *there must be a `runs` folder in the same location as the notebook* you're using to train the model. To make things easier, I created a `runs` folder in the repository, so you get it out-of-the-box already. After running it, you'll see a message like this one (the version of TensorBoard may be different than yours, though):

Output

```
TensorFlow installation not found - running with reduced
feature set.
Serving TensorBoard on localhost; to expose to the network,
use a proxy or pass --bind_all
TensorBoard 2.2.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

You see, it "*complains*" about not finding Tensorflow :-) Nonetheless, it is up and running! If you throw the address `http://localhost:6006/` at your favorite browser, you'll likely see something like this:


TensorBoard - Mozilla Firefox
File Edit View History Bookmarks Tools Help
TensorBoard
localhost:6006/#
TensorBoard
INACTIVE

No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Fri Apr 10 2020 10:27:25 GMT+0200 (Central European Summer Time)

Data location: runs

Figure 2.3 - Empty TensorBoard

It doesn't show you anything yet because it cannot find any data inside the `runs` folder, as we haven't sent anything there yet. It will be automatically updated when we send some data to it so, let's send some data to TensorBoard!

Running It Separately (Binder)

If you chose to follow this book using *Binder*, you'll **need** to run TensorBoard separately.

But you won't have to actually do much. Configuring TensorBoard for running inside Binder's environment is a bit *tricky* (it involves Jupyter's server extensions), so I took care of that for you :-)

Moreover, I've provided an **automatically generated link** that will **open a new tab** pointing to the TensorBoard instance running in your Binder environment.

The link looks like this (the actual URL is generated on the spot, this one is just a dummy):

Click here to open TensorBoard

The only downside is that the **folder** where TensorBoard will look for logs is **fixed**: `runs`.

SummaryWriter

It all starts with the creation of a `SummaryWriter`:

SummaryWriter

```
writer = SummaryWriter('runs/test')
```

Since we told TensorBoard to look for logs inside the `runs` folder, it only makes sense to **actually log to that folder**. Moreover, to be able to *distinguish* between different experiments or models, we should also specify a sub-folder: `test`.

If we do not specify any folder, TensorBoard will default to `runs/CURRENT_DATETIME_HOSTNAME`, which is not such a great name if you'd be looking for your experiment results in the future.

So, it is recommended to **try to name it in a more meaningful way**, like `runs/test` or `runs/simple_linear_regression`. It will then create a subfolder inside `runs` (the folder we specified when we started TensorBoard).

Even better, you should name it in a meaningful way **and add datetime or a sequential number as a suffix**, like `runs/test_001` or `runs/test_20200502172130`, to avoid writing data of multiple runs into the same folder (we'll see why this is bad in the **add_scalars** section below).

The summary writer implements several methods to allow us sending information to TensorBoard:

`add_graph`	`add_scalars`	`add_scalar`
`add_histogram`	`add_images`	`add_image`
`add_figure`	`add_video`	`add_audio`

add_text	add_embedding	add_pr_curve
add_custom_scalars	add_mesh	add_hparams

It also implements two other methods for effectively writing data to disk:

- flush
- close

We'll be using the first two methods (add_graph and add_scalars) to send our model's graph (not quite the same as the *Dynamic Computation Graph* we drew using make_dot, though...) and, of course, both scalars: **training** and **validation losses**.

add_graph

Let's start with add_graph: unfortunately, its documentation seems to be absent (as at the time of writing), and its default values for arguments lead you to believe you don't need to provide any inputs (input_to_model=None). What happens if we try it?

```
writer.add_graph(model)
```

We'll get an enormous **error message** that ends with:

Output

```
...
TypeError: 'NoneType' object is not iterable
```

So, we **do** need to send it some **inputs** together with our model... Let's fetch a mini-batch of data points from our train_loader and then pass it as input to add_graph:

Adding the model's graph

```
# Fetching a tuple of feature (dummy_x) and label (dummy_y)
dummy_x, dummy_y = next(iter(train_loader))

# Since our model was sent to device, we need to do the same
# with the data.
# Even here, both model and data need to be on the same device!
writer.add_graph(model, dummy_x.to(device))
```

If you open (or refresh) again your browser (or re-run the cell containing the magic `%tensorboard --logdir runs` inside a notebook) to look at TensorBoard, it should look like this:

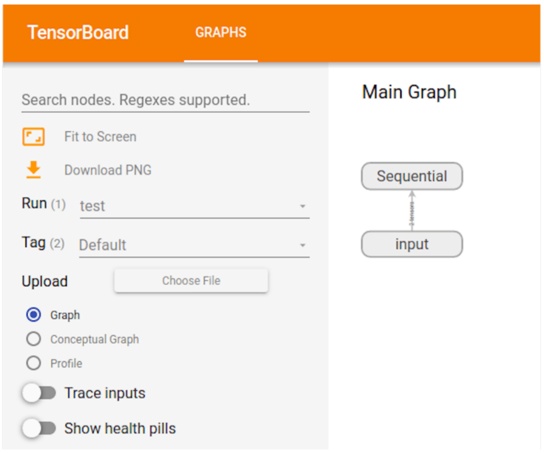


Figure 2.4 - Dynamic Computation Graph on TensorBoard

add_scalars

What about sending the **loss values** to TensorBoard? I'm on it! We can use `add_scalars` method to send multiple scalar values at once, and it needs three arguments:

- **main_tag**: the parent name of the tags or, the "group tag", if you will
- **tag_scalar_dict**: the dictionary containing the `key: value` pairs for the scalars

you want to keep track of (in our case, training and validation losses)

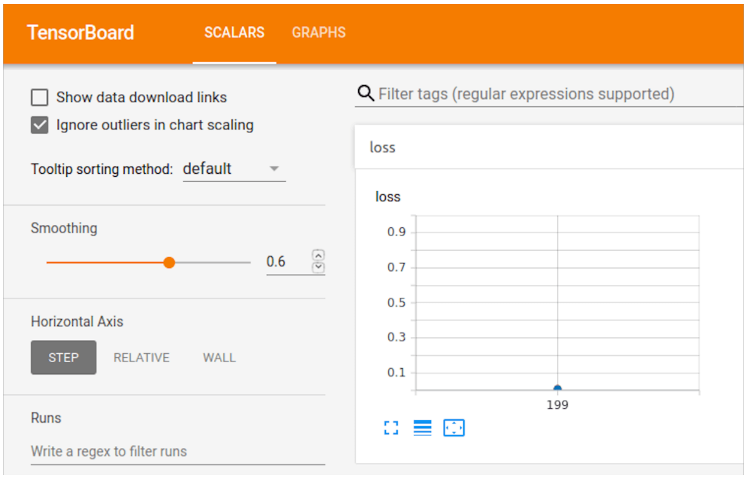
- **global_step**: step value, that is, the index you're associating with the values you're sending in the dictionary - the **epoch** comes to mind in our case, as losses are computed for each epoch

How does it translate into code? Let's check it out:

Adding losses

```
writer.add_scalars(
    main_tag='loss',
    tag_scalar_dict={'training': loss,
                     'validation': val_loss},
    global_step=epoch
)
```

If you run the code above after performing the model training, it will just send both loss values computed for the last epoch (199). Your TensorBoard will look like this (don't forget to refresh it - it may take a while if you're running it on Google Colab):



TensorBoard
SCALARS
GRAPHS
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
Filter tags (regular expressions supported)
loss
loss
0.9
0.7
0.5
0.3
0.1
199

Figure 2.5 - Scalars on TensorBoard

Not very useful, uh? We need to incorporate these elements into our **Model Configuration** and **Model Training** codes, which look like this now:

Run - Data Preparation V2

```
%run -i data_preparation/v2.py
```

Define - Model Configuration V3

```
%%writefile model_configuration/v3.py

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model,
# loss function and optimizer
train_step = make_train_step(model, loss_fn, optimizer)

# Creates the val_step function for our model and loss function
val_step = make_val_step(model, loss_fn)

# Creates a Summary Writer to interface with TensorBoard
writer = SummaryWriter('runs/simple_linear_regression') ①
# Fetches a single mini-batch so we can use add_graph
x_dummy, y_dummy = next(iter(train_loader))
writer.add_graph(model, x_dummy.to(device))
```

① Creating SummaryWriter to interface with TensorBoard

Run - Model Configuration V3

```
%run -i model_configuration/v3.py
```

Define - Model Training V5

```
%%writefile model_training/v5.py

# Defines number of epochs
n_epochs = 200

losses = []
val_losses = []

for epoch in range(n_epochs):
    # inner loop
    loss = mini_batch(device, train_loader, train_step)
    losses.append(loss)

    # VALIDATION - no gradients in validation!
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step)
        val_losses.append(val_loss)

    # Records both losses for each epoch under tag "loss"
    writer.add_scalars(main_tag='loss',      ①
                       tag_scalar_dict={
                            'training': loss,
                            'validation': val_loss},
                       global_step=epoch)

# Closes the writer
writer.close()
```

① Sending losses to TensorBoard

Run - Model Training V5

```
%run -i model_training/v5.py
```

After the last update of both model configuration and training parts, our current state of development is:

- **Data Preparation V2**
- **Model Configuration V3**
- **Model Training V5**

You probably noticed we **did not** throw the two lists (**losses** and **val_losses**) away… there is a reason for it, which will be clear in the next section.

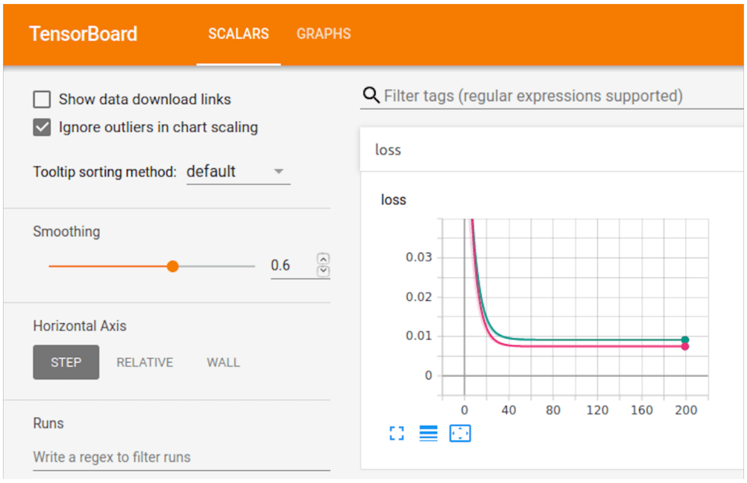
Let’s inspect the model’s state:

```
# Checks model's parameters
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9448]], device='cuda:0')),
('0.bias', tensor([1.0295], device='cuda:0'))])
```

Now, let’s inspect **TensorBoard**. You should see something like this:

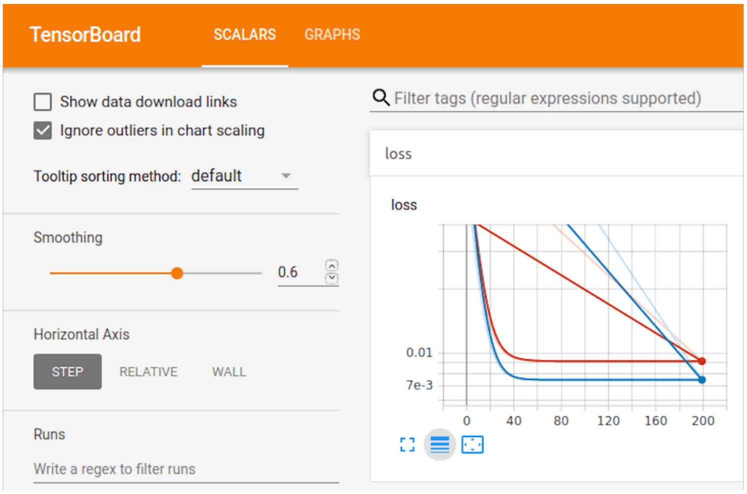

TensorBoard
SCALARS
GRAPHS
Filter tags (regular expressions supported)
loss
loss
0.03
0.02
0.01
0
0
40
80
120
160
200
Runs
Write a regex to filter runs

Show data download links: ☐
Ignore outliers in chart scaling: ✓
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: [✓ STEP] RELATIVE WALL

Figure 2.6 - Finally, losses on Tensorboard

This is the same plot we've built before using our lists and Matplotlib. If our model were big or complex enough to take at least a couple of minutes to train, we would be able to see the *evolution* of our losses in TensorBoard during training.

If, by any chance, you ended up with something like the weird plot below, don't worry just yet!


TensorBoard
SCALARS
GRAPHS
Filter tags (regular expressions supported)
loss
loss
0.01
7e-3
0
40
80
120
160
200
Runs
Write a regex to filter runs

Show data download links: ☐
Ignore outliers in chart scaling: ✓
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: [✓ STEP] RELATIVE WALL

Figure 2.7 - Weird results on TensorBoard :P

Remember, I said writing data of multiple runs into the same folder was bad? This is it…

Since we're writing data to the folder `runs/simple_linear_regression`, if we do not **change the name** of the folder (or **erase the data** there) before running the code a second time, TensorBoard gets somewhat confused, as you can guess from its output:

- *Found more than one graph event per run* (because we ran `add_graph` more than once)
- *Found more than one "run metadata" event with tag step1* (because we ran `add_scalars` more than once)

If you are using a local installation, you can see those messages in the **terminal window or Anaconda Prompt** you used to run `tensorboard --log_dir=runs`.

So, you finished training your model, you inspected TensorBoard plots, and you're happy with the losses you got.

Congratulations! Your job is done; you successfully trained your model!

There is only **one more thing** you need to know, and that is how to handle…

Saving and Loading Models

Training a model successfully is great, no doubt about that, but not all models will be that fast to be trained, and maybe **training gets interrupted** (computer crashing, timeout after 12h of continuous GPU usage on Google Colab, etc.). It would be a pity to have to start over, right?

So, it is important to be able to **checkpoint or save** our model, that is, saving it to disk, in case we'd like to **restart training later** or **deploy it** as an application to **make predictions**.

Model State

To checkpoint a model, we basically have to **save its state** to a file so that it can be **loaded** back later - nothing special, actually.

What defines the **state of a model**?

- `model.state_dict()`: kinda obvious, right?
- `optimizer.state_dict()`: remember, optimizers have a `state_dict` as well
- **losses**: after all, you should keep track of its evolution
- **epoch**: it is just a number, so why not? :-)
- **anything else you'd like to have restored later**...

Saving

Now, **wrap everything into a Python dictionary** and use `torch.save()` to dump it all into a file. Easy peasy! We have just **saved** our model to a file named `model_checkpoint.pth`.

Notebook Cell 2.4 - Saving checkpoint

```
checkpoint = {'epoch': n_epochs,
              'model_state_dict': model.state_dict(),
              'optimizer_state_dict': optimizer.state_dict(),
              'loss': losses,
              'val_loss': val_losses}

torch.save(checkpoint, 'model_checkpoint.pth')
```

The procedure is exactly the **same**, either if you are **checkpointing a partially trained model** to resume training later or if you are **saving a fully trained model** to deploy it and make predictions.

OK, what about **loading** it back? In that case, it will be a **bit different**, depending on what you're doing…

Resuming Training

If we're starting fresh (as if we had just turned on the computer and started Jupyter), we have to **set up the stage** before actually loading the model. This means we need to **load the data** and **configure the model**.

Luckily, we have code for that already: **Data Preparation V2** and **Model Configuration V3**:

Notebook Cell 2.5

```
%run -i data_preparation/v2.py
%run -i model_configuration/v3.py
```

Let's double-check that we do have an **untrained model**:

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[0.7645]], device='cuda:0')),
('0.bias', tensor([0.8300], device='cuda:0'))])
```

Now we are **ready** to load the model back, which is easy:

- load the dictionary back using `torch.load()`
- load **model** and **optimizer** state dictionaries back using the `load_state_dict()` method
- load everything else into their corresponding variables

Notebook Cell 2.6 - Loading checkpoint to resume training

```
checkpoint = torch.load('model_checkpoint.pth')

model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

saved_epoch = checkpoint['epoch']
saved_losses = checkpoint['loss']
saved_val_losses = checkpoint['val_loss']

model.train() # always use TRAIN for resuming training ①
```

① Never forget to set the mode!

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9448]], device='cuda:0')),
('0.bias', tensor([1.0295], device='cuda:0'))])
```

Cool, we recovered our **model's state**, and we can **resume training**.

After **loading a model to resume training**, make sure you **ALWAYS** set it to **training mode**:

```
model.train()
```

In our example, this is going to be redundant because our `train_step` function already does it. But it is important to pick up the habit of **setting the mode** of the model accordingly.

Next, we can run **Model Training V5** to train it for another 200 epochs.

"*Why* ***200*** *more epochs? Can't I choose a* ***different*** *number?*"

Well, you could, but you'd have to change the code in **Model Training V5**… this clearly **isn't** ideal, but we will make our model training code more flexible very soon, so please bear with me for now.

Notebook Cell 2.7

```
%run -i model_training/v5.py
```

How does the model look like after training another 200 epochs?

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9448]], device='cuda:0')),
('0.bias', tensor([1.0295], device='cuda:0'))])
```

Well, it didn't change at all, which is **no surprise**: the original model had **converged** already; that is, the loss was at a **minimum**. These extra epochs served an educational purpose only; they did not improve the model. But, since we are at it, let's check the evolution of the losses, **before** and **after** checkpointing:

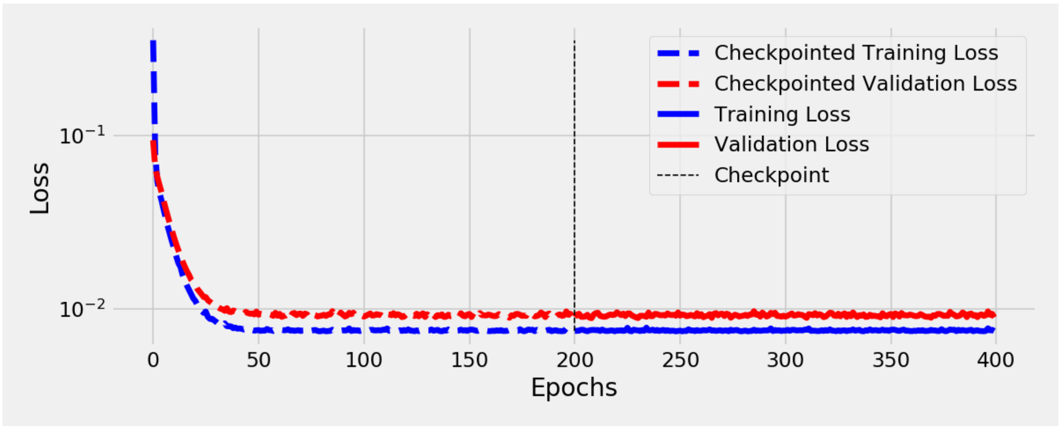


Figure 2.8 - Losses, before and after resuming training

Clearly, the loss was already at a minimum before the checkpoint so, nothing has changed!

It turns out, the model we saved to disk was a **fully trained model** so that we can use it for…

Deploying / Making Predictions

Again, if we're starting fresh (as if we had just turned on the computer and started Jupyter), we have to **set up the stage** before actually loading the model. But, this time, we **only** need to **configure the model**:

Notebook Cell 2.8

```
%run -i model_configuration/v3.py
```

Once again, we have an **untrained model** at this point. The loading procedure is simpler, though:

- load the dictionary back using `torch.load()`
- load **model** state dictionary back using its method `load_state_dict()`

Since the model is **fully trained**, we don't need to load the optimizer or anything else.

Notebook Cell 2.9 - Loading a fully trained model to make predictions

```
checkpoint = torch.load('model_checkpoint.pth')

model.load_state_dict(checkpoint['model_state_dict'])

print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9448]], device='cuda:0')),
('0.bias', tensor([1.0295], device='cuda:0'))])
```

After recovering our **model's state**, we can finally use it to **make predictions** for **new inputs**:

Notebook Cell 2.10

```
new_inputs = torch.tensor([[.20], [.34], [.57]])

model.eval() # always use EVAL for fully trained models! ①
model(new_inputs.to(device))
```

① Never forget to set the mode!

Output

```
tensor([[1.4185],
        [1.6908],
        [2.1381]], device='cuda:0', grad_fn=<AddmmBackward>)
```

Since **Model Configuration V3** created a model and sent it automatically to our **device**, we need to do the same with our **new inputs**.

After **loading a fully trained model for deployment / to make predictions**, make sure you **ALWAYS** set it to **evaluation mode**:

`model.eval()`

Congratulations, you "*deployed*" your first model :-)

Setting the Model's Mode

I know, I am probably a bit obsessive about this, but here we go one more time:

After loading the model, **DO NOT FORGET** to **SET THE MODE** :

- **checkpointing**: `model.train()`
- **deploying / making predictions**: `model.eval()`

Putting It All Together

We have **updated** each one of the three **fundamental parts** of our code at least twice. It is time to put it all together to get an overall view of what we have achieved so far.

Behold your pipeline: **Data Preparation V2**, **Model Configuration V3**, and **Model Training V5**!

Run - Data Preparation V2

```
# %load data_preparation/v2.py

torch.manual_seed(13)

# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(x).float()
y_tensor = torch.as_tensor(y).float()

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train
train_data, val_data = random_split(dataset, [n_train, n_val])
# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_data,
    batch_size=16,
    shuffle=True,
)
val_loader = DataLoader(dataset=val_data, batch_size=16)
```

Run - Model Configuration V3

```python
# %load model_configuration/v3.py

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model,
# loss function and optimizer
train_step = make_train_step(model, loss_fn, optimizer)

# Creates the val_step function for our model and loss function
val_step = make_val_step(model, loss_fn)

# Creates a Summary Writer to interface with TensorBoard
writer = SummaryWriter('runs/simple_linear_regression')
# Fetches a single mini-batch so we can use add_graph
x_dummy, y_dummt = next(iter(train_loader))
writer.add_graph(model, x_dummy.to(device))
```

Run - Model Training V5

```python
# %load model_training/v5.py

# Defines number of epochs
n_epochs = 200

losses = []
val_losses = []

for epoch in range(n_epochs):
    # inner loop
    loss = mini_batch(device, train_loader, train_step)
    losses.append(loss)

    # VALIDATION - no gradients in validation!
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step)
        val_losses.append(val_loss)

    # Records both losses for each epoch under tag "loss"
    writer.add_scalars(main_tag='loss',
                       tag_scalar_dict={
                            'training': loss,
                            'validation': val_loss},
                       global_step=epoch)

# Closes the writer
writer.close()
```

```python
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9440]], device='cuda:0')),
('0.bias', tensor([1.0249], device='cuda:0'))])
```

This is the **general structure** you'll use *over and over again* for **training PyTorch models**.

Sure, a **different dataset and problem** will require a **different model and loss function**, and you may want to try a different optimizer and a cycling learning rate (we'll get to that later!), but the rest is likely to remain exactly the same.

Recap

We have covered a lot of ground in this first chapter:

- writing a **higher-order function** that builds functions to perform **training steps**
- understanding PyTorch's `Dataset` and `TensorDataset` classes, implementing its `__init__`, `__get_item__` and `__len__` methods
- using PyTorch's `DataLoader` class to **generate mini-batches** out of a dataset
- modifying our **training loop** to incorporate **mini-batch gradient descent** logic
- writing a **helper function** to handle the **mini-batch inner loop**
- using PyTorch's `random_split` method to generate training and validation datasets
- writing a **higher-order function** that builds functions to perform **validation steps**
- realizing the **importance** of including `model.eval()` inside the **validation loop**
- remembering the purpose of `no_grad()` and using it to **prevent** any kind of **gradient computation during validation**

- using `SummaryWriter` to **interface** with *TensorBoard* for logging
- adding a graph representing our model to **TensorBoard**
- sending **scalars** to *TensorBoard* to track the **evolution of training and validation losses**
- **saving/checkpointing** and **loading** models to and from disk to allow **resuming model training** or **deployment**
- realizing the importance of **setting the mode** of the model: `train()` or `eval()`, for **checkpointing** or **deploying** for prediction, respectively

Congratulations! You now possess the necessary **knowledge** and **tools** to tackle more interesting (and complex!) problems using PyTorch. We'll put them to good use in the next chapters.

[50] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter02.ipynb

[51] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter02.ipynb

[52] https://bit.ly/3jJtJeT

[53] https://bit.ly/37vJVdG

[54] https://bit.ly/39u1tbo

[55] https://bit.ly/39ovRUx

[56] https://www.geeksforgeeks.org/context-manager-in-python/

[57] https://www.tensorflow.org/tensorboard/tensorboard_in_notebooks

Chapter 2.1
Going Classy

Spoilers

In this chapter, we will:

- define a **class** to handle **model training**
- implement the **constructor** method
- understand the difference between **public**, **protected**, and **private** methods of a class
- **integrate the code** we've developed so far into our class
- **instantiate** our class and use it to run a **classy** pipeline

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 2.1[58] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[59].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 2.1's notebook. If not, just click on `Chapter02.1.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
import datetime

import torch
import torch.optim as optim
import torch.nn as nn
import torch.functional as F
from torch.utils.data import DataLoader, TensorDataset, random_split
from torch.utils.tensorboard import SummaryWriter

import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('fivethirtyeight')
```

Going Classy

So far, the `%%writefile` magic has helped us to organize the code into three distinct parts: *data preparation*, *model configuration*, and *model training*. At the end of Chapter 2, though, we bumped into some of its **limitations**, like being unable to choose a different number of epochs without having to **edit** the model training code.

Clearly, this situation is **not** ideal. We need to do better. We need to **go classy**; that is, we need to build a **class** to handle the **model training** part.

I am assuming you have a *working knowledge* of **Object-Oriented Programming (OOP)** in order to benefit the most from this chapter. If that's not the case, and if you haven't done it in Chapter 1, now it is the time to follow tutorials like Real Python's Objected-Oriented Programming (OOP) in Python 3[60] and Supercharge Your Classes With Python super()[61] before proceeding.

The Class

Let's start defining our class with a rather unoriginal name: `StepByStep`. We're starting it from scratch: either we don't specify a parent class, or we inherit it from the fundamental `object` class. I personally prefer the latter, so our class definition looks like this:

```
# A completely empty (and useless) class
class StepByStep(object):
    pass
```

Boring, right? Let's make it more interesting by further building it.

The Constructor

"*From where do we start building a class?*"

That would be the **constructor**, the `__init__(self)` method that we've already seen a couple of times when handling both **model** and **dataset** classes.

The constructor **defines the parts that make up the class**. These parts are the **attributes** of the class. Typical attributes include:

- **arguments** provided by the user
- **placeholders** for other objects that are not available at the moment of creation

(pretty much like *delayed* arguments)

- **variables** we may want to keep track of
- **functions** that are dynamically built using some of the arguments and **higher-order functions**

Let's see how each one of those applies to our problem.

Arguments

Let's start with the **arguments**, the part that **needs to be specified by the user**. At the beginning of Chapter 2, we asked ourselves: "*Would the code inside the training loop **change** if we were using a **different optimizer**, or **loss**, or even **model**?*". The answer was and still is, **no, it wouldn't change**.

So, these **three** elements, **optimizer**, **loss**, and **model**, will be our main **arguments**. The user **needs** to specify those; we can't figure them on our own.

But there is one more piece of information needed: the **device** to be used for training the model. Instead of asking the user to inform it, we'll *automatically check* if there is a GPU available and fall back to a CPU if there isn't. But we still want to give the user a chance to use a different device (whatever the reason may be); thus, we add a very simple method (conveniently named `to`) that allows the user to specify a device.

Our constructor (`__init__`) method will initially look like this:

```
class StepByStep(object):
    def __init__(self, model, loss_fn, optimizer):
        # Here we define the attributes of our class
        # We start by storing the arguments as attributes
        # to use them later
        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        # Let's send the model to the specified device right away
        self.model.to(self.device)

    def to(self, device):
        # This method allows the user to specify a different device
        # It sets the corresponding attribute (to be used later in
        # the mini-batches) and sends the model to the device
        self.device = device
        self.model.to(self.device)
```

Placeholders

Next, let’s tackle the **placeholders** or *delayed arguments*. We expect the user to **eventually** provide **some** of those, as they are *not necessarily required*. There are another three elements that fall into that category: **train and validation data loaders** and a **summary writer** to interface with TensorBoard.

We need to append the following code to the constructor method above (I am not reproducing the rest of the method here for the sake of simplicity - in the Jupyter notebook you’ll find the full code):

```
        # These attributes are defined here, but since they are
        # not available at the moment of creation, we keep them None
        self.train_loader = None
        self.val_loader = None
        self.writer = None
```

The **train data loader** is obviously required. How could we possibly train a model without it?

> *"Why don't we make the train data loader an **argument** then?"*

Conceptually speaking, the data loader (and the dataset it contains) is **not** part of the model. It is the **input** we use to train the model. Since we **can** specify a model without it, it shouldn't be made an argument of our class.

In other words, our `StepByStep` class is **defined by a particular combination of arguments** (model, loss function, and optimizer), which can then be used to perform model training on any (compatible) dataset.

The **validation data loader** is not required (although it is recommended), and the **summary writer** is definitely optional.

The class should implement **methods** to allow the user to inform those at a later time (both methods should be placed *inside* the `StepByStep` class, after the constructor method):

```
    def set_loaders(self, train_loader, val_loader=None):
        # This method allows the user to define which train_loader
        # (and val_loader, optionally) to use
        # Both loaders are then assigned to attributes of the class
        # So they can be referred to later
        self.train_loader = train_loader
        self.val_loader = val_loader

    def set_tensorboard(self, name, folder='runs'):
        # This method allows the user to create a SummaryWriter to
        # interface with TensorBoard
        suffix = datetime.datetime.now().strftime('%Y%m%d%H%M%S')
        self.writer = SummaryWriter('{}/{}_{}'.format(
            folder, name, suffix
        ))
```

"*Why do we need to specify a default value to the* `val_loader`? *Its placeholder value is already* `None`."

Since the validation loader is **optional**, setting a **default value** for a particular argument in the method's definition frees the user from having to provide that argument when calling the method. The best default value, in our case, is the same value we chose when specifying the placeholder for the validation loader: `None`.

Variables

Then, there are **variables** we may want to keep track of. Typical examples are the **number of epochs**, and the training and validation **losses**. These variables are likely to be computed and updated internally by the class.

We need to **append the following code to the constructor** method once again (like we did with the placeholders):

```
        # These attributes are going to be computed internally
        self.losses = []
        self.val_losses = []
        self.total_epochs = 0
```

"Can't we just set these variables whenever we use them for the first time?"

Yes, we could, and we would probably get away with it just fine since our class is quite simple. As classes grow more complex, though, it may lead to problems. So, it is **best practice** to **define all attributes of a class in the constructor method**.

Functions

For convenience, sometimes it is useful to create **attributes** that are **functions**, which will be called somewhere else inside the class. In our case, we can create both `train_step` and `val_step` using the higher-order functions we defined in Chapter 2 (Helper Functions #1 and #3, respectively). Both of them take a model, a loss function, and an optimizer as arguments, and all of those are already known attributes of our `StepByStep` class at construction time.

The code below will be the last addition to our constructor method (once again, as we did with the placeholders):

```
        # Creates the train_step function for our model,
        # loss function and optimizer
        # Note: there are NO ARGS there! It makes use of the class
        # attributes directly
        self.train_step = self._make_train_step()
        # Creates the val_step function for our model and loss
        self.val_step = self._make_val_step()
```

If you have patched together the pieces of code above, your code should look like this:

StepByStep Class

```
class StepByStep(object):
    def __init__(self, model, loss_fn, optimizer):
        # Here we define the attributes of our class
        # We start by storing the arguments as attributes
        # to use them later
        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        # Let's send the model to the specified device right away
        self.model.to(self.device)

        # These attributes are defined here, but since they are
        # not available at the moment of creation, we keep them None
        self.train_loader = None
        self.val_loader = None
        self.writer = None

        # These attributes are going to be computed internally
        self.losses = []
        self.val_losses = []
        self.total_epochs = 0

        # Creates the train_step function for our model,
        # loss function and optimizer
        # Note: there are NO ARGS there! It makes use of the class
        # attributes directly
        self.train_step = self._make_train_step()
        # Creates the val_step function for our model and loss
        self.val_step = self._make_val_step()

    def to(self, device):
        # This method allows the user to specify a different device
        # It sets the corresponding attribute (to be used later in
        # the mini-batches) and sends the model to the device
        self.device = device
        self.model.to(self.device)

    def set_loaders(self, train_loader, val_loader=None):
        # This method allows the user to define which train_loader
        # (and val_loader, optionally) to use
        # Both loaders are then assigned to attributes of the class
        # So they can be referred to later
        self.train_loader = train_loader
        self.val_loader = val_loader

    def set_tensorboard(self, name, folder='runs'):
        # This method allows the user to create a SummaryWriter to
        # interface with TensorBoard
        suffix = datetime.datetime.now().strftime('%Y%m%d%H%M%S')
        self.writer = SummaryWriter('{}/{}_{}'.format(
            folder, name, suffix
        ))
```

Sure, we are still missing both `_make_train_step` and `_make_val_step` functions… both are pretty much the same as before, except for the fact that they refer to the class attributes **`self.model`**, **`self.loss_fn`**, and **`self.optimizer`**, instead of taking them as arguments. They look like this now:

Step Methods

```
def _make_train_step(self):
    # This method does not need ARGS... it can use directly
    # the attributes: self.model, self.loss_fn and self.optimizer

    # Builds function that performs a step in the train loop
    def perform_train_step(x, y):
        # Sets model to TRAIN mode
        self.model.train()
```

```
            # Step 1 - Computes model's predicted output - forward pass
            yhat = self.model(x)
            # Step 2 - Computes the loss
            loss = self.loss_fn(yhat, y)
            # Step 3 - Computes gradients for "b" and "w" parameters
            loss.backward()
            # Step 4 - Updates parameters using gradients and the
            # learning rate
            self.optimizer.step()
            self.optimizer.zero_grad()

            # Returns the loss
            return loss.item()

        # Returns the function that will be called inside the train loop
        return perform_train_step

    def _make_val_step(self):
        # Builds function that performs a step in the validation loop
        def perform_val_step(x, y):
            # Sets model to EVAL mode
            self.model.eval()

            # Step 1 - Computes model's predicted output - forward pass
            yhat = self.model(x)
            # Step 2 - Computes the loss
            loss = self.loss_fn(yhat, y)
            # There is no need to compute Steps 3 and 4,
            # since we don't update parameters during evaluation
            return loss.item()

        return perform_val_step
```

"Why these methods have an underscore as a prefix? How is this different than the double underscore in the `__init__` *method?"*

Methods, _methods and __methods

Some programming languages, like Java, have three kinds of methods: public, protected, and private. **Public methods** are the kind you're most familiar with: they can be **called by the user**.

Protected methods, on the other hand, **shouldn't** be called by the user - they are supposed to be called either **internally** or by the **child class** (the child class can call a protected method from its parent class).

Finally, **private methods** are supposed to be called **exclusively internally**. They should be invisible even to a child class.

These rules are strictly enforced in Java, but **Python** takes a more relaxed approach: **all methods are public**, meaning you can call whatever method you want. But you can **suggest** the appropriate usage by **prefixing the method name** with a **single underscore** (for **protected methods**) or a **double underscore** (for **private methods**). This way, the user is aware of the programmer's intention.

In our example, both `_make_train_step` and `_make_val_step` are defined as **protected methods**. I expect users **not to call them directly**, but if someone decides to define a class that inherits from `StepByStep`, they **should feel entitled** to do so.

In order to make the **additions** to our code **visually simpler**, that is, without having to *replicate the full class* every time I introduce a new method, I am resorting to something that **shouldn't be used in regular circumstances**: `setattr`[62].

```
# ATTENTION! Using SETATTR for educational purposes only :-)
setattr(StepByStep, '_make_train_step', _make_train_step)
setattr(StepByStep, '_make_val_step', _make_val_step)
```

setattr

The setattr function sets the value of the specified attribute of a given object. But **methods** are also **attributes**, so we can use this function to "attach" a method to an existing class and to all its existing instances in one go!

Yes, this is a hack! No, you should not use it in your regular code! Using setattr to build a class by appending methods to it incrementally serves educational purposes only.

To illustrate how it works and why it may be dangerous, I will show you a little example. Let's create a simple Dog class, which takes only the dog's name as argument:

```
class Dog(object):
    def __init__(self, name):
        self.name = name
```

Next, let's **instantiate** our class, that is, we are *creating* a dog. Let's call it Rex. Its name is going to be stored in the name attribute:

```
rex = Dog('Rex')
print(rex.name)
```

Output

```
Rex
```

Then, let's create a bark function that takes an **instance of Dog** as argument:

```
def bark(dog):
    print('{} barks: "Woof!"'.format(dog.name))
```

Sure enough, we can call this function to make Rex bark:

```
bark(rex)
```

Output

```
Rex barks: "Woof!"
```

But that's **not** what we want. We want our dogs to be able to bark out of the box! So we will use `setattr` to give dogs the ability to bark. There is **one thing we need to change**, though, and that's the function's argument. Since we want the bark function to be a method of the `Dog` class itself, the **argument** needs to be the **method's own instance**: `self`.

```
def bark(self):
    print('{} barks: "Woof!"'.format(self.name))

setattr(Dog, 'bark', bark)
```

Does it work? Let's create a new dog:

```
fido = Dog('Fido')
fido.bark()
```

Output

```
Fido barks: "Woof!"
```

Of course it works! Not only new dogs can bark now, but **all dogs can bark**:

```
rex.bark()
```

Output

```
Rex barks: "Woof!"
```

See? We effectively **modified the underlying `Dog` class** and **all its instances** at once! It looks very cool, sure. And it can wreak havoc too!

Using `setattr` is a **hack**, I can't stress this enough! **Please don't use `setattr` in your regular code**.

Instead of creating an attribute or method directly in the class, as we've been doing so far, it is possible to use `setattr` to create them dynamically. In our `StepByStep` class, the last two lines of code created two methods in the class, each having the same name of the function used to create the method.

OK, but there are still some parts missing in order to perform model training… Let's keep adding more methods.

Training Methods

The next method we need to add corresponds to the **Helper Function #2** in Chapter 2: the **mini-batch loop**. We need to **change it** a bit, though… there, both the **data loader** and the **step function** were arguments. This is not the case anymore since we have both of them as attributes: `self.train_loader` and `self.train_step`, for training; `self.val_loader` and `self.val_step`, for validation. The only thing this method needs to know is if it is handling training or validation data. The code should look like this:

Mini-Batch

```python
 1 def _mini_batch(self, validation=False):
 2     # The mini-batch can be used with both loaders
 3     # The argument `validation`defines which loader and
 4     # corresponding step function is going to be used
 5     if validation:
 6         data_loader = self.val_loader
 7         step = self.val_step
 8     else:
 9         data_loader = self.train_loader
10         step = self.train_step
11 
12     if data_loader is None:
13         return None
14 
15     # Once the data loader and step function are set, this is the
16     # same mini-batch loop we had before
17     mini_batch_losses = []
18     for x_batch, y_batch in data_loader:
19         x_batch = x_batch.to(self.device)
20         y_batch = y_batch.to(self.device)
21 
22         mini_batch_loss = step(x_batch, y_batch)
23         mini_batch_losses.append(mini_batch_loss)
24 
25     loss = np.mean(mini_batch_losses)
26 
27     return loss
28 
29 setattr(StepByStep, '_mini_batch', _mini_batch)
```

Moreover, if the user decides **not** to provide a validation loader, it will retain its initial `None` value from the constructor method. If that's the case, we don't have a corresponding loss to compute, and it returns `None` instead (line 13 in the snippet above).

What's left to do? The **training loop**, of course! This is similar to our **Model Training V5** in Chapter 2, but we can make it more flexible, taking the **number of epochs** and the **random seed** as arguments.

This solves the issue we faced in Chapter 2, when we *had* to train for another 200 epochs after loading a checkpoint, just because it was *hard-coded* into the training loop. Well, not anymore!

Moreover, we need to ensure the **reproducibility of the training loop**. We have already set up seeds to ensure the reproducibility of the *random split* (Data Preparation) and the *model initialization* (Model Configuration). So far, we were running the full pipeline in order, so the training loop yielded the same results every time. Now, to gain flexibility without compromising reproducibility, we need to set yet another random seed.

We're building a method to take care of seed-setting only, following PyTorch's guidelines on reproducibility[63]:

Seeds

```
def set_seed(self, seed=42):
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.manual_seed(seed)
    np.random.seed(seed)

setattr(StepByStep, 'set_seed', set_seed)
```

It is also time to use the variables we defined as attributes in the constructor method: `self.total_epochs`, `self.losses` and `self.val_losses`. All of them are being updated inside the training loop.

Training Loop

```
def train(self, n_epochs, seed=42):
    # To ensure reproducibility of the training process
    self.set_seed(seed)

    for epoch in range(n_epochs):
        # Keeps track of the numbers of epochs
        # by updating the corresponding attribute
        self.total_epochs += 1

        # inner loop
        # Performs training using mini-batches
        loss = self._mini_batch(validation=False)
        self.losses.append(loss)

        # VALIDATION
        # no gradients in validation!
        with torch.no_grad():
            # Performs evaluation using mini-batches
            val_loss = self._mini_batch(validation=True)
            self.val_losses.append(val_loss)

        # If a SummaryWriter has been set...
        if self.writer:
            scalars = {'training': loss}
            if val_loss is not None:
                scalars.update({'validation': val_loss})
            # Records both losses for each epoch under tag "loss"
            self.writer.add_scalars(main_tag='loss',
                                    tag_scalar_dict=scalars,
                                    global_step=epoch)

    if self.writer:
        # Flushes the writer
        self.writer.flush()

setattr(StepByStep, 'train', train)
```

Did you notice this function **does not return anything**? It doesn't need to! Instead of returning values, it simply updates several class attributes: `self.losses`, `self.val_losses`, and `self.total_epochs`.

The current state of development of our `StepByStep` class already allows us to train a model fully. Now, let's give our class the ability to save and load models as well.

Saving and Loading Methods

Most of the code here is exactly the same as the code we had in Chapter 2. The only difference is that we use the class' attributes instead of local variables.

The methods for saving and loading checkpoints should look like this now:

Saving

```python
def save_checkpoint(self, filename):
    # Builds dictionary with all elements for resuming training
    checkpoint = {
        'epoch': self.total_epochs,
        'model_state_dict': self.model.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        'loss': self.losses,
        'val_loss': self.val_losses
    }

    torch.save(checkpoint, filename)

setattr(StepByStep, 'save_checkpoint', save_checkpoint)
```

Loading

```python
def load_checkpoint(self, filename):
    # Loads dictionary
    checkpoint = torch.load(filename)

    # Restore state for model and optimizer
    self.model.load_state_dict(checkpoint['model_state_dict'])
    self.optimizer.load_state_dict(
        checkpoint['optimizer_state_dict']
    )

    self.total_epochs = checkpoint['epoch']
    self.losses = checkpoint['loss']
    self.val_losses = checkpoint['val_loss']

    self.model.train() # always use TRAIN for resuming training

setattr(StepByStep, 'load_checkpoint', load_checkpoint)
```

Notice that the model is set to **training mode** after loading the checkpoint.

What about making predictions? To make it easier for the user to make predictions for any new data points, we will be handling all the *Numpy* to *PyTorch* back and forth conversion inside the function.

Making Predictions

```python
def predict(self, x):
    # Set it to evaluation mode for predictions
    self.model.eval()
    # Takes a Numpy input and make it a float tensor
    x_tensor = torch.as_tensor(x).float()
    # Send input to device and uses model for prediction
    y_hat_tensor = self.model(x_tensor.to(self.device))
    # Set it back to train mode
    self.model.train()
    # Detaches it, brings it to CPU and back to Numpy
    return y_hat_tensor.detach().cpu().numpy()

setattr(StepByStep, 'predict', predict)
```

First, we set the model to **evaluation mode**, as it is required to make predictions. Then, we convert the `x` argument (assumed to be a *Numpy* array) to a float PyTorch tensor, send it to the configured device, and use the model to make a prediction.

Next, we set the model back to the **training mode**. The last step includes detaching the tensor containing the predictions and making it a *Numpy* array to be returned to the user.

We have already covered most of what was developed in the previous chapters, except for a couple of visualization functions. Let's tackle them now.

Visualization Methods

Since we have kept track of both training and validation losses as attributes, let's build a simple plot for them:

Losses

```
def plot_losses(self):
    fig = plt.figure(figsize=(10, 4))
    plt.plot(self.losses, label='Training Loss', c='b')
    if self.val_loader:
        plt.plot(self.val_losses, label='Validation Loss', c='r')
    plt.yscale('log')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    return fig

setattr(StepByStep, 'plot_losses', plot_losses)
```

Finally, if both training loader and TensorBoard were already configured, we can use the former to fetch a single mini-batch and the latter to build the model graph in TensorBoard:

Model Graph

```
def add_graph(self):
    if self.train_loader and self.writer:
        # Fetches a single mini-batch so we can use add_graph
        x_dummy, y_dummy = next(iter(self.train_loader))
        self.writer.add_graph(self.model, x_dummy.to(self.device))

setattr(StepByStep, 'add_graph', add_graph)
```

The Full Code

If you'd like to check how the full code of the class looks like, you can check it here[64] or in the Jupyter Notebook of this chapter.

We are **classy** now, so let's build a **classy pipeline** too!

Classy Pipeline

In Chapter 2, our pipeline was composed of three steps: **Data Preparation V2**, **Model Configuration V3**, and **Model Training V5**. The last step, model training, was already integrated into our `StepByStep` class. Let's take a look at the other two steps:

But, first, let's generate our synthetic data once again.

Run - Data Generation

```
# Runs data generation - so we do not need to copy code here
%run -i data_generation/simple_linear_regression.py
```

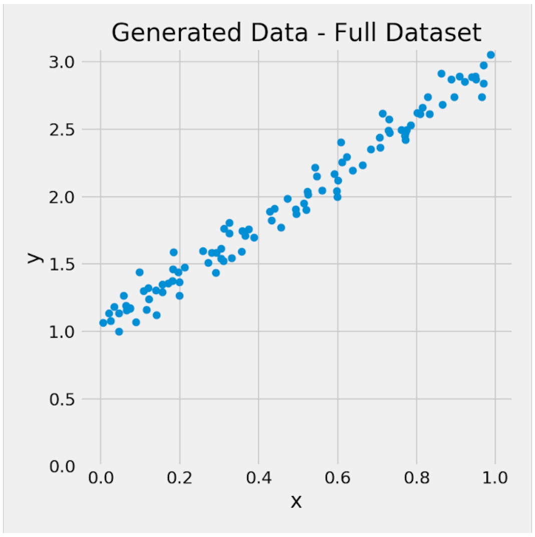


Figure 2.1.1 - Full Dataset

Looks familiar, doesn't it? :-)

The first part of the pipeline is the **Data Preparation**. It turns out; we can still keep it exactly the way it was.

Run - Data Preparation V2

```
# %load data_preparation/v2.py

torch.manual_seed(13)

# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(x).float()
y_tensor = torch.as_tensor(y).float()

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train

train_data, val_data = random_split(dataset, [n_train, n_val])

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_data,
    batch_size=16,
    shuffle=True
)
val_loader = DataLoader(dataset=val_data, batch_size=16)
```

Next in line is the **Model Configuration**. Some of its code got integrated into our class already: both `train_step` and `val_step` functions, the `SummaryWriter`, and adding the model graph.

So, we strip the model configuration code down to its bare minimum, that is, we keep only the elements we need to pass as **arguments** to our `StepByStep` class: **model**, **loss function**, and **optimizer**. Notice that we do not send the model to the device at this point anymore since this is going to be handled by our class' constructor.

Define - Model Configuration V4

```
%%writefile model_configuration/v4.py

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model
model = nn.Sequential(nn.Linear(1, 1))

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

Run - Model Configuration V4

```
%run -i model_configuration/v4.py
```

Let’s inspect the randomly initialized parameters of our model:

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[0.7645]])),
             ('0.bias', tensor([0.8300]))])
```

These are **CPU tensors**, since our model wasn't sent anywhere (yet).

And now the **fun** begins: let's put our `StepByStep` class to good use and **train our model**.

Model Training

We start by **instantiating** the `StepByStep` class with the corresponding arguments. Next, we set its loaders using the appropriately named function `set_loaders`. Then, we set up an interface with TensorBoard and name our experiment **classy** (what else could it be?!).

Notebook Cell 2.1.1

```
sbs = StepByStep(model, loss_fn, optimizer)
sbs.set_loaders(train_loader, val_loader)
sbs.set_tensorboard('classy')
```

One important thing to notice is that the `model` attribute of the `sbs` object is **the same object** as the `model` variable created in the model configuration. It is **not a copy**! We can easily verify this:

```
print(sbs.model == model)
print(sbs.model)
```

Output

```
True
Sequential(
  (0): Linear(in_features=1, out_features=1, bias=True)
)
```

As expected, the equality holds. If we print the model itself, we get our simple **one input-one output** model.

Let's **train the model** now, using the same 200 epochs as before:

Notebook Cell 2.1.2

```
sbs.train(n_epochs=200)
```

Done, it is trained! Really? Really! Let's check it out:

```
print(model.state_dict()) # remember, model == sbs.model
print(sbs.total_epochs)
```

Output

```
OrderedDict([('0.weight', tensor([[1.9414]], device='cuda:0')),
             ('0.bias', tensor([1.0233], device='cuda:0'))])
200
```

Our class sent the model to the available device (a GPU, in this case), and now the model's parameters are **GPU tensors**.

The weights of our trained model are quite close to the ones we got in Chapter 2. They are slightly different, though, because we are now using yet another *random seed* before starting the training loop. The total number of epochs was tracked by the `total_epochs` attribute, as expected.

Let's take a look at the losses:

```
fig = sbs.plot_losses()
```

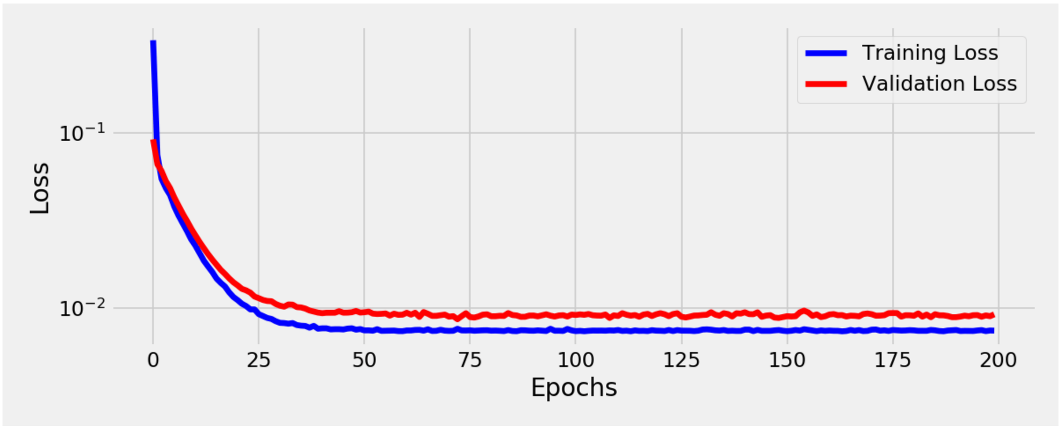


Figure 2.1.2 - Losses

Again, no surprises here… what about making predictions for new, never seen before, data points?

Making Predictions

Let's make up some data points for our feature **x**, and shape them as a single-column matrix:

```
new_data = np.array([.5, .3, .7]).reshape(-1, 1)
```

Output

```
array([[0.5],
       [0.3],
       [0.7]])
```

Since the *Numpy* array to PyTorch tensor conversion is already handled by the `predict` method, we can call the method right away, passing the array as its argument:

```
predictions = sbs.predict(new_data)
predictions
```

Output

```
array([[1.9939734],
       [1.6056864],
       [2.3822603]], dtype=float32)
```

And now we have predictions! Easy, right?

What if, instead of making predictions, we wanted to **checkpoint** the model to resume training later?

Checkpointing

That's a no-brainer… the `save_checkpoint` method handles the state dictionaries for us and save them to a file:

Notebook Cell 2.1.3

```
sbs.save_checkpoint('model_checkpoint.pth')
```

Resuming Training

Remember, when we did this in Chapter 2, we had to **set up the stage** before actually loading the model, loading the data, and configuring the model. We still need to do this, but we are now using the latest version of **model configuration**:

Run - Model Configuration V4

```
%run -i model_configuration/v4.py
```

Let's double-check that we do have an **untrained model**:

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[0.7645]], device='cuda:0')),
             ('0.bias', tensor([0.8300], device='cuda:0'))])
```

Good, same as before! Besides, the model configuration part has created the **three elements** we need to pass as **arguments** to **instantiate** our `StepByStep` class:

Notebook Cell 2.1.4

```
new_sbs = StepByStep(model, loss_fn, optimizer)
```

Next, let's **load the trained model** back using the `load_checkpoint` method and inspect the model's weights:

Notebook Cell 2.1.5

```
new_sbs.load_checkpoint('model_checkpoint.pth')
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9414]], device='cuda:0')),
             ('0.bias', tensor([1.0233], device='cuda:0'))])
```

Great, these are the weights of our trained model. Let's **train it a bit further** then…

In Chapter 2, we could only train it for another 200 epochs since the number of epochs was hard-coded. Not anymore! Thanks to our `StepByStep` class, we have the flexibility to train the model for as many epochs as we please.

But we are still missing one thing… the data! First, we need to **set the data loader(s)**, and then we can train our model for another, say, 50 epochs.

Notebook Cell 2.1.6

```
new_sbs.set_loaders(train_loader, val_loader)
new_sbs.train(n_epochs=50)
```

Let's take a look at the losses:

```
fig = new_sbs.plot_losses()
```

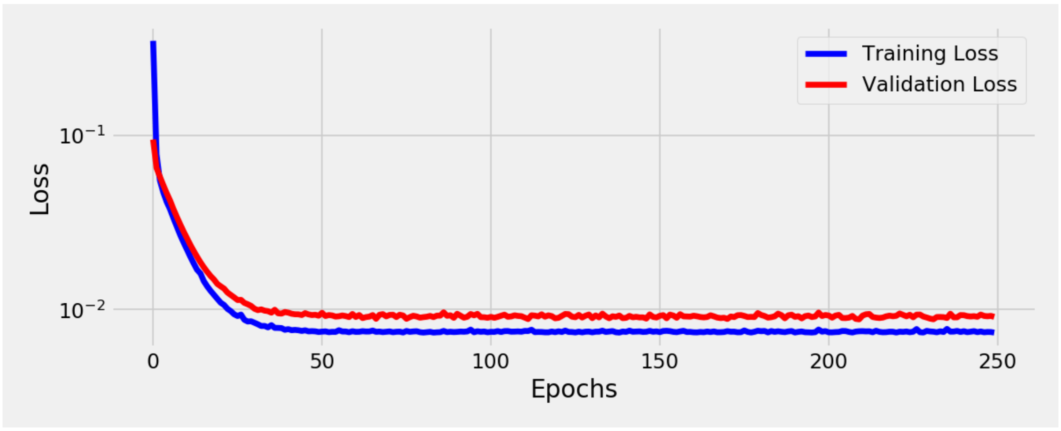


Figure 2.1.3 - More Losses!

We have loss values over 250 epochs now. The losses for the first 200 epochs were loaded from the checkpoint, and the losses for the last 50 epochs were computed after training was resumed. Once again, as in Chapter 2, the overall levels of the losses didn't change much.

If the losses haven't changed, it means the training loss was at a **minimum** already. So, we expect the **weights to remain unchanged**. Let's check it out:

```
print(sbs.model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9414]], device='cuda:0')),
('0.bias', tensor([1.0233], device='cuda:0'))])
```

No changes, indeed.

Putting It All Together

In this chapter, we have heavily modified the training pipeline. Even though the data preparation part was left *unchanged*, the model configuration part was reduced to its bare minimum, and the model training part was fully integrated into the `StepByStep` class. In other words, our pipeline went **classy** :-)

Run - Data Preparation V2

```
# %load data_preparation/v2.py

torch.manual_seed(13)

# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(x).float()
y_tensor = torch.as_tensor(y).float()

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train

train_data, val_data = random_split(dataset, [n_train, n_val])

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_data,
    batch_size=16,
    shuffle=True
)
val_loader = DataLoader(dataset=val_data, batch_size=16)
```

Run - Data Configuration V4

```
# %load model_configuration/v4.py

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model
model = nn.Sequential(nn.Linear(1, 1))

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

Run - Model Training

```
n_epochs = 200

sbs = StepByStep(model, loss_fn, optimizer)
sbs.set_loaders(train_loader, val_loader)
sbs.set_tensorboard('classy')
sbs.train(n_epochs=n_epochs)
```

```
print(model.state_dict())
```

Output

```
OrderedDict([('0.weight', tensor([[1.9414]], device='cuda:0')),
             ('0.bias', tensor([1.0233], device='cuda:0'))])
```

Recap

In this chapter, we've revisited and reimplemented many methods. This is what we've covered:

- defining our `StepByStep` **class**
- understanding the purpose of the **constructor** (`__init__`) method
- defining the **arguments** of the constructor method
- defining **class' attributes** to store *arguments*, *placeholders*, and *variables* we need to keep track of
- defining **functions as attributes**, using higher-order functions and the class' attributes to build functions that perform training and validation steps
- understanding the **difference** between **public**, **protected**, and **private methods**; and Python's "relaxed" approach to it
- creating methods to set **data loaders** and **TensorBoard** integration
- (re)implementing **training** methods: `_mini_batch` and `train`
- implementing **saving and loading** methods: `save_checkpoint` and `load_checkpoint`
- implementing a method for **making predictions** that takes care of all boilerplate code regarding *Numpy* to PyTorch conversion and back
- implementing methods to **plot losses** and add the **model's graph** to TensorBoard
- **instantiating** our `StepByStep` class and running a **classy** pipeline: configuring the model, loading the data, training the model, making predictions, checkpointing, and resuming training. The whole nine yards!

Congratulations! You have developed a **fully-functioning class** that implements all methods relevant to model training and evaluation. From now on, we'll use it over and over again to tackle different tasks and models. Next stop: classification!

[58] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter02.1.ipynb

[59] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter02.1.ipynb

[60] https://realpython.com/python3-object-oriented-programming/

[61] https://realpython.com/python-super/

[62] https://www.w3schools.com/python/ref_func_setattr.asp

[63] https://pytorch.org/docs/stable/notes/randomness.html

[64] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/stepbystep/v0.py

Chapter 3
A Simple Classification Problem

Spoilers

In this chapter, we will:

- build a model for **binary classification**
- understand the concept of **logits** and how it is related to **probabilities**
- use **binary cross-entropy loss** to train a model
- use the loss function to handle **imbalanced datasets**
- understand the concepts of **decision boundary** and **separability**
- learn how the **choice of a classification threshold** impacts evaluation metrics
- build **ROC** and **precision-recall** curves

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 3[65] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[66].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 3's notebook. If not, just click on `Chapter03.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.functional as F
from torch.utils.data import DataLoader, TensorDataset

from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_curve, \
precision_recall_curve, auc

from stepbystep.v0 import StepByStep
```

A Simple Classification Problem

It is time to handle a different **class** of problems: **classification problems** (pun intended). In a classification problem, we're trying to predict **which class a data point belongs to**.

Let's say we have **two classes** of points: they are either **red** or **blue**. These are the **labels (*y*)** of the points. Sure enough, we need to assign **numeric values** to them. We could assign **zero** to **red** and **one** to **blue**. The class associated with **zero** is the **negative class**, while **one** corresponds to the **positive class**.

In a nutshell, for **binary classification**, we have:

Color	Value	Class
Red	0	Negative
Blue	1	Positive

IMPORTANT: in a classification model, the **output** is the predicted probability of the **positive class**. In our case, the model will predict the *probability of a point being blue*.

The choice of *which class is positive* and *which class is negative* **does not** affect model performance. If we reverse the mapping, making *red the positive class*, the only difference would be that the model would predict the *probability of a point being red*. But, since **both probabilities have to add up to one**, we could easily convert between them, so the **models are equivalent**.

Instead of defining a model first and *then* generating synthetic data for it, we'll do it **the other way around**…

Data Generation

Let's make the data a *bit* more interesting by using **two features (x_1 and x_2)** this time. We'll use Scikit-Learn's make_moons to generate a **toy dataset with 100 data points**. We will also add some *Gaussian noise* and set a *random seed* to ensure reproducibility.

Data Generation

```
X, y = make_moons(n_samples=100, noise=0.3, random_state=0)
```

Then, we'll perform the **train-validation split** using Scikit-Learn's train_test_split for convenience (we'll get back to *splitting indexes* later) :

Train-validation split

```
X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=.2,
    random_state=13
)
```

Remember, the split should **always** be the **first thing** you do - no preprocessing, no transformations, **nothing happens before the split**.

Next, we'll **standardize the features** using Scikit-Learn's `StandardScaler`:

Feature Standardization

```
sc = StandardScaler()
sc.fit(X_train)

X_train = sc.transform(X_train)
X_val = sc.transform(X_val)
```

Remember, you should use **only the training set** to fit the `StandardScaler`, and then use its `transform` method to apply the preprocessing step to **all datasets**: training, validation, and test. Otherwise, you'll be **leaking** information from the validation and/or test sets to your model!

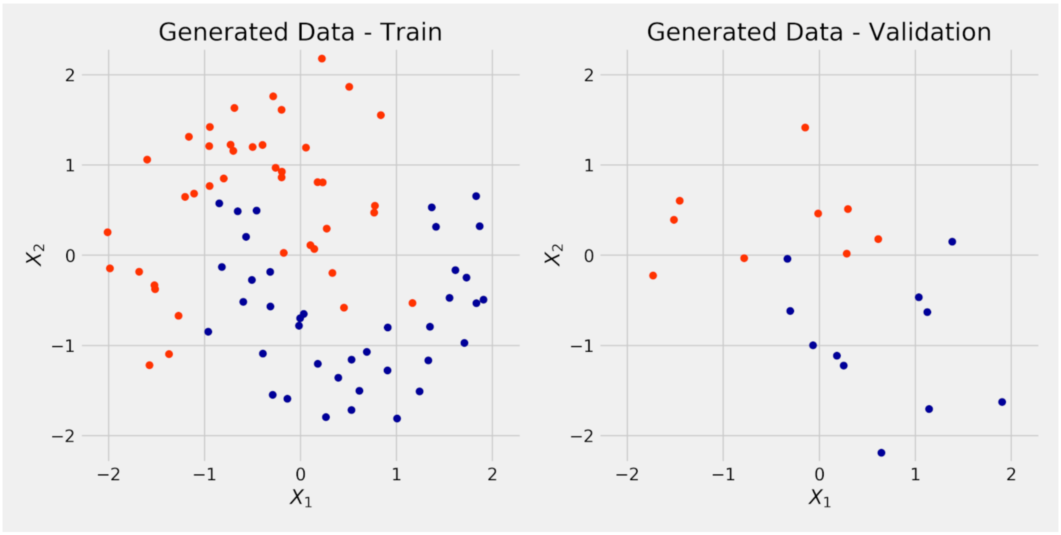


Figure 3.1 - Moons Dataset

Data Preparation

Hopefully, this step feels familiar to you already :-) As usual, the data preparation step converts *Numpy* arrays into PyTorch tensors, builds *TensorDatasets* for them, and creates the corresponding *data loaders*.

Data Preparation

```
torch.manual_seed(13)

# Builds tensors from numpy arrays
x_train_tensor = torch.as_tensor(X_train).float()
y_train_tensor = torch.as_tensor(y_train.reshape(-1, 1)).float()

x_val_tensor = torch.as_tensor(X_val).float()
y_val_tensor = torch.as_tensor(y_val.reshape(-1, 1)).float()

# Builds dataset containing ALL data points
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
val_dataset = TensorDataset(x_val_tensor, y_val_tensor)

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=16,
    shuffle=True
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

There are *80 data points* (*N* = 80) in our training set. We have **two features**, $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, and the **labels (*y*)** are either **zero (red)** or **one (blue)**. We have a dataset; now we need a…

Model

Given a **classification problem**, one of the more straightforward models is the **logistic regression**. But, instead of simply *presenting* it and using it right away, I am going to **build up to it**. The rationale behind this approach is twofold: first, it will make clear why this algorithm is called logistic *regression* if it is used for classification; second, you'll get a **clear understanding of what a *logit* is**.

Well, since it is called logistic **regression**, I would say that **linear regression** is a good starting point for us to build up to it. How would a linear regression model with two features look like?

$$y = b + w_1 x_1 + w_2 x_2 + \epsilon$$

Equation 3.1 - A Linear Regression model with two features

There is one obvious **problem** with the model above: our **labels (*y*)** are **discrete**; that is, they are either **zero** or **one**; no other value is allowed. We need to **change the model slightly** to adapt it to our purposes…

*"What if we assign the **positive** outputs to **one** and the **negative** outputs to **zero**?"*

Makes sense, right? We're already calling them **positive** and **negative** classes anyway; why not put their names to good use? Our model would look like this:

$$y = \begin{cases} 1, \ if \ \ b + w_1 x_1 + w_2 x_2 \geq 0 \\ 0, \ if \ \ b + w_1 x_1 + w_2 x_2 < 0 \end{cases}$$

Equation 3.2 - Mapping a Linear Regression model to **discrete labels**

Logits

To make our lives easier, let's give the right-hand side of the equation above a name: **logit (*z*)**.

$$z = b + w_1 x_1 + w_2 x_2$$

Equation 3.3 - Computing **logits**

The equation above is strikingly similar to the original **linear regression model**, but we're calling the resulting value ***z***, or **logit**, instead of ***y***, or **label**.

*"Does it mean a **logit** is the same as **linear regression**?"*

Not quite… there is one **fundamental difference** between them: there is **no error term (*epsilon*)** in Equation 3.3.

*"If there is no error term, where does the **uncertainty** come from?"*

I am glad you asked :-) That's the role of the **probability**: instead of assigning a data point to a **discrete label (zero or one)**, we'll compute the **probability of a data point belonging to the positive class**.

Probabilities

If a data point has a **logit** equals **zero**, it is exactly at the decision boundary since it is neither positive nor negative. For the sake of completeness, we assigned it to the **positive class**, but this assignment has **maximum uncertainty**, right? So, the corresponding **probability needs to be 0.5** (50%), since it could go either way…

Following this reasoning, we would like to have **large *positive* logit values** assigned to ***higher* probabilities** (of being in the positive class) and **large *negative* logit values** assigned to ***lower probabilities*** (of being in the positive class).

For *really large* positive and negative **logit values (z)**, we would like to have:

$$
\begin{aligned}
P(y=1) &\approx 1.0, \;\; if \;\; z \gg 0 \\
P(y=1) &= 0.5, \;\; if \;\; z = 0 \\
P(y=1) &\approx 0.0, \;\; if \;\; z \ll 0
\end{aligned}
$$

Equation 3.4 - Probabilities assigned to different logit values (z)

We still need to figure out a **function** that maps **logit values** into **probabilities**. We'll get there soon enough, but first, we need to talk about…

Odds Ratio

"What are the odds?!"

This is a colloquial expression meaning something very unlikely has happened. But **odds** do not have to refer to an unlikely event or a slim chance. The odds of getting **heads** in a (fair) coin flip are 1 to 1 since there is a 50% chance of success and a 50% chance of failure.

Let's imagine we are betting on the winner of the World Cup's final. There are two countries: **A** and **B**. Country **A** is the **favorite**: it has a 75% chance of winning. So, Country **B** has only a 25% chance of winning. If you bet on Country **A**, your chances of winning, that is, your **odds (in favor)** are **3 to 1** (75 to 25). If you decide to test your luck and bet on Country **B**, your chances of winning, that is, your **odds (in favor)** are **1 to 3** (25 to 75), or **0.33 to 1**.

The **odds ratio** is given by the **ratio** between the **probability of success** (*p*) and the **probability of failure** (*q*):

$$odds\ ratio(p) = \frac{p}{q} = \frac{p}{1-p}$$

Equation 3.5 - Odds ratio

In code, our `odds_ratio` function looks like this:

```python
def odds_ratio(prob):
    return prob / (1 - prob)

p = .75
q = 1 - p
odds_ratio(p), odds_ratio(q)
```

Output

```
(3.0, 0.3333333333333333)
```

We can also **plot** the resulting **odds ratios** for probabilities ranging from 1% to 99%. The *red dots* correspond to the probabilities of 25% (*q*), 50%, and 75% (*p*).

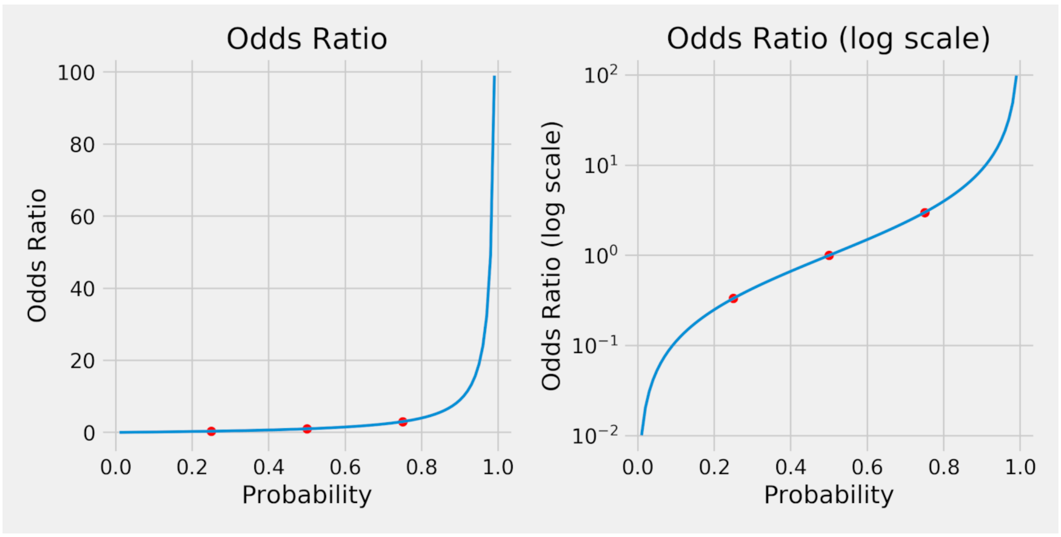


Figure 3.2 - Odds ratio

Clearly, the odds ratios (left plot) are **not symmetrical**. But, in a **log scale** (right plot), **they are**. This serves us very well since we're looking for a **symmetrical function** that maps **logit values** into **probabilities**.

"*Why does it **need** to be **symmetrical**?*"

If the function **weren't** symmetrical, different choices for the **positive class** would produce models that were **not** equivalent. But, using a symmetrical function, we could train **two equivalent models** using the **same dataset**, just flipping the classes:

- **Blue Model** (the positive class (**y=1**) corresponds to **blue** points)
- Data Point #1: **P(y=1) = P(blue) = .83** (which is the same as **P(red) = .17**)

- **Red Model** (the positive class (**y=1**) corresponds to **red** points)
- Data Point #1: **P(y=1) = P(red) = .17** (which is the same as **P(blue) = .83**)

Log Odds Ratio

By taking the **logarithm** of the **odds ratio**, the function is not only **symmetrical**, but it also maps **probabilities** into **real numbers**, instead of only the positive ones:

$$log\ odds\ ratio(p) = log\left(\frac{p}{1-p}\right)$$

Equation 3.6 - Log odds ratio

In code, our `log_odds_ratio` function looks like this:

```
def log_odds_ratio(prob):
    return np.log(odds_ratio(prob))

p = .75
q = 1 - p
log_odds_ratio(p), log_odds_ratio(q)
```

Output

```
(1.0986122886681098, -1.0986122886681098)
```

As expected, **probabilities that add up to 100%** (like 75% and 25%) correspond to **log odds ratios** that are the **same in absolute value**. Let's plot it:

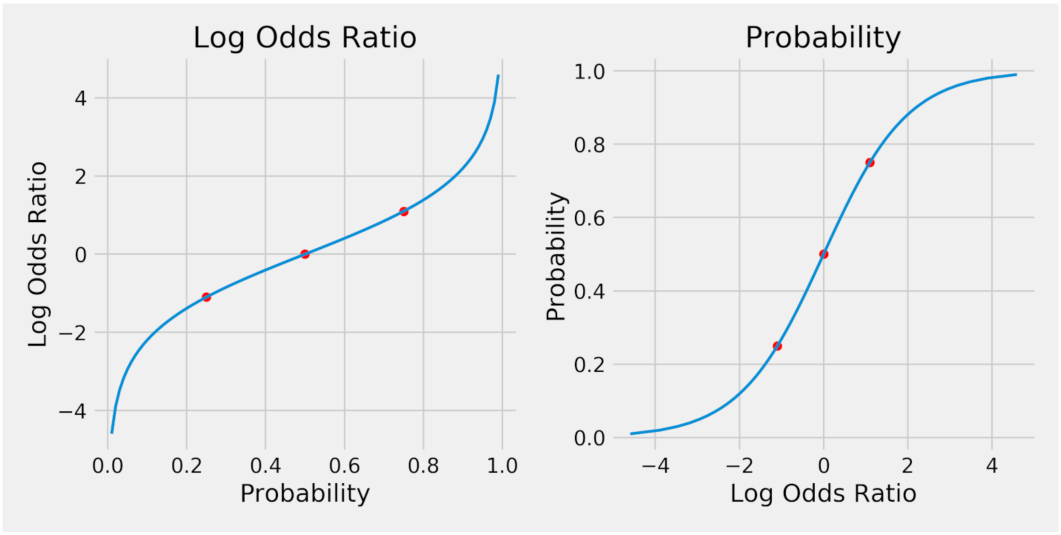


Figure 3.3 - Log odds ratio and probability

On the left, **each probability maps into a log odds ratio**. The *red dots* correspond to probabilities of 25%, 50%, and 75%, the same as before.

If we **flip** the horizontal and vertical axes (right plot), we are **inverting the function**, thus mapping **each log odds ratio into a probability**. That's the function we were looking for!

Does its shape look familiar? Wait for it…

From Logits to Probabilities

In the previous section, we were trying to **map logit values into probabilities**, and we've just found out, graphically, a function that **maps log odds ratios into probabilities**.

Clearly, our **logits are log odds ratios** :-) Sure, drawing conclusions like this is not very scientific, but the purpose of this exercise is to illustrate how the results of a regression, represented by the **logits (z)**, get to be mapped into probabilities.

So, that's where we arrived at:

$$
\begin{aligned}
b + w_1x_1 + w_2x_2 = z \quad &= log\left(\frac{p}{1-p}\right) \\
e^{b+w_1x_1+w_2x_2} = e^z \quad &= \frac{p}{1-p}
\end{aligned}
$$

Equation 3.7 - Regression, logits, and log odds ratios

Let's work this equation out a bit, inverting, rearranging, and simplifying some terms to **isolate p**:

$$
\begin{aligned}
\frac{1}{e^z} &= \frac{1-p}{p} \\
e^{-z} &= \frac{1}{p} - 1 \\
1 + e^{-z} &= \frac{1}{p} \\
p &= \frac{1}{1+e^{-z}}
\end{aligned}
$$

Equation 3.8 - From logits (z) to probabilities (p)

Does it look familiar? That's a **sigmoid function**! It is the **inverse of the log odds ratio**.

$$
p = \sigma(z) = \frac{1}{1+e^{-z}}
$$

Equation 3.9 - Sigmoid function

Sigmoid Function

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

p = .75
q = 1 - p
sigmoid(log_odds_ratio(p)), sigmoid(log_odds_ratio(q))
```

Output

```
(0.75, 0.25)
```

Sigmoid

There is no need to implement our own sigmoid function, though. PyTorch provides two different ways of using a **sigmoid**: `torch.sigmoid` and `nn.Sigmoid`.

The first one is a simple **function**, like the one above, but taking a tensor as input and returning another tensor:

```
torch.sigmoid(torch.tensor(1.0986)), torch.sigmoid(torch.tensor(
-1.0986))
```

Output

```
(tensor(0.7500), tensor(0.2500))
```

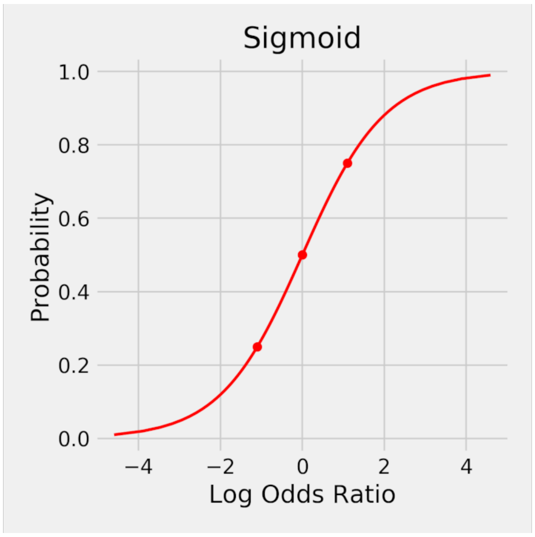


Figure 3.4 - Sigmoid function

The second one is a full-fledged **class** inherited from `nn.Module`: it is, for all intents and purposes, a **model on its own**. It is quite a simple and straightforward model: it **only** implements a `forward` method which, surprise, surprise, calls `torch.sigmoid`.

"*Why do you need a **model** for a **sigmoid** function?*"

Remember, models can be used as **layers** of another, larger model. That's exactly what we're going to do with the **sigmoid class**.

Sigmoid, nonlinearities, and activation functions

The sigmoid function is **nonlinear**. It can be used to map **logits** into **probabilities**, as we've just figured out. But this **is not** its only purpose!

Nonlinear functions play a fundamental role in neural networks. We know these nonlinearities by their usual name: **activation functions**.

The sigmoid is the "biologically-inspired" and the first activation function to be used back in the old days. It was followed by the hyperbolic-tangent (TanH) and, more recently, by the Rectified Linear Unit (ReLU) and a whole family of functions it spawned.

Moreover, there would be **no neural networks without a nonlinear function.** Have you ever wondered *what would happen to a neural network*, no matter how many layers deep, **if all its activation functions were removed**?

I will get back to this topic in the next chapter, but I will spoil the answer already: the network would be **equivalent to linear regression**. True story!

Logistic Regression

Given **two features**, x_1 and x_2, the model will fit a **linear regression** such that its outputs are **logits (*z*)**, which are converted into **probabilities** using a **sigmoid function**.

$$P(y=1) = \sigma(z) = \sigma(b + w_1 x_1 + w_2 x_2)$$

Equation 3.10 - Logistic Regression

A picture is worth a thousand words so, let's visualize it:

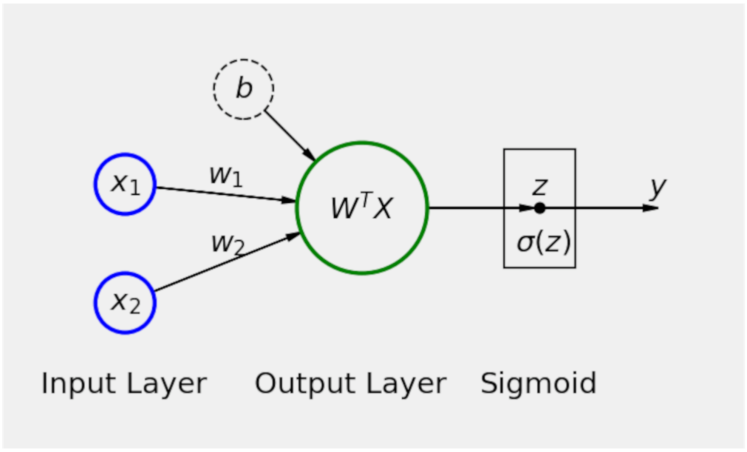


Figure 3.5 - The (second) simplest of all neural networks

We can think of the **logistic regression** as the **second simplest neural network possible**. It is pretty much the **same as the linear regression**, but with a **sigmoid** applied to the results of the output layer (*z*).

Now let's use the `Sequential` model to build our logistic regression in PyTorch:

```
torch.manual_seed(42)
model1 = nn.Sequential()
model1.add_module('linear', nn.Linear(2, 1))
model1.add_module('sigmoid', nn.Sigmoid())
print(model1.state_dict())
```

Output

```
OrderedDict([('linear.weight', tensor([[0.5406, 0.5869]])),
             ('linear.bias', tensor([-0.1657]))])
```

Did you notice that the `state_dict` contains parameters from the `linear` layer only? Even though the model has a second `sigmoid` layer, this layer does not contain any parameters since it does not need to learn anything: the sigmoid function will be the same regardless of which model it is a part of.

A Note on Notation

So far, we've handled either **one feature** (up to Chapter 2) or **two features** (this chapter). It allowed us to spell equations out, listing all terms.

But the number of features will soon increase *fast* when we tackle **images as inputs**. So we need to agree on notation for **vectorized features**. Actually, I've already used it in Figure 3.5 above.

The vectorized representations of the **weights (*W*)** and **features (*X*)** are:

$$W = \underset{(3\times1)}{\begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}}; X = \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}$$

I will always place the dimensions below the vectors to make it more clear.

The **logits (*z*)**, as shown in Figure 3.5, are given by the expression below:

$$z = W^T \cdot X = \underset{(1\times3)}{\begin{bmatrix} - & w^T & - \end{bmatrix}} \cdot \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}} = \underset{(1\times3)}{\begin{bmatrix} b & w_1 & w_2 \end{bmatrix}} \cdot \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}$$

$$= b + w_1x_1 + w_2x_2$$

From now on, instead of using the final and *long* expression, we'll use the first and more concise one.

Loss

We already have a model, and now we need to define an appropriate **loss** for it. A **binary classification** problem calls for the **binary cross-entropy (BCE) loss**, sometimes known as **log loss**.

The **BCE loss** requires the **predicted probabilities**, as returned by the **sigmoid function**, and the **true labels (*y*)** for its computation. For each data point ***i*** in the training set, it starts by computing the **error** corresponding to the point's **true class**.

If the data point belongs to the **positive class (*y*=1)**, we would like our model to **predict a probability close to one**, right? A **perfect one** would result in the **logarithm of one**, which is **zero**. It makes sense; **a perfect prediction means zero loss**. It goes like this:

$$y_i = 1 \Rightarrow error_i = log(P(y_i = 1))$$

Equation 3.11 - Error for a data point in the positive class

What if the data point belongs to the **negative class (*y*=0)**? Then we **cannot** simply use the predicted probability. Why not? Because the model outputs the probability of a point belonging to the *positive*, not the *negative*, class. Luckily, the latter can be easily computed:

$$P(y_i = 0) = 1 - P(y_i = 1)$$

Equation 3.12 - Probability of a data point belonging to the negative class

And thus, the **error** associated with a data point belonging to the **negative class** goes like this:

$$y_i = 0 \Rightarrow error_i = log(1 - P(y_i = 1))$$

Equation 3.13 - Error for a data point in the negative class

Once all errors are computed, they are **aggregated into a loss value**. For the binary-cross entropy loss, we simply take the **average of the errors** and **invert its sign**.

$$BCE(y) = -\frac{1}{(N_{pos} + N_{neg})}\left[\sum_{i=1}^{N_{pos}} log(P(y_i = 1)) + \sum_{i=1}^{N_{neg}} log(1 - P(y_i = 1))\right]$$

Equation 3.14 - Binary Cross-Entropy formula, the intuitive way

Let’s assume we have two dummy data points, one for each class. Then, let’s pretend our model made predictions for them: 0.9 and 0.2. The predictions are not bad since it predicts a 90% probability of being positive for an actual positive, and only 20% of being positive for an actual negative. How does this look like in code? Here it is:

```
dummy_labels = torch.tensor([1.0, 0.0])
dummy_predictions = torch.tensor([.9, .2])

# Positive class (labels == 1)
positive_pred = dummy_predictions[dummy_labels == 1]
first_summation = torch.log(positive_pred).sum()
# Negative class (labels == 0)
negative_pred = dummy_predictions[dummy_labels == 0]
second_summation = torch.log(1 - negative_pred).sum()
# n_total = n_pos + n_neg
n_total = dummy_labels.size(0)

loss = -(first_summation + second_summation) / n_total
loss
```

Output

```
tensor(0.1643)
```

The first summation adds up the errors corresponding to the points in the positive class. The second summation adds up the errors corresponding to the points in the negative class. I believe the **formula above** is quite **straightforward** and **easy to understand**. Unfortunately, it is usually *skipped* over, and only its equivalent is presented:

$$BCE(y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot log(P(y_i = 1)) + (1 - y_i) \cdot log(1 - P(y_i = 1))]$$

Equation 3.15 - Binary Cross-Entropy formula, the clever way

The formula above is a *clever way* of computing the loss in a single expression, sure, but the split of positive and negative points is less obvious. If you pause for a minute, you'll realize that points in the **positive class (*y*=*1*)** have its **second term**

equals zero, while points in the **negative class (*y=0*)** have its **first term equals zero**.

Let's see how it looks like in code:

```
summation = torch.sum(
    dummy_labels * torch.log(dummy_predictions) +
    (1 - dummy_labels) * torch.log(1 - dummy_predictions)
)
loss = -summation / n_total
loss
```

Output

```
tensor(0.1643)
```

Of course, we got the same loss (0.1643) as before.

For a *very* detailed explanation of the rationale behind this loss function, make sure to check my post: Understanding binary cross-entropy / log loss: a visual explanation[67].

BCELoss

Sure enough, PyTorch implements the *binary cross-entropy loss*, `nn.BCELoss`. Just like its regression counterpart, `MSELoss`, introduced in Chapter 1, it is a *higher-order function* that **returns the actual loss function**.

The **BCELoss** higher-order function takes two **optional** arguments (the others are deprecated, and you can safely ignore them):

- `reduction`: it takes either `mean`, `sum`, or `none`. The default **`mean`** corresponds to our **Equation 3.15** above. As expected, `sum` will return the sum of the errors, instead of the average. The last option, **`none`**, corresponds to the **unreduced** form, that is, it returns the full **array of errors**.

- `weight`: the default is `none`, meaning every data point has equal weight. If informed, it needs to be a tensor with size equals to the number of elements in a mini-batch, representing the weights assigned to each element in the batch. In other words, this argument allows you to assign different weights to each element of the current batch, based on its position. So, the **first element** would have a given weight, the **second element** would have a different weight, and so on… **regardless of the actual class of that particular data point**. Sounds confusing? Weird? Yes, this is weird, I think so too. Of course, this is not useless or a mistake, but the proper usage of this argument is a more advanced topic and outside the scope of this book.

This argument **DOES NOT help with weighting imbalanced datasets**! We'll see how to handle that shortly.

We'll be sticking with the default arguments, corresponding to Equation 3.15 above.

```
loss_fn = nn.BCELoss(reduction='mean')

loss_fn
```

Output

```
BCELoss()
```

As expected, `BCELoss` returned another function, that is, the actual loss function. The latter takes both predictions and labels to compute the loss.

IMPORTANT: make sure to pass the **predictions first** and then the **labels** to the loss function. The **order matters** in the implementation of this loss function differently from the mean squared error.

Let’s check this out:

```
dummy_labels = torch.tensor([1.0, 0.0])
dummy_predictions = torch.tensor([.9, .2])

# RIGHT
right_loss = loss_fn(dummy_predictions, dummy_labels)

# WRONG
wrong_loss = loss_fn(dummy_labels, dummy_predictions)

print(right_loss, wrong_loss)
```

Output

```
tensor(0.1643) tensor(15.0000)
```

Clearly, the order matters. It matters because the `BCELoss` takes the logarithm of the probabilities, which is expected as the **first argument**. If we swap the arguments, it will yield different results. In Chapter 1, we followed the same convention when using `MSELoss` - **first predictions**, **then labels** - even though it wouldn’t make any difference there.

So far, so good. But there is yet **another** binary cross-entropy loss available, and it is **very important** to know **when to use one or the other**, so you don’t end up with an inconsistent combination of model and loss function. Moreover, you’ll understand why I made such a fuss about the **logits**…

BCEWithLogitsLoss

The former loss function took probabilities as an argument (together with the labels, obviously). This loss function takes **logits** as an argument, instead of probabilities.

"What does that mean, in practical terms?"

It means you **should NOT add a sigmoid as the last layer of your model** when using this loss function. This loss combines both the **sigmoid layer and the former binary cross-entropy loss into one**.

IMPORTANT: I can't stress this enough: you **must** use the **right combination of model and loss function**.

Option 1: `nn.Sigmoid` as the **last** layer, meaning your model is producing **probabilities**, combined with `nn.BCELoss` function

Option 2: **no sigmoid** in the last layer, meaning your model is producing **logits**, combined with `nn.BCEWithLogitsLoss` function.

Mixing `nn.Sigmoid` and `nn.BCEWithLogitsLoss` is just **wrong**.

Besides, **Option 2 is preferred** since it is numerically more stable than Option 1.

Now that the difference in the arguments is clear, let's take a closer look at the `nn.BCEWithLogitsLoss` function. It is also a higher-order function, but it takes three **optional** arguments (the others are deprecated, and you can safely ignore them):

- `reduction`: it takes either `mean`, `sum`, or `none`, and it works just like in `nn.BCELoss`. The default is **`mean`**.
- `weight`: this argument also works just like in `nn.BCELoss`, and it is unlikely to be used.
- `pos_weight`: the weight of positive samples, it must be a tensor with length equals to the **number of labels associated with a data point** (the documentation refers to *classes*, instead of labels, which just makes everything even more confusing).

To make it clear: in this chapter, we're dealing with a **single label binary classification** (we have only **one label** per data point), and the **label is binary** (there are only two possible values for it, zero or one). If the label is **zero**, we say it belongs to the **negative class**. If the label is **one**, it belongs to the **positive class**.

Please do not confuse the positive and negative classes of our single label with ***c***, the so-called **class number** in the documentation. That ***c*** corresponds to the **number of different labels associated with a data point**. In our example, ***c* = 1**.

You *can* use this argument to handle **imbalanced datasets**, but there's more to it than it meets the eye... we'll get back to it in the next sub-section.

Enough talking (or writing!)... let's see how to use this loss in code. We start by creating the loss function itself:

```
loss_fn_logits = nn.BCEWithLogitsLoss(reduction='mean')

loss_fn_logits
```

Output

```
BCEWithLogitsLoss()
```

Next, we use **logits** and **labels** to compute the loss. Following the same principle as before, **logits first**, **then labels**. To keep the example consistent, let's get the values of the logits corresponding to the probabilities we used before, 0.9 and 0.2, using our `log_odds_ratio` function:

```
logit1 = log_odds_ratio(.9)
logit2 = log_odds_ratio(.2)

dummy_labels = torch.tensor([1.0, 0.0])
dummy_logits = torch.tensor([logit1, logit2])

print(dummy_logits)
```

Output

```
tensor([ 2.1972, -1.3863])
```

We have logits, and we have labels. Time to compute the loss:

```
loss = loss_fn_logits(dummy_logits, dummy_labels)
loss
```

Output

```
tensor(0.1643)
```

OK, we got the same result, as expected.

Imbalanced Dataset

In our dummy example with two data points, we had one of each class: positive and negative. The dataset was perfectly balanced. Let's create another dummy example with an imbalance, adding **two extra data points belonging to the negative class**. For the sake of simplicity and to illustrate a *quirk* in the behavior of `BCEWithLogitsLoss`, I will give those two extra points the **same logits** as the other data point in the negative class. It looks like this:

```
dummy_imb_labels = torch.tensor([1.0, 0.0, 0.0, 0.0])
dummy_imb_logits = torch.tensor([logit1, logit2, logit2, logit2])
```

Clearly, this is an **imbalanced dataset**. There are **three times more** data points in the negative class than in the positive one. Now, let's turn to the `pos_weight` argument of `BCEWithLogitsLoss`. To compensate for the imbalance, one can set the weight equals the ratio of negative to positive examples:

$$pos\_weight = \frac{\#points\ in\ negative\ class}{\#points\ in\ positive\ class}$$

In our imbalanced dummy example, the result would be 3.0. This way, every point in the *positive class* would have its corresponding **loss multiplied by three**. Since there is a **single label** for each data point ($c = 1$), the **tensor used as an argument for `pos_weight`** has only **one element**: `tensor([3.0])`. We could compute it like this:

```
n_neg = (dummy_imb_labels == 0).sum().float()
n_pos = (dummy_imb_labels == 1).sum().float()

pos_weight = (n_neg / n_pos).view(1,)
pos_weight
```

Output

```
tensor([3])
```

Now, let's create yet another loss function, including the `pos_weight` argument this time:

```
loss_fn_imb = nn.BCEWithLogitsLoss(
    reduction='mean',
    pos_weight=pos_weight
)
```

Then, we can use this **weighted** loss function to compute the loss for our **imbalanced dataset**. I guess one would expect the **same loss** as before; after all, this is a *weighted* loss. Right?

```
loss = loss_fn_imb(dummy_imb_logits, dummy_imb_labels)
loss
```

Output

```
tensor(0.2464)
```

Wrong! It was 0.1643 when we had two data points, one of each class. Now it is 0.2464, **even though we assigned a weight** to the positive class.

"Why is it different?"

Well, it turns out, PyTorch **does not compute a weighted average**. That's what you would expect from a weighted average:

$$weighted\ average = \frac{pos\_weight \cdot \sum_{i=1}^{N_{pos}} loss_i + \sum_{i=1}^{N_{neg}} loss_i}{pos\_weight \cdot N_{pos} + N_{neg}}$$

Equation 3.16 - Weighted average of losses

But this is what PyTorch does:

$$BCEWithLogitsLoss = \frac{pos\_weight \cdot \sum_{i=1}^{N_{pos}} loss_i + \sum_{i=1}^{N_{neg}} loss_i}{N_{pos} + N_{neg}}$$

Equation 3.17 - PyTorch's BCEWithLogitsLoss

See the difference in the denominator? Of course, if you **multiply the losses** of the positive examples **without multiplying their count (*N*)**, you'll end up with a number **larger than an actual weighted average**.

*"What if I **really** want the weighted average?"*

It is not that hard, to be honest. Remember the `reduction` argument? If we set it to `sum`, our loss function will only return the **numerator** of the equation above. And then we can divide it by the weighted counts ourselves:

```
loss_fn_imb_sum = nn.BCEWithLogitsLoss(
    reduction='sum',
    pos_weight=pos_weight
)

loss = loss_fn_imb_sum(dummy_imb_logits, dummy_imb_labels)

loss = loss / (pos_weight * n_pos + n_neg)
loss
```

Output

```
tensor([0.1643])
```

There we go!

Model Configuration

In Chapter 2.1, we ended up with a *lean* **Model Configuration** part: we only need to define a **model**, an appropriate **loss function**, and an **optimizer**. Let's define a model that **produces logits** and use `BCEWithLogitsLoss` as the loss function. Since we have **two features**, and we are producing *logits* instead of probabilities, our model has **one layer** and one layer alone: `Linear(2, 1)`. We will keep using the SGD optimizer with a learning rate of 0.1 for now.

This is what the model configuration looks like for our classification problem:

Model Configuration

```python
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(2, 1))

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a BCE with logits loss function
loss_fn = nn.BCEWithLogitsLoss()
```

Model Training

Time to **train** our model! We can leverage the `StepByStep` class we built in Chapter 2.1 and use pretty much the same code as before:

Model Training

```
n_epochs = 100

sbs = StepByStep(model, loss_fn, optimizer)
sbs.set_loaders(train_loader, val_loader)
sbs.train(n_epochs)
```

```
fig = sbs.plot_losses()
```

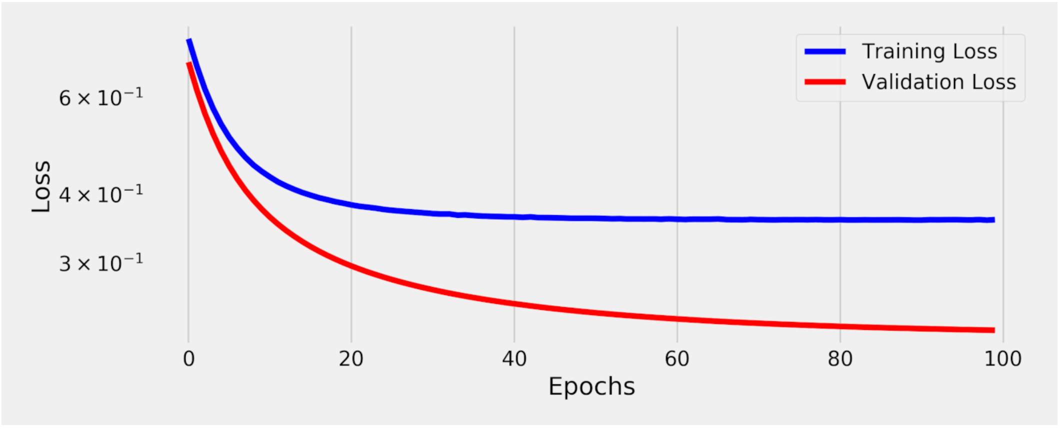


Figure 3.6 - Training and validation Losses

"*Wait, there is something weird with this plot…*" you say.

You're right; the **validation loss** is **smaller** than the **training loss**. Shouldn't it be the other way around?! Well, generally speaking, *YES*, it should… but you can learn more about situations where this *swap* happens at this great post[68]. In our case, it was simply the case that the **validation set is easier** to classify: if you check Figure 3.1 at the beginning of the chapter, you'll notice that the red and blue points in the right plot (validation) are not so mixed up as the ones in the left plot (training).

Having cleared that, it is time to inspect the model's trained parameters:

```
print(model.state_dict())
```

Output

```
OrderedDict([('linear.weight', tensor([[ 1.1822, -1.8684]], device
='cuda:0')),
             ('linear.bias', tensor([-0.0587], device='cuda:0'))])
```

Our model produced **logits**, right? So we can plug the weights above in the corresponding **logit equation** (Equation 3.3), and we end up with:

$$\begin{aligned} z &= b + w_1x_1 + w_2x_2 \\ z &= -0.0587 + 1.1822x_1 - 1.8684x_2 \end{aligned}$$

Equation 3.18 - Model's output

The value ***z*** above is the **output of our model**. It is a "*glorified* linear regression"! And this is a classification problem! How come?! Hold that thought; it will become more clear in the next section, *Decision Boundary*.

But, before going down that road, I would like to use our model (and the `StepByStep` class) to **make predictions** for, say, the first four data points in our training set:

Making Predictions (Logits)

```
predictions = sbs.predict(x_train_tensor[:4])
predictions
```

Output

```
array([[ 0.20252657],
       [ 2.944347  ],
       [ 3.6948545 ],
       [-1.2356305 ]], dtype=float32)
```

Clearly, these are not probabilities, right? These are **logits**, as expected.

We can still get the corresponding probabilities, though.

"*How do we go from logits to probabilities*", you ask, just to make sure you got it right.

That's what the **sigmoid function** is good for.

Making Predictions (Probabilities)

```
probabilities = sigmoid(predictions)
probabilities
```

Output

```
array([[0.5504593 ],
       [0.94999564],
       [0.9757515 ],
       [0.22519748]], dtype=float32)
```

Now we're talking! These are the **probabilities**, given our model, of those four points being positive examples.

Lastly, we need to go from probabilities to classes. If the **probability is greater than or equal to a threshold**, it is a **positive** example. If it is **less than the threshold**, it is a **negative** example. Simple enough. The trivial choice of a **threshold** is **0.5**:

$$y = \begin{cases} 1, \ if \ P(y=1) \geq 0.5 \\ 0, \ if \ P(y=1) < 0.5 \end{cases}$$

Equation 3.19 - From probabilities to classes

But the probability itself is just the **sigmoid** function applied to the **logit (*z*)**:

$$y = \begin{cases} 1, \ if \ \sigma(z) \geq 0.5 \\ 0, \ if \ \sigma(z) < 0.5 \end{cases}$$

Equation 3.20 - From logits to classes, via sigmoid function.

But the **sigmoid** function has a value of **0.5** only when the **logit (*z*)** has a value of **zero**:

$$y = \begin{cases} 1, \ if \ z \geq 0 \\ 0, \ if \ z < 0 \end{cases}$$

Equation 3.21 - From logits to classes, directly

Thus, if we don't care about the probabilities, we could use the **predictions (logits)** directly to get the **predicted classes** for the data points:

Making Predictions (Classes)

```
classes = (predictions >= 0).astype(np.int)
classes
```

Output

```
array([[1],
       [1],
       [1],
       [0]])
```

Clearly, the points where the **logits (*z*) equal zero** determine the **boundary** between **positive** and **negative** examples.

*"Why 0.5? Can I choose a **different threshold**?"*

Sure, you can! **Different thresholds** will give you **different confusion matrices** and, therefore, **different metrics**, like accuracy, precision, and recall. We'll get back to that in the "*Decision Boundary*" section.

By the way, are you still *holding that thought* about the "*glorified linear regression*"? Good!

Decision Boundary

We have just figured out that whenever ***z* equals zero**, we are in the **decision boundary**. But ***z*** is given by a **linear combination** of features ***x*₁** and ***x*₂**. If we work out some basic operations, we arrive at:

$$
\begin{aligned}
z = 0 &= b + w_1 x_1 + w_2 x_2 \\
-w_2 x_2 &= b + w_1 x_1 \\
x_2 &= -\frac{b}{w_2} - \frac{w_1}{w_2} x_1
\end{aligned}
$$

Equation 3.22 - Decision boundary for logistic regression with two features

Given our model (***b***, ***w*₁**, and ***w*₂**), for any value of the first feature (***x*₁**), we can compute the corresponding value of the second feature (***x*₂**) that sits **exactly at the**

decision boundary.

Look at the expression above: this is a **straight line**. It means the **decision boundary is a straight line**.

Let's plug the **weights** of our **trained model** in it:

$$\begin{aligned} x_2 &= -\frac{0.0587}{1.8684} + \frac{1.1822}{1.8684}x_1 \\ x_2 &= -0.0314 + 0.6327x_1 \end{aligned}$$

An image is worth a thousand words, right? Let's plot it!

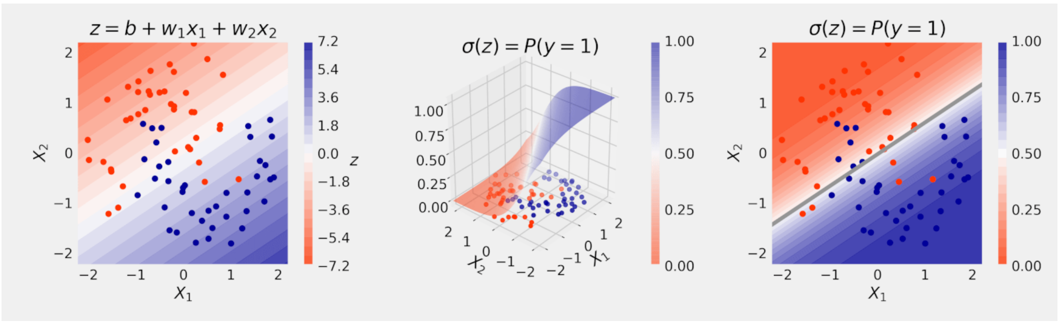


Figure 3.7 - Decision boundary

The figure above tells the whole story! It contains only **data points** in the **training set**. So, that's what the model "sees" when it is training. It will try to achieve the **best possible separation** between the two classes, depicted as **red** (negative class) and **blue** (positive class) points.

In the left plot, we have a **contour plot** (remember those from the loss surfaces in Chapter 0?) of the **logits (z)**.

In the center plot, we have a 3D plot of the **probabilities** resulting from **applying a sigmoid function** to the logits. You can even see the **shape** of the sigmoid function in 3D, approaching zero to the left and one to the right.

Finally, in the right plot, we have a **contour plot** of the **probabilities**, so it is the same as the center plot but without the cool 3D effect. Maybe it is not as cool, but it is surely easier to understand what's going on. **Darker blue (red) colors** mean **higher (lower) probabilities**, and we have the **decision boundary as a straight gray line**, corresponding to a **probability of 50%** (and a logit value of zero).

A logistic regression always separates two classes with a straight line.

Our model produced a straight line that does quite a good job separating red and blue points, right? Well, it was not *that* hard anyway, since the blue points were more concentrated on the bottom right corner, while the red points were mostly on the top left corner. In other words, the classes were quite **separable**.

The more **separable** the **classes** are, the **lower** the **loss** will be.

Now we can make sense of the **validation loss** being **lower** than the training loss. In the *validation set*, the classes are **more separable** than in the *training set*. The **decision boundary** obtained using the training set can do an *even better* job separating red and blue points. Let's check it out, plotting the **validation set** against the **same contour plots** as above:

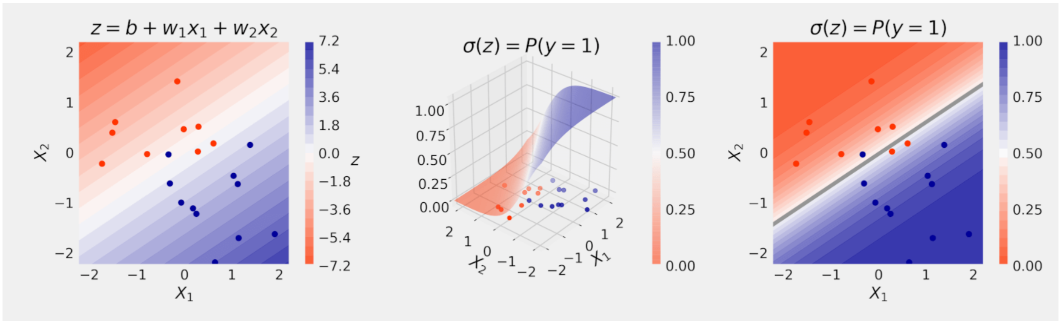


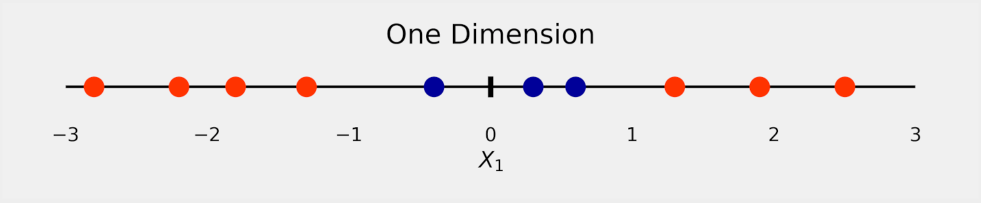
Figure 3.8 - Decision boundary (validation dataset)

See? Apart from three points, two red and one blue, which are *really* close to the decision boundary, the data points are correctly classified. **More separable indeed**.

Are my data points separable?

That's the million-dollar question! In the example above, we can clearly see that data points in the validation set are **more separable** than those in the training set.

What happens if the points are **not separable at all**? Let's take a quick detour and look at another tiny dataset with 10 data points, seven red, three blue. The colors are the **labels (*y*)**, and each data point has a **single feature (*x_1*)**. We could plot them **along a line**; after all, we have only **one dimension**.



Can you **separate the blue points from the red ones with one straight line**? Obviously not… these points **are not separable** (in one dimension, that is).

Should we give up, then?

"*Never give up, never surrender!*"

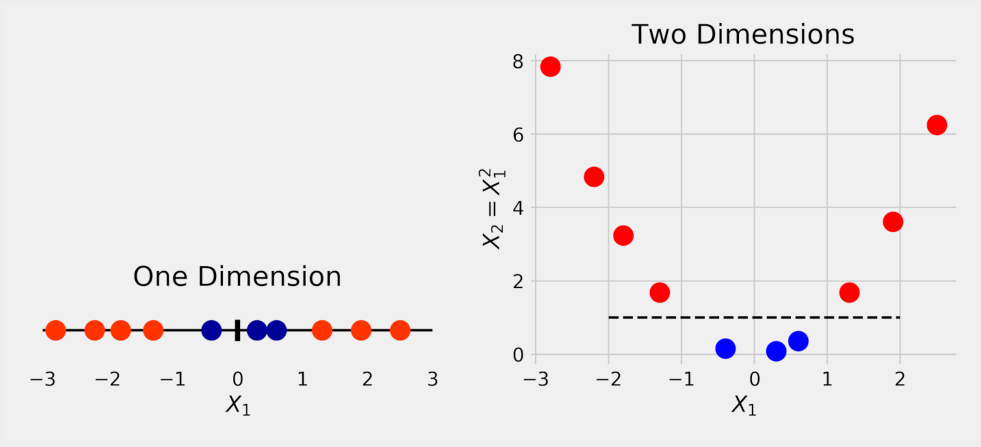
Commander Taggart

If it doesn't work in one dimension, try using two! There is just one problem, though... where do the other dimension come from? We can use a **trick** here: we apply a **function** to the **original dimension (feature)** and use the result as a **second dimension (feature)**. Quite simple, right?

For the tiny dataset at hand, we could try the **square function**:

$$X_2 = f(X_1) = X_1^2$$

How does it look like?



Back to the original question: "*can you separate the blue points from the red ones with one straight line?*"

In two dimensions, that's a piece of cake!

The more dimensions, the more separable the points are.

It is beyond the scope of this book to explain *why* this trick works. The important thing is to **understand the general idea** here: as the **number of dimensions increases**, there is **more and more empty space**. If the data points are farther apart, it is likely easier to separate them. In two dimensions, the decision boundary is a line. In three dimensions, it is a plane. In four dimensions and more, it is a hyper-plane (fancier wording for a plane you can't draw).

Have you heard of the **kernel trick** for Support Vector Machines (SVMs)? That's pretty much what it does! The **kernel** is nothing else but the **function** we used to create additional dimensions. The square function we used is a **polynomial**, so we used a **polynomial kernel**.

"*Why are we talking about SVMs in a Deep Learning book?*"

Excellent question! It turns out; **neural networks** may also **increase the dimensionality**. That's what happens if you add a **hidden layer** with **more units** than the **number of features**. For instance:

```
model = nn.Sequential()
model.add_module('hidden', nn.Linear(2, 10))
model.add_module('activation', nn.ReLU())
model.add_module('output', nn.Linear(10, 1))
model.add_module('sigmoid', nn.Sigmoid())

loss_fn = nn.BCELoss()
```

The model above increases dimensionality **from two dimensions** (two features) to **ten dimensions** and then uses those **ten dimensions to compute logits**. But it **only works if there is an activation function between the layers**.

I suppose you may have two questions right now: "why is that?", and "what actually is an activation function?". Fair enough. But these are topics for the next chapter.

Classification Threshold

This section is **optional**. In it, I will dive deeper into using different thresholds for classification and how it affects the confusion matrix. I will explain the most common classification metrics: true and false positive rates, precision and recall, and accuracy. Finally, I will show you how these metrics can be combined to build ROC and Precision-Recall curves.

If you are already comfortable with these concepts, feel free to *skip* this section.

So far, we've been using the trivial threshold of 50% to classify our data points, given the probabilities predicted by our model. Let's dive a bit deeper into this and see the **effects of choosing different thresholds**. We'll be working on the data points in the **validation set**. There are only **20 data points** in it, so we can easily **keep track of all of them**.

First, let's compute the logits and corresponding probabilities:

Evaluation

```
logits_val = sbs.predict(X_val)
probabilities_val = sigmoid(logits_val).squeeze()
```

Then, let's **visualize the probabilities on a line**. It means we're going from the fancy contour plot to a **simpler plot**:

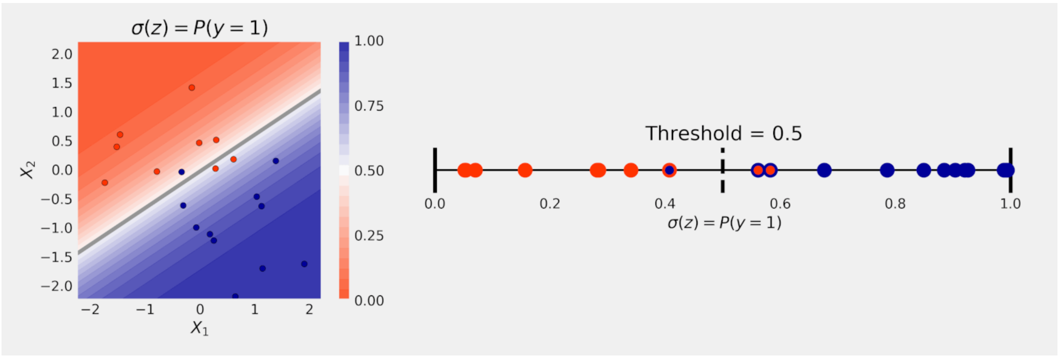


Figure 3.9 - Probabilities on a line

The left plot comes from Figure 3.8. It shows the **contour plot of the probabilities** and the **decision boundary as a straight gray line**. We place the data points **on a line**, according to their **predicted probabilities**. That's the plot on the right.

The **decision boundary** is shown as a **vertical dashed line** placed at the **chosen threshold** (0.5). Points to the **left** of the dashed line are **classified as red**, and therefore have **red edges around them**, while those to the **right** are **classified as blue**, having **blue edges around them**.

The points are **filled with their actual color**, meaning that those having **distinct colors for edge and filling** are **misclassified.** In the figure above, we have **one blue point classified as red (left)** and **two red points classified as blue (right)**.

Now, let's make a *tiny* change to our plot to make it **more visually interesting**: we'll plot **blue (positive) points below the probability line** and **red (negative) points above the probability line**. It looks like this:

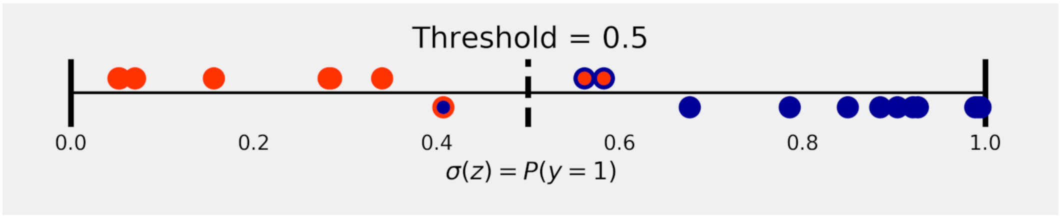


Figure 3.10 - Split probability line

"*Why is it more visually interesting?*" you ask.

Well, now **all correctly classified** and **all misclassified** points are in **different quadrants**. There is something else that **looks exactly like this**…

Confusion Matrix

Those quadrants have names: **true negative (TN)** and **false positive (FP)**, above the line, **false negative (FN)** and **true positive (TP)**, below the line.

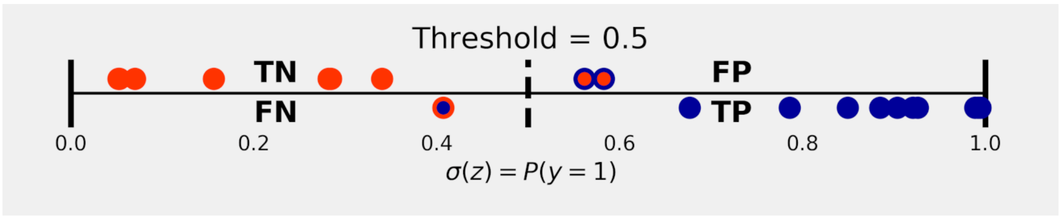


Figure 3.11 - Probability line as a confusion matrix

Points **above the line** are **actual negatives**, points **below the line** are **actual positives**.

Points to the **right of the threshold** are **classified as positive**, points to the **left of**

the threshold are **classified as negative**.

Cool, right? Let's double-check it with Scikit-Learn's `confusion_matrix` method:

```
cm_thresh50 = confusion_matrix(y_val, (probabilities_val >= 0.5))
cm_thresh50
```

Output

```
array([[ 7,  2],
       [ 1, 10]])
```

All 20 points in our validation set are accounted for. There are **three misclassified points**: one false negative and two false positives, just like in the figure above. I chose to move the **blue points (positive) below** the line to **match Scikit-Learn's convention for the confusion matrix**.

Confusion matrices are already **confusing enough** on their own, but what's even **worse** is that you'll find all sorts of layouts around. Some people list positives first and negatives last. Some people even *flip* actuals and predicted classes, effectively transposing the confusion matrix. Make sure to always **check the layout** before drawing conclusions from matrices you see "in the wild".

To make your life, and mine, simpler, I am just sticking with Scikit-Learn's convention throughout this book.

There is one more thing I hope you noticed already: **the confusion matrix depends on the threshold**. If you **shift the threshold** along the probability line, you'll end up **changing the number of points in each quadrant**.

There are **many confusion matrices, one for each threshold**.

Moreover, **different confusion matrices** mean **different metrics**. We need the individual components of the confusion matrix, namely, TN, FP, FN, and TP, to construct those metrics. The function below *splits* the confusion matrix accordingly:

True and False Positives and Negatives

```python
def split_cm(cm):
    # Actual negatives go in the top row,
    # above the probability line
    actual_negative = cm[0]
    # Predicted negatives go in the first column
    tn = actual_negative[0]
    # Predicted positives go in the second column
    fp = actual_negative[1]

    # Actual positives go in the bottow row,
    # below the probability line
    actual_positive = cm[1]
    # Predicted negatives go in the first column
    fn = actual_positive[0]
    # Predicted positives go in the second column
    tp = actual_positive[1]

    return tn, fp, fn, tp
```

Metrics

Starting with these four numbers, TN, FP, FN, and TP, you may construct **a ton of metrics**. We're focusing here on the most commonly used: **True and False Positive Rates** (TPR and FPR), **precision**, **recall**, and **accuracy**.

True and False Positive Rates

Let's start with the first two:

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

For both of them, you **divide one value on the right column (positive)** by the **sum of the corresponding row**. So, the **true positive rate** is computed by dividing the value on the **bottom right** by the sum of the **bottom row**. Similarly, the **false positive rate** is computed by dividing the value on the **top right** by the sum of the **top row**. Fine, but what do they mean?

The **true positive rate** tells you, from **all points you *know* to be positive, how many your model got right**. In our example, we **know** there are **11 positive** examples. Our model **got ten right**. The **TPR** is 10 out of 11, or roughly 91%. There is yet another name for this metric: **recall**. Makes sense, right? From all the positive examples, how many does your model **recall**?

If **false negatives** are bad for your application, you need to **focus on improving the TPR/recall** metric of your model.

When is a false negative *really* bad? Take airport security screening, for example, where **positive means the existence of a threat**. False positives are common: you have nothing to hide, and still, your bag will eventually be more thoroughly inspected due to the extreme sensitivity of the machinery. A **false negative** means that the machine **failed to detect an actual threat**. I don't have to explain why this is **bad**.

The **false positive rate** tells you, from **all points you *know* to be negative, how many your model got wrong**. In our example, we **know** there are **nine negative** examples. Our model **got two wrong**. The **FPR** is 2 out of 9, or roughly 22%.

If **false positives** are bad for your application, you need to **focus on reducing the FPR** metric of your model.

When is a false positive *really* bad? Take an investment decision, for example, where **positive means a profitable investment**. False negatives are missed opportunities: they seemed like bad investments, but they weren't. You did not make a profit, but you didn't sustain any losses either. A **false positive** means that you chose to invest but ended up **losing your money**.

We can use the function below to compute both metrics, given a confusion matrix:

True and False Positive Rates

```
def tpr_fpr(cm):
    tn, fp, fn, tp = split_cm(cm)

    tpr = tp / (tp + fn)
    fpr = fp / (fp + tn)

    return tpr, fpr
```

```
tpr_fpr(cm_thresh50)
```

Output

```
(0.9090909090909091, 0.2222222222222222)
```

The trade-off between TPR and FPR

As always, there is a *trade-off* between the two metrics.

Let's say **false negatives** are bad for our application, and we want to **improve TPR**. Here is one quick idea: let's make a model that **only predicts the positive class**, using a **threshold of zero**. We get **no false negatives whatsoever** (because there aren't any negatives in the first place). Our **TPR is 100%**. Awesome, right?

Wrong! If all points are predicted to be positive, **every negative example will be a false positive, and there are no true negatives**. Our **FPR is 100% too**.

There is no free lunch: the model is useless.

What if **false positives** are the problem instead? We would like to **reduce FPR**. Another *brilliant* idea comes to mind: let's make a model that **only predicts the negative class**, using a **threshold of one**. We get **no false positives whatsoever** (because there aren't any positives in the first place). Our **FPR is 0%**. Mission accomplished, right?

Guess what? Wrong again! If all points are predicted to be negative, **every positive example will be a false negative, and there are no true positives**. Our **TPR is 0% too**.

It turns out; you cannot have the cake and eat it too.

Precision and Recall

Moving on to the next pair of metrics, we have:

$$Recall = \frac{TP}{TP + FN} \quad Precision = \frac{TP}{TP + FP}$$

We can *skip* the **recall** because, as I mentioned above, it is the **same as TPR**: from all the positive examples, how many does your model **recall**?

What about **precision**? We compute it in the **right column (positive) only**. We divide the value on the **bottom right** by the sum of the **right column**. Its meaning is somewhat complementary to that of recall: from **all points classified as positive by *your model*, how many your model got right**. In our example, the **model classified 12 points as positive**. The model **got ten right**. The **precision** is 10 out of 12, or roughly 83%.

If **false positives** are bad for your application, you need to **focus on improving the precision** metric of your model.

We can use the function below to compute both metrics, given a confusion matrix:

Precision and Recall

```
def precision_recall(cm):
    tn, fp, fn, tp = split_cm(cm)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)

    return precision, recall
```

```
precision_recall(cm_thresh50)
```

Output

```
(0.8333333333333334, 0.9090909090909091)
```

The trade-off between Precision and Recall

Here, too, there is no free lunch. The trade-off is a bit different, though.

Let's say **false negatives** are bad for our application, and we want to **improve recall**. Once again, let's make a model that **only predicts the positive class**, using a **threshold of zero**. We get **no false negatives whatsoever** (because there aren't any negatives in the first place). Our **recall is 100%**. Now you're probably waiting for the bad news, right?

If all points are predicted to be positive, **every negative example will be a false positive**. The **precision** is exactly the **proportion of positive samples in the dataset**.

What if **false positives** are the problem instead? We would like to **increase precision**. It's time to make a model that **only predicts the negative class** by using a **threshold of one**. We get **no false positives whatsoever** (because there aren't any positives in the first place). Our **precision is 100%**.

Of course, this is too good to be true… If all points are predicted to be negative, **there are no true positives**. Our **recall is 0%**.

No free lunch, no cake, just another couple of useless models.

There is one metric left to explore.

Accuracy

This is the simplest and most intuitive of them all: how many times your model got it right, considering all data points. Totally straightforward!

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In our example, the model got 17 points right out of a total of 20 data points. Its accuracy is 85%. Not bad, right? The higher the accuracy, the better, but it does not tell the whole story. If you have an imbalanced dataset, relying on accuracy can be misleading.

Let's say we have 1,000 data points: 990 points are negative, and only ten are positive. Now, let's take that model that uses a threshold of one and **only predicts the negative class**. This way, we get **all 990 negative points right** at the cost of **ten false negatives**. This model's accuracy is **99%**. But the model is still useless because it will *never* get a positive example right.

Accuracy may be misleading because it does not involve a trade-off with another metric, like the previous ones.

Talking about trade-offs…

Trade-offs and Curves

We already know there are trade-offs between true and false positive rates, as well as between precision and recall. We also know that there are many confusion matrices, one for each threshold. What if we combine these two pieces of information? I present to you the **Receiver Operating Characteristic (ROC)** and **Precision-Recall (PR)** curves! Well, they are not curves *yet*, but they will be soon enough :-)

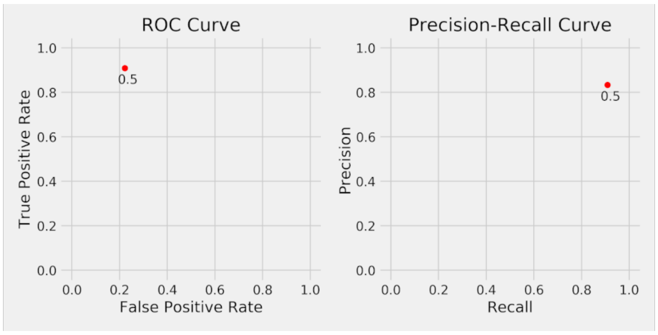


Figure 3.12 - Trade-offs for a threshold of 50%

We've already computed TPR/recall (91%), FPR (22%), and precision (83%) for our model using the threshold of 50%. If we plot them, we'll get the figure above.

Time to try **different thresholds**.

Low Threshold

What about 30%? If the predicted probability is greater than or equal to 30%, we classify the data point as positive, and as negative otherwise. That's a very **loose** threshold since we don't require the model to be very confident to consider a data point to be positive. What can we expect from it? **More false positives, less false negatives**.

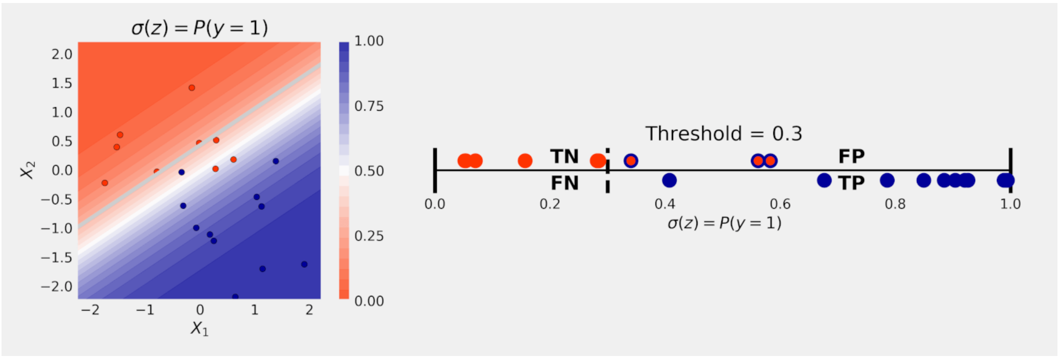


Figure 3.13 - Using a low threshold

You can see in the figure above that **lowering the threshold** (moving it to the left on the probability line) **turned one false negative into a true positive** (blue point close to 0.4), but it also **turned one true negative into a false positive** (red point close to 0.4).

Let's double-check it with Scikit-learn's confusion matrix:

```
confusion_matrix(y_val, (probabilities_val >= 0.3))
```

Output

```
array([[ 6,  3],
       [ 0, 11]])
```

OK, now let’s plot the corresponding metrics one more time:

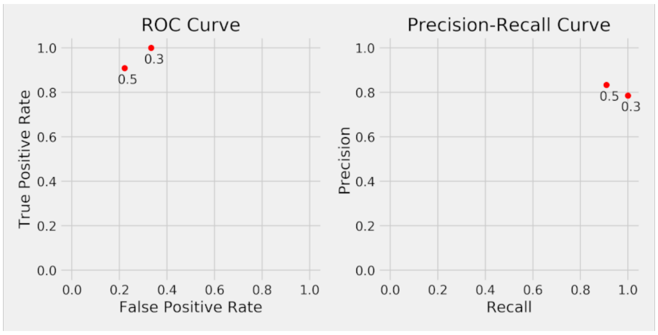


Figure 3.14 - Trade-offs for two different thresholds

Still not a curve, I know… but we can already learn something from having these two points.

Lowering the threshold moves you to the right along both curves.

Let’s move to the other side now.

High Threshold

What about 70%? If the predicted probability is greater than or equal to 70%, we classify the data point as positive, and as negative otherwise. That’s a very **strict** threshold since we require the model to be very confident to consider a data point to be positive. What can we expect from it? **Less false positives, more false negatives**.

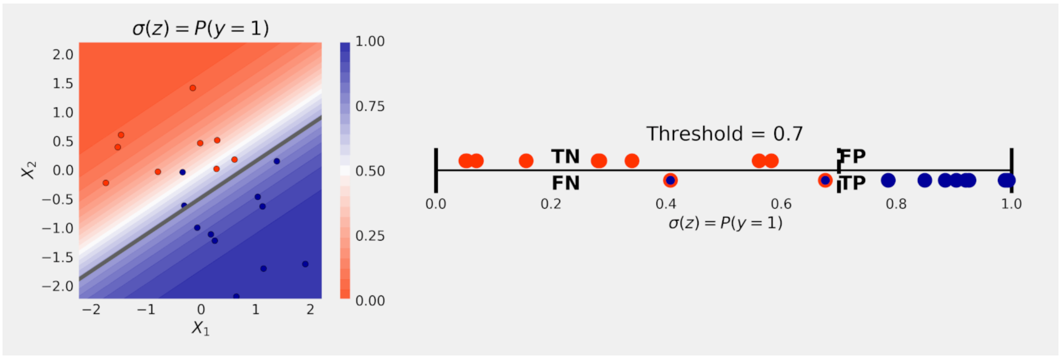


Figure 3.15 - Using a high threshold

You can see in the figure above that **raising the threshold** (moving it to the right on the probability line) **turned two false positives into true negatives** (red points close to 0.6), but it also **turned one true positive into a false negative** (blue point close to 0.6).

Let's double-check it with Scikit-learn's confusion matrix:

```
confusion_matrix(y_val, (probabilities_val >= 0.7))
```

Output

```
array([[9, 0],
       [2, 9]])
```

OK, now let's plot the corresponding metrics again:

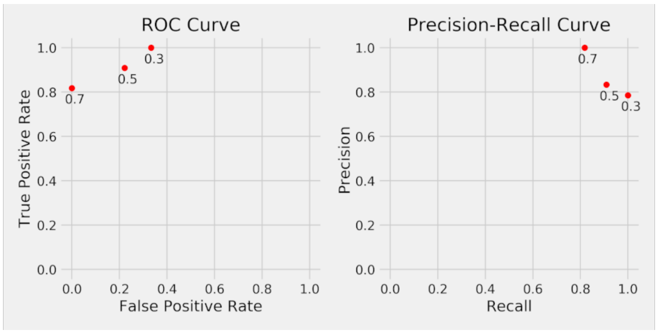


Figure 3.16 - Trade-offs for two different thresholds

I guess we earned the right to call it a curve now :-)

Raising the threshold moves you to the left along both curves.

Can we just *connect the dots* and call it a curve for real? Actually, no, not yet…

ROC and PR Curves

We need to try out **more thresholds** to actually build a curve. Let's try multiples of 10%:

```
threshs = np.linspace(0,1,11)
```

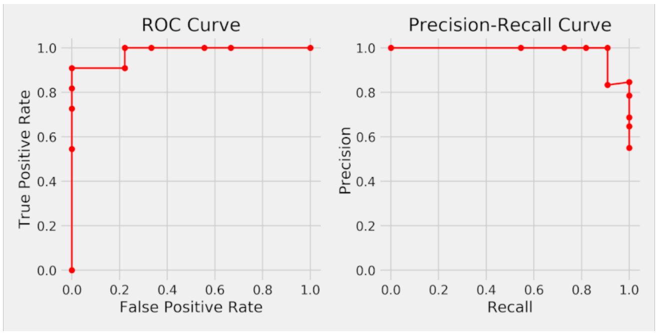


Figure 3.17 - Full curves

Cool! We finally have proper curves! I have some questions for you:

- in each plot, which point corresponds to a **threshold of zero (every prediction is positive)**?
- in each plot, which point corresponds to a **threshold of one (every prediction is negative)**?
- what does the **right-most point in the PR curve** represent?
- if I **raise the threshold**, how do I **move along the curve**?

You should be able to answer all of those questions by referring to the *Metrics* section. But, if you are eager to get the answers, here they are:

- the **threshold of zero** corresponds to the **right-most** point in both curves
- the **threshold of one** corresponds to the **left-most** point in both curves
- the **right-most point in the PR curve** represents the **proportion of positive examples in the dataset**
- if I **raise the threshold**, I am **moving to the left** along both curves

Now, let's double-check our curves with Scikit-Learn's `roc_curve` and `precision_recall_curve` methods:

```
fpr, tpr, thresholds1 = roc_curve(y_val, probabilities_val)
prec, rec, thresholds2 = precision_recall_curve(y_val,
probabilities_val)
```

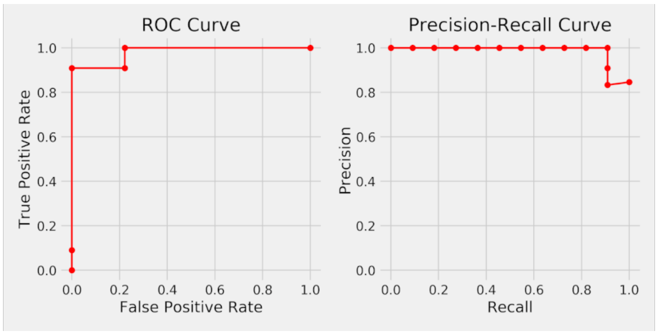


Figure 3.18 - Scikit-Learn's curves

Same shapes, different points.

*"Why do these curves have **different points** than ours?"*

Simply put, Scikit-Learn uses only the **meaningful thresholds**, that is, those thresholds that **actually make a difference to the metrics**. If moving the threshold a bit does not modify the classification of any points, it doesn't matter for building a curve. Also, notice that the **two curves** have a **different number of points** because different metrics have a different set of meaningful thresholds. Moreover, these **thresholds do not necessarily include the extremes, zero, and one**. In Scikit-Learn's PR curve, the right-most point is clearly different than ours.

*"How come the PR curve **dips to lower precision**? Shouldn't it always go up as we raise the threshold, moving to the left along the curve?"*

The Precision Quirk

Glad you asked! This is very annoying and somewhat counterintuitive, but it happens often, so let's take a closer look at it. To illustrate **why this happens**, I will plot the probability lines for three distinct thresholds: 0.4, 0.5, and 0.57.

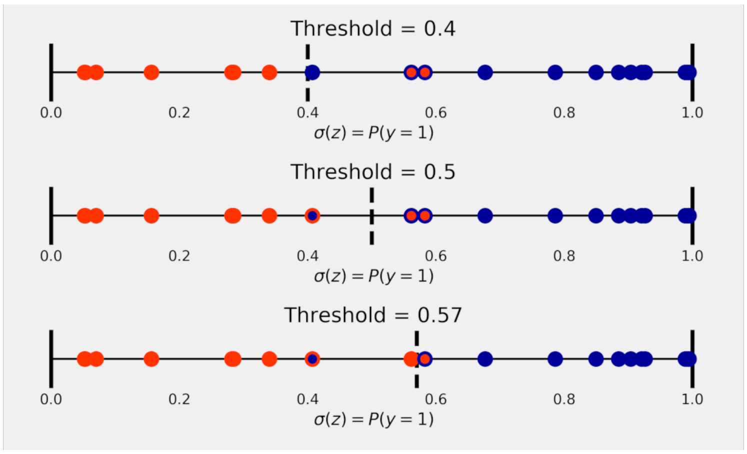


Figure 3.19 - The precision quirk

At the top, with a threshold of 0.4, we have **15 points** on the right (classified as

positive), **two of which are false positives**. The **precision** is given by:

$$Precision_{thresh=0.40} = \frac{13}{13+2} = 0.8666$$

But if we move the threshold to the right, up to 0.5, we **lose one true positive**, effectively **reducing precision**:

$$Precision_{thresh=0.50} = \frac{(13-1)}{(13-1)+2} = \frac{12}{12+2} = 0.8571$$

This is a **temporary side effect**, though. As we raise the threshold even further to 0.57, we get the benefit of **getting rid of a false positive**, thus **increasing precision**:

$$Precision_{thresh=0.57} = \frac{12}{12+(2-1)} = \frac{12}{12+1} = 0.9230$$

In general, **raising the threshold** will **reduce the number of false positives** and **increase precision**.

But, along the way, we **may lose some of the true positives**, which will **temporarily reduce precision**. *Quirky*, right?

Best and Worst Curves

Let's ask ourselves: what would the **best possible** (and, of course, the **worst possible**) curves look like?

The **best** curve belongs to a model that predicts everything perfectly: it gives us a 100% probability to all actual positive data points and 0% probability to all actual negative data points. Of course, such a model *does not exist* in real life. But *cheating* does exist. So, let's cheat and use the **true labels** as the **probabilities**. These are the curves we get:

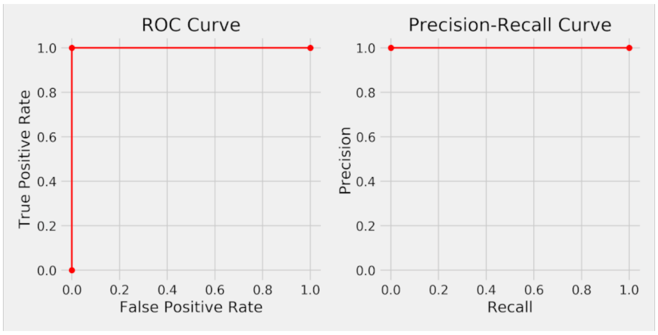


Figure 3.20 - Perfect curves

Nice! If a perfect model exists, its curves are actually **squares**! The **top-left corner** on the **ROC curve**, as well as the **top-right corner** on the **PR curve**, are the (unattainable) sweet spots. Our logistic regression was *not bad*, actually… but, of course, our validation set was ridiculously easy.

"*And the Oscar for the* ***worst curve*** *goes to…*"

"*…the* ***random model****!*"

If a model spits out **probabilities all over the place**, without any regard to the actual data, it is as bad as it can be. We can simply **generate uniformly distributed values between zero and one** as our **random probabilities**:

```
np.random.seed(39)
random_probs = np.random.uniform(size=y_val.shape)

fpr_random, tpr_random, thresholds1_random = roc_curve(y_val,
random_probs)
prec_random, rec_random, thresholds2_random =
precision_recall_curve(y_val, random_probs)
```

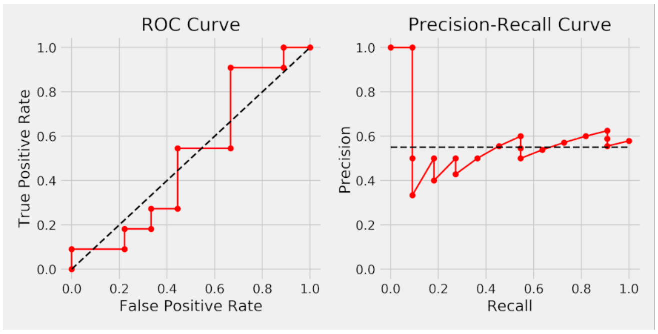


Figure 3.21 - Worst curves ever

We have only 20 data points, so our curves are not **as bad as they theoretically are** :-) The black dashed lines are the *theoretical worst* for both curves. On the left, the **diagonal line** is as bad as it can be. On the right, it is a *bit* more nuanced: the **worst is a horizontal line**, but the **level** is given by the **proportion of positive samples** in the dataset. In our example, we have 11 positive examples out of 20 data points, so the line sits at the level of 0.55.

Comparing Models

"If I have two models, how do I choose the best one?"

"The best model is the one with the best curve."

Captain Obvious

Thank you, Captain. The real question here is: how to compare curves? The **closer** they are to **squares**, the **better** they are, this much we already know. Besides, if **one curve has all its points above all the points of another curve**, the one above is clearly the best. The problem is, two different models may produce curves that **intersect each other** at some point. If that's the case, **there is no clear winner**.

One possible solution to this dilemma is to look at the **area under the curve**. The curve with **more area** under it **wins**! Luckily, Scikit-Learn has an `auc` (area under the curve) method, which we can use to compute the area under the curves for our (good) model:

```
# Area under the curves of our model
auroc = auc(fpr, tpr)
aupr = auc(rec, prec)
print(auroc, aupr)
```

Output

```
0.9797979797979798 0.9854312354312356
```

Very close to the perfect value of one! But then again, this is a toy example… you shouldn't expect figures so high in real-life problems. What about the random model? The theoretical minimum for the **area under the worst ROC curve is 0.5**, which is the area under the diagonal. The theoretical minimum for the **area under the worst PR curve is the proportion of positive samples in the dataset**, which is 0.55 in our case.

```
# Area under the curves of the random model
auroc_random = auc(fpr_random, tpr_random)
aupr_random = auc(rec_random, prec_random)
print(auroc_random, aupr_random)
```

Output

```
0.5050505050505050 0.570559046216941
```

Close enough, after all, the curves produced by our random model were only roughly approximating the theoretical ones.

Further Reading

If you want to learn more about both curves, you can check Scikit-Learn's documentation for Receiver Operating Characteristic (ROC)[69] and Precision-Recall[70]. Another good resource is Jason Brownlee's Machine Learning Mastery blog: How to Use ROC Curves and Precision-Recall Curves for Classification in Python[71] and ROC Curves and Precision-Recall Curves for Imbalanced Classification[72].

Putting It All Together

In this chapter, we haven't modified the training pipeline much. The data preparation part is roughly the same as in the previous chapter, except for the fact that we performed the split using Scikit-Learn this time. The model configuration part is largely the same as well, but we **changed the loss function**, so it is the appropriate one for a **classification** problem. The model training part is quite straightforward since the development of the `StepByStep` class in the last chapter.

But now, after training a model, we can use our class' `predict` method to get predictions for our validation set and use Scikit-Learn's `metrics` module to compute a wide range of classification metrics, like the confusion matrix, for example.

Data Preparation

```
torch.manual_seed(13)

# Builds tensors from numpy arrays
x_train_tensor = torch.as_tensor(X_train).float()
y_train_tensor = torch.as_tensor(y_train.reshape(-1, 1)).float()

x_val_tensor = torch.as_tensor(X_val).float()
y_val_tensor = torch.as_tensor(y_val.reshape(-1, 1)).float()

# Builds dataset containing ALL data points
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
val_dataset = TensorDataset(x_val_tensor, y_val_tensor)

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=16,
    shuffle=True
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Model Configuration

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(2, 1))

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a BCE loss function
loss_fn = nn.BCEWithLogitsLoss()
```

Model Training

```
n_epochs = 100

sbs = StepByStep(model, loss_fn, optimizer)
sbs.set_loaders(train_loader, val_loader)
sbs.train(n_epochs)
```

```
print(model.state_dict())
```

Output

```
OrderedDict([('linear.weight', tensor([[ 1.1822, -1.8684]], device
='cuda:0')),
             ('linear.bias', tensor([-0.0587], device='cuda:0'))])
```

Evaluating

```
logits_val = sbs.predict(X_val)
probabilities_val = sigmoid(logits_val).squeeze()
cm_thresh50 = confusion_matrix(y_val, (probabilities_val >= 0.5))
cm_thresh50
```

Output

```
array([[ 7,  2],
       [ 1, 10]])
```

Recap

In this chapter, we've gone through many concepts related to classification problems. This is what we've covered:

- defining a **binary classification problem**
- generating and preparing a toy dataset using Scikit-Learn's `make_moons` method
- defining **logits** as the result of a **linear combination of features**
- understanding what **odds ratios** and **log odds ratios** are
- figuring we can **interpret logits as log odds ratios**
- mapping **logits into probabilities** using a **sigmoid function**
- defining a **logistic regression** as a **simple neural network with a sigmoid function in the output**
- understanding the **binary cross-entropy loss** and its PyTorch implementation `BCELoss`
- understanding the difference between the `BCELoss` and `BCEWithLogitsLoss`
- highlighting the **importance of choosing the correct combination of the last layer and loss function**

- using PyTorch’s loss functions' arguments to handle **imbalanced datasets**
- **configuring** model, loss function, and optimizer for a classification problem
- **training** a model using the `StepByStep` class
- understanding that the validation loss **may be smaller** than the training loss
- **making predictions** and mapping **predicted logits to probabilities**
- **using a classification threshold** to convert **probabilities into classes**
- understanding the definition of a **decision boundary**
- understanding the concept of **separability of classes** and how it’s related to **dimensionality**
- exploring **different classification thresholds** and its effect on the **confusion matrix**
- reviewing typical **metrics** for evaluating classification algorithms, like true and false positive rates, precision, and recall
- building **ROC** and **precision-recall** curves out of **metrics computed for multiple thresholds**
- understanding the reason behind the **quirk of losing precision** while raising the classification threshold
- defining the **best** and **worst** possible ROC and PR curves
- using the **area under the curve** to **compare different models**

Wow! That’s a *whole lot* of material! Congratulations on finishing yet another big step in your journey! What’s next? We’ll **build upon this knowledge** to tackle an **image classification problem** first and then a **multiclass classification problem** later.

[65] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter03.ipynb

[66] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter03.ipynb

[67] https://bit.ly/2GlmLO0

[68] http://pyimg.co/kku35

[69] https://bit.ly/34lPAlx

[70] https://bit.ly/30xB9JZ

[71] https://bit.ly/30vF7TE

[72] https://bit.ly/2GCEL6A

Part II
Computer Vision

Chapter 4

Classifying Images

Spoilers

In this chapter, we will:

- build models to **classify images**
- use Torchvision to apply **transformations on images**
- **compose transformations** and apply them to datasets
- perform **data augmentation** in the training set
- use **samplers** to handle **imbalanced datasets**
- understand **why** we need **activation functions**
- build a **deeper model** using activation functions

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 4[73] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[74].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 4's notebook. If not, just click on `Chapter04.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import random
import numpy as np
from PIL import Image

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset, random_split, \
WeightedRandomSampler, SubsetRandomSampler
from torchvision.transforms import Compose, ToTensor, Normalize,\
ToPILImage, RandomHorizontalFlip, Resize

import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
%matplotlib inline

from data_generation.image_classification import generate_dataset
from stepbystep.v0 import StepByStep
```

Classifying Images

Enough already with simple data points: let's classify **images**! Although the data is different, it is still a classification problem, so we will try to predict **which class an image belongs to**.

First, let's generate some images to work with (so we don't have to use MNIST[75]!).

Data Generation

Our images are quite simple: they have black backgrounds and white lines drawn on top of them. The lines can be drawn either in a **diagonal** or in a **parallel** (to one of the edges, so they could be either horizontal or vertical) way. So, our **classification problem** can be simply stated as: **is the line diagonal**?

If the line **is diagonal**, then we assume it belongs to the **positive class**. If it is **not diagonal**, it belongs to the **negative class**. We got our **labels (*y*),** and we can summarize them like this:

Line	Value	Class
Not Diagonal	0	Negative
Diagonal	1	Positive

Let's generate 300 random images, each one five-by-five pixels in size:

Data Generation

```
images, labels = generate_dataset(
  img_size=5, n_images=300, binary=True, seed=13
)
```

And then let's visualize the first 30 images:

```
fig = plot_images(images, labels, n_plot=30)
```

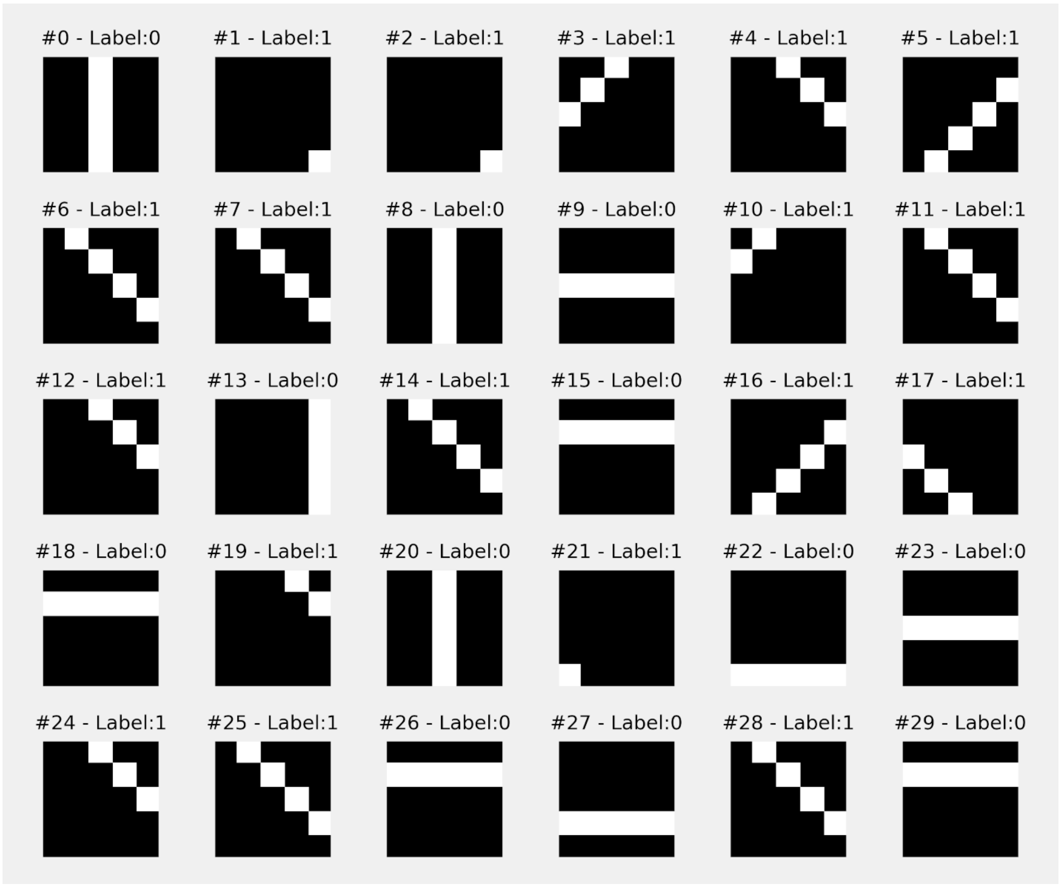


Figure 4.1 - Image dataset

Since our images are quite small, there aren't *that* many possibilities for drawing lines on top of them. There are actually 18 different configurations for diagonal lines (nine to the left, nine to the right), and another 10 different configurations for horizontal and vertical lines (five each). That's a total of 28 possibilities in a 300-image dataset. So there will be lots of duplicates (like images #1 and #2, or #6 and #7, for example), but that's fine.

Images and Channels

In case you're not familiar with the meaning of channels, pixel values, and how images are represented as tensors, this is a **brief** overview of these topics.

To illustrate how images are represented, let's create three **separate** images first:

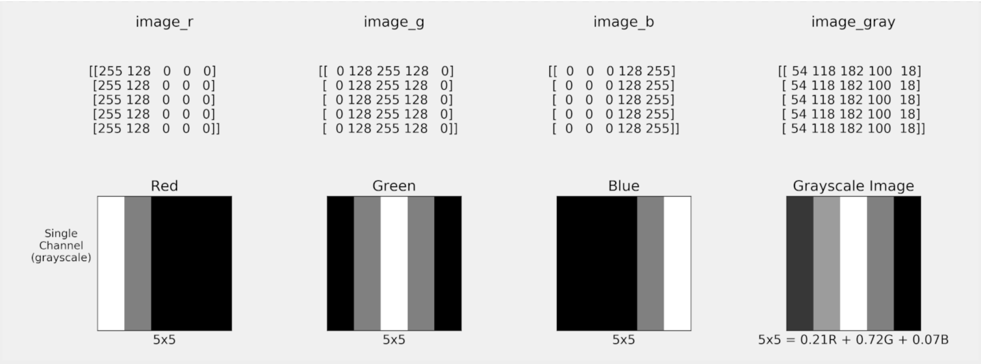
```
image_r  = np.zeros((5, 5), dtype=np.uint8)
image_r[:, 0] = 255
image_r[:, 1] = 128

image_g = np.zeros((5, 5), dtype=np.uint8)
image_g[:, 1] = 128
image_g[:, 2] = 255
image_g[:, 3] = 128

image_b = np.zeros((5, 5), dtype=np.uint8)
image_b[:, 3] = 128
image_b[:, 4] = 255
```

Each one of these images is five-by-five pixels and is represented by a five-by-five matrix. It is a **two-dimensional** representation, which means it is a **single-channel image**. Moreover, its `dtype` is `np.uint8`, which only accepts **values from zero to 255**.

If an image has **only one channel**, it is a **grayscale image**.

The **range of pixel values** goes from **zero (black)** to **255 (white)**, and everything **in between** is a **shade of gray**.

Let's see what the matrices above represent:



```
[[255 128   0   0   0]
 [255 128   0   0   0]
 [255 128   0   0   0]
 [255 128   0   0   0]
 [255 128   0   0   0]]
```

```
[[  0 128 255 128   0]
 [  0 128 255 128   0]
 [  0 128 255 128   0]
 [  0 128 255 128   0]
 [  0 128 255 128   0]]
```

```
[[  0   0   0 128 255]
 [  0   0   0 128 255]
 [  0   0   0 128 255]
 [  0   0   0 128 255]
 [  0   0   0 128 255]]
```

```
[[ 54 118 182 100  18]
 [ 54 118 182 100  18]
 [ 54 118 182 100  18]
 [ 54 118 182 100  18]
 [ 54 118 182 100  18]]
```

Taken individually, they are three images with vertical stripes. But we can **pretend** they represent **different colors**: **red**, **green**, and **blue**. These three colors are the **base colors** used to build all others. That's where the **RGB** acronym comes from!

If we perform a **weighted average** of these three colors, we'll get **another grayscale** image. This should be no surprise since it still has **only one channel**.

```
image_gray = .2126*image_r + .7152*image_g + .0722*image_b
```

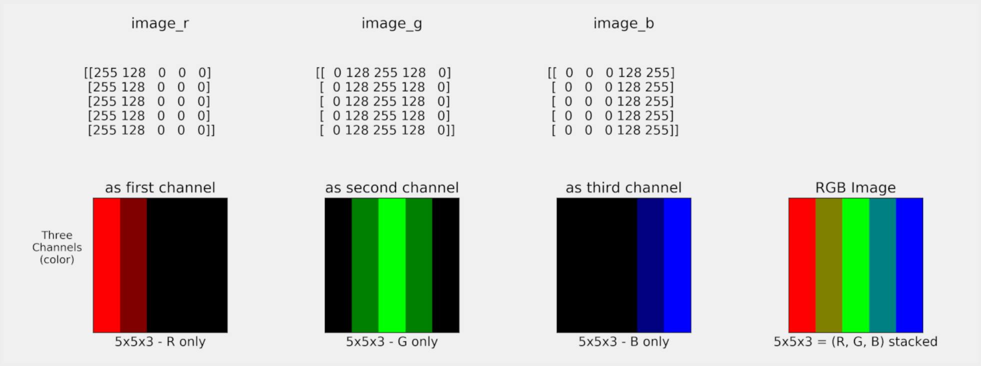
By the way, these weights are not arbitrary: they are considered to best preserve the original characteristics of the image. If you use an image editor to convert a colored image to grayscale, this is what the software is doing under the hood.

Grayscale images are **boring**, though. May I have some **color**, please? It turns out, we only need to **stack** the three images representing the three colors, **each image becoming a channel**.

Colored images have **three channels**, one for each color: red, green, and blue, in that order.

```python
image_rgb = np.stack([image_r, image_g, image_b], axis=2)
```

Let's see what those same matrices represent, once we consider them channels (unfortunately, the visual impact is completely lost in the printed version):



Before moving on with our classification problem, we need to address the **shape issue**: different frameworks (and Python packages) use different conventions for the shape of the images.

NCHW vs NHWC

"*What do these acronyms stand for?*"

It's quite simple, actually:

- **N** stands for the **N**umber of images (in a mini-batch, for instance)
- **C** stands for the number of **C**hannels (or **filters**) in each image
- **H** stands for each image's **H**eight
- **W** stands for each image's **W**idth

Thus the acronyms indicate the expected **shape of the mini-batch**:

- **NCHW**: (number of images, channels, height, width)
- **NHWC**: (number of images, height, width, channel)

Basically, everyone agrees that the **number of images comes first**, and that **height and width are an inseparable duo**. It all comes down to the **channels (or filters)**: for each individual image, it may be either the **first dimension** (*before* HW) or the **last dimension** (*after* HW).

There are endless discussions about which format is better, faster, or whatever. We're not getting into this discussion here. Nonetheless, we need to address this difference because it is a common source of **confusion** and **error** since each package or framework uses a different one:

- *PyTorch* uses **NCHW**
- *TensorFlow* uses **NHWC**
- *PIL* images are **HWC**

Annoying, right? I think so, too. But it is only a matter of paying close attention to which format is coming in, and which format is coming out after an operation. Let's work our way through it.

Our dataset generates images following the **PyTorch format**, that is, **NCHW**. What's the **shape** of our dataset, then?

```
images.shape
```

Output

```
(300, 1, 5, 5)
```

As expected, 300 images, single-channel, five pixels wide, five pixels high. Let's take a closer look at one image, say, image #7:

```
example = images[7]
example
```

Output

```
array([[[  0, 255,   0,   0,   0],
        [  0,   0, 255,   0,   0],
        [  0,   0,   0, 255,   0],
        [  0,   0,   0,   0, 255],
        [  0,   0,   0,   0,   0]]], dtype=uint8)
```

That’s fairly straightforward, we can even "*see*" the diagonal line of values equal to 255 representing the white pixels.

What would an image in the **HWC** (PIL image format) look like? We can **transpose** the *first* dimension to become the *last* using **Numpy**'s `transpose`:

```
example_hwc = np.transpose(example, (1, 2, 0))
example_hwc.shape
```

Output

```
(5, 5, 1)
```

The shape is correct: **HWC**. What about the content?

```
example_hwc
```

Output

```
array([[[  0],
        [255],
        [  0],
        [  0],
        [  0]],
        ...
       [[  0],
        [  0],
        [  0],
        [  0],
        [  0]]], dtype=uint8)
```

Say what you want, but this **HWC** format is surely **not** intuitive to look at, array-style!

The good thing is, PyTorch's default shape for a single image (**CHW**) is much better to look at. And, if you need a PIL image at some point, PyTorch provides means for transforming images back and forth between the two shapes.

It is time to introduce you to…

Torchvision

Torchvision is a package containing popular datasets, model architectures, and common image transformations for computer vision.

Datasets

Many of the popular and common **datasets** are included out of the box, like MNIST, ImageNet, CIFAR, and many more. All these datasets inherit from the original `Dataset` class, so they can be naturally used with a `DataLoader` in exactly the same way we've been doing so far.

There is one particular dataset we should pay more attention to: `ImageFolder`. This is *not* a dataset itself, but a **generic dataset** that you can use with your own images, provided that they are properly organized into sub-folders, each sub-folder named after a class and containing the corresponding images.

We'll get back to it in Chapter 6 when we use "*Rock-Paper-Scissors*" images to build a dataset using `ImageFolder`.

Models

PyTorch also includes the most popular **model architectures**, including its **pre-trained weights**, for tackling many tasks like image classification, semantic segmentation, object detection, instance segmentation, and person keypoint detection, and video classification.

Among the many models, we can find the well-known AlexNet, VGG (in its many incarnations: VGG11, VGG13, VGG16, and VGG19), ResNet (also in many flavors: ResNet18, ResNet34, ResNet50, ResNet101, ResNet152), and Inception V3.

In Chapter 7, we will load a pre-trained model and fine-tune it to our particular task. In other words, we'll use **transfer learning**.

Transforms

Torchvision has some common image transformations on its **transforms** module. It is important to realize there are two main groups of transformations:

- transformations based on **images** (either in PIL or PyTorch shapes)
- transformations based on **Tensors**

Obviously, there are conversion transforms to convert from tensor **`ToPILImage`** and from PIL image **`ToTensor`**.

Let's start using `ToTensor` to convert a *Numpy* array (in PIL shape) to a PyTorch tensor. We can create a "*tensorizer*" (for lack of a better name), and feed it our example image (#7) in HWC shape:

```
tensorizer = ToTensor()
example_tensor = tensorizer(example_hwc)
example_tensor.shape
```

Output

```
torch.Size([1, 5, 5])
```

Cool, we got the expected CHW shape. So, its content should be easy to associate with the underlying image:

```
example_tensor
```

Output

```
tensor([[[0., 1., 0., 0., 0.],
         [0., 0., 1., 0., 0.],
         [0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 1.],
         [0., 0., 0., 0., 0.]]])
```

And indeed it is: once again we can "*see*" the diagonal line. But, this time, its values are **not equal to 255** anymore, they are **equal to 1.0** instead.

`ToTensor` may **scale** the values **from a [0, 255] range to a [0.0, 1.0] range**, if the input is either a *Numpy* array with `dtype` equals to `uint8` (as in our example) or a PIL image belonging to one of the following modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1).

To learn more about image modes, check this[76] documentation out.

"So, I convert PIL images and Numpy arrays to PyTorch tensors from the start and it is all good?"

That's pretty much it, yes. It wasn't always like that, though, since earlier versions of TorchVision implemented the **interesting transformations** for **PIL images** only.

"What do you mean by ***interesting transformations****? What do they do?"*

These transformations **modify the training images** in many different ways: rotating, shifting, flipping, cropping, blurring, zooming in, adding noise to it, erasing parts of it…

"Why would I ever want to modify my training images like that?"

That's what's called **data augmentation**. It is a clever technique to **expand a dataset** (augment it) **without collecting more data**. In general, deep learning models are very **data-hungry**, requiring a massive amount of examples to perform well. But collecting large datasets is often challenging, and sometimes impossible.

Enter data augmentation: **rotate an image** and **pretend it is a brand new image**. Flip an image and do the same. Even better, **do it randomly** during model training, so the model sees many different versions of it.

Let's say we have an **image of a dog**. If we **rotate it**, it is **still a dog**, but from a **different angle**. Instead of taking two pictures of the dog, one from each angle, we take the picture we already have and use data augmentation to **simulate many different angles**. Not quite the same as the real deal, but close enough to improve our model's performance. Needless to say, data augmentation is **not suited for every task**: if you are trying to perform object detection, that is, detecting the **position of an object** in a picture, you **shouldn't** do anything that changes its position, like flipping or shifting. Adding noise would still be fine, though.

This is just a brief overview of data augmentation techniques, so you understand the reasoning behind including this kind of transformations in a training set.

There is also "test-time augmentation", which can be used to improve the performance of a model after it's deployed. This is more advanced, though, and beyond the scope of this book.

The bottom line is, these transformations are important. To more easily visualize the resulting images, we may use `ToPILImage` to convert a tensor to a PIL image:

```
example_img = ToPILImage()(example_tensor)
print(type(example_img))
```

Output

```
<class 'PIL.Image.Image'>
```

Notice that it is a **real PIL image**, not a *Numpy* array anymore, so we can use Matplotlib to visualize it:

```
plt.imshow(example_img, cmap='gray')
plt.grid(False)
```

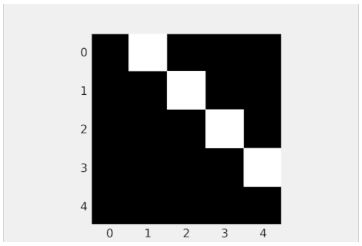


Figure 4.2 - Image #7

ToPILImage may take either a **tensor in PyTorch shape (CHW)** or a ***Numpy* array in PIL shape (HWC)** as inputs.

Transforms on Images

These transforms include the typical things you'd like to do with an image for the purpose of data augmentation: Resize, CenterCrop, GrayScale, RandomHorizontalFlip, and RandomRotation, to name a few. Let's use our example image above and try some **random horizontal flipping**. But, just to make sure we flip it, let's ditch the randomness and make it flip 100% of the time:

```
flipper = RandomHorizontalFlip(p=1.0)
flipped_img = flipper(example_img)
```

OK, the image should be flipped horizontally now. Let's check it out:

```
plt.imshow(flipped_img, cmap='gray')
plt.grid(False)
```

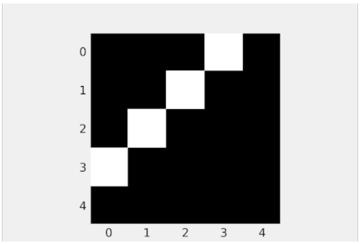


Figure 4.3 - Flipped Image #7

Transforms on Tensor

There are **only four** transforms that take (non-image) tensors as inputs: LinearTransformation, Normalize, RandomErasing (although I believe this one was a better fit for the other group of transforms…), and ConvertImageDtype.

First, let's transform our flipped image to a tensor using the `tensorizer` we've already created:

```
img_tensor = tensorizer(flipped_img)
img_tensor
```

Output

```
tensor([[[0., 0., 0., 1., 0.],
         [0., 0., 1., 0., 0.],
         [0., 1., 0., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]]])
```

Normalize Transform

Now we can apply one of the most common transformations: `Normalize`. In its documentation, we can get a brief description of this transformation:

> Normalize a tensor image with mean and standard deviation. Given mean: `(mean[1],...,mean[n])` and std: `(std[1],..,std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

Does it look familiar? Remember the `StandardScaler` we have used in previous chapters to **standardize** our features? That's the tensor-based version of it, operating independently on each image channel.

"*Why is it called* ***normalize*** *then?*"

Unfortunately, there are *many names* for the procedure that involves subtracting the mean value first, and then dividing the result by the standard deviation. In my opinion, it should always be called **standardization**, as in Scikit-Learn, since normalizing means something else (transforming the features such that every data point has a unit norm). But, in many cases, and Torchvision is one of those cases, the standardization procedure is called **normalization** (coming from normal distribution, not from the unit norm).

Regardless of its name, standardization or normalization, this transformation modifies the range of values of a given feature or set of features. As we have seen in Chapter 0, having features in well-behaved ranges greatly improves the performance of gradient descent. Moreover, as we'll see shortly, it is better to have features with symmetrical ranges of values (from -1 to 1, for example) when training neural networks.

By definition, **pixel values can only be positive**, usually in the range [0, 255]. From our **image tensor**, we see its values are in the **[0, 1] range**, and we have only **one channel**. We can use the `normalize` transform to have its values mapped to a symmetrical range.

But, **instead of computing mean and standard deviation** first (as we did in previous chapters), let's **set the mean to 0.5** and **set the standard deviation to 0.5** as well.

"*Wait a moment... why?!*"

By doing so, we'll be effectively performing a **min-max scaling** (like Scikit-Learn's `MinMaxScaler`) such that the resulting range is [-1, 1]. It is easy to see why, if we compute the resulting values for the extremes of our original range [0, 1]:

$$
\begin{aligned}
input = 0 &\implies \frac{0 - mean}{std} = \frac{0 - 0.5}{0.5} = -1 \\
input = 1 &\implies \frac{1 - mean}{std} = \frac{1 - 0.5}{0.5} = 1
\end{aligned}
$$

Normalizer

There we go: the resulting range is [-1, 1]. Actually, we could set it to *anything* we want. Had we chosen a standard deviation of 0.25, we would get a [-2, 2] range instead. If we had chosen a mean value different than the midpoint of the original range, we would end up with an asymmetrical range as a result.

Now, if we take the trouble of **actually computing the real mean and standard deviation of the training data**, we would have achieved an actual **standardization**, that is, our training data would have **zero mean** and **unit standard deviation**.

For now, let's stick with the lazy approach and use the `Normalize` transformation as a **min-max scaler** to the [-1, 1] range:

```
normalizer = Normalize(mean=(.5,), std=(.5,))
normalized_tensor = normalizer(img_tensor)
normalized_tensor
```

Output

```
tensor([[[-1., -1., -1.,  1., -1.],
         [-1., -1.,  1., -1., -1.],
         [-1.,  1., -1., -1., -1.],
         [ 1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.]]])
```

Notice that the transformation takes **two tuples** as arguments, one tuple for the means, another one for the standard deviations. Each tuple has **as many values as channels** in the image. Since we have single-channel images, our tuples have a single element each.

It is also easy to see that we achieved the desired range of values: the transformation simply converted "*zeros*" into negative ones and preserved the original "*ones*". Good for illustrating the concept, but surely not exciting.

In Chapter 6, we'll use `Normalize` to **standardize real (3-channel) images**.

Composing Transforms

No one expects you to run these transformations one by one, that's what `Compose` can be used for: **composing several transformations** into a single, big, composed, transformation. Also, I guess I could have composed a better sentence to explain it (pun intended).

It is quite simple, actually: just line up all desired transformations in a list. This works pretty much the same way as a pipeline in Scikit-Learn. We only need to make sure the **output** of a given **transformation** is an **appropriate input** for the **next one**.

Let's compose a new transformation using the following list of transformations:

- first, let's **flip an image** using `RandomHorizontalFlip`
- next, let's perform some **min-max scaling** using `Normalize`

In code, the sequence above looks like this:

```
composer = Compose([RandomHorizontalFlip(p=1.0),
                    Normalize(mean=(.5,), std=(.5,))])
```

If we use the composer above to transform the example **tensor**, we should get the **same normalized tensor as output**. Let's double check it:

```
composed_tensor = composer(example_tensor)
(composed_tensor == normalized_tensor).all()
```

Output

```
tensor(True)
```

Great! We can use a single, composed, transformation from now on!

Notice that we have not used the original `example`, a *Numpy* array already in PyTorch shape (CHW), as input. To understand why, let's briefly compare it to the `example_tensor` we used as the actual input (a PyTorch tensor, also in CHW shape):

```
print(example)
print(example_tensor)
```

Output

```
[[[  0 255   0   0   0]
  [  0   0 255   0   0]
  [  0   0   0 255   0]
  [  0   0   0   0 255]
  [  0   0   0   0   0]]]
tensor([[[0., 1., 0., 0., 0.],
         [0., 0., 1., 0., 0.],
         [0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 1.],
         [0., 0., 0., 0., 0.]]])
```

As you can see, the only differences between them are the scale (255 vs one) and the type (integer and float). We can convert the former into the latter using a one-liner:

```
example_tensor = torch.as_tensor(example / 255).float()
```

Moreover, we can use this line of code to convert our whole *Numpy* dataset into tensors, so they become an appropriate input to our composed transformation.

Data Preparation

The first step of data preparation, as in previous chapters, is to convert our features and labels from *Numpy* arrays to PyTorch tensors:

```
# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels.reshape(-1, 1)).float()
```

The only difference is that we scaled the images to get them into the expected [0.0, 1.0] range.

Dataset Transforms

Next, we use both tensors to build a `Dataset`, but not a simple `TensorDataset` like before. Once again, we'll build our own **custom dataset**, that is capable of **handling transformations**. Its code is actually quite simple:

Transformed Dataset

```python
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]

        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

It takes **three arguments**: a tensor for **features (*x*)**, another tensor for **labels (*y*)**, and an **optional transformation**. These arguments are then stored as **attributes** of the class. Of course, if *no transformation* is given, it will behave similarly to a regular `TensorDataset`.

The main difference is in the `__getitem__` method: instead of simply returning the elements corresponding to a given index in both tensors, it **transforms the features**, if a transformation is defined.

*"Do I **have to** create a custom dataset to perform transformations?"*

Not necessarily, no. The **`ImageFolder` dataset**, which you'll likely use for handling real images, **handles transformations out-of-the-box**. The mechanism is essentially the same: if a transformation is defined, the dataset applies it to the images. The **purpose** of using yet another custom dataset here is to **illustrate this mechanism**.

So, let's redefine our composed transformations (so it *actually* flips the image *randomly* instead of every time) and create our dataset:

```
composer = Compose([RandomHorizontalFlip(p=0.5),
                    Normalize(mean=(.5,), std=(.5,))])

dataset = TransformedTensorDataset(x_tensor, y_tensor, composer)
```

Cool! But we still have to **split** the dataset as usual. But we'll do it a *bit differently* this time…

SubsetRandomSampler

Previously, when creating a data loader for the training set, we used to set its argument `shuffle` to `True` (since shuffling data points, in most cases, improves the performance of gradient descent). This was a very convenient way of **shuffling** the data, which was **implemented using a `RandomSampler`** under the hood. Every time a new mini-batch was required, it **sampled** some indices **randomly**, and the data points corresponding to those indices were returned.

Even when there was **no shuffling** involved, as in the data loader used for the validation set, a **`SequentialSampler`** was used. In this case, whenever a new mini-batch was required, this sampler simply returned a sequence of **indices**, **in order**, and the data points corresponding to those indices were returned.

In a nutshell, a **sampler** can be used to **return sequences of indices** to be used for data loading. In the two examples above, each sampler would take a `Dataset` as an argument. But not all samplers are like that.

The `SubsetRandomSampler` samples indices **from a list**, given as argument, without replacement. As in the other samplers, these indices will be used to load data from a dataset. If an **index is not on the list**, the corresponding **data point will never be used**.

So, if we have **two disjoint lists of indices** (that is, no intersection between them, and they cover all elements if added together), we can create **two samplers** to effectively **split a dataset**. Let's put this into code, to make it more clear.

First, we need to generate **two shuffled lists of indices**, one corresponding to the points in the **training set**, the other, to the points in the **validation set**. We have done this already using *Numpy*. Let's make it a bit more *interesting* and *useful* this time by assembling the **Helper Function #4**, aptly named `index_splitter`, to split the indices:

Helper Function #4

```python
def index_splitter(n, splits, seed=13):
    idx = torch.arange(n)
    # Makes the split argument a tensor
    splits_tensor = torch.as_tensor(splits)
    # Finds the correct multiplier, so we don't have
    # to worry about summing up to N (or one)
    multiplier = n / splits_tensor.sum()
    splits_tensor = (multiplier * splits_tensor).long()
    # If there is a difference, throws at the first split
    # so random_split does not complain
    diff = n - splits_tensor.sum()
    splits_tensor[0] += diff
    # Uses PyTorch random_split to split the indices
    torch.manual_seed(seed)
    return random_split(idx, splits_tensor)
```

The function above takes three arguments:

- `n`: the **number of data points** to generate indices for
- `splits`: a list of values representing the **relative weights** of the split sizes
- `seed`: a random seed to ensure **reproducibility**

It always bugged me a little that PyTorch's `random_split` needs a list with the *exact* number of data points in each split. I wish I could give it **proportions**, like `[80, 20]`, or [.8, .2], or even [4, 1], and then it would **figure out how many points** go into each split on its own. That's the main reason index_splitter exists: we can give it relative weights, and it figures the number of points out.

Sure, it still calls random_split to split a tensor containing a list of indices (in previous chapters, we used it to split Dataset objects instead). The resulting splits are Subset objects:

```
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
train_idx
```

Output

```
<torch.utils.data.dataset.Subset at 0x7fc6e7944290>
```

Each subset contains the corresponding indices as an attribute:

```
train_idx.indices
```

Output

```
[118,
 170,
 ...
 10,
 161]
```

Next, each Subset object is used as an argument to the corresponding sampler:

```
train_sampler = SubsetRandomSampler(train_idx)
val_sampler = SubsetRandomSampler(val_idx)
```

So, we can use a **single dataset** to load the data from since the split is controlled by the samplers. But we still need **two data loaders**, each one using its corresponding sampler:

```
# Builds a loader of each set
train_loader = DataLoader(
    dataset=dataset, batch_size=16, sampler=train_sampler
)
val_loader = DataLoader(
    dataset=dataset, batch_size=16, sampler=val_sampler
)
```

If you're using a **sampler**, you **cannot** set `shuffle=True`.

We can also check if the loaders are returning the correct number of mini-batches:

```
len(iter(train_loader)), len(iter(val_loader))
```

Output

```
(15, 4)
```

There are 15 mini-batches in the training loader (15 mini-batches * 16 batch size = 240 data points), and four mini-batches in the validation loader (4 mini-batches * 16 batch size = 64 data points). In the validation set, the last mini-batch will have only 12 points, since there are only 60 points in total.

OK, cool, this means we don't need two (split) datasets anymore, we only need two samplers. Right? Well, it **depends**.

Data Augmentation Transforms

No, I did not change topics :-) The reason why we **may still need two split datasets** is exactly that: **data augmentation**. In general, we want to apply data augmentation to the **training data only** (yes, there is test-data augmentation too, but that's a different matter). But data augmentation is accomplished using **composing transforms**, which will be **applied to all points in the dataset**. See the problem?

If we need **some** data points to be **augmented**, but not others, the **easiest way** to accomplish this is to create **two composers** and use them in **two different datasets**. We can still use the indices, though:

```
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]
```

Then, here come the two composers: the `train_composer` augments the data, and then scales it (min-max); the `val_composer` only scales the data (min-max).

```
train_composer = Compose([RandomHorizontalFlip(p=.5),
                          Normalize(mean=(.5,), std=(.5,))])

val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
```

Next, we use them to create two datasets and their corresponding data loaders:

```
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor, transform=train_composer
)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor, transform=val_composer
)

# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16, shuffle=True
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

And, since we're not using a sampler to perform the split anymore, we can (and should) set `shuffle` to `True`.

If you **do not** perform **data augmentation**, you may **keep using samplers** and a **single dataset**.

Disappointed with the apparently short-lived use of samplers? Don't be! I saved the best sampler for last.

WeightedRandomSampler

We have already talked about **imbalanced datasets** when learning about binary cross-entropy losses in Chapter 3. We adjusted the **loss weight** for points in the **positive class** to compensate for the imbalance. It wasn't quite the *weighted average* one would expect, though. Now, we can tackle the **imbalance** using a different approach: a **weighted sampler**.

The reasoning is pretty much the same but, *instead of weighted losses*, we use **weights for sampling**: the class with **fewer data points (minority class)** should get **larger weights**, while the class with **more data points (majority class)** should get **smaller weights**. This way, on average, we'll end up with mini-batches containing roughly the same number of data points in each class: a **balanced dataset**.

"How are the weights computed?"

First, we need to find **how imbalanced** the dataset is, that is, how many data points belong to each label. We can use PyTorch's `unique` method on our training set labels (`y_train_tensor`), with `return_counts` equals `True`, to get a list of the **existing labels** and the corresponding **number of data points**:

```
classes, counts = y_train_tensor.unique(return_counts=True)
print(classes, counts, weights)
```

Output

```
tensor([0., 1.]) tensor([ 80, 160])
```

Ours is a **binary classification**, so it is no surprise we have **two classes**: zero (**not diagonal**) and one (**diagonal**). There are 80 images with lines that are not diagonal, and 160 images with diagonal lines. Clearly, an imbalanced dataset.

Next, we use these counts to compute the **weights**, by **inverting the counts**. It is as simple as that:

```
weights = 1.0 / counts.float()
weights
```

Output

```
tensor([0.0125, 0.0063])
```

The first weight (0.0125) corresponds to the negative class (not diagonal). Since this class has only 80 out of 240 images in our training set, it is also the **minority**

class. The other weight (0.0063) corresponds to the positive class (diagonal), which has the remaining 160 images thus making it the **majority class**.

The **minority class** should have the **largest weight**, so each data point belonging to it gets **overrepresented** to compensate for the imbalance.

"But these weights do not sum up to one, isn't it wrong?"

It is usual to have weights summing up to one, sure, but this is **not required** by PyTorch's weighted sampler. We can get away with **weights inversely proportional to the counts**. In this sense, the sampler is very "forgiving". But it is not without its own quirks, unfortunately.

It is *not enough* to provide a sequence of weights corresponding to each different class in the training set. It **requires** a sequence containing **the corresponding weight for each and every data point** in the training set. Even though this is a bit annoying, it is not so hard to accomplish: we can use the labels as indexes of the weights we computed above. It is probably easier to see it in code:

```
sample_weights = weights[y_train_tensor.squeeze().long()]

print(sample_weights.shape)
print(sample_weights[:10])
print(y_train_tensor[:10].squeeze())
```

Output

```
torch.Size([240])
tensor([0.0063, 0.0063, 0.0063, 0.0063, 0.0063, 0.0125, 0.0063,
        0.0063, 0.0063, 0.0063])
tensor([1., 1., 1., 1., 1., 0., 1., 1., 1., 1.])
```

Since there are 240 images in our training set, we need 240 weights. We squeeze our labels (`y_train_tensor`) to a single dimension and cast them to `long` type since we want to use them as indices. The code above shows the first 10 elements, so you can actually see the correspondence between class and weight in the resulting tensor.

The sequence of weights is the main argument used to create the `WeightedRandomSampler` but not the only one. Let's take a look at its arguments:

- `weights`: a sequence of weights like the one we have just computed
- `num_samples`: how many samples are going to be drawn from the dataset
    - a typical value is the **length** of the sequence of weights, as you're likely sampling from the whole training set
- `replacement`: if `True` (the default value), it draws samples *with* replacement
    - if `num_samples` equals the length, that is, if the whole training set is used, it makes sense to draw samples **with** replacement to effectively compensate for the imbalance
    - it only makes sense to set it to `False` **if `num_samples` < length** of the dataset
- `generator`: optional, it takes a (pseudo) random number `Generator` that will be used for drawing the samples
    - to ensure *reproducibility*, we **need** to create and assign a **generator** (which has its own seed) to the sampler, since the **manual seed** we've already set is **not enough**

OK, we'll sample from the whole training set, and we got our sequence of weights ready. We are still missing a generator, though. Let's create both the generator and the sampler now:

```
generator = torch.Generator()

sampler = WeightedRandomSampler(
    weights=sample_weights,
    num_samples=len(sample_weights),
    generator=generator,
    replacement=True
)
```

"Didn't you say we need to set a seed for the generator?! Where is it?"

Indeed, I said it. We'll set it soon, **after** assigning the sampler to the data loader. You'll understand the reasoning behind this choice shortly, please bear with me. So, let's (re-)create the data loaders using the weighted sampler with the training set:

```
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16,  sampler=sampler
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Once again, if we're using a sampler, we cannot use the `shuffle` argument.

There is a lot of **boilerplate code** here, right? Let's build yet another function, **Helper Function #5**, to wrap it all up:

Helper Function #5

```
def make_balanced_sampler(y):
    # Computes weights for compensating imbalanced classes
    classes, counts = y.unique(return_counts=True)
    weights = 1.0 / counts.float()
    sample_weights = weights[y.squeeze().long()]
    # Builds sampler with compute weights
    generator = torch.Generator()
    sampler = WeightedRandomSampler(
        weights=sample_weights,
        num_samples=len(sample_weights),
        generator=generator,
        replacement=True
    )
    return sampler
```

```
sampler = make_balanced_sampler(y_train_tensor)
```

Much better! Its only argument is the tensor containing the **labels**: the function will compute the weights and build the corresponding weighted sampler on its own.

Seeds and more (seeds)

Time to set the **seed** for the **generator** used in the **sampler** assigned to the **data loader**. It is a long sequence of objects, but we can work our way through it to retrieve the generator and call its `manual_seed` method:

```
train_loader.sampler.generator.manual_seed(42)
random.seed(42)
```

Now we can check if our sampler is doing its job correctly. Let's have it sample a full run (240 data points in 15 mini-batches of 16 points each), and sum up the labels so we know **how many points are in the positive class**:

```
torch.tensor([t[1].sum() for t in iter(train_loader)]).sum()
```

Output

```
tensor(123.)
```

Close enough! We have 160 images of the positive class, and now, thanks to the weighted sampler, we're sampling only 123 of them. It means we're oversampling the negative class (which has 80 images) to a total of 117 images, adding up to 240 images. Mission accomplished, our dataset is **balanced** now.

"*Wait a minute! Why on Earth there was **an extra seed** in the code above? Don't we have enough already?*"

I agree, **too many seeds**. Besides one specific seed for the generator, we also have to **set yet another seed** for Python's `random` module.

Honestly, this came to me as a surprise too when I found out about it! As weird as it may sound, in Torchvision versions prior to 0.8, there was *still* some code that **depended upon Python's native random module**, instead of PyTorch's *own* random generators. The problem happened when some of the **random transformations** for data augmentation were used, like `RandomRotation`, `RandomAffine`, and others.

It's better to be **safe** than sorry, so we better **set yet another seed to ensure the reproducibility** of our code.

And that's **exactly** what we're going to do! Remember the `set_seed` method we implemented in Chapter 3? Let's **update it** to include **more seeds**:

StepByStep Method

```
def set_seed(self, seed=42):
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)
    try:
        self.train_loader.sampler.generator.manual_seed(seed)
    except AttributeError:
        pass

setattr(StepByStep, 'set_seed', set_seed)
```

Four seeds and counting! Our updated method **tries to update** the seed of the generator used by the sampler assigned to the data loader of the training set. But, if there is no generator (the argument is *optional*, after all), it fails silently.

Putting It Together

We've gone through **a lot** of things regarding the data preparation step. Let's put them all together to get a better view of the *big picture* here.

First, we've built a **custom dataset** to handle **transforms on tensors**, and two helper functions to handle boilerplate code for **splitting indices** and building a **weighted random sampler**.

Then, we performed **many different processing steps** as **data preparation**:

- modifying the **scale of pixel value**s from [0, 255] to [0, 1]
- **splitting** indices and tensors into training and validation sets
- building **composed transforms**, including data augmentation in the training set
- using a custom dataset to **apply transforms to tensors**

- creating a **weighted random sampler** to handle **class imbalance**
- creating **data loaders**, using the **sampler** together with the training set

Data Preparation

```python
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels.reshape(-1, 1)).float()
# Uses index_splitter to generate indices
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]
# Builds different composers because of data augmentation on
training set
train_composer = Compose([RandomHorizontalFlip(p=.5),
                          Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor, transform=train_composer
)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor, transform=val_composer
)
# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)
# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16, sampler=sampler
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

We're *almost* finished with the data preparation section! There is one last thing to discuss…

Pixels as Features

So far, we've been handling our data as either *PIL images* or *three-dimensional tensors* (CHW) with shape (1, 5, 5). It is also possible to consider **each pixel and channel as an individual feature** by **flattening** the pixels with a `Flatten` layer. Let's take one mini-batch of images from our training set to illustrate how it works:

```
dummy_xs, dummy_ys = next(iter(train_loader))
dummy_xs.shape
```

Output

```
torch.Size([16, 1, 5, 5])
```

Our dummy mini-batch has 16 images, one channel each, dimensions five-by-five pixels. What if we *flatten* this mini-batch?

```
flattener = nn.Flatten()
dummy_xs_flat = flattener(dummy_xs)

print(dummy_xs_flat.shape)
print(dummy_xs_flat[0])
```

Output

```
torch.Size([16, 25])
tensor([-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,  1., -1.,
-1., -1.,
        -1., -1.,  1., -1., -1., -1., -1., -1.,  1., -1., -1.])
```

By default, it preserves the first dimension such that we **keep the number of data points** in the mini-batch, but it collapses the remaining dimensions. If we look at the first element of the flattened mini-batch, we find a long tensor with 25 (1 x 5 x 5) elements in it. If our images had three channels, the tensor would be 75 (3 x 5 x 5) elements long.

"*Don't we **lose information** when we **flatten pixels**?*"

Sure we do! And that's why **Convolutional Neural Networks (CNNs)**, which we'll cover in the next chapter, are so successful: they **do not** lose this information. But, for now, let's do it *really old style* and flatten the pixels. It will make it much simpler to illustrate a couple of other concepts before we get to the fancier CNNs.

Now, assuming the flattened version of our dataset, I ask you:

"*How is this different from the datasets we worked with in previous chapters?*"

It isn't! Before, our data points were tensors with one or two elements in it, that is, one or two features. Now, our data points are tensors with 25 elements in it, each one corresponding to a pixel/channel in the original image, as if they were 25 "features".

And, since it is *not different*, apart from the number of features, we can start from what we already know about defining a model to handle a binary classification task.

Shallow Model

Guess what? It is a **logistic regression**:

$$P(y=1) = \sigma(z) = \sigma(w_0x_0 + w_1x_1 + \cdots + w_{24}x_{24})$$

Equation 4.1 - Logistic Regression

Given **25 features**, $\boldsymbol{x_0}$ through $\boldsymbol{x_{24}}$, each one corresponding to the **value of a pixel** in a given **channel**, the model will fit a **linear regression** such that its outputs are

logits (*z*), which are converted into **probabilities** using a **sigmoid function**.

*"Oh no, not **this** again… where are the **deep** models?"*

Don't worry, this section was named *shallow* model for a reason… in the next one, we'll build a **deeper** model with a *hidden layer* in it - finally!

How does our model look like? Visualization always helps, so here we go:

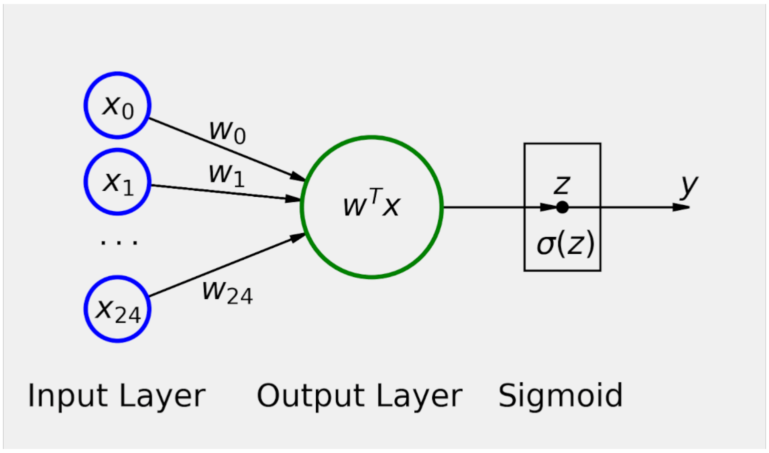


Figure 4.4 - Yet another logistic regression…

*"Wait, where is the **bias**?"*

I am glad you noticed. I **removed it on purpose**! I want to *illustrate* the **difference** between a **shallow** model like this, and a **deeper one**, and it is much easier to work it out if we ditch the bias. Moreover, I would also like to recall the corresponding **notation** for our model, since I am planning on using this notation to illustrate that point.

Notation

The vectorized representations of the **weights (*W*)** and **features (*X*)** are:

$$W = \underset{(25\times 1)}{\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{24} \end{bmatrix}} ; X = \underset{(25\times 1)}{\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}}$$

The **logits (*z*)**, as shown in Figure 4.4 are given by the expression below:

$$\begin{aligned} z = W^T \cdot X &= \underset{(1\times 25)}{\begin{bmatrix} - & w^T & - \end{bmatrix}} \cdot \underset{(25\times 1)}{\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}} = \underset{(1\times 25)}{\begin{bmatrix} w_0 & w_1 & \cdots & w_{24} \end{bmatrix}} \cdot \underset{(25\times 1)}{\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}} \\ &= w_0x_0 + w_1x_1 + \cdots + w_{24}x_{24} \end{aligned}$$

Model Configuration

As usual, we only need to define a **model**, an appropriate **loss function**, and an **optimizer**. Since we have **five-by-five single-channel images** as inputs now, we need to **flatten** them first, so they can be proper inputs to our **linear layer (without bias)**. We will keep using the SGD optimizer with a learning rate of 0.1 for now.

This is what the model configuration looks like for our classification problem:

Model Configuration

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(17)
# Now we can create a model
model_logistic = nn.Sequential()
model_logistic.add_module('flatten', nn.Flatten())
model_logistic.add_module('output', nn.Linear(25, 1, bias=False))
model_logistic.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
optimizer_logistic = optim.SGD(
    model_logistic.parameters(), lr=lr
)
# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

Model Training

Let's train our model for 100 epochs using the StepByStep class and visualize the losses:

Model Training

```
n_epochs = 100

sbs_logistic = StepByStep(
    model_logistic, binary_loss_fn, optimizer_logistic
)
sbs_logistic.set_loaders(train_loader, val_loader)
sbs_logistic.train(n_epochs)
```

```
fig = sbs_logistic.plot_losses()
```

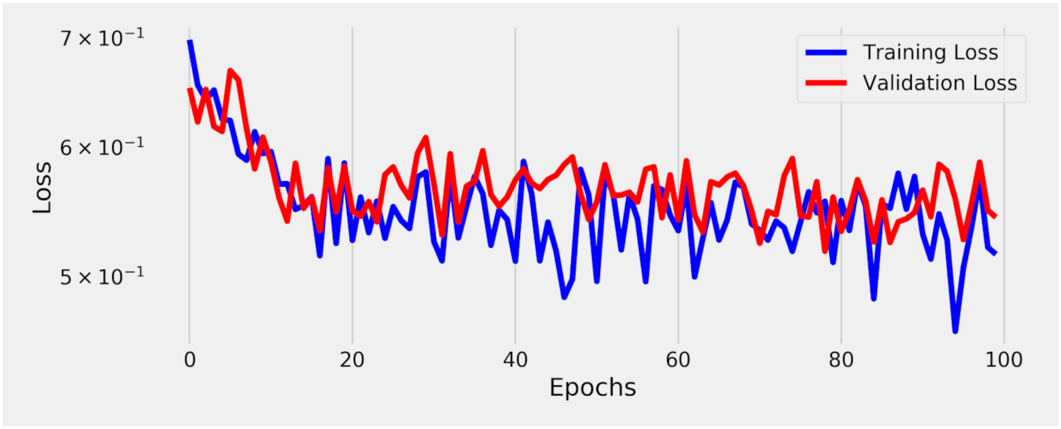


Figure 4.5 - Losses for the logistic regression model

Awful, right? It seems our model is barely learning anything! Maybe a **deeper** model can do better.

Deep-ish Model

There we go, let's add not one, but **two hidden layers** to our model and make it **deep-ish**. We still start with a `Flatten` layer, and the last part of our model still is a `Sigmoid`, but there are **two extra `Linear` layers** before the already existing output layer.

Let's visualize it:

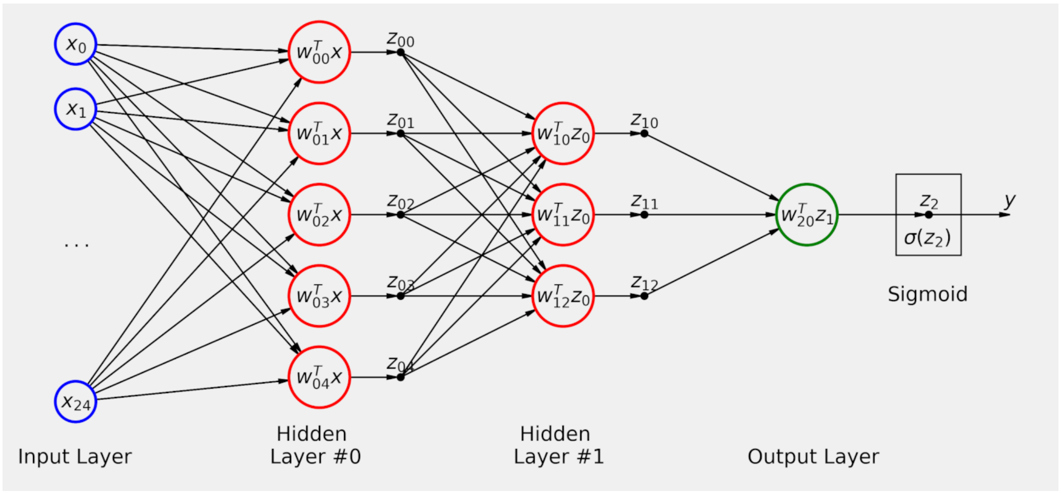


Figure 4.6 - Deep-ish model

By the way, in the figure above, the subscripts for both ***w*** and ***z*** represent the zero-based indices for layer and unit: in the output layer, for instance, ***w***$_{20}$ represents the weights corresponding to the first unit (#0) of the third layer (#2).

What's happening here? Let's work out the **forward pass**, that is, the path **from inputs (*x*) to output (*y*)**:

1. an image is **flattened** to a tensor with **25 features**, from ***x***$_0$ to ***x***$_{24}$ (not depicted in the figure above)
2. the 25 **features** are **forwarded** to each one of the **five units in Hidden Layer #0**
3. each unit in Hidden Layer #0 use its **weights**, from ***w***$_{00}$ to ***w***$_{04}$, and the **features** from the Input Layer to compute its corresponding **outputs**, from ***z***$_{00}$ to ***z***$_{04}$
4. the **outputs of Hidden Layer #0** are **forwarded** to each one of the **three units in Hidden Layer #1** (in a way, the outputs of Hidden Layer #0 work as if they were *features* to the Hidden Layer #1)
5. each unit in Hidden Layer #1 uses its **weights**, from ***w***$_{10}$ to ***w***$_{12}$, and the ***z***$_0$ values from the preceding hidden layer to compute its corresponding **outputs**, from ***z***$_{10}$ to ***z***$_{12}$
6. the **outputs of Hidden Layer #1** are **forwarded** to the **single unit in the output**

layer (again, the outputs of Hidden Layer #1 work as if they were *features* to the Output Layer)

7. the unit in the Output Layer uses its **weights (*w_{20}*)**, and the ***z_1*** values from the preceding hidden layer to compute its corresponding **output (*z_2*)**
8. ***z_2*** is a **logit**, which is converted to a **probability** using a **sigmoid function**

There are a couple of things to highlight:

- **all units** in the hidden layers, and the one in the output layer, take a set of inputs (***x*** or ***z***) and perform the same operation (***w^Tx*** or ***w^Tz***, each using its own weights, of course), producing an **output** (***z***)
- in the hidden layers, these operations are **exactly** like the logistic regression models we used so far, up to the point where the logistic regression produced a **logit**
- it is perfectly fine to think of the **outputs of one layer** as **features of the next layer**; actually, this is at the heart of the **transfer learning** technique we'll see in Chapter 7
- for a binary classification problem, the **output layer** is a **logistic regression**, where the "*features*" are the outputs produced by the previous hidden layer

Not *so* complicated, right? It actually seems like a natural extension of the logistic regression. Let's see how it performs in practice.

Model Configuration

We can easily translate the model depicted above to code:

Model Configuration

```python
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(17)
# Now we can create a model
model_nn = nn.Sequential()
model_nn.add_module('flatten', nn.Flatten())
model_nn.add_module('hidden0', nn.Linear(25, 5, bias=False))
model_nn.add_module('hidden1', nn.Linear(5, 3, bias=False))
model_nn.add_module('output', nn.Linear(3, 1, bias=False))
model_nn.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
optimizer_nn = optim.SGD(model_nn.parameters(), lr=lr)

# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

I've kept the names of the modules consistent with the captions in the figure, so it is easier to follow. The rest of the code should be already familiar to you.

Model Training

Let's train our new, deep-ish, model for 100 epochs using the `StepByStep` class and visualize the losses:

Model Training

```python
n_epochs = 100

sbs_nn = StepByStep(model_nn, binary_loss_fn, optimizer_nn)
sbs_nn.set_loaders(train_loader, val_loader)
sbs_nn.train(n_epochs)
```

```
fig = sbs_nn.plot_losses()
```

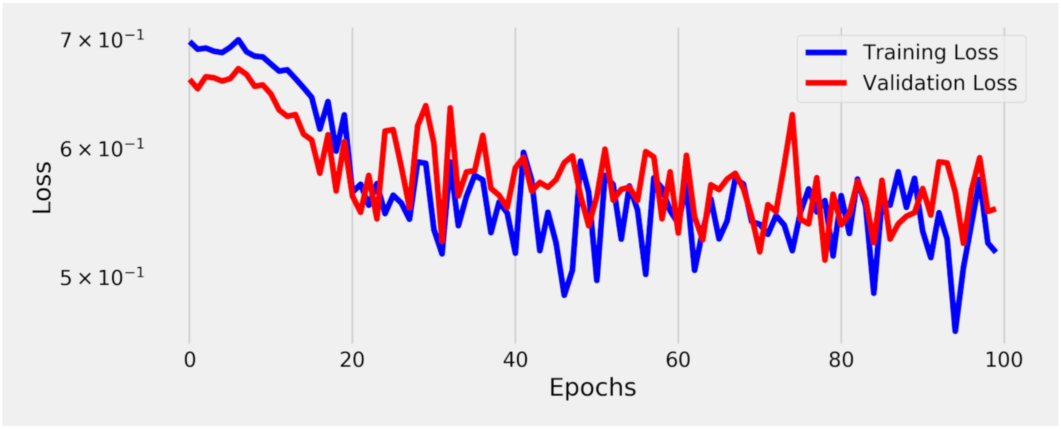


Figure 4.7 - Losses for deep-ish model

Well, that **does not** look good at all! It seems **even worse than the logistic regression**. Or is it? Let's plot them both on the same chart to more easily compare them:

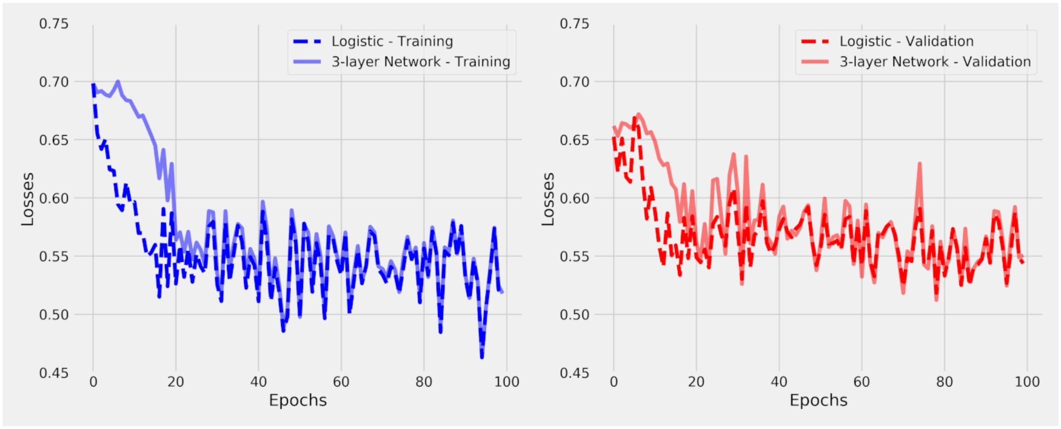


Figure 4.8 - Comparing losses of shallow and deep-ish models

"How could it be? They are... the ***same****?"*

Apparently, the deep-ish model is neither better nor worse, it is **unbelievably**

similar. There's got to be **something wrong**, after all, a deeper model **should** perform, if not *better*, at least, *differently* from a plain logistic regression.

"*What are we missing?*"

We are missing the **activation functions**!

An activation function is a **nonlinear function** that **transforms the outputs of the hidden layers**, in a similar way the **sigmoid function transforms the logits** in the output layer.

Actually, the **sigmoid is one of many activation functions**. There are others, like the hyperbolic-tangent (**tanh**) and the Rectified Linear Unit (**ReLU**).

A deeper model **without activation functions** in its hidden layers is **no better** than a **linear or logistic regression**. That's what I wanted to illustrate with the two models we've trained, the shallow and the deep. That's why I **removed the bias** in both models too: it makes the comparison more straightforward.

Show Me the Math!

This subsection is **optional**. If you're curious to understand, using matrix multiplications, **why** our deep-ish model is **equivalent** to a logistic regression, check the sequence of equations below.

The **deep-ish** model is **above the line**, each **row** corresponding to a **layer**. The data **flows from right to left** (since that's how one multiplies a sequence of matrices), starting with the **25 features on the right** and finishing with a single **logit output on the left**. Looking at each layer (row) individually, it should also be clear that the outputs of a given layer (each row's left-most vector) are the inputs of the next layer, the same way the features are the inputs of the first layer.

$$\begin{array}{lcccccc}
\textit{Hidden \#0} & & & & \underset{(5\times1)}{\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}} = & \underset{(5\times25)}{\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}} \cdot & \underset{(25\times1)}{\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}} \\
\textit{Hidden \#1} & & \underset{(3\times1)}{\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}} = & \underset{(3\times5)}{\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}} \cdot & \underset{(5\times1)}{\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}} & & \\
\textit{Output} & \underset{(1\times1)}{\begin{bmatrix} z_2 \end{bmatrix}} = \underset{(1\times3)}{\begin{bmatrix} - & w_{20}^T & - \end{bmatrix}} \cdot & \underset{(3\times1)}{\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}} & & & & \\
\hline
\textit{substituting } z\textit{'s...} & \underset{(1\times1)}{\begin{bmatrix} z_2 \end{bmatrix}} = \underbrace{\underset{(1\times3)}{\begin{bmatrix} - & w_{20}^T & - \end{bmatrix}}}_{Output\ Layer} & \cdot & \underbrace{\underset{(3\times5)}{\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}}}_{Hidden\ Layer\#1} & \cdot & \underbrace{\underset{(5\times25)}{\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}}}_{Hidden\ Layer\#0} \cdot & \underset{(25\times1)}{\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}} \\
\textit{multiplying...} & = \underbrace{\underset{(1\times25)}{\begin{bmatrix} - & w^T & - \end{bmatrix}}}_{Matrices\ Multiplied} & \cdot & & & & \underset{(25\times1)}{\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}}
\end{array}$$

Equation 4.2 - Equivalence of deep and shallow models

The **first row below the line** shows the sequence of matrices. The **bottom row** shows the **result of the matrix multiplications**. This result is exactly the **same** operation shown in the "*Notation*" subsection of the shallow model, that is, the logistic regression.

In a nutshell, a model with **any number of hidden layers** has an **equivalent** model with **no hidden layers**. Sure, we're not including the bias here because it would make it much harder to illustrate this point.

Show Me the Code!

If equations are not your favorite way of looking at this, let's try using some code. First, we need to get the **weights** for the layers in our deep-ish model. We can use the `weight` attribute of each layer, without forgetting to `detach()` it from the computation graph, so we can freely use them on other operations:

```
w_nn_hidden0 = model_nn.hidden0.weight.detach()
w_nn_hidden1 = model_nn.hidden1.weight.detach()
w_nn_output = model_nn.output.weight.detach()

w_nn_hidden0.shape, w_nn_hidden1.shape, w_nn_output.shape
```

Output

```
(torch.Size([5, 25]), torch.Size([3, 5]), torch.Size([1, 3]))
```

The shapes should match both our model's definition and the weight matrices in the equations above the line.

We can compute the **bottom row**, that is, the **equivalent model** using matrix multiplication (which happens from right to left, as in the equations):

```
w_nn_equiv = w_nn_output @ w_nn_hidden1 @ w_nn_hidden0
w_nn_equiv.shape
```

Output

```
torch.Size([1, 25])
```

"What is @ doing in the expression above?"

It is performing a matrix multiplication, exactly like `torch.mm` does. We could have written the expression above like this:

```
w_nn_equiv = w_nn_output.mm(w_nn_hidden1.mm(w_nn_hidden0))
```

In my opinion, the sequence of operations looks more clear using "@" for matrix multiplication.

Next, we need to compare them to the weights of the shallow model, that is, the logistic regression:

```
w_logistic_output = model_logistic.output.weight.detach()

w_logistic_output.shape
```

Output

```
torch.Size([1, 25])
```

Same shape, as expected. If we compare the values, one by one, we'll find that they are *similar*, but not quite the same. Let's try to grasp the **full picture** by looking at a picture (yes, pun intended!):

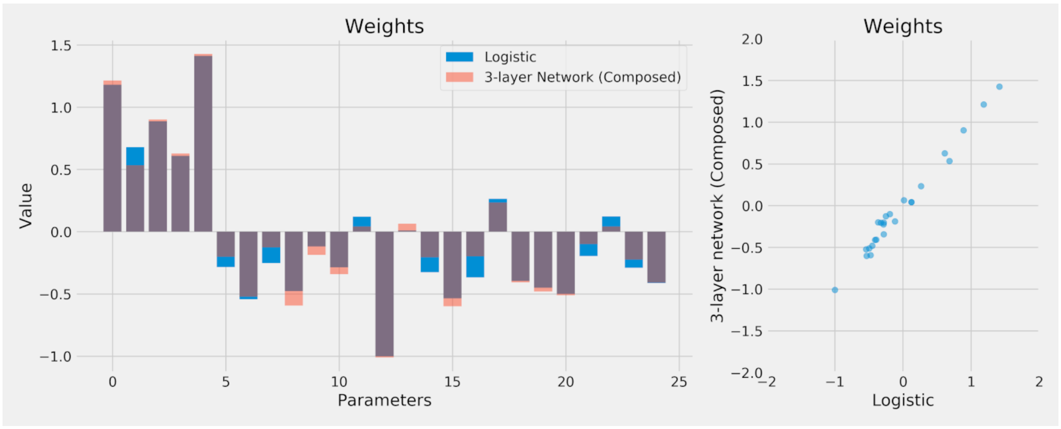


Figure 4.9 - Comparing weights of deep-ish and shallow models

On the left, we plot all the 25 weights/parameters for both models. Even though they are **not quite the same**, the similarity is striking. On the right, we can appreciate that the weights are, indeed, highly correlated.

"If the models are equivalent, how come the weights ended up being slightly different?"

That's a fair question. First, remember that every model is **randomly initialized**. We *did use* the **same random seed**, but this was **not enough** to make both models identical at the beginning. Why not? Simply put, the deep-ish model had **many more weights** to be initialized, so they couldn't have been identical at the start.

It is fairly straightforward that the logistic regression model has 25 weights. But how many weights does the deep-ish model have? We could work it out: 25 features times five units in Hidden Layer #0 (125), plus those five units times three units in Hidden Layer #1 (15), plus the last three weights from Hidden Layer #1 to the Output Layer, adding up to a total of 143.

Or we could just use PyTorch's `numel` instead to return the total **num**ber of **el**ements (clever, right?) in a tensor:

StepByStep Method

```
def count_parameters(self):
    return sum(p.numel()
               for p in self.model.parameters()
               if p.requires_grad)

setattr(StepByStep, 'count_parameters', count_parameters)
```

Even better, let's make it a method of our `StepByStep` class, and take **only gradient-requiring tensors**, so we count only those weights that need to be updated. Right now, it is **all of them**, sure, but it will not necessarily be the case anymore when we use *transfer learning* in Chapter 7.

```
sbs_logistic.count_parameters(), sbs_nn.count_parameters()
```

Output

```
(25, 143)
```

Weights as Pixels

During data preparation, we *flattened* the inputs from five-by-five images to 25-element long tensors. Here is a crazy idea: what if we take some **other tensor** with **25 elements in it** and try to **visualize it as an image**?

We have some perfect candidates for this: the **weights** used by **each unit** in **Hidden Layer #0**. Each unit uses 25 weights since each unit receives values from 25 features. We even have these weights in a variable already:

```
w_nn_hidden0.shape
```

Output

```
torch.Size([5, 25])
```

Five units, 25 weights each. Perfect! We only need to use `view` to make the 25-element long **tensors representing the weights** into two-dimensional tensors (5x5), and visualize them **as if they were images**:

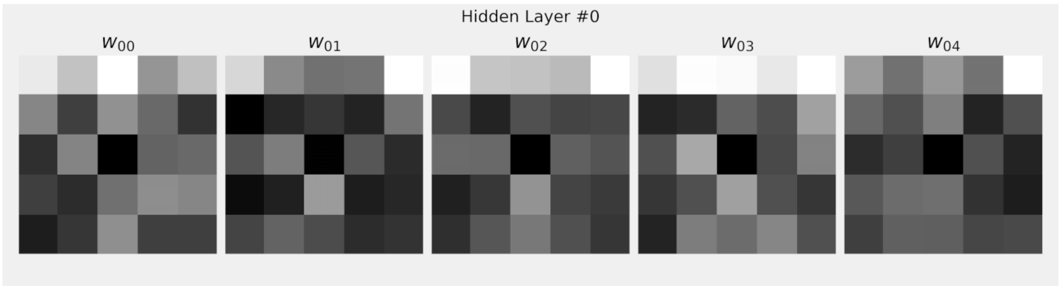


Figure 4.10 - Weights as pixels

"What's the point of doing that?"

Visualizing weights as images is commonplace when using **Convolutional Neural Networks (CNNs)**. These images will be called **filters**, and trained models will likely exhibit more **recognizable characteristics** in its filters. Since our model was poorly trained, it's no wonder the images above are not very informative. Moreover, in our case, these are **not quite** "*filters*", since they have the **same size as the input image**. In CNN-based models, **real filters** cover only **part of the image**. We'll get back to it in the next chapter.

Activation Functions

"*What **are** activation functions?*"

Activation functions are **nonlinear functions**. They either **squash** or **bend** straight lines. They will **break the equivalence** between the *deep-ish* and the *shallow* model.

"*What exactly do you mean by **squash** or **bend** straight lines?*"

Excellent question! Please hold this thought, I will illustrate this in the next chapter, "*Feature Space*". First, let's take a look at some common activation functions. PyTorch has plenty of activation functions to choose from, but we are focusing on five of them only.

Sigmoid

Let's start with the most traditional of the ***activation functions***, the **sigmoid**, which we've already used to transform **logits** into **probabilities**. Nowadays, that is pretty much its only usage, but in the early days of neural networks, one would find it everywhere!

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

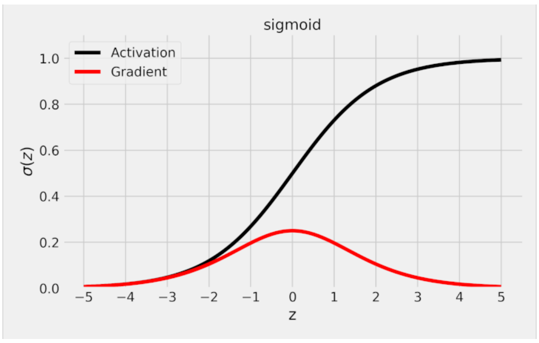


Figure 4.11 - Sigmoid function and its gradient

Let's quickly recap the shape of a **sigmoid**: as you can see in the figure above, a **sigmoid activation function** "**squashes**" its input values (*z*) into the **range (0, 1)** (same range probabilities can take, the reason why it is used in the output layer for binary classification tasks). It is also possible to verify that its **gradient peak value** is only **0.25** (for *z* = 0) and that it gets already close to zero as the absolute value of *z* reaches a value of five.

Also, remember that the activation values of any given layer are the inputs of the following layer and, given the range of the **sigmoid**, the **activation values** are going to be **centered around 0.5**, instead of zero. This means that, even if we normalize our inputs to feed the first layer, it will not be the case anymore for the other layers.

"Why does it matter if the outputs are centered around zero or not?"

In previous chapters, we **standardized** features (zero mean, unit standard deviation) to improve the performance of gradient descent. The same reasoning applies here since the outputs of any given layer are the inputs of the following layer. There is actually more to it, and we'll briefly touch this topic again in the ReLU activation function when talking about the "*internal covariate shift*".

PyTorch has the sigmoid function available in **two** flavors, as we've already seen it in Chapter 3: `torch.sigmoid` and `nn.Sigmoid`. The first one is a simple **function**, and the second one is a full-fledged **class** inherited from `nn.Module`, thus being, for all intents and purposes, a **model on its own**.

```
dummy_z = torch.tensor([-3., 0., 3.])
torch.sigmoid(dummy_z)
```

Output

```
tensor([0.0474, 0.5000, 0.9526])
```

```
nn.Sigmoid()(dummy_z)
```

Output

```
tensor([0.0474, 0.5000, 0.9526])
```

Hyperbolic Tangent (TanH)

The **hyperbolic tangent activation function** was the evolution of the **sigmoid**, as its outputs are values with a **zero mean**, differently from its predecessor.

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

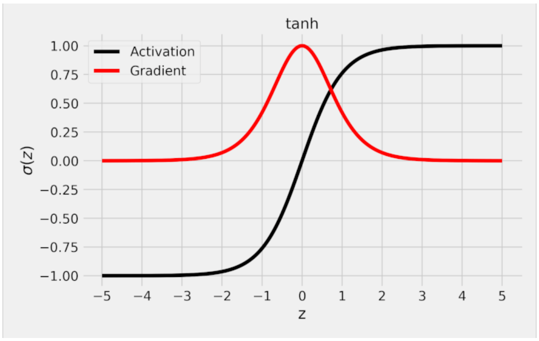


Figure 4.12 - Tanh function and its gradient

As you can see in the figure above, the **tanh activation function** "**squashes**" the input values into the **range (-1, 1)**. Therefore, being **centered at zero**, the activation values are already (somewhat) normalized inputs for the next layer, making the hyperbolic tangent a better activation function than the sigmoid.

Regarding the **gradient**, it has a much **bigger peak value of 1.0** (again, for *z* = 0), but its decrease is even faster, approaching zero to absolute values of *z* as low as three. This is the underlying cause of what is referred to as the problem of **vanishing gradients**, which causes the training of the network to be progressively slower.

Just like the sigmoid function, the hyperbolic tangent also comes in two flavors: `torch.tanh` and `nn.Tanh`.

```
dummy_z = torch.tensor([-3., 0., 3.])
torch.tanh(dummy_z)
```

Output

```
tensor([-0.9951,  0.0000,  0.9951])
```

```
nn.Tanh()(dummy_z)
```

Output

```
tensor([-0.9951,  0.0000,  0.9951])
```

Rectified Linear Unit (ReLU)

Maybe "*squashing*" is not the way to go… what if we *bend* the rules a bit and use an activation function that **bends** (yay, another pun!) the line? The **ReLU** was born like that, and it spawned a whole *family* of similar functions! The **ReLU**, or one of its relatives, is the commonplace choice of activation function nowadays. It addresses the problem of **vanishing gradients** of its two predecessors, while also being the **fastest** to compute gradients for.

$$\sigma(z) = \begin{cases} z, \; if \quad z \geq 0 \\ 0, \; if \quad z < 0 \end{cases}$$

$$or$$

$$\sigma(z) = max(0, z)$$

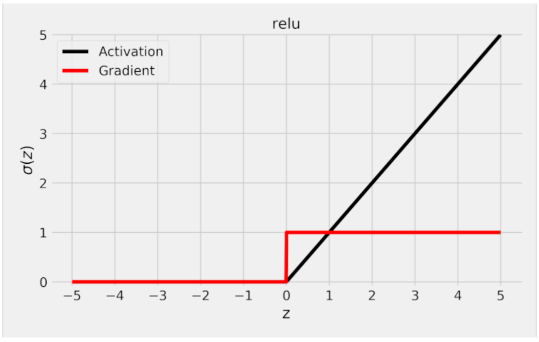


Figure 4.13 - ReLU function and its gradient

As you can see in the figure above, the **ReLU** is a totally different beast: it does not "**squash**" the values into a range — it simply **preserves positive values** and turns all **negative values into zero**.

The upside of using a **ReLU** is that its **gradient** is either **one** (for positive values) or **zero** (for negative values) — **no more vanishing gradients**! This pattern leads to a **faster convergence** of the network. On the other hand, this behavior can also lead to what it is called a **"dead neuron"**, that is, a neuron whose inputs are consistently negative and, therefore, always has an **activation value of zero**. Worse yet, the **gradient is also zero** for negative inputs, meaning the **weights are not updated**. It's like the neuron got **stuck**.

The activation values of the ReLU are obviously **not centered at zero**. Does it make it *worse* than the hyperbolic tangent? It surely **doesn't**, otherwise it would not have become such a popular activation function among practitioners. The ReLU, having its comparably bigger gradients, is able to achieve better and faster results than the other two activation functions *despite* the fact its outputs are not centered at zero.

For deeper and more complex models, this *may* become an issue commonly called "**internal covariate shift**", which is just *fancy* for **different distributions of activation values** in different layers. In general, we would like to have **all layers** producing **activation values** with **similar distributions**, ideally **zero centered and with unit standard deviation**.

To address this issue, you can use **normalization layers**, such as `BatchNorm`. We'll get back to it in Chapter 7.

There are **three** different ways of implementing a **ReLU** in PyTorch: `F.ReLU`, `nn.ReLU`, and `clamp`.

"*What is this **F**? Why isn't it* `torch` *anymore?*"

The **F** stands for **functional**, and it is a common abbreviation for `torch.nn.functional` (as we did in the "*Imports*" at the beginning of this chapter). The functional module has lots of, well… functions, many of them performing the operation of the corresponding module. In this case, there is a `F.ReLU`, which is actually **called** by the forward method of its corresponding module `nn.ReLU`.

Some functions, like `sigmoid` and `tanh`, have been *deprecated* from the functional module and moved to the `torch` module. This is not the case for ReLU and its relatives, though, which remain functional:

```
dummy_z = torch.tensor([-3., 0., 3.])
F.relu(dummy_z)
```

Output

```
tensor([0., 0., 3.])
```

As before, we can still use the full-fledged module version:

```
nn.ReLU()(dummy_z)
```

Output

```
tensor([0., 0., 3.])
```

And, in the particular case of the ReLU, we can use `clamp` to directly **cap *z* at a minimum value of zero**:

```
dummy_z.clamp(min=0)
```

Output

```
tensor([0., 0., 3.])
```

Leaky ReLU

How can you give a "dead neuron" a chance to come back to life? If the underlying problem is the fact that it got **stuck**, we need to **nudge** it a bit. And that's what a **Leaky ReLU** does: for negative inputs, it returns a **tiny activation value** and yields a **tiny gradient**, instead of a **fixed zero** for both. The multiplier for negative values, 0.01, is called the **coefficient of leakage**.

It may not be much, but it gives the neuron a **chance** to get **unstuck**. And it **keeps** the nice properties of the ReLU: bigger gradients and faster convergence.

$$\sigma(z) = \begin{cases} z, \ if \quad z \ge 0 \\ 0.01z, \ if \quad z < 0 \end{cases}$$

$$or$$

$$\sigma(z) = max(0, z) + 0.01min(0, z)$$

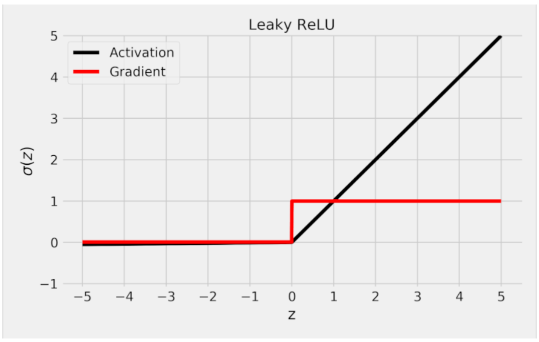


Figure 4.14 - Leaky ReLU function and its gradient

As you can see in the figure above, the **Leaky ReLU** is pretty much the **same as the ReLU**, except for the **tiny**, barely visible, slope on the left-hand side.

Once again, we have two options. Functional (F.leaky_relu):

```
dummy_z = torch.tensor([-3., 0., 3.])
F.leaky_relu(dummy_z, negative_slope=0.01)
```

Output

```
tensor([-0.0300,  0.0000,  3.0000])
```

And module (nn.LeakyReLU):

```
nn.LeakyReLU(negative_slope=0.02)(dummy_z)
```

Output

```
tensor([-0.0600,  0.0000,  3.0000])
```

As you can see, in PyTorch, the coefficient of leakage is called `negative_slope`, with a default value of 0.01.

> *"Any particular reason to choose 0.01 as the coefficient of leakage?"*

Not really, it is just a **small number** that **enables the update of the weights**, which leads to another question: why not trying a **different coefficient**? Sure enough, people started using other coefficients to improve performance.

> *"Maybe the model can **learn** the **coefficient of leakage** too?"*

Sure it can!

Parametric ReLU (PReLU)

The **Parametric ReLU** is the natural evolution of the **Leaky ReLU**: instead of arbitrarily choosing a **coefficient of leakage** (such as 0.01), let's make it a **parameter** (*a*). Hopefully, the model will learn how to prevent dead neurons, or how to bring them back to life (zombie neurons?!). Jokes aside, that's an ingenious solution to the problem.

$$\sigma(z) = \begin{cases} z, \; if \;\; z \geq 0 \\ az, \; if \;\; z < 0 \end{cases}$$

$$or$$

$$\sigma(z) = max(0, z) + a \; min(0, z)$$

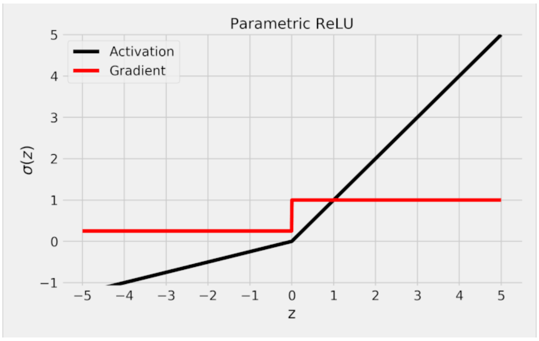


Figure 4.15 - Parametric ReLU function and its gradient

As you can see in the figure above, the **slope** on the left-hand side is much bigger now, 0.25 to be precise, PyTorch's default value for the parameter *a*.

We can set the parameter *a* using the functional version (argument `weight` in `F.prelu`):

```
dummy_z = torch.tensor([-3., 0., 3.])
F.prelu(dummy_z, weight=torch.tensor(0.25))
```

Output

```
tensor([-0.7500,  0.0000,  3.0000])
```

But, in the regular module (`nn.PReLU`), it doesn't make sense to *set it*, since it is going to be **learned**, right? We can still set the **initial value** for it, though:

```
nn.PReLU(init=0.25)(dummy_z)
```

Output

```
tensor([-0.7500,  0.0000,  3.0000], grad_fn=<PreluBackward>)
```

Did you notice the `grad_fn` attribute on the resulting tensor? It shouldn't be a surprise, after all, where there is **learning**, there is a **gradient**.

Deep Model

Now that we've learned that activation functions **break the equivalence** to a shallow model, let's use them to transform our former *deep-ish* model into a **real** deep model. It has the **same architecture** as the previous model, except for the **activation functions** applied to the **outputs of the hidden layers**. Here is the diagram of the updated model:

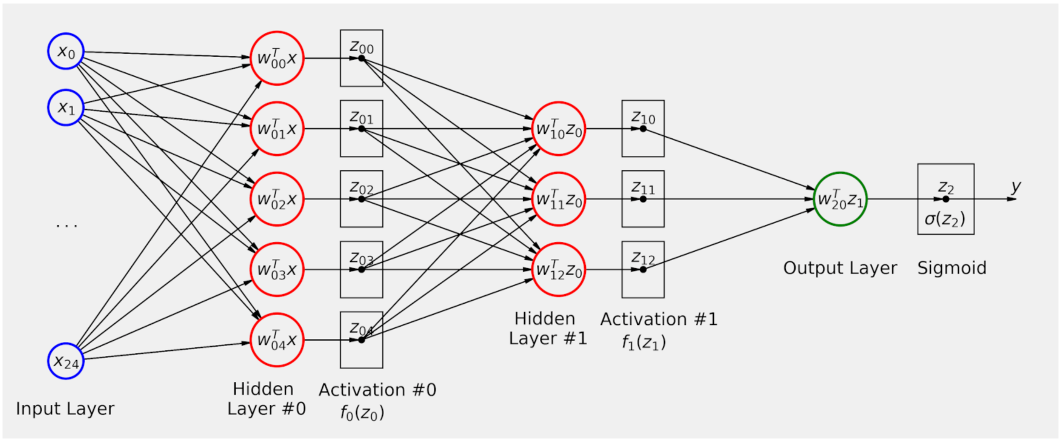


Figure 4.16 - Deep model (for real)

Let's see how it performs now.

Model Configuration

First, we translate the model above to code:

Model Configuration

```python
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(17)
# Now we can create a model
model_relu = nn.Sequential()
model_relu.add_module('flatten', nn.Flatten())
model_relu.add_module('hidden0', nn.Linear(25, 5, bias=False))
model_relu.add_module('activation0', nn.ReLU())
model_relu.add_module('hidden1', nn.Linear(5, 3, bias=False))
model_relu.add_module('activation1', nn.ReLU())
model_relu.add_module('output', nn.Linear(3, 1, bias=False))
model_relu.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer_relu = optim.SGD(model_relu.parameters(), lr=lr)

# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

The chosen activation function is the **Rectified Linear Unit (ReLU)**, one of the most commonly used functions.

We kept the bias out of the picture for the sake of comparing this model to the previous one, which is completely identical except for the activation functions introduced after each hidden layer.

In real problems, as a general rule, you should **keep** `bias=True`.

Model Training

Let's train our new, deep, and activated model for 50 epochs using the `StepByStep` class and visualize the losses:

Model Training

```
n_epochs = 50

sbs_relu = StepByStep(model_relu, binary_loss_fn, optimizer_relu)
sbs_relu.set_loaders(train_loader, val_loader)
sbs_relu.train(n_epochs)
```

```
fig = sbs_relu.plot_losses()
```

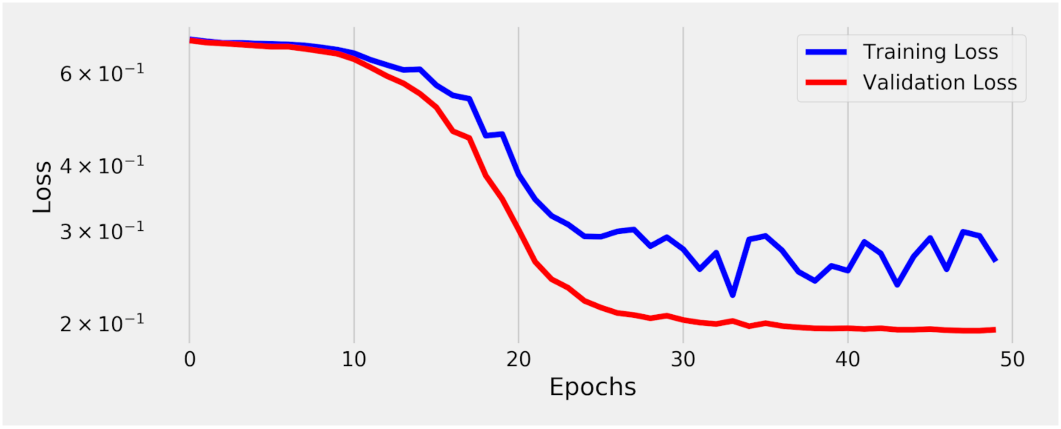


Figure 4.17 - Losses

This is more like it! But, to really grasp the difference made by the activation functions, let's plot all models on the same chart:

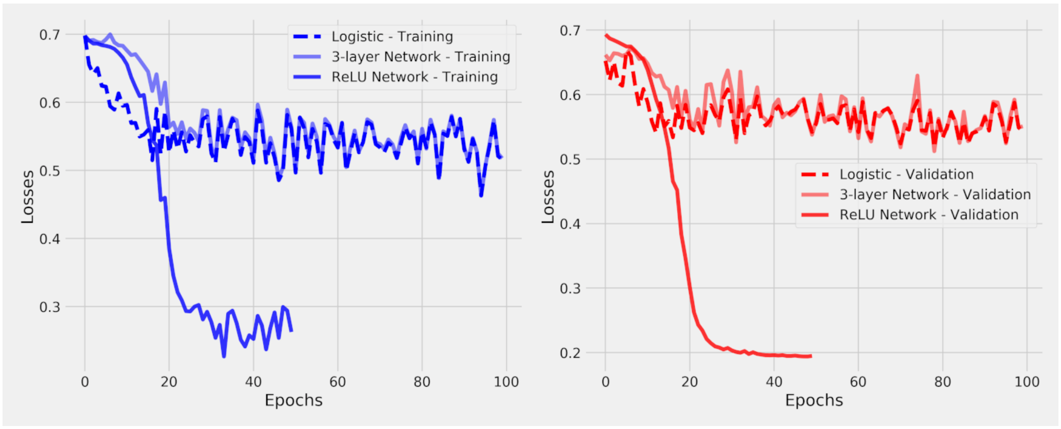


Figure 4.18 - Losses (before and after activations)

It took only a *handful* of epochs for our new model to outperform the previous one. Clearly, this model is **not equivalent** to a logistic regression: it is **much, much better**.

To be completely honest with you, both models are kinda crappy. They perform quite poorly if you look at their accuracies (ranging from 43% to 65% for the validation set). The sole purpose of this exercise is to demonstrate that activation functions, by breaking the equivalence to a logistic regression, are **capable** of achieving better results in minimizing losses.

This particular model also exhibits a **validation loss lower** than the **training loss**, which **isn't** what you generally expect. We've already seen a case like this in Chapter 3: the validation set was **easier** than the training set. The current example is a **bit** more nuanced than that… here is the explanation:

- short version: this is a quirk!
- long version: first, our model is not so great and has a tendency to predict more points in the **positive class** (high **FPR** and **TPR**); second, one of the mini-batches from the validation set has almost all of its points in the positive class, so the **loss is very low**; third, there are **only four mini-batches** in the validation set, so the average loss is easily affected by a single mini-batch.

It's time to ask ourselves two questions:

- **why** is the equivalence to a logistic regression broken?
- **what** exactly are the activation functions doing under the hood?

The first question is answered in the next subsection, "*Show Me the Math Again!*". The other, and more interesting question, is answered in the next chapter, "*Feature Space*".

Show Me the Math Again!

This subsection is also **optional**. If you're curious to understand, using matrix multiplications, **why** our deep model is **not equivalent** to a logistic regression anymore, check the sequence of equations below.

As before, the data **flows from right to left** (since that's how one multiplies a sequence of matrices), starting with the **25 features on the right** and finishing with a single **logit output on the left**. Looking at each layer (row) individually, it should also be clear that the outputs of a given layer (the row's left-most vector) are **transformed by an activation function** before turning into inputs of the next layer.

The **row below the line** shows the result of composing all operations above the line. There is **no way to further simplify the expression** due to the existence of the two **activation functions** (f_0 and f_1). They indeed **break the equivalence** to a logistic regression.

$$Hidden\ \#0 \qquad \underset{(5\times1)}{\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}} = \underset{(5\times25)}{\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}} \cdot \underset{(25\times1)}{\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}}$$

$$Hidden\ \#1 \qquad \underset{(3\times1)}{\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}} = \underset{(3\times5)}{\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}} \cdot \underbrace{f_0 \left(\underset{(5\times1)}{\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}} \right)}_{Activation\ \#0}$$

$$Output \qquad \underset{(1\times1)}{\begin{bmatrix} z_2 \end{bmatrix}} = \underset{(1\times3)}{\begin{bmatrix} - & w_{20}^T & - \end{bmatrix}} \cdot \underbrace{f_1 \left(\underset{(3\times1)}{\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}} \right)}_{Activation\ \#1}$$

$$substituting\ z's\ldots \qquad \underset{(1\times1)}{\begin{bmatrix} z_2 \end{bmatrix}} = \underbrace{\underset{(1\times3)}{\begin{bmatrix} - & w_{20}^T & - \end{bmatrix}}}_{Output\ Layer} \cdot f_1 \left(\underbrace{\underset{(3\times5)}{\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}}}_{Hidden\ Layer\#1} \cdot f_0 \left(\underbrace{\underset{(5\times25)}{\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}}}_{Hidden\ Layer\#0} \cdot \underbrace{\underset{(25\times1)}{\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}}}_{Inputs} \right) \right)$$

Equation 4.3 - Activation functions break the equivalence

Putting It All Together

In this chapter, we have focused mostly on the **data preparation** part of our pipeline. Sure, we got a fancier and deeper model, activation functions, and all, but the model configuration part hasn't changed, and neither the model training.

This should *not* come as a surprise since it is somewhat common knowledge that a data scientist spends more time on **data preparation** than on **actual model training**.

Transformed Dataset

```python
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]

        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

Helper Function #4

```python
def index_splitter(n, splits, seed=13):
    idx = torch.arange(n)
    # Makes the split argument a tensor
    splits_tensor = torch.as_tensor(splits)
    # Finds the correct multiplier, so we don't have
    # to worry about summing up to N (or one)
    multiplier = n / splits_tensor.sum()
    splits_tensor = (multiplier * splits_tensor).long()
    # If there is a difference, throws at the first split
    # so random_split does not complain
    diff = n - splits_tensor.sum()
    splits_tensor[0] += diff
    # Uses PyTorch random_split to split the indices
    torch.manual_seed(seed)
    return random_split(idx, splits_tensor)
```

Helper Function #5

```python
def make_balanced_sampler(y):
    # Computes weights for compensating imbalanced classes
    classes, counts = y.unique(return_counts=True)
    weights = 1.0 / counts.float()
    sample_weights = weights[y.squeeze().long()]
    # Builds sampler with compute weights
    generator = torch.Generator()
    sampler = WeightedRandomSampler(
        weights=sample_weights,
        num_samples=len(sample_weights),
        generator=generator,
        replacement=True
    )
    return sampler
```

Data Preparation

```
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels.reshape(-1, 1)).float()

# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]

# Builds different composers because of data augmentation on
training set
train_composer = Compose([RandomHorizontalFlip(p=.5),
                          Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor, transform=train_composer
)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor, transform=val_composer
)

# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)

# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16, sampler=sampler
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Model Configuration

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model
model_relu = nn.Sequential()
model_relu.add_module('flatten', nn.Flatten())
model_relu.add_module('hidden0', nn.Linear(25, 5, bias=False))
model_relu.add_module('activation0', nn.ReLU())
model_relu.add_module('hidden1', nn.Linear(5, 3, bias=False))
model_relu.add_module('activation1', nn.ReLU())
model_relu.add_module('output', nn.Linear(3, 1, bias=False))
model_relu.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
optimizer_relu = optim.SGD(model_relu.parameters(), lr=lr)

# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

Model Training

```
n_epochs = 50

sbs_relu = StepByStep(model_relu, binary_loss_fn, optimizer_relu)
sbs_relu.set_loaders(train_loader, val_loader)
sbs_relu.train(n_epochs)
```

Recap

Once again, we've covered a lot of ground, from transforming images to the inner workings of activation functions. This is what we've covered:

- generating a **dataset** with 300 tiny, simple, **images**
- understanding the difference between **NCHW** and **NHWC** shapes for data
- learning about **Torchvision**, its built-in datasets, and model architectures
- using Torchvision to **transform images**, from PIL to Tensor, and vice-versa
- performing **data augmentation**, like rotating, cropping, and flipping images
- **normalizing** a dataset of images
- **composing transformations** to use them with `Datasets`
- using **samplers** to perform dataset **splits**, and to handle **imbalanced** datasets
- using **pixels as individual features** to build a shallow model (logistic regression) for image classification
- **trying to make a model deeper** by adding an extra hidden layer
- realizing, using math and code, that the model was still **equivalent to a logistic regression**
- **visualizing the weights** of a hidden layer **as pixels and images**
- learning what an **activation function** does, and going over the most common ones: sigmoid, Tanh, ReLU, leaky ReLU, and PReLU
- using **activation functions to make our model effectively deeper**, observing a huge improvement in loss minimization

Well, that was a *long* one for sure! **Congratulations** on finishing yet another step towards understanding the main concepts used in developing and training deep learning models. In this chapter, we've trained simple models to classify images; in Chapter 5, we'll learn about and use **Convolutional Neural Networks (CNNs)**, and perform **multiclass classification**.

[73] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter04.ipynb

[74] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter04.ipynb

[75] http://yann.lecun.com/exdb/mnist/

[76] https://bit.ly/3kyYvY7

Bonus Chapter
Feature Space

This chapter is *different* from the others. We're not coding anything.

The purpose of this chapter is to **illustrate the effect activation functions have on the feature space**.

> *"What do you mean by 'feature space'?"*

The feature space is the *n-dimensional* space where our features "live". In Chapter 3 we used **two features** for our binary classification, so the feature space there was two-dimensional, a **plane**. In Chapter 4, we used 25 features, so the feature space was 25-dimensional, a **hyper-plane**.

Two-Dimensional Feature Space

Let's **forget** about hyper-planes (**phew!**), and go back to the comfy and **familiar world of two dimensions** for now.

> This is an **entirely new dataset**, it is **not** the image dataset anymore!

Our new, two-dimensional, data has 2,000 points, evenly split into two classes: **red** (negative) and **blue** (positive), neatly organized in two distinct parabolas, as in the figure below:

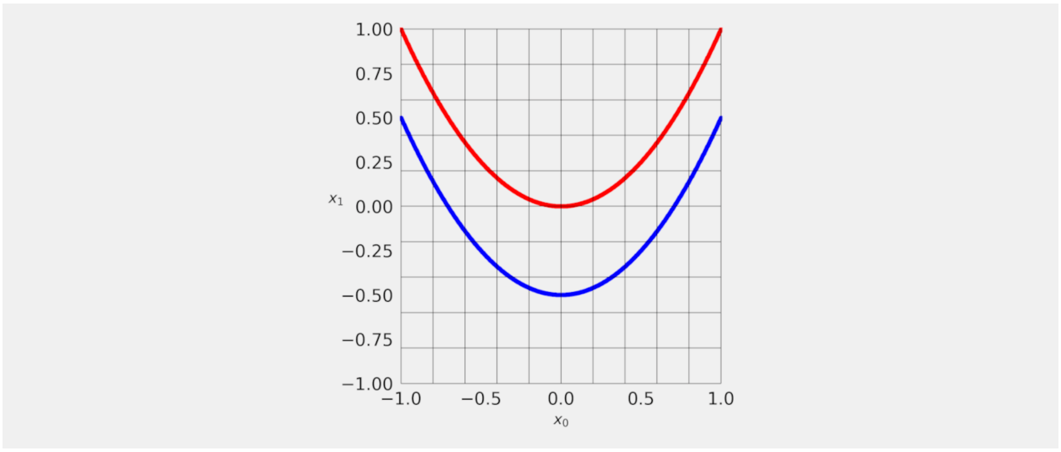


Figure B.1 - Two-dimensional feature space

Our goal is to train a binary classifier that is able to **separate** the two curves, drawing a **decision boundary between them**. In Chapter 3, we figured that the decision boundary for a binary classification problem was a **straight line**.

So I ask you: is it possible to draw a **straight line that separates the parabolas**? Obviously not… but does it mean the problem is unsolvable? Same answer: obviously not. It only means we need to look at the problem from a **different perspective**!

Transformations

In the "*Are My Data Points Separable?*" section of Chapter 3, we talked briefly about dimensionality, the kernel trick in Support Vector Machines, and the separability of data points. In a way, that *was* a *different perspective* already. There, we would **transform** the feature space, mapping it into a **higher-dimensional** one, and hoping to more easily separate the data points.

In Chapter 4, we've already established that, **without activation functions**, a **deeper model** has an **equivalent shallow model** (a logistic regression, in case of a binary classification). This means **we're NOT actually increasing dimensionality**.

You may be thinking: "*how is this different from the example in Chapter 3 where we took the* ***square*** *of the feature values?*"

There **is** a difference: neurons can only perform **affine transformations** in the form $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$. Therefore, an operation like $\boldsymbol{x}^2$, although simple, is still impossible.

Affine Transformations

An **affine transformation** is simply a **linear transformation** (w^Tx), such as **rotating**, **scaling**, **flipping**, or **shearing**, followed by a **translation** (b).

If you want to learn more about it, and *really* understand the role of matrices in linear transformations, including **beautiful visualizations**, make sure to check **3Blue1Brown**'s channel on YouTube, especially the **Essence of linear algebra**[77] series. It is amazing! Seriously, don't miss it!

If you have time to watch the **entire series**, great, but if you need to keep it to a minimum, stick with these three:

- Linear transformations and matrices - Chapter 3[78]
- Matrix multiplication as composition - Chapter 4[79]
- Nonsquare matrices as transformations between dimensions - Chapter 8[80]

This means we need **a different transformation** to be able to **effectively increase dimensionality** and, more importantly, to **twist and turn** the feature space. That's the role of the **activation function**, as you've probably already guessed.

A Two-Dimensional Model

To visualize these effects, we'll have to keep everything in a two-dimensional feature space. Our model will be like this:

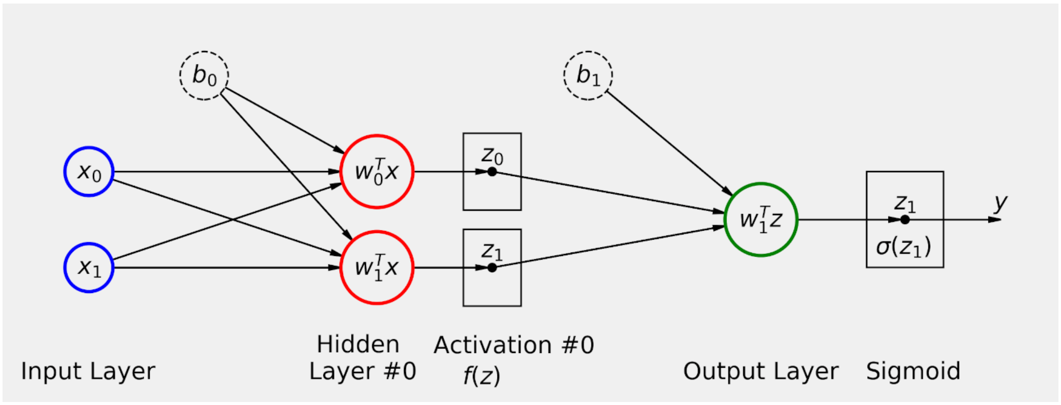


Figure B.2 - Model diagram

It has **one hidden layer** with an **activation function** (and here we can try any of our choosing), and an output layer followed by a sigmoid function, typical of a binary classification. The model above corresponds to the following code:

```
fs_model = nn.Sequential()
fs_model.add_module('hidden', nn.Linear(2, 2))
fs_model.add_module('activation', nn.Sigmoid())
fs_model.add_module('output', nn.Linear(2, 1))
fs_model.add_module('sigmoid', nn.Sigmoid())
```

Let's take a quick look at "*Hidden Layer #0*", which performs an **affine transformation**:

- first, it uses the **weights** to perform a **linear transformation** of the **feature space** (features x_0 and x_1), such that the resulting feature space is a rotated, scaled, maybe flipped, and likely sheared version of the original
- then, it uses the **bias** to **translate** the whole feature space to a **different origin**, resulting in a **transformed feature space** (z_0 and z_1)

The equation below shows the whole operation, from inputs (x) to "*logits*" (z):

$$\underbrace{\underbrace{\begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}}_{Linear\ Transformation} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}}_{Affine\ Transformation} = \underbrace{\underbrace{x_0 \begin{bmatrix} w_{00} \\ w_{10} \end{bmatrix} + x_1 \begin{bmatrix} w_{01} \\ w_{11} \end{bmatrix}}_{Linear\ Transformation} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}}_{Affine\ Transformation} = \begin{bmatrix} w_{00}x_0 + w_{01}x_1 + b_0 \\ w_{10}x_0 + w_{11}x_1 + b_1 \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \end{bmatrix}$$

Equation B.1 - From inputs to "logits" using an affine transformation

It is on top of this *transformed feature space* that the **activation function** will work its magic, **twisting and turning** the feature space **beyond recognition**.

Next, the resulting *activated feature space* will **feed** the **output layer**. But, if we look at the **output layer alone**, it is like a **logistic regression**, right? This means that the output layer will use its inputs (z_0 and z_1) to **draw a decision boundary**.

We can *annotate* the model diagram to, hopefully, make it more clear:

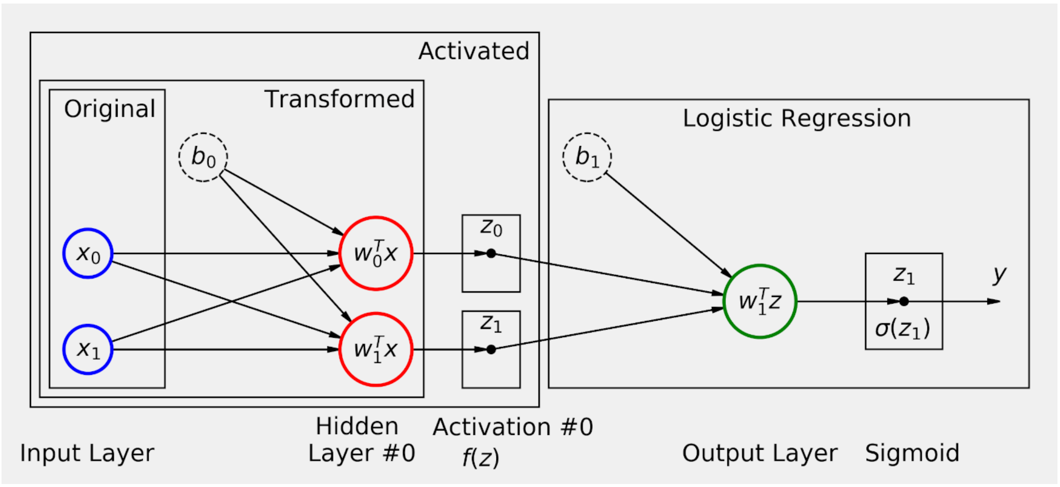


Figure B.3 - Annotated model diagram

Decision Boundary, Activation Style!

And then I have a question for you:

"How does the decision boundary drawn by a logistic regression look like?"

It is a **straight line**! We've seen this in Chapter 3 already. Before proceeding, let's make a quick summary to organize our findings so far:

- in the *original feature space* (x_0 and x_1), depicted in Figure B.1, it is **impossible to draw a straight line** that separates the red and blue curves
- in the *transformed feature space* (z_0 and z_1), it is **still impossible to draw a straight line** that separates both curves, since the **affine transformation preserves parallel lines**
- in the *activated feature space*, ($f(z_0)$ and $f(z_1)$), where f is an activation function of our choice, **it becomes possible to draw a straight line** that separates the red and blue curves

Even better, let's look at the result of a trained model (using a *sigmoid* as activation function *f*):

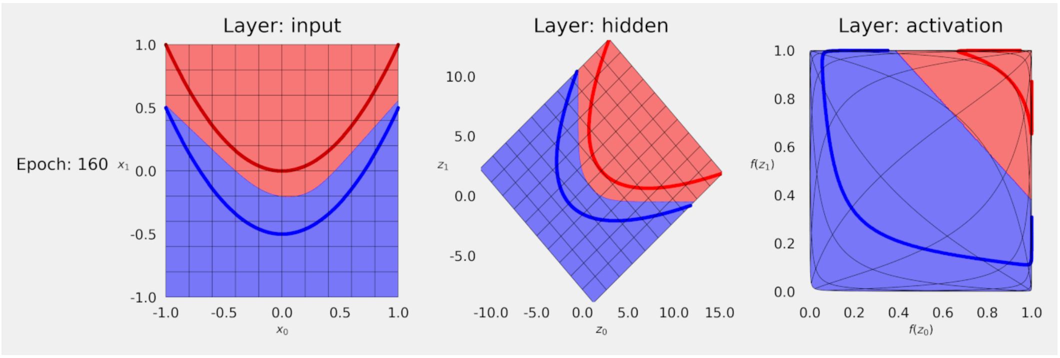


Figure B.4 - From original to activated feature space

On the left, we have the *original feature space*, followed by the *transformed feature space* in the center (corresponding to the output of the "*Hidden Layer #0*", *before* the activation), and the *activated feature space* on the right.

Let's focus on the **right plot**: as promised, the **decision boundary is a straight line**. Now, pay attention to the **grid lines** there: they are **twisted and turned beyond recognition**, as promised. This is the **work of the activation function**.

Moreover, I've plotted the **decision boundary** in the **first two features spaces** as well: they are **curves** now!

It turns out, a **curved decision boundary in the original feature space** corresponds to a **straight line in the activated feature space**.

Cool, right? The first time I looked at those, many years ago, it was a defining moment in my own understanding of the role and importance of activation functions.

I showed you the **trained model** first to make an impact. At the beginning of the training process, the visuals are not nearly as impressive:

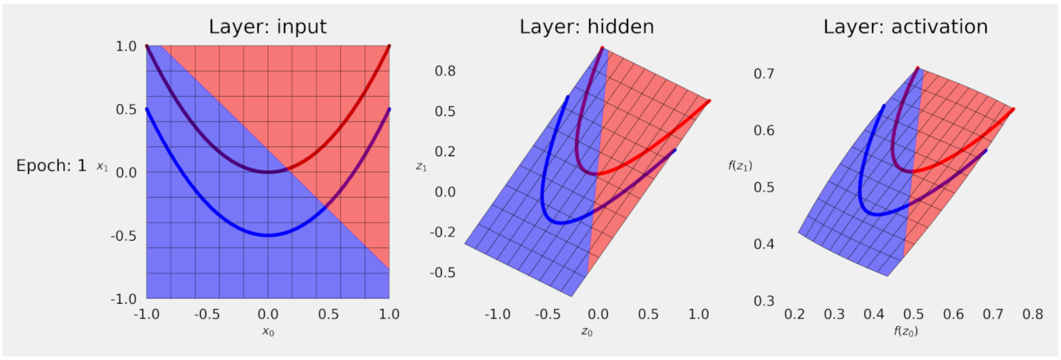


Figure B.5 - In the beginning…

But it gets better as the training progresses (pay attention to the **scale** of the **center** plot):

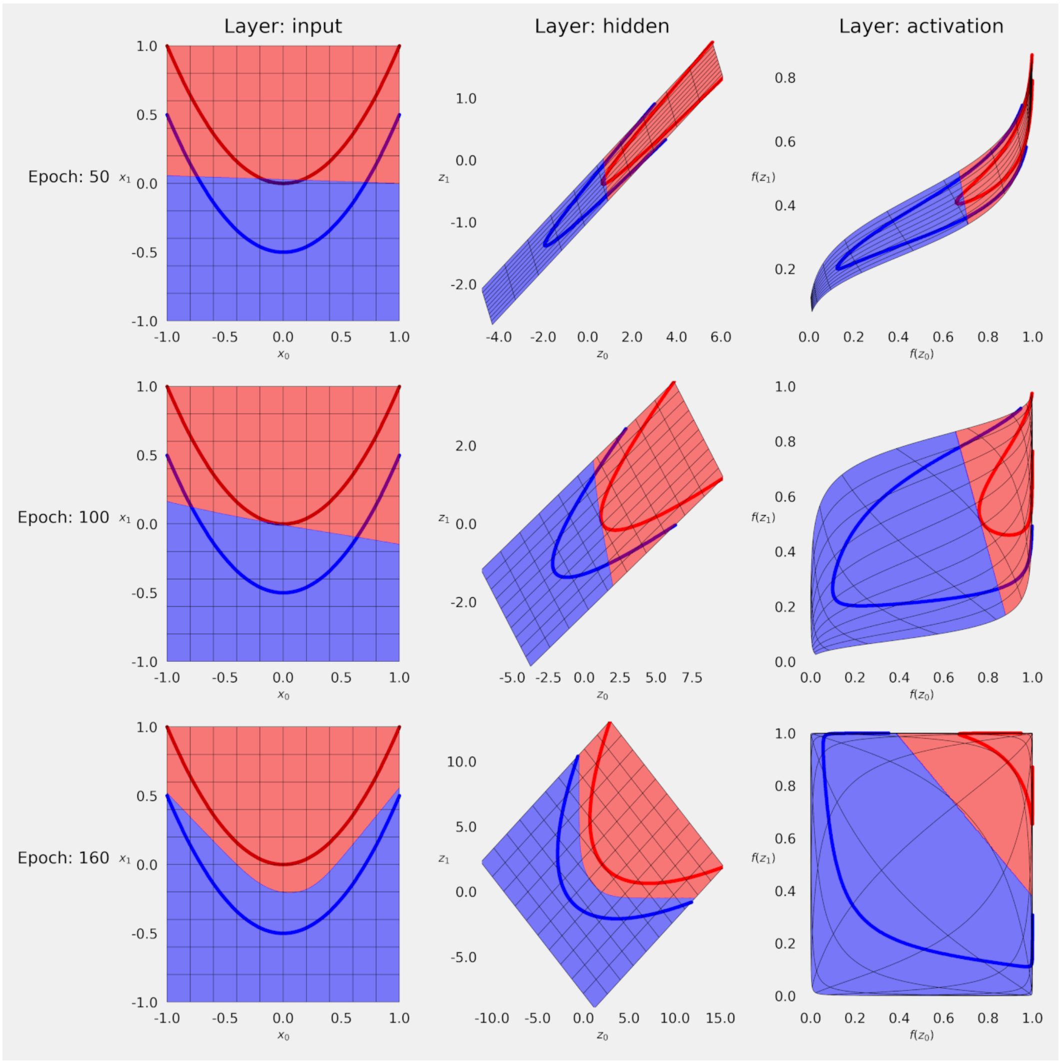


Figure B.6 - After a while…

The solution found by the model was to **rotate it to the right** and **scale it up** (the linear transformation), and then **translate it to the right and up** (making it an affine transformation). That was achieved by **Hidden Layer #0**. Then the **sigmoid activation function** turned that transformed feature space into these **oddly shaped** figures on the right column. In the final plot, the resulting activated feature space looks like it was "*zoomed in*" in its center as if we were looking at it through a magnifying glass.

Now, notice the **range** in the final plot: it is restricted to the (0, 1) interval. That's the range of the **sigmoid**. What if we try a **different activation function**?

More Functions, More Boundaries

Let's try the **hyperbolic tangent**:

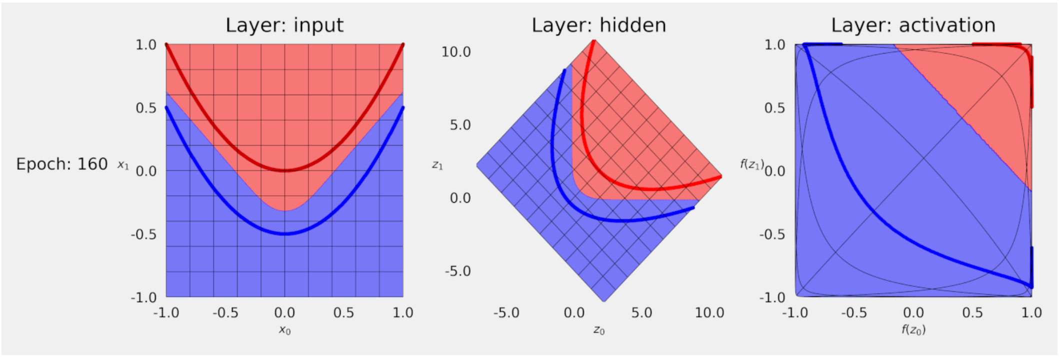


Figure B.7 - Activated feature space - Tanh

It is actually quite similar… especially the *transformed feature space* of the hidden layer. The **range** of the *activated feature space* is different though: it is restricted to the (-1, 1) interval, corresponding to the range of the hyperbolic tangent.

What about the famous **ReLU**?

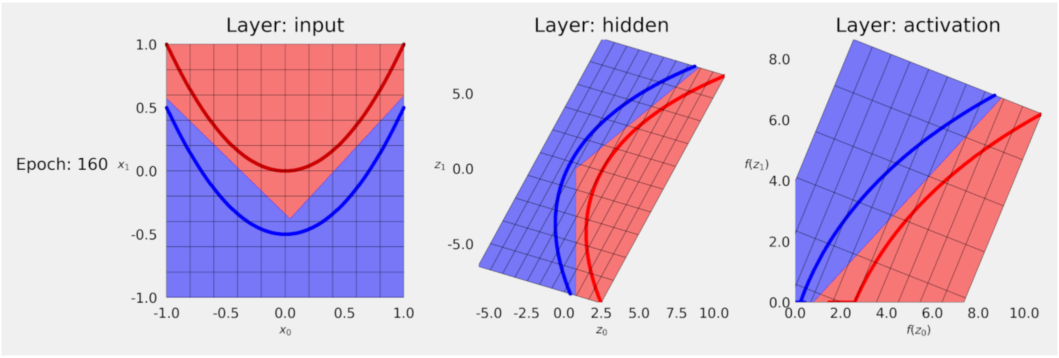


Figure B.8 - Activated feature space - ReLU

OK, now we can clearly see a difference: the **decision boundary on the original**

feature space has a **corner**, a direct consequence of the ReLU's own *corner* when its input is zero. On the right, we can also verify that the range of the activated feature space has only **positive values**, as expected.

Next, let's try the **Parametric ReLU (PReLU)**:

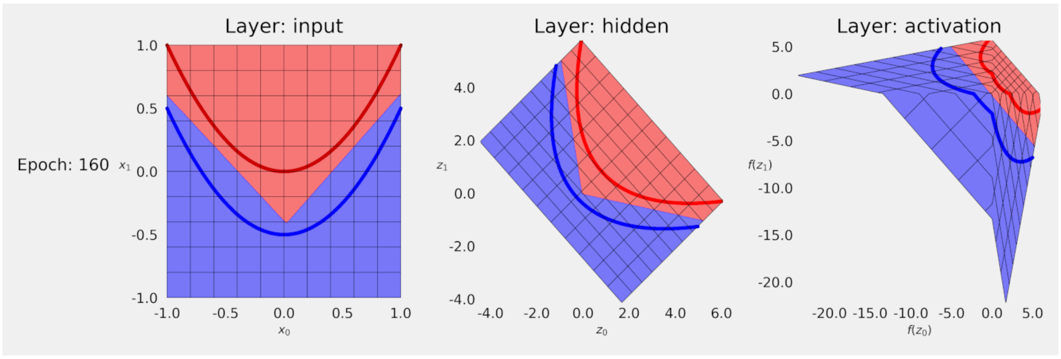


Figure B.9 - Activated feature space - PReLU

This is even more different! Given the fact that the PReLU learns a **slope for the negative values**, effectively **bending** the feature space instead of simply *chopping off* parts of it like the plain ReLU, the result looks like the feature space was **folded** in two different places. I don't know about you, but I find this *really* cool!

So far, all models were trained for 160 epochs, which was about enough training for them to converge to a solution that would completely separate both curves. This seems quite a lot of epochs to solve a rather simple problem, right? But, keep in mind what we discussed in Chapter 3: increasing dimensionality makes it easier to separate the classes. So, we're actually imposing a severe restriction on these models by **keeping it two-dimensional** (two units in the hidden layer) and **performing only one transformation** (only one hidden layer).

Let's cut our models some slack and give them more power…

More Layers, More Boundaries

One option to give a model more power is to **make it deeper**. We can make it deeper, while **keeping it strictly two-dimensional**, by adding **another hidden layer** with two units. It looks like this in the diagram:

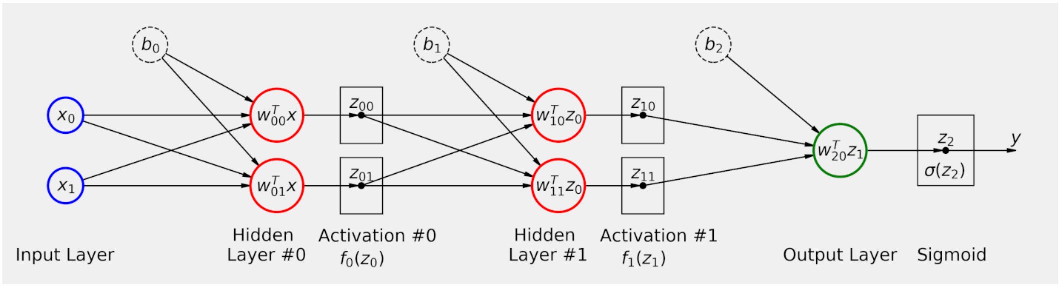


Figure B.10 - Deeper model

A sequence of one or more hidden layers, all with the same size as the input layer, as in the figure above (up to Activation #1), is a typical architecture used to model the **hidden state** in **recurrent neural networks (RNNs)**. We'll get back to it in a later chapter.

And it looks like this in code (we're using a hyperbolic tangent as an activation function because it looks good when visualizing a sequence of transformations):

```python
fs_model = nn.Sequential()
fs_model.add_module('hidden0', nn.Linear(2, 2))
fs_model.add_module('activation0', nn.Tanh())
fs_model.add_module('hidden1', nn.Linear(2, 2))
fs_model.add_module('activation1', nn.Tanh())
fs_model.add_module('output', nn.Linear(2, 1))
fs_model.add_module('sigmoid', nn.Sigmoid())
```

In the model above, the **sigmoid** function **isn't an activation** function: it is there to convert logits into probabilities only.

You may be wondering: "*can I mix different activation functions in the same model?*". It is definitely possible but it is also highly unusual. In general, models are built using the same activation function across all hidden layers. The ReLU or one of its variants are the most common choices because they lead to faster training, while Tanh and Sigmoid activation functions are used in very specific cases (recurrent neural networks, for instance).

But, more importantly, since it can perform **two transformations** now (and activations, obviously), this is how the model is working:

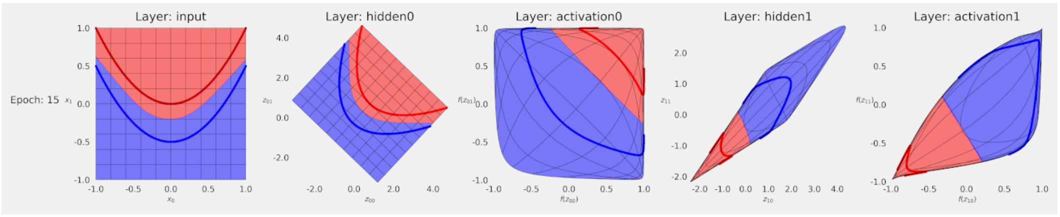


Figure B.11 - Activated feature space - Deeper model

First of all, these plots were built using a model trained for **15 epochs only** (compared to 160 epochs in all previous models). Adding another hidden layer surely makes the model more powerful, thus leading to a satisfactory solution in a much shorter amount of time.

"*Great, let's just make ridiculously deep models and solve everything! Right?*"

Not so fast… as models grow **deeper**, other issues start popping up, like the (in)famous **vanishing gradients** problem. We'll get back to that later. For now, adding one or two extra layers is likely *safe*, but please don't get carried away with it.

More Dimensions, More Boundaries

We can also make a model more powerful by adding **more units** to a hidden layer. By doing this, we're **increasing dimensionality**, that is, mapping our **two-dimensional feature space** into a, say, **10-dimensional feature space** (which we cannot visualize). But we *can* **map it back to two dimensions** in a second hidden layer with the only purpose of taking a peek at it.

I am skipping the diagram, but here is the code:

```
fs_model = nn.Sequential()
fs_model.add_module('hidden0', nn.Linear(2, 10))
fs_model.add_module('activation0', nn.PReLU())
fs_model.add_module('hidden1', nn.Linear(10, 2))
fs_model.add_module('output', nn.Linear(2, 1))
fs_model.add_module('sigmoid', nn.Sigmoid())
```

Its **first hidden layer** has **10 units** now and uses PReLU as an activation function. The **second hidden layer**, however, has **no activation function**: this layer is working as a **projection of 10D into 2D**, such that the **decision boundary** can be visualized in two dimensions.

In practice, this extra hidden layer is **redundant**. Remember, **without an activation function between two layers**, they are **equivalent to a single layer**. We are doing this here with the sole purpose of **visualizing it**.

And here are the results, after training it for **ten epochs** only:

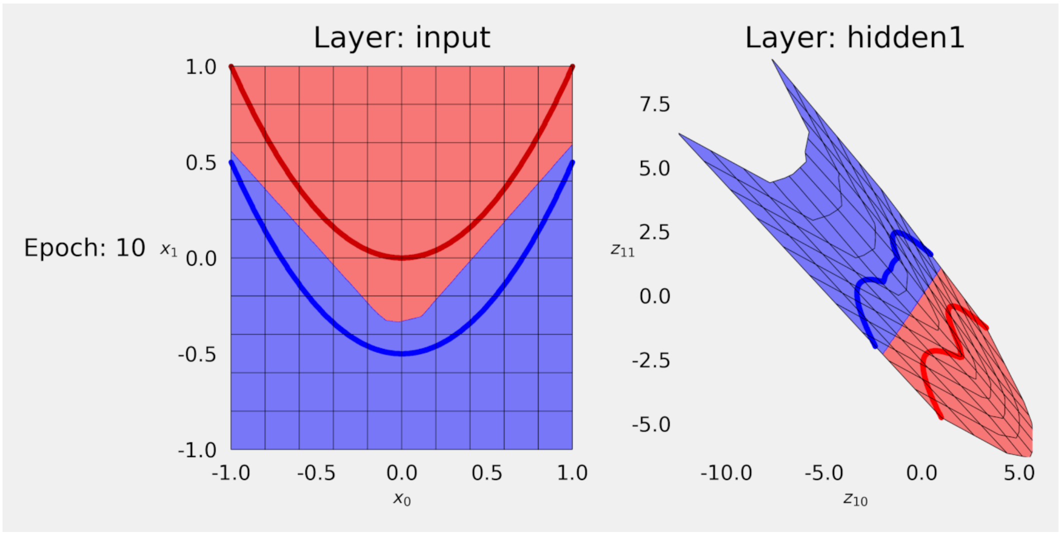


Figure B.12 - Activated feature space - Wider model

By mapping the original feature space into some crazy 10-dimensional one, we make it easier for our model to figure out a way of **separating the data**. Remember, **the more dimensions, the more separable the points are**, as we've seen it in the "*Are My Data Points Separable?*" section in Chapter 3.

Then, by projecting it back into two-dimensions, we can visualize the decision boundary in the modified feature space. The overall shape is more complex as if it had gone through multiple foldings, as a result of the increase in dimensionality.

Personally, this is one of my favorite topics, and it was the subject of my very first blog post: "*Hyper-parameters in Action! Part I - Activation Functions*"[81]. You can also check out some **animations** I built back then for visualizing the training process using different activation functions: sigmoid[82], hyperbolic tangent[83], and ReLU[84].

Recap

And that's enough for our feature space visualization journey! I hope you liked it. This is what we've covered:

- learning what a **feature space** is, and how a **hidden layer** performs **affine**

transformations to modify the feature space

- **visualizing the effect of activation functions** on the feature space
- learning that the **decision boundary** is a **straight line in the activated feature space**, while a **curve in the original** feature space
- visualizing **different decision boundaries** (in original feature space) for **different activation functions**
- making a more powerful model by **making it deeper**
- making a more powerful model by **making it wider**, thus **increasing dimensionality**

Now, let's get back to the **main track**, tackling a **multiclass classification problem** using **Convolutional Neural Networks (CNNs)**.

[77] https://bit.ly/3l5XVkN

[78] https://bit.ly/2QgEmYR

[79] https://bit.ly/34mw0ai

[80] https://bit.ly/3hgfIU6

[81] https://towardsdatascience.com/hyper-parameters-in-action-a524bf5bf1c

[82] https://youtu.be/4RoTHKKRXgE

[83] https://youtu.be/PFNp8_V_Apg

[84] https://youtu.be/Ji_05nOFLE0

Chapter 5
Convolutions

Spoilers

In this chapter, we will:

- understand the **arithmetic of convolutional layers** in detail
- build a model for **multiclass classification**
- understand the role of the **softmax function**
- use **negative log-likelihood** and **cross-entropy** losses
- **visualize filters** learned by our convolutional neural network
- understand and use **hooks** to capture outputs from intermediate layers
- **visualize feature maps** to better understand what's happening inside the model

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 5[85] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[86].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 5's notebook. If not, just click on `Chapter05.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import random
import numpy as np
from PIL import Image

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import Compose, Normalize

from data_generation.image_classification import generate_dataset
from helpers import index_splitter, make_balanced_sampler
from stepbystep.v1 import StepByStep
```

Convolutions

In the last chapter, we talked about **pixels as features**: we considered each pixel as an individual, independent, feature, thus **losing information** while **flattening** the image. We also talked about **weights as pixels**, and how we could interpret the weights used by a neuron as an **image**, or, more specifically, a **filter**.

Now, it is time to take that one step further and learn about **convolutions**. A convolution is "*a mathematical operation on two functions (f and g) that produces a third function (f * g) expressing how the shape of one is modified by the other*"[87]. In image processing, a **convolution matrix** is also called a **kernel** or **filter**. Typical image processing operations, like *blurring*, *sharpening*, *edge detection*, and more are accomplished by performing a **convolution between a kernel and an image**.

Filter / Kernel

Simply put, one defines a **filter** (or *kernel*, but we're sticking with filter here), and **applies** this filter to an image (that is, convolving an image). Usually, the filters are **small square matrices**. The convolution itself is performed by **applying the filter on the image repeatedly**. Let's try a *concrete example* to make it more clear.

We're using a *single-channel image*, and the **most boring filter** ever, the **identity filter**:

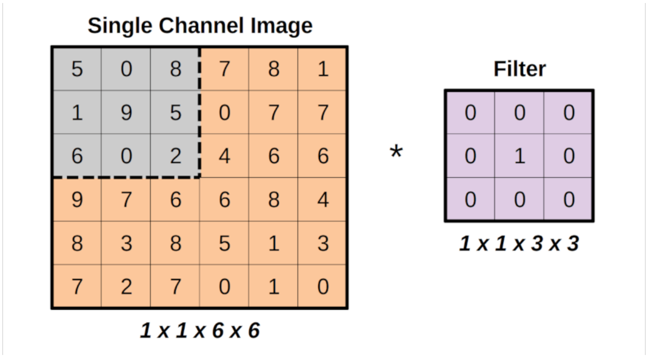


Figure 5.1 - Identity filter

See the **gray region** on the top left corner of the image, which has the **same size as the filter**? That's the **region the filter is being applied to** and it is called the **receptive field**, drawing an analogy to the way human vision works.

Moreover, look at the **shapes** underneath the images: the shapes follow the **NCHW** shape convention used by PyTorch. There is one image, one channel, six by six pixels in size. There is one filter, one channel, three by three pixels in size.

Finally, the **asterisk** is representing the **convolution** operation between the two.

Let's create *Numpy* arrays to follow the operations, after all, everything gets easier to understand in code, right?

```
single = np.array(
    [[[[5, 0, 8, 7, 8, 1],
       [1, 9, 5, 0, 7, 7],
       [6, 0, 2, 4, 6, 6],
       [9, 7, 6, 6, 8, 4],
       [8, 3, 8, 5, 1, 3],
       [7, 2, 7, 0, 1, 0]]]]
)
single.shape
```

Output

```
(1, 1, 6, 6)
```

```
identity = np.array(
    [[[[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]]]]
)
identity.shape
```

Output

```
(1, 1, 3, 3)
```

Convolving

"How does the filter modify the selected region/receptive field?"

It is actually quite simple: it performs an **element-wise multiplication** between the two, **region and filter**, and **adds everything up**. That's it! Let's check it out, zooming in on the selected region:

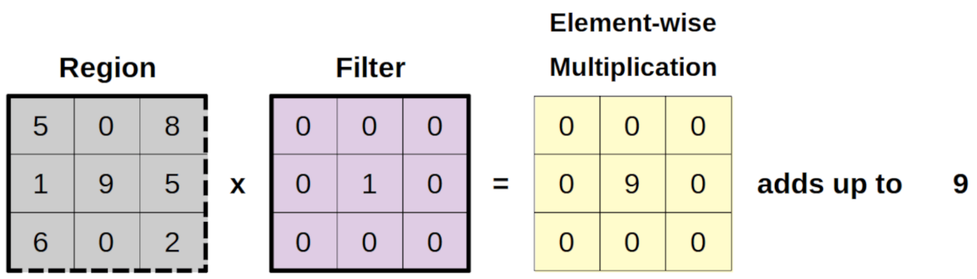


Figure 5.2 - Element-wise multiplication

In code, we have to **slice** the corresponding region (remember the NCHW shape, so we're operating on the *last two* dimensions):

```
region = single[:, :, 0:3, 0:3]
filtered_region = region * identity
total = filtered_region.sum()
total
```

Output

```
9
```

And we're **done** for the **first region** of the image!

"Wait, there are nine pixel values coming in, but only ONE value coming out!"

Good point, you're absolutely right! Doing a convolution produces an image with a **reduced size**. It is easy to see why, if we zoom out back to the full image:

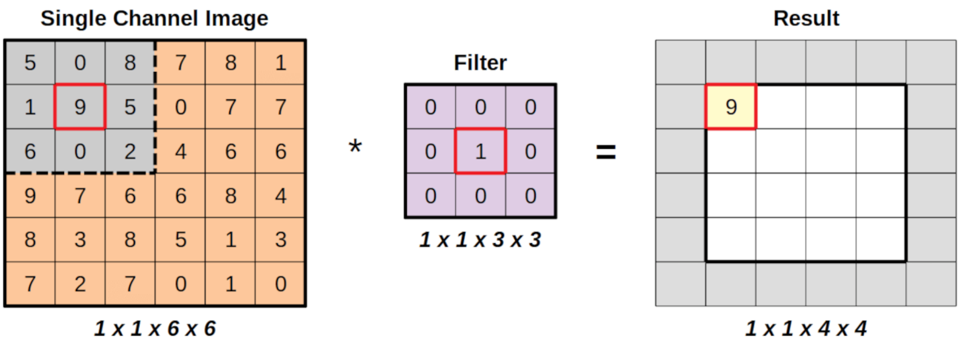


Figure 5.3 - Shrinking images with convolutions

Since the filter gets applied to the gray region, and we're using an **identity filter**, it is fairly straightforward to see it is simply copying the **value in the center of the region**. The remaining values are simply multiplied by zero and do not make to the sum. But even if they did, it **wouldn't change** the fact that the **result of one operation is a single value**.

Moving Around

Next, we **move the region one step to the right**, that is, we change the receptive field, and **apply the filter again**:

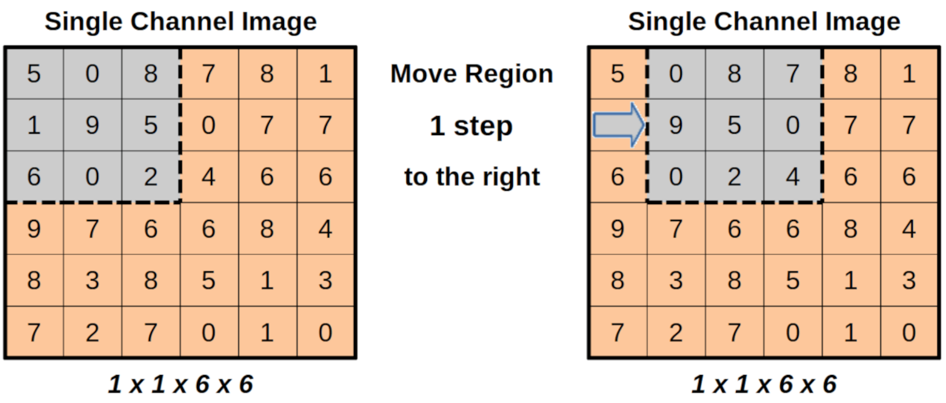


Figure 5.4 - Striding the image, one step at a time

The **size of the movement**, in pixels, is called a **stride**. In our example, the stride is one.

In code, it means we're **changing the slice** of the input image:

```
new_region = single[:, :, 0:3, (0+1):(3+1)]
```

But the operation remains the same: first, an element-wise multiplication, and then adding up the elements of the resulting matrix:

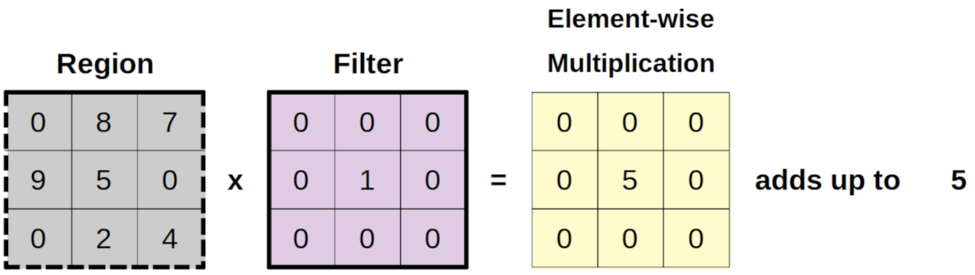


Figure 5.5 - Element-wise multiplication

```
new_filtered_region = new_region * identity
new_total = new_filtered_region.sum()
new_total
```

Output

```
5
```

Great! We have a **second pixel value** to add up to our resulting image:

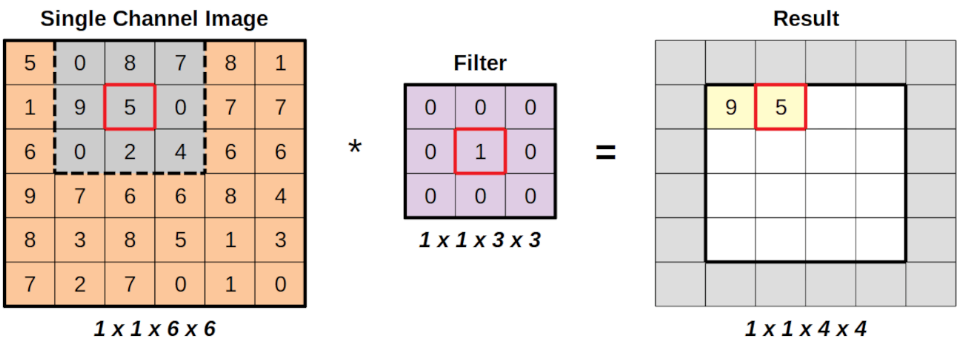


Figure 5.6 - Taking one step to the right

We can **keep moving the gray region to the right** until we can't move it anymore:

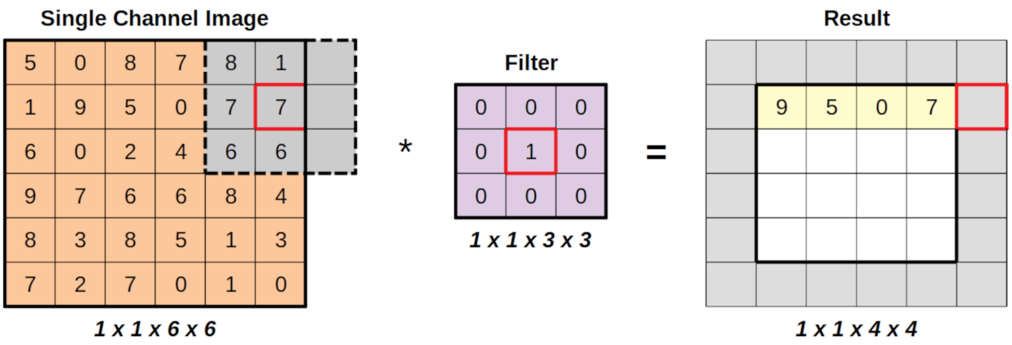


Figure 5.7 - An invalid step!

The **fourth step to the right** will actually place the region **partially outside the input image**. That's a **big no-no**!

```
last_horizontal_region = single[:, :, 0:3, (0+4):(3+4)]
```

The selected region **does not match** the shape of the filter anymore. So, if we try to perform the element-wise multiplication, it fails:

```
last_horizontal_region * identity
```

Output

```
---------------------------------------------------------------
ValueError                      Traceback (most recent call last)
<ipython-input-9-fa0fcce9e228> in <module>
----> 1 last_horizontal_region * identity

ValueError: operands could not be broadcast together with shapes
 (1,1,3,2) (1,1,3,3)
```

Shape

Next, we go back to the **left** side and **move down one step**. If we **repeat the operation**, covering **all valid regions**, we'll end up with a resulting image that is **smaller** (on the right):

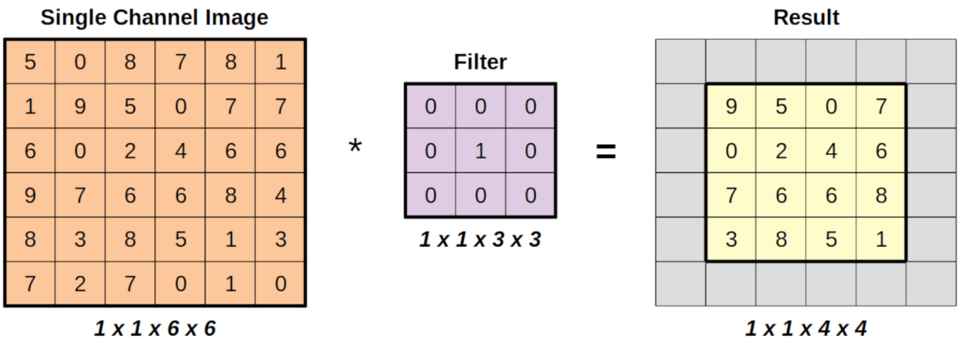


Figure 5.8 - Fully convolved

"How much smaller is it going to be?"

It depends on the **size of the filter**.

The **bigger the filter**, the **smaller the resulting image**.

Since applying **a filter** always produces a **single value**, the reduction is equal to the **filter size minus one**. If the input image has (h_i, w_i) shape (we're disregarding the channel dimension for now), and the filter has (h_f, w_f) shape, the shape of the resulting image is given by:

$$(h_i, w_i) * (h_f, w_f) = (h_i - (h_f - 1), w_i - (w_f - 1))$$

Equation 5.1 - Shape after a convolution

If we assume the filter is a **square matrix** of **size f**, we can simplify the expression above to:

$$(h_i, w_i) * f = (h_i - f + 1, w_i - f + 1)$$

Equation 5.2 - Shape after a convolution (square filter)

Makes sense, right? The filter has its dimensions reduced from (*f*, *f*) to (1, 1), so the operation reduces the original size by (*f* - 1).

*"But I'd like to **keep** the image size, is it possible?"*

Sure it is! **Padding** comes to our rescue in this case. We'll get to that in a couple of sections.

Convolving in PyTorch

Now that we know how a convolution works, let's try it out using PyTorch. First, we need to convert our image and filter to tensors:

```
image = torch.as_tensor(single).float()
kernel_identity = torch.as_tensor(identity).float()
```

Since *kernel* and *filter* are used interchangeably, especially when it comes to arguments of different methods, I am calling the variable `kernel_identity`, even though it is exactly the same identity filter we have used so far.

Just like the activation functions we've seen in Chapter 4, **convolutions** also come in two flavors: **functional** and **module**. There is a fundamental difference between the two, though: the **functional** convolution takes the **kernel/filter as an argument** while the **module** has **weights** to represent the **kernel/filter**.

Let's use the functional convolution, `F.conv2d`, to apply the **identity filter** to our input image (notice we're using `stride=1` since we moved the region around one pixel at a time):

```
convolved = F.conv2d(image, kernel_identity, stride=1)
convolved
```

Output

```
tensor([[[[9., 5., 0., 7.],
          [0., 2., 4., 6.],
          [7., 6., 6., 8.],
          [3., 8., 5., 1.]]]])
```

As expected, we got the same result shown in the previous section. No surprises here.

Now, let's turn our attention to PyTorch's convolution module, `nn.Conv2d`. It has many arguments, let's focus on the first four of them:

- `in_channels`: number of channels of the input image
- `out_channels`: number of channels produced by the convolution
- `kernel_size`: size of the (square) convolution filter/kernel
- `stride`: the size of the movement of the selected region

There is a couple of things to notice here. First, there is **no argument for the kernel/filter itself**, there is only a **`kernel_size`** argument.

The **actual filter**, that is, the **square matrix** used to perform element-wise multiplication is **learned** by the module.

Second, it is possible to produce **multiple channels** as output. It simply means the module is going to learn **multiple filters**. Each filter is going to produce a different **result**, which is being called a **channel** here.

So far, we've been using a **single channel image** as input, and applying **one filter** (size three by three) to it, **moving one pixel** at a time, resulting in **one output/channel**. Let's do it in code:

```
conv = nn.Conv2d(
    in_channels=1, out_channels=1, kernel_size=3, stride=1
)
conv(image)
```

Output

```
tensor([[[[-4.2000, -6.6859, -4.9735, -3.5615],
          [-1.2363,  0.5150, -1.8602, -4.7287],
          [-2.1209, -4.1894, -4.3694, -5.5897],
          [-4.3954, -6.1578, -4.5968, -5.0000]]]],
       grad_fn=<MkldnnConvolutionBackward>)
```

These results are gibberish now (and yours are going to be different than mine) because the convolutional module **randomly** initializes the weights representing the **kernel/filter**.

That’s the whole point of the convolutional module: it will **learn the kernel/filter** on its own.

In traditional computer vision, people would develop different **filters** for different **purposes**: blurring, sharpening, edge detection, and so on.

But, instead of being clever and trying to *manually devise a filter* that does the trick for a given problem, why not *outsource* the **filter definition** to the **neural network** as well? This way. the network will come up with filters that highlight **features** that are relevant to the task at hand.

It’s no surprise that the resulting image shows a `grad_fn` attribute now: it will be used to compute gradients so the network can actually learn how to change the weights representing the filter.

"*Can we tell it to learn* ***multiple filters*** *at once?*"

Sure we can, that’s the role of the `out_channels` argument. If we set it to two, it will generate two (randomly initialized) filters:

```
conv_multiple = nn.Conv2d(
    in_channels=1, out_channels=2, kernel_size=3, stride=1
)
conv_multiple.weight
```

Output

```
Parameter containing:
tensor([[[[ 0.0009, -0.1240, -0.0231],
          [-0.2259, -0.2288, -0.1945],
          [-0.1141, -0.2631,  0.2795]]],


        [[[-0.0662,  0.2868,  0.1039],
          [-0.2823,  0.2307, -0.0917],
          [-0.1278, -0.2767, -0.3314]]]], requires_grad=True)
```

See? There are **two** filters represented by **three-by-three** matrices of weights (your values are going to be different than mine).

Even if you have only **one channel as input**, you can have **many channels as output**.

Spoiler alert: the filters learned by the network are going to show edges, patterns, and even more complex shapes (sometimes resembling faces, for instance). We'll get back to **visualizing those filters** later in this chapter.

We can also **force** a convolutional module to use a **particular filter** by **setting its weights**:

```
with torch.no_grad():
    conv.weight[0] = kernel_identity ①
    conv.bias[0] = 0                 ①
```

① `weight[0]` and `bias[0]` are indexing the first (and only) output channel in this convolutional layer

IMPORTANT: setting the weights is a **strictly no-gradient operation**, so you should **always use the `no_grad` context manager**.

In the code snippet above, we are forcing the module to use the (boring) identity kernel we have used so far. As expected, if we convolve our input image we'll get the familiar result:

```
conv(image)
```

Output

```
tensor([[[[9., 5., 0., 7.],
          [0., 2., 4., 6.],
          [7., 6., 6., 8.],
          [3., 8., 5., 1.]]]], grad_fn=<MkldnnConvolutionBackward>)
```

Setting the weights to get specific filters is at the heart of **transfer learning**. Someone else trained a model and that model learned lots of useful filters, so we **don't have to learn them again**. We can **set the corresponding weights** and go from there. We'll see this in practice in Chapter 7.

Striding

So far, we've been moving the region of interest one pixel at a time: a stride of one. Let's try a **stride of two** for a change and see what happens to the resulting image. I am not reproducing the first step here because it is always the same: the **gray region** centered at the number nine.

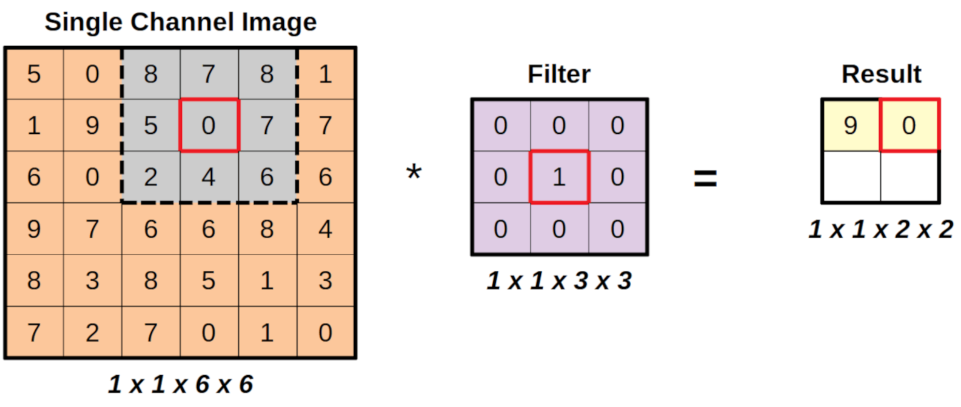


Figure 5.9 - Increasing stride

The **second step**, depicted above, shows the **gray region** moved **two pixels to the right**: that's a **stride of two**.

Moreover, notice that, if we take *another step of two pixels*, the gray region would be placed **partially outside the underlying image**. It was and still is a *big no-no*, so there are only two valid operations while moving horizontally. The same will eventually happen when we move vertically. The first stride of two pixels down is fine, but the second will be, once again, a failed operation.

The resulting image, after the **only four valid operations**, looks like this:

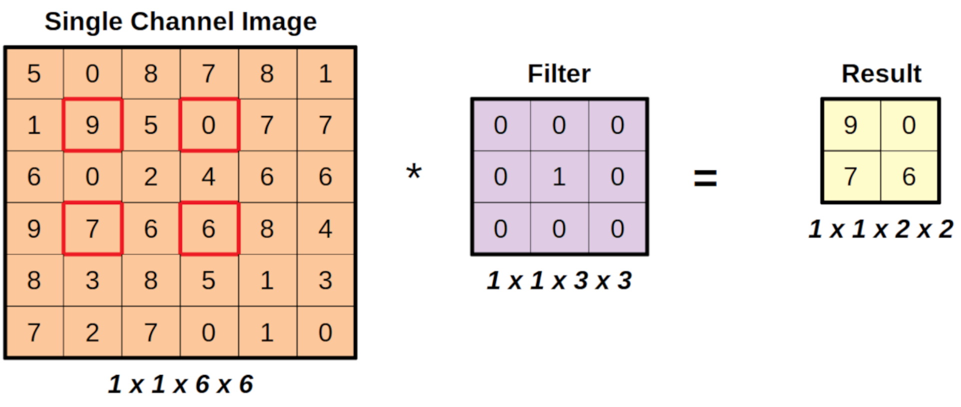


Figure 5.10 - Shrinking even more!

The identity kernel may be *boring*, but it is definitely **useful** to highlight the inner workings of the convolutions. It is crystal clear in the figure above where the pixel values in the resulting image come from.

Also, notice that using a **bigger stride** made the **shape** of the resulting image **even smaller**.

The **bigger the stride**, the **smaller the resulting image**.

Once again, it makes sense: if we are **skipping pixels** in the input image, there are fewer regions of interest to apply the filter to. We can extend our previous formula to include the **stride size (*s*)**:

$$(h_i, w_i) * f = \left(\frac{h_i - f + 1}{s}, \frac{w_i - f + 1}{s}\right)$$

Equation 5.3 - Shape after a convolution with stride

As we've seen before, the **stride** is only an **argument** of the convolution, so let's use PyTorch's functional convolution to double-check the results:

```
convolved_stride2 = F.conv2d(image, kernel_identity, stride=2)
convolved_stride2
```

Output

```
tensor([[[[9., 0.],
          [7., 6.]]]])
```

Cool, it works!

So far, the operations we performed have been **shrinking the images**. What about **restoring them to their original glory, I mean, size**?

Padding

Padding means **stuffing**. We need to **stuff the original image** so it can sustain the "*attack*" on its size.

*"How do I **stuff** an image?"*

Glad you asked! We may simply **add zeros around it**. An image is worth a thousand words in this case:

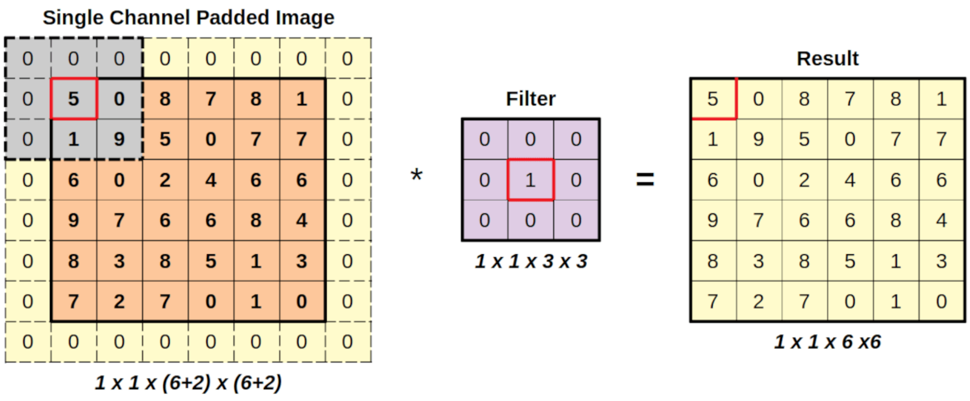


Figure 5.11 - Zero-padded image

See what I mean? By adding columns and rows of zeros around it, we **expand the input image** such that the gray region **starts centered in the actual top left corner** of the input image. This simple trick can be used to **preserve the original size** of the image.

In code, as usual, PyTorch gives us two options: functional (`F.pad`) and module (`nn.ConstantPad2d`). Let's start with the module version this time:

```
constant_padder = nn.ConstantPad2d(padding=1, value=0)
constant_padder(image)
```

Output

```
tensor([[[[0., 0., 0., 0., 0., 0., 0., 0.],
          [0., 5., 0., 8., 7., 8., 1., 0.],
          [0., 1., 9., 5., 0., 7., 7., 0.],
          [0., 6., 0., 2., 4., 6., 6., 0.],
          [0., 9., 7., 6., 6., 8., 4., 0.],
          [0., 8., 3., 8., 5., 1., 3., 0.],
          [0., 7., 2., 7., 0., 1., 0., 0.],
          [0., 0., 0., 0., 0., 0., 0., 0.]]]])
```

There are two arguments: `padding`, for the number of columns and rows to be stuffed in the image; and `value`, for the value that is filling these new columns and rows. One can also do **asymmetric padding**, by specifying a **tuple** in the padding argument representing `(left, right, top, bottom)`. So, if we were to stuff our image on left and right sides only, the argument would go like this: `(1, 1, 0, 0)`.

We can achieve the same result using the functional padding:

```
padded = F.pad(image, pad=(1, 1, 1, 1), mode='constant', value=0)
```

In the functional version, one **must specify the padding as a tuple**. The `value` argument is straightforward, and there is yet **another argument**: **`mode`**, which was set to **constant** to match the module version above.

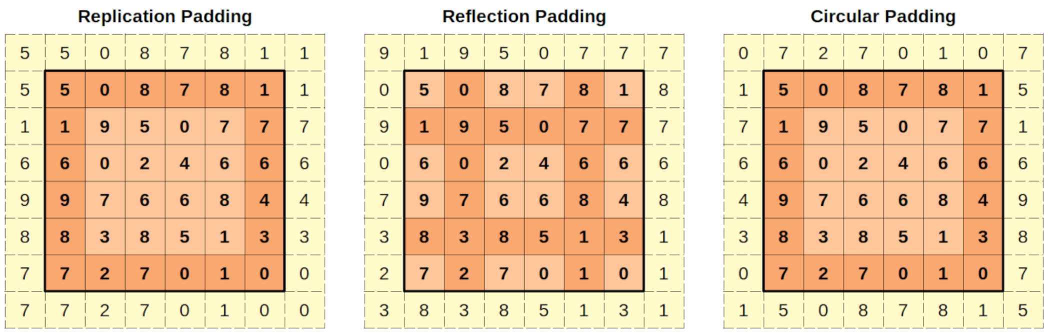
In PyTorch's documentation there is a **note** warning about possible reproducibility issues while using padding:

"*When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on Reproducibility for background.*"

It strikes me a bit odd that such a straightforward operation, of all things, would jeopardize reproducibility. Go figure!

"What are the other available modes?"

There are three other modes: `replicate`, `reflect`, and `circular`. Let's take a look at them, starting with the visualization:


Replication Padding

5	5	0	8	7	8	1	1
5	5	0	8	7	8	1	1
1	1	9	5	0	7	7	7
6	6	0	2	4	6	6	6
9	9	7	6	6	8	4	4
8	8	3	8	5	1	3	3
7	7	2	7	0	1	0	0
7	7	2	7	0	1	0	0

Reflection Padding

9	1	9	5	0	7	7	7
0	5	0	8	7	8	1	8
9	1	9	5	0	7	7	7
0	6	0	2	4	6	6	6
7	9	7	6	6	8	4	8
3	8	3	8	5	1	3	1
2	7	2	7	0	1	0	1
3	8	3	8	5	1	3	1

Circular Padding

0	7	2	7	0	1	0	7
1	5	0	8	7	8	1	5
7	1	9	5	0	7	7	1
6	6	0	2	4	6	6	6
4	9	7	6	6	8	4	9
3	8	3	8	5	1	3	8
0	7	2	7	0	1	0	7
1	5	0	8	7	8	1	5

Figure 5.12 - Paddings modes

In the **replication** padding, the padded pixels will have the **same value** as the **closest real pixel**. The padded corners will have the same value as the real corners. The other columns (left and right) and rows (top and bottom) will **replicate** the corresponding values of the original image. The values used in the replication are in a darker shade of orange.

In PyTorch, one can use the functional form `F.pad` with `mode="replicate"`, or use the module version `nn.ReplicationPad2d`:

```
replication_padder = nn.ReplicationPad2d(padding=1)
replication_padder(image)
```

Output

```
tensor([[[[5., 5., 0., 8., 7., 8., 1., 1.],
          [5., 5., 0., 8., 7., 8., 1., 1.],
          [1., 1., 9., 5., 0., 7., 7., 7.],
          [6., 6., 0., 2., 4., 6., 6., 6.],
          [9., 9., 7., 6., 6., 8., 4., 4.],
          [8., 8., 3., 8., 5., 1., 3., 3.],
          [7., 7., 2., 7., 0., 1., 0., 0.],
          [7., 7., 2., 7., 0., 1., 0., 0.]]]])
```

In the **reflection** padding, it gets a bit trickier. It is like the outer columns and rows are used as axes for the reflection. So, the **left padded column** (forget about the corners for now) will **reflect** the **second column** (since the first column is the axis of reflection). The same reasoning goes for the right padded column. Similarly, the **top padded row** will **reflect** the **second row** (since the first row is the axis of reflection), and the same reasoning goes for the bottom padded row. The values used in the reflection are in a darker shade of orange. The **corners** will have the same values as the **intersection of the reflected rows and columns of the original image**. Hopefully, the image can convey the idea better than my words.

In PyTorch, you can use the functional form `F.pad` with `mode="reflect"`, or use the module version `nn.ReflectionPad2d`:

```
reflection_padder = nn.ReflectionPad2d(padding=1)
reflection_padder(image)
```

Output

```
tensor([[[[9., 1., 9., 5., 0., 7., 7., 7.],
          [0., 5., 0., 8., 7., 8., 1., 8.],
          [9., 1., 9., 5., 0., 7., 7., 7.],
          [0., 6., 0., 2., 4., 6., 6., 6.],
          [7., 9., 7., 6., 6., 8., 4., 8.],
          [3., 8., 3., 8., 5., 1., 3., 1.],
          [2., 7., 2., 7., 0., 1., 0., 1.],
          [3., 8., 3., 8., 5., 1., 3., 1.]]]])
```

In the **circular** padding, the **left-most (right-most) column** gets copied as the **right (left) padded column** (forget about the corners for now too). Similarly, the **top-most (bottom-most) row** gets copied as the **bottom (top) padded row**. The **corners** will receive the values of the **diametrically opposed corner**: the top-left padded pixel receives the value of the bottom-right corner of the original image. Once again, the values used in the padding are in a darker shade of orange.

In PyTorch, you must use the **functional form** `F.pad` with `mode="circular"` since there is **no module version** of the circular padding (at time of writing):

```
F.pad(image, pad=(1, 1, 1, 1), mode='circular')
```

Output

```
tensor([[[[0., 7., 2., 7., 0., 1., 0., 7.],
          [1., 5., 0., 8., 7., 8., 1., 5.],
          [7., 1., 9., 5., 0., 7., 7., 1.],
          [6., 6., 0., 2., 4., 6., 6., 6.],
          [4., 9., 7., 6., 6., 8., 4., 9.],
          [3., 8., 3., 8., 5., 1., 3., 8.],
          [0., 7., 2., 7., 0., 1., 0., 7.],
          [1., 5., 0., 8., 7., 8., 1., 5.]]]])
```

By means of **padding** an image, it is possible to get **resulting images** with the **same shape as input images**, or even *bigger*, should you choose to stuff more and more rows and columns to the input image. Assuming we're doing **symmetrical padding of size p**, the **resulting shape** is given by the formula below:

$$(h_i, w_i) * f = \left(\frac{(h_i + 2p) - f}{s} + 1, \frac{(w_i + 2p) - f}{s} + 1 \right)$$

Equation 5.4 - Shape after a convolution with stride and padding

We're basically extending the original dimensions by ***2p*** pixels each.

A REAL Filter

Enough with the identity filter! Let's try an **edge detector**[88] filter from traditional computer vision for a change:

```
edge = np.array(
    [[[[0,  1, 0],
       [1, -4, 1],
       [0,  1, 0]]]]
)
kernel_edge = torch.as_tensor(edge).float()
kernel_edge.shape
```

Output

```
torch.Size([1, 1, 3, 3])
```

And let's apply it to a different region of our (padded) input image as well:

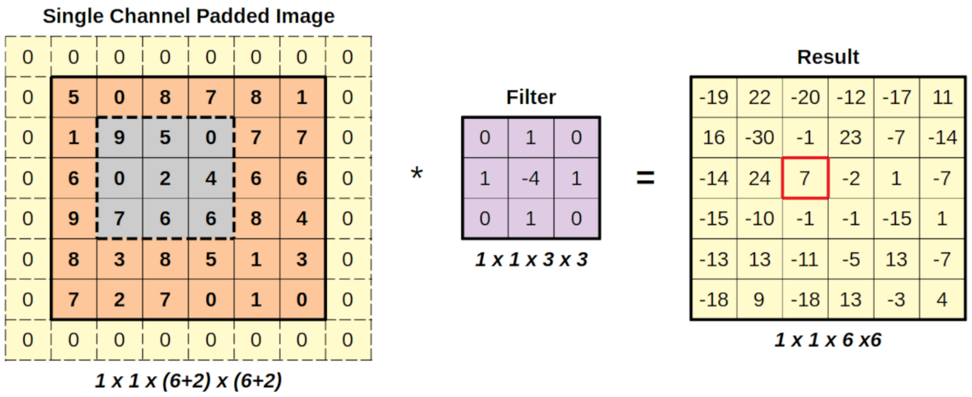


Figure 5.13 - Convolving a padded image - no shrinking!

As you can see, filters, other than the identity one, will **not simply copy** the value at the center. The element-wise multiplication finally means something:

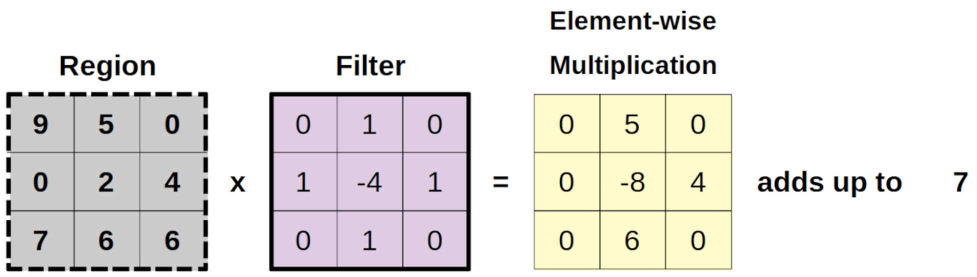


Figure 5.14 - Element-wise multiplication - edge filter

Let’s apply this filter to our image, so we can use the resulting image in our next operation:

```
padded = F.pad(image, (1, 1, 1, 1), mode='constant', value=0)
conv_padded = F.conv2d(padded, kernel_edge, stride=1)
```

Pooling

Now we're back in the business of **shrinking images**. Pooling is different than the former operations: it **splits the image into tiny chunks**, **performs an operation on each chunk** (that yields a **single value**), and **puts the chunks together** as the resulting image. Again, an image is worth a thousand words:

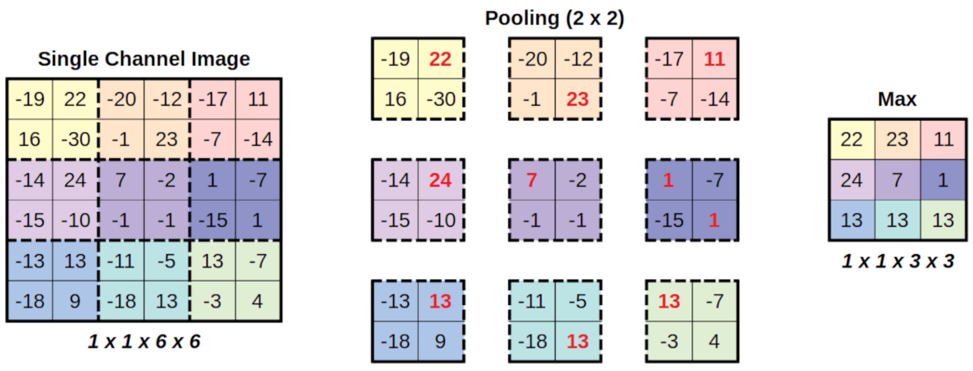


Figure 5.15 - Max pooling

In the image above, we're performing a **max-pooling** with a **kernel size of two**. Even though these are not quite the same filters as the ones we've already seen, it is still called **kernel**.

In this example, the **stride** is assumed to be the **same size** as the **kernel**.

Our input image is **split** into nine chunks, and we perform a simple **max** operation (hence, max-pooling) on each chunk (really, it is just taking the biggest value in each chunk). Then these values are put together, **in order**, to produce a **smaller resulting image**.

The **bigger the pooling kernel**, the **smaller the resulting image**.

A pooling kernel of two-by-two results in an image whose dimensions are half of the original. A pooling kernel of three-by-three makes the resulting image one third the size of the original, and so on. Moreover, only **full chunks** count: if we try a kernel of four-by-four in our six-by-six image, only **one chunk** fits, and the resulting image would have a **single pixel**.

In PyTorch, as usual, we have both forms: `F.max_pool2d` and `nn.MaxPool2d`. Let's use functional form to replicate the max-pooling in the figure above:

```
pooled = F.max_pool2d(conv_padded, kernel_size=2)
pooled
```

Output

```
tensor([[[[22., 23., 11.],
          [24.,  7.,  1.],
          [13., 13., 13.]]]])
```

And then let's use the module version to illustrate the big four-by-four pooling:

```
maxpool4 = nn.MaxPool2d(kernel_size=4)
pooled4 = maxpool4(conv_padded)
pooled4
```

Output

```
tensor([[[[24.]]]])
```

A single pixel as promised!

"*Can I perform some **other operation**?*"

Sure, besides *max-pooling*, **average pooling** is also fairly common. As the name suggests, it will output the **average pixel value** for each chunk. In PyTorch, we have `F.avg_pool2d` and `nn.AvgPool2d`.

"*Can I use a* ***stride of a different size****?*"

Of course, you can! In this case, there will be an **overlap** between regions instead of a *clean split into chunks*. So, it looks like a regular kernel of a convolution, but the **operation is already defined** (max or average, for instance). Let's go through a quick example:

```
F.max_pool2d(conv_padded, kernel_size=3, stride=1)
```

Output

```
tensor([[[[24., 24., 23., 23.],
          [24., 24., 23., 23.],
          [24., 24., 13., 13.],
          [13., 13., 13., 13.]]]])
```

The max-pooling kernel, sized three-by-three, will move over the image (just like the convolutional kernel) and compute the maximum value of each region it goes over. The resulting shape follows the formula on Equation 5.4.

Flattening

We've already seen this one! It simply **flattens** a tensor, preserving the first dimension such that we **keep the number of data points** while collapsing all other dimensions. It has a module version `nn.Flatten`:

```
flattened = nn.Flatten()(pooled)
flattened
```

Output

```
tensor([[22., 23., 11., 24.,  7.,  1., 13., 13., 13.]])
```

It has **no functional version**, but there is no need for one since we can accomplish the same thing using `view`:

```
pooled.view(1, -1)
```

Output

```
tensor([[22., 23., 11., 24.,  7.,  1., 13., 13., 13.]])
```

Dimensions

We've performed convolutions, padding, and pooling in **two dimensions** because we're handling images. But there are **one** and **three**-dimensional versions of some of them as well:

- `nn.Conv1d` and `F.conv1d`; `nn.Conv3d` and `F.conv3d`
- `nn.ConstandPad1d` and `nn.ConstandPad3d`
- `nn.ReplicationPad1d` and `nn.ReplicationPad3d`
- `nn.ReflectionPad1d`
- `nn.MaxPool1d` and `F.max_pool1d`; `nn.MaxPool3d` and `F.max_pool3d`
- `nn.AvgPool1d` and `F.avg_pool1d`; `nn.AvgPool3d` and `F.avg_pool3d`

We will not venture into the third dimension in this book, but we'll get back to **one-dimension** operations later.

"Aren't color images ***three-dimensional*** *since they have* ***three channels****?"*

Well, yes, but we will **still** be applying **two-dimensional convolutions** to them. We'll go through a detailed example using a three-channel image in the next chapter.

Typical Architecture

The typical architecture uses a **sequence** of one or more **typical convolutional blocks**, each block consisting of three operations:

1. Convolution
2. Activation function
3. Pooling

As images go through these operations, they will **shrink in size**. After three of these blocks (assuming kernel size of two for pooling), for instance, an image will be reduced to 1/8 or less of its original dimensions (and thus 1/64 of its total number of pixels). The **number of channels/filters** produced by each block, though, is usually **increased** as more blocks are added.

After the sequence of blocks, the image gets **flattened**: hopefully, at this stage, there is no loss of information by considering **each value** in the flattened tensor **a feature** on its own.

Once the features are dissociated from pixels, it becomes a fairly standard problem, like the ones we've been handling in this book: the features feed **one or more hidden layers**, and an **output layer** produces the **logits** for classification.

If you think of it, what those **typical convolutional blocks** do is akin to **pre-processing images** and **converting them into features**. Let's call this part of the network a **featurizer** (the one that generates features).

The **classification** itself is handled by the familiar and well-known **hidden and output layers**.

In **transfer learning**, which we'll see in Chapter 7, this will become even more clear.

LeNet-5

LeNet-5 is a 7-level convolutional neural network developed by Yann LeCun in 1998 to recognize hand-written digits in 28x28 pixel images - the famous **MNIST** dataset! That's when it all started (kinda). In 1989, LeCun himself used back-propagation (chained gradient descent, remember?) to **learn the convolution filters**, as we discussed above, instead of painstakingly developing them manually.

His network had this architecture:

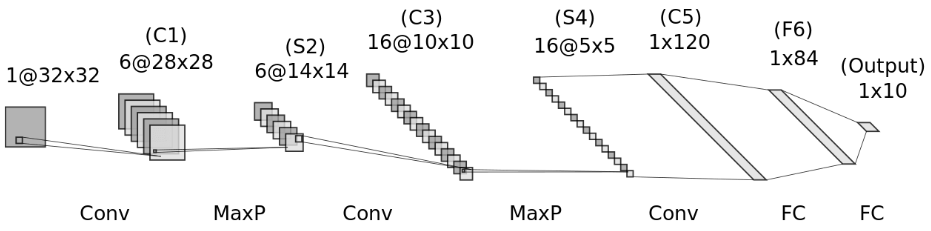


Figure 5.15 - LeNet-5 architecture

Source: Generated using Alexander Lenail's NN-SVG and adapted by the author. For more details, see LeCun, Y., et al (1998). "Gradient-based learning applied to document recognition". Proceedings of the IEEE, 86(11), 2278–2324[89]

Do you see anything familiar? The **typical convolutional blocks** are **already there** (to some extent): convolutions (**C** layers), activation functions (not shown), and subsampling (**S** layers). There are some **differences**, though:

- back then, the **subsampling was more complex** than today's **max-pooling**, but the general idea still holds
- the **activation function**, a sigmoid at the time, was applied ***after* the subsampling instead of before**, as it is typical today
- the F6 and OUTPUT layers were connected by something called "*Gaussian connections*", which is also **more complex than the typical activation function** one would use today

Adapting LeNet-5 to today's standards, it could be implemented like this:

```
lenet = nn.Sequential()

# Featurizer
# Block 1: 1@28x28 -> 6@28x28 -> 6@14x14
lenet.add_module('C1',
  nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)
)
lenet.add_module('func1', nn.ReLU())
lenet.add_module('S2', nn.MaxPool2d(kernel_size=2))
# Block 2: 6@14x14 -> 16@10x10 -> 16@5x5
lenet.add_module('C3',
  nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
)
lenet.add_module('func2', nn.ReLU())
lenet.add_module('S4', nn.MaxPool2d(kernel_size=2))
# Block 3: 16@5x5 -> 120@1x1
lenet.add_module('C5',
  nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5)
)
lenet.add_module('func2', nn.ReLU())
# Flattening
lenet.add_module('flatten', nn.Flatten())

# Classification
# Hidden Layer
lenet.add_module('F6', nn.Linear(in_features=120, out_features=84))
lenet.add_module('func3', nn.ReLU())
# Output Layer
lenet.add_module('OUTPUT',
  nn.Linear(in_features=84, out_features=10)
)
```

LeNet-5 used **three convolutional blocks**, although the last one does not have a max-pooling because the **convolution already produces a single pixel**. Regarding the number of channels, they increase as the image size decreases:

- input image: single-channel 28x28 pixels
- first block: produces 6-channel 14x14 pixels
- second block: produces 16-channel 5x5 pixels
- third block: produces 120-channel single-pixel (1x1)

Then, these 120 values (or features) are **flattened** and fed to a typical hidden layer with 84 units. The last step is, obviously, the output layer, which produces **10 logits** to be used for **digit classificatio**n (from 0 to 9, there are 10 classes).

"Wait, we ***haven't*** *seen any of those* ***multiclass*** *classification problems yet!"*

You're right, it is about time. But we're still *not* using MNIST for this.

A Multiclass Classification Problem

A problem is considered a **multiclass** classification problem if there are **more than two classes**. So, let's keep it as simple as possible and build a model to **classify images into three classes**.

Data Generation

Our images are going to have either a diagonal or a parallel line, BUT this time we will make a distinction between a **diagonal line tilted to the right**, a **diagonal line tilted to the left**, and a **parallel line** (it doesn't matter if it is horizontal or vertical). We can summarize the **labels (*y*)** like this:

Line	Label / Class Index
Parallel (Horizontal OR Vertical)	0
Diagonal, Tilted to the Right	1

Line	Label / Class Index
Diagonal, Tilted to the Left	2

Also, let's generate **more and bigger images**: one thousand images, each one ten-by-ten pixels in size.

Data Generation

```
images, labels = generate_dataset(
  img_size=10, n_images=1000, binary=False, seed=17
)
```

```
fig = plot_images(images, labels, n_plot=30)
```

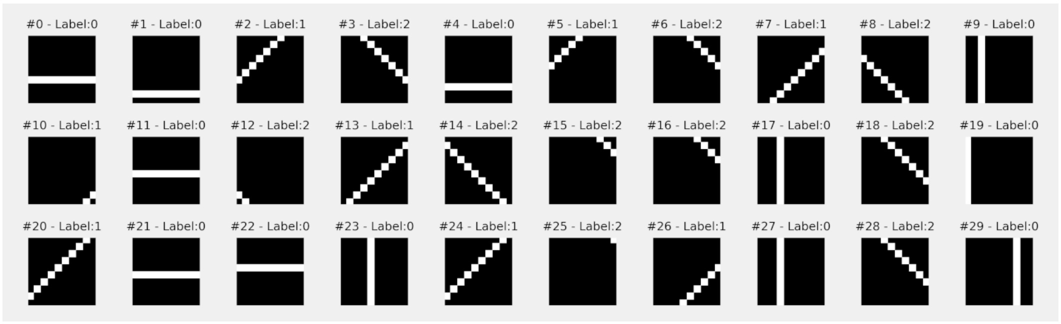


Figure 5.16 - Generated dataset

Just like in Chapter 4, the dataset is generated following PyTorch's format: **NCHW**.

Data Preparation

The data preparation step would be identical to the one we used in Chapter 4 if it weren't for **one change**: we will **not perform data augmentation** this time.

"Why not?"

In our particular problem, **flipping an image** is potentially **ruining the label**. If we have an image containing a diagonal line **tilted to the right** (thus labeled as class index #1), and we **flip it**, the diagonal line would **end up tilted to the left**. But data augmentation **does not change the labels**, so the result is an image with a **wrong label** (class index #1, even though it would contain a left-tilted diagonal line).

Data augmentation may be useful, but it **should not** produce images that are **inconsistent with their labels**.

That being said, we're only keeping the **min-max scaling** using the `Normalize` transform. All the rest remains the same: splitting, datasets, sampler, and data loaders.

Transformed Dataset

```python
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]

        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

Data Preparation

```python
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
```

```python
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels).long()

# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]

# We're not doing any data augmentation now
train_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])

# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor,
    transform=train_composer
)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor,
    transform=val_composer
)

# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)

# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16,
    sampler=sampler
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Before defining a model to classify our images, we need to discuss something else: the **loss function**.

Loss

New problem, new loss. Since we're embracing **multiclass** classification now, we need to use a **different loss**. And, once again, it all starts with our "*favorite*" subject: **logits**.

Logits

In binary classification problems, the model would produce **one logit**, and one logit only, for each data point. It makes sense, binary classification is about answering a simple question: "***does a given data point belong to the positive class?***".

The **logit** output represented the **log odds ratio** (remember that?) of answering "**yes**" to the question above. The log odds ratio of a "**no**" answer was simply the inverse. There was **no need** to pose any other question to make a decision. And we used a **sigmoid function** to map logits to **probabilities**. It was a simple world :-)

But a **multiclass classification** is more complex: we need to **ask more questions**, that is, we need to get **log odds ratios for every possible class**. In other words, we need **as many logits as there are classes**.

"*But a **sigmoid** takes **only one logit**. I guess we need something else to get probabilities, right?*"

Absolutely correct! The function we're looking for here is called **softmax**.

Softmax

The **softmax** function returns, for each class, the **contribution** that a given class had to the **sum of odds ratios**. The class with a **higher odds ratio** will have the **biggest contribution** and thus the **highest probability**.

Since the **softmax** is computed using **odds ratios** instead of **log odds ratios** (logits), we need to **exponentiate the logits**!

$$z = logit(p) = \ log\ odds\ ratio(p) = log\left(\frac{p}{1-p}\right)$$
$$e^z = e^{logit(p)} = \qquad odds\ ratio(p) = \left(\frac{p}{1-p}\right)$$

Equation 5.5 - Logit and odds ratio

The softmax formula itself is quite simple:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{C-1} e^{z_j}}$$

Equation 5.6 - Softmax function

In the equation above, *C* stands for the **number of classes** and *i* corresponds to the index of a particular class. In our example, we have **three classes**, so our model needs to **output three logits** (z_0, z_1, z_2). Applying **softmax** to these logits, we would get:

$$softmax(z) = \left[\frac{e^{z0}}{e^{z0}+e^{z1}+e^{z2}}, \frac{e^{z1}}{e^{z0}+e^{z1}+e^{z2}}, \frac{e^{z2}}{e^{z0}+e^{z1}+e^{z2}}\right]$$

Equation 5.7 - Softmax for a three-class classification problem

Simple, right? Let's see it in code now. Assuming our model produces this tensor containing three logits:

```
logits = torch.tensor([ 1.3863,  0.0000, -0.6931])
```

We **exponentiate the logits** to get the corresponding **odds ratios**:

```
odds_ratios = torch.exp(logits)
odds_ratios
```

Output

```
tensor([4.0000, 1.0000, 0.5000])
```

The resulting tensor is telling us that the first class has much better odds than the other two, and the second one has better odds than the third. So we take these odds and **add them together**, and then compute the **each class' contribution to the sum**:

```
softmaxed = odds_ratios / odds_ratios.sum()
softmaxed
```

Output

```
tensor([0.7273, 0.1818, 0.0909])
```

Voilà! Our logits were **softmaxed**: the **probabilities are proportional to the odds ratios**. This data point most likely belongs to the first class since it has a probability of 72.73%.

But there is absolutely no need to compute it manually, of course. PyTorch provides the typical implementations: functional (`F.softmax`) and module (`nn.Softmax`):

```
nn.Softmax(dim=-1)(logits), F.softmax(logits, dim=-1)
```

Output

```
(tensor([0.7273, 0.1818, 0.0909]), tensor([0.7273, 0.1818, 0.0909]))
```

In both cases, it asks you to provide **which dimension** the softmax function should be applied to. In general, our models will produce logits with the shape **(number of data points, number of classes)**, so the right dimension to apply softmax to is the **last one** (`dim=-1`).

LogSoftmax

The logsoftmax function returns, well, the **logarithm of the softmax** function above. But, instead of manually taking the logarithm, PyTorch provides `F.log_softmax` and `nn.LogSoftmax` out of the box.

These functions are **faster** and also have *better numerical properties*. But, I guess your main question at this point is:

> ❓ "*Why do I need to **take the log** of the softmax?*"

The simple and straightforward reason is that the loss function expects **log-probabilities** as input.

Negative Log-Likelihood Loss

Since the softmax returns **probabilities**, the logsoftmax returns **log-probabilities**. And that's the input for computing the negative log-likelihood loss, or `NLLLoss` for short. This loss is simply an **extension of the binary cross-entropy loss to handle multiple classes**.

This was the formula for computing **binary cross-entropy**:

$$BCE(y) = -\frac{1}{(N_{pos}+N_{neg})}\left[\sum_{i=1}^{N_{pos}} log(P(y_i=1)) + \sum_{i=1}^{N_{neg}} log(1-P(y_i=1))\right]$$

Equation 5.8 - Binary cross-entropy

See the **log-probabilities** in the summation terms? In our example, there are **three classes**, that is, our **labels (*y*)** could be either **zero**, **one**, or **two**. So, the loss function will look like this:

$$NLLLoss(y) = -\frac{1}{(N_0+N_1+N_2)}\left[\sum_{i=1}^{N_0} log(P(y_i=0)) + \sum_{i=1}^{N_1} log(P(y_i=1)) \sum_{i=1}^{N_2} log(P(y_i=2))\right]$$

Equation 5.9 - Negative log-likelihood loss for a three-class classification problem

Take, for instance, the first class (***y=0***). For every data point belonging to this class (there are N_0 of them), we take the **logarithm of the predicted probability for that point and class** (***log(P(*y_i*=0))***) and add them all up. Next, we repeat the process for the other two classes, add all three results up, and divide by the total number of data points.

The loss **only considers the predicted probability for the true class**.

If a data point is labeled as belonging to class index **two**, the loss will consider the **probability assigned to class index two only**. The other probabilities will be completely ignored.

For a total of **C** classes, the formula can be written like this:

$$NLLLoss(y) = -\frac{1}{(N_0+\cdots+N_{C-1})}\sum_{c=0}^{C-1}\sum_{i=1}^{N_c} log(P(y_i=c))$$

Equation 5.10 - Negative log-likelihood loss for a classification problem with C classes

Since the **log-probabilities** are obtained by applying **logsoftmax**, this loss isn't doing much more than **looking up the inputs corresponding to the true class** and adding them up. Let's see this in code:

```
log_probs = F.log_softmax(logits, dim=-1)
log_probs
```

Output

```
tensor([-0.3185, -1.7048, -2.3979])
```

These are the **log-probabilities for each class** we computed using logsoftmax for our **single data point**. Now, let's assume its **label** is **two**: what is the corresponding loss?

```
label = torch.tensor([2])
F.nll_loss(log_probs.view(-1, 3), label)
```

Output

```
tensor(2.3979)
```

It is the **negative** of the **log-probability** corresponding to the **class index** (two) of the **true label**.

As you've probably noticed, I used the *functional* version of the loss in the snippet of code above: `F.nll_loss`. But, as we've done with the binary cross-entropy loss in Chapter 3, we're likely using the module version: `nn.NLLLoss`.

Just like before, this loss function is a higher-order function, and this one takes three **optional** arguments (the others are deprecated and you can safely ignore them):

- `reduction`: it takes either `mean`, `sum`, or `none`. The default, **`mean`**, corresponds to our **Equation 5.10** above. As expected, `sum` will return the sum of the errors, instead of the average. The last option, **`none`**, corresponds to the **unreduced** form, that is, it returns the full **array of errors**.
- `weight`: it takes a tensor of length **C**, that is, containing as many weights as there are classes.

> **IMPORTANT**: this argument **can be used to handle imbalanced datasets**, unlike the `weight` argument in the binary cross-entropy losses we've seen in Chapter 3.
>
> Also, **unlike** the `pos_weight` argument of `BCEWithLogitsLoss`, **the `NLLLoss` computes a true weighted average** when this argument is used.

- `ignore_index`: it takes **one integer**, corresponding to the **one (and only one) class index that should be ignored** when computing the loss. It can be used to **mask a particular label** that is not relevant to the classification task.

Let's go through some quick examples using the arguments above. First, we need to generate some *dummy logits* (we'll keep using three classes, though), and the corresponding log-probabilities:

```
torch.manual_seed(11)
dummy_logits = torch.randn((5, 3))
dummy_labels = torch.tensor([0, 0, 1, 2, 1])
dummy_log_probs = F.log_softmax(dummy_logits, dim=-1)
dummy_log_probs
```

Output

```
tensor([[-1.5229, -0.3146, -2.9600],
        [-1.7934, -1.0044, -0.7607],
        [-1.2513, -1.0136, -1.0471],
        [-2.6799, -0.2219, -2.0367],
        [-1.0728, -1.9098, -0.6737]])
```

Can you hand-pick the log-probabilities that are going to be **actually used** in the loss computation?

```
relevant_log_probs = torch.tensor([-1.5229, -1.7934, -1.0136,
-2.0367, -1.9098])
-relevant_log_probs.mean()
```

Output

```
tensor(1.6553)
```

Now let's use `nn.NLLLoss` to create the actual loss function, and then use predictions and labels to check if we got the relevant log-probabilities right:

```
loss_fn = nn.NLLLoss()
loss_fn(dummy_log_probs, dummy_labels)
```

Output

```
tensor(1.6553)
```

Right indeed! What if we want to **balance** our dataset, giving data points with **label (*y*=2) double the weight** of the other classes?

```
loss_fn = nn.NLLLoss(weight=torch.tensor([1., 1., 2.]))
loss_fn(dummy_log_probs, dummy_labels)
```

Output

```
tensor(1.7188)
```

And what if we want to **simply ignore** data points with **label (y=2)**?

```
loss_fn = nn.NLLLoss(ignore_index=2)
loss_fn(dummy_log_probs, dummy_labels)
```

Output

```
tensor(1.5599)
```

And, once again, there is yet **another** loss function available for multiclass classification. And, once again, it is **very important** to know **when to use one or the other**, so you don't end up with an inconsistent combination of model and loss function.

Cross-Entropy Loss

The former loss function took log-probabilities as an argument (together with the labels, obviously). Guess what *this* function takes? **Logits**, of course! This is the multiclass version of `nn.BCEWithLogitsLoss`.

"What does it mean, in practical terms?"

It means you **should NOT add a logsoftmax as the last layer of your model** when using this loss function. This loss combines both **the logsoftmax layer and the former negative log-likelihood loss into one**.

IMPORTANT: I can't stress this enough: you **must** use the **right combination of model and loss function**.

Option 1: `nn.LogSoftmax` as the **last** layer, meaning your model is producing **log-probabilities**, combined with `nn.NLLLoss` function

Option 2: **no logsoftmax** in the last layer, meaning your model is producing **logits**, combined with `nn.CrossEntropyLoss` function.

Mixing `nn.LogSoftmax` and `nn.CrossEntropyLoss` is just **wrong**.

Now that the difference in the arguments is clear, let's take a closer look at the `nn.CrossEntropyLoss` function. It is also a higher-order function, and it takes the **same three optional** arguments as `nn.NLLLoss`:

- `reduction`: it takes either `mean`, `sum`, or `none`, and the default is **`mean`**.
- `weight`: it takes a tensor of length **C**, that is, containing as many weights as there are classes.
- `ignore_index`: it takes **one integer**, corresponding to the **one (and only one) class index that should be ignored**.

Let's see a quick example of its usage, taking the *dummy logits* as input:

```
torch.manual_seed(11)
dummy_logits = torch.randn((5, 3))
dummy_labels = torch.tensor([0, 0, 1, 2, 1])

loss_fn = nn.CrossEntropyLoss()
loss_fn(dummy_logits, dummy_labels)
```

Output

```
tensor(1.6553)
```

No logsoftmax whatsoever but the same resulting loss, as expected.

Classification Losses Showdown!

Honestly, I always felt this whole thing is a bit confusing, especially for someone who's learning it for the first time.

Which loss functions take logits as inputs? Should I add a (log)softmax layer or not? Can I use the `weight` argument to handle imbalanced datasets? Too many questions, right?

So, here is a **table** to help you figure out the landscape of loss functions for classification problems, both binary and multiclass:

	BCE Loss	**BCE With Logits Loss**	**NLL Loss**	**Cross-Entropy Loss**
Classification	binary	binary	multiclass	multiclass
Input (each data point)	probability	logit	array of log-probabilities	array of logits
Label (each data point)	float (0.0 or 1.0)	float (0.0 or 1.0)	long (class index)	long (class index)
Model's last layer	Sigmoid	-	LogSoftmax	-
`weight` argument	**not** class weights	**not** class weights	class weights	class weights
`pos_weight` argument	n/a	"weighted" loss	n/a	n/a

Model Configuration

Let's build our first **convolutional neural network** for real! We can use the **typical convolutional block**: convolutional layer, activation function, pooling layer. Our images are quite **small**, so we only need **one of those**.

We still need to decide **how many channels** our convolutional layer is going to produce. In general, the number of channels **increases** with each convolutional block. For the sake of simplicity (and later visualization), let's **keep a single channel**.

We also need to decide on a **kernel size** (the **receptive field** or *gray regions* in the figures at the beginning of this chapter). Let's stick with a kernel size of **three**, which will **reduce the image size by two pixels in each dimension** (we are not using padding here).

Our **featurizer**, which will encode our images into features using convolutions, should look like this:

Model Configuration - Featurizer

```
torch.manual_seed(13)
model_cnn1 = nn.Sequential()

# Featurizer
# Block 1: 1@10x10 -> n_channels@8x8 -> n_channels@4x4
n_channels = 1
model_cnn1.add_module('conv1', nn.Conv2d(
  in_channels=1, out_channels=n_channels, kernel_size=3
))
model_cnn1.add_module('relu1', nn.ReLU())
model_cnn1.add_module('maxp1', nn.MaxPool2d(kernel_size=2))
# Flattening: n_channels _ 4 _ 4
model_cnn1.add_module('flatten', nn.Flatten())
```

I am keeping the number of channels as a variable, so you can try different values for it if you like.

Let's follow what happens to an input image (single-channel, 10x10 pixels in size - 1@10x10):

- the image is **convolved** with the kernel and the resulting image has one channel, and it is 8x8 pixels in size (1@8x8)
- a ReLU activation function is applied to the resulting image
- the "*activated*" image goes under a **max-pooling** operation with a kernel size of two, so it is divided into **16 chunks** of size two-by-two, resulting in an image with one channel, but 4x4 pixels in size (1@4x4)
- these 16 values can be considered **features**, and are **flattened** into a tensor with 16 elements

The next part of our model, the **classifier** part, uses these features to feed what would be a simple neural network with a single hidden layer if considered on its own:

Model Configuration - Classifier

```
# Classification
# Hidden Layer
model_cnn1.add_module('fc1',
  nn.Linear(in_features=n_channels*4*4, out_features=10)
)
model_cnn1.add_module('relu2', nn.ReLU())
# Output Layer
model.add_module('fc2', nn.Linear(in_features=10, out_features=3))
```

See? There is a hidden layer that takes the 16 features as inputs and maps them into a 10-dimensional space that is going to be "activated" by the ReLU.

Then, the output layer produces **three distinct linear combinations of the ten activation values**, each combination corresponding to a different class. The figure below, depicting the second half of the model, should make it more clear:

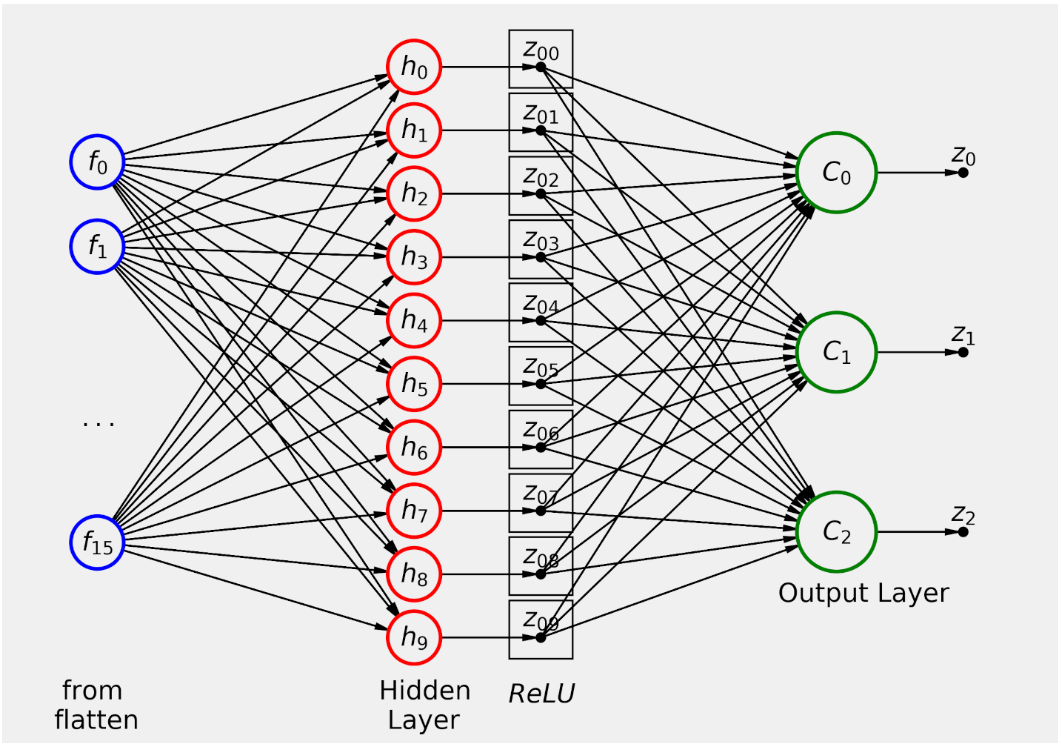


Figure 5.17 - Classifier with softmax output

The three units in the output layer produce **three logits**, one for each class (C_0, C_1, and C_2). We could have added a `nn.LogSoftmax` layer to the model, and it would convert the three logits to log-probabilities.

Since our model produces logits, we **must** use the `nn.CrossEntropyLoss` function:

Model Configuration - Loss and Optimizer

```
lr = 0.1
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn1 = optim.SGD(model_cnn1.parameters(), lr=lr)
```

And then we create an optimizer (SGD) with a given learning rate (0.1), as usual. Boring, right? No worries, we'll finally **change the optimizer** in the "*Rock-Paper-Scissors*" classification problem in the next chapter.

Model Training

This part is completely straightforward. First, we instantiate our class, and set the loaders:

Model Training

```
sbs_cnn1 = StepByStep(model_cnn1, multi_loss_fn, optimizer_cnn1)
sbs_cnn1.set_loaders(train_loader, val_loader)
```

Then, we train it for 20 epochs, and visualize the losses:

Model Training

```
sbs_cnn1.train(20)
```

```
fig = sbs_cnn1.plot_losses()
```

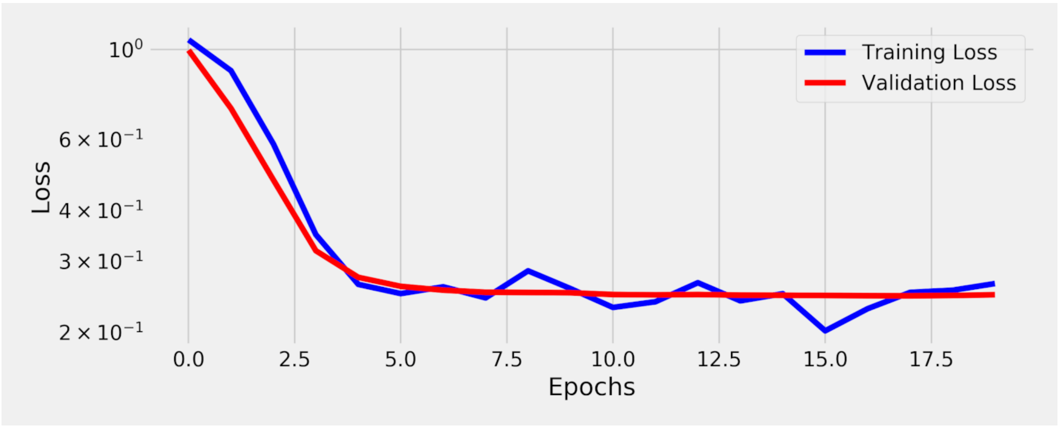


Figure 5.18 - Losses

OK, it seems to have reached a minimum at the fifth epoch already.

Visualizing Filters and More!

In Chapter 4, we briefly discussed **visualizing weights as pixels**. We're going to dive deeper into the visualization of filters (weights), as well as the transformed images produced by each one of our model's layers.

First, let's add another method to our tool belt:

StepByStep Method

```
@staticmethod
def _visualize_tensors(axs, x, y=None, yhat=None,
                       layer_name='', title=None):
    # The number of images is the number of subplots in a row
    n_images = len(axs)
    # Gets max and min values for scaling the grayscale
    minv, maxv = np.min(x[:n_images]), np.max(x[:n_images])
    # For each image
    for j, image in enumerate(x[:n_images]):
        ax = axs[j]
        # Sets title, labels, and removes ticks
        if title is not None:
            ax.set_title(f'{title} #{j}', fontsize=12)
        shp = np.atleast_2d(image).shape
        ax.set_ylabel(
            f'{layer_name}\n{shp[0]}x{shp[1]}',
            rotation=0, labelpad=40
        )
        xlabel1 = '' if y is None else f'\nLabel: {y[j]}'
        xlabel2 = '' if yhat is None else f'\nPredicted: {yhat[j]}'
        xlabel = f'{xlabel1}{xlabel2}'
        if len(xlabel):
            ax.set_xlabel(xlabel, fontsize=12)
        ax.set_xticks([])
        ax.set_yticks([])

        # Plots weight as an image
```

```
            ax.imshow(
                np.atleast_2d(image.squeeze()),
                cmap='gray',
                vmin=minv,
                vmax=maxv
            )
        return

setattr(StepByStep, '_visualize_tensors', _visualize_tensors)
```

Most of the function's body is handling titles, labels, and axes' ticks before using `imshow` to actually plot the image, so it is not *that* interesting. Let's check its arguments:

- `axs`: an **array of subplots**, corresponding to **one row of subplots** as returned by Matplotlib's `subplot`
- `x`: a *Numpy* array containing at least as many **images/filters** as subplots in `axs`
- `y`: optional, a *Numpy* array containing at least as many **labels** as subplots in `axs`
- `yhat`: optional, a *Numpy* array containing at least as many **predicted labels** as subplots in `axs`
- `layer_name`: label for the row of subplots
- `title`: title prefix for each subplot

"*What is that* `@staticmethod` *thingy above the method's definition?*"

Static Method

The “@” indicates that the method sitting below it, `_visualize_tensors` is being **decorated** by the **`staticmethod` decorator function**.

"What is a decorator?"

Python decorators is a **big** topic on its own, too long to explain here. If you want to learn more about it, check Real Python’s Primer on Python Decorators[90]. But I am not leaving you without a working knowledge of what that particular (and somewhat common) decorator is doing.

The `@staticmethod` decorator allows the method to be **called on an uninstantiated class object**. It is as if we’re **attaching** a method to a class that **does not depend on an instance of the class it is attached to.**

It is easy to see **why**: in **every other method** we have created so far for the `StepByStep` class, the **first argument was ALWAYS `self`**. So, those methods had access to the class they belonged to, better yet, they had access to **a particular instance and its attributes**. Remember the `Dog` class? The `bark` method **knew the name of the dog** because its first argument was the instance representing the dog (`self`).

A **static method does not have a `self` argument**. The inner workings of the function **must be independent of the instance of the class it belongs to**.

The static method **can be executed** from the class itself instead of one of its instances.

Let me illustrate it with yet another silly example:

```python
class Cat(object):
    def __init__(self, name):
        self.name = name

    @staticmethod
    def meow():
        print('Meow')
```

The `meow` method is *totally independent* of the `Cat` class. We **do not even need to create a cat**! That's what I meant with "*called on an uninstantiated class object*":

```python
Cat.meow()
```

Output

```
Meow
```

See? The `meow` method could well be an independent function because it works like one. But, in the context of a cat class, it makes sense to have that method attached to it since they *belong* together, conceptually speaking.

Back to our own static method, we'll call it from other (regular) methods to plot the images we're interested in. Let's start with the *filters*.

Visualizing Filters

We could apply the same principle to the **weights** of the **filter learned by our convolutional layer**. We can access the weights of any given layer using dot notation:

```
weights_filter = model_cnn1.conv1.weight.data.cpu().numpy()
weights_filter.shape
```

Output

```
(1, 1, 3, 3)
```

Each layer has its own `weight` attribute, which is a `nn.Parameter`. We could use it directly, but then we would also have to `detach` the parameter from the computation graph before converting it to *Numpy*. So, it is easier to use the `data` attribute of `weight` because it is simply a tensor, and no detaching is needed.

The shape of the weights (representing the filters) of a two-dimensional convolutional layer is given by **(out_channels, in_channels, kernel_size, kernel_size)**. In our case, the kernel size is three, and we have a single channel, both in and out, hence the shape of the weights is (1, 1, 3, 3).

And that's when the **static method** we developed in the previous section comes in handy: we can loop through the **filters** (**output channels**) that the model learned to convolve each one of the input channels.

StepByStep Method

```
def visualize_filters(self, layer_name, **kwargs):
    try:
        # Gets the layer object from the model
        layer = self.model
        for name in layer_name.split('.'):
            layer = getattr(layer, name)
        # We are only looking at filters for 2D convolutions
        if isinstance(layer, nn.Conv2d):
            # Takes the weight information
            weights = layer.weight.data.cpu().numpy()
            # weights -> (channels_out (filter), channels_in, H, W)
            n_filters, n_channels, _, _ = weights.shape

            # Builds a figure
            size = (2 * n_channels + 2, 2 * n_filters)
            fig, axes = plt.subplots(n_filters, n_channels,
                                     figsize=size)
            axes = np.atleast_2d(axes)
            axes = axes.reshape(n_filters, n_channels)
            # For each channel_out (filter)
            for i in range(n_filters):
                StepByStep._visualize_tensors(
                    axes[i, :],                  ①
                    weights[i],                  ②
                    layer_name=f'Filter #{i}',
                    title='Channel'
                )

            for ax in axes.flat:
                ax.label_outer()

            fig.tight_layout()
            return fig
    except AttributeError:
        return

setattr(StepByStep, 'visualize_filters', visualize_filters)
```

① The i-th row of subplots corresponds to a particular filter, each row has as many columns as there are input channels

② The i-th element of the weights corresponds to the i-th filter, which learned different weights to convolve each one of the input channels

OK, let's see how the filter looks like:

```
fig = sbs_cnn1.visualize_filters('conv1', cmap='gray')
```

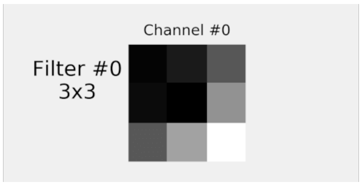


Figure 5.19 - Our model's only filter

Is *this* a filter one could come up with to try *distinguishing between the different classes* we have? Maybe, but just by looking at this filter, it is not easy to grasp what it is effectively accomplishing.

To **really understand the effect this filter** has on each image, we need to **visualize the intermediate values** produced by our model, namely, the **output of each and every layer**!

"*How can we* ***visualize the output*** *of each layer? Do we have to modify our* `StepByStep` *class to capture those?*"

It is much **easier** than that: we can use **hooks**!

Hooks

A **hook** is simply a way to **force a model to execute a function** either after its **forward** or after its **backward** pass. Hence, there are **forward hooks** and **backward hooks**. We're using *only forward* hooks here, but the idea is the same for both.

First, we create a **function** that is going to be, guess what, **hooked** to the **forward pass**. Let's illustrate the process with a **dummy model**:

```
dummy_model = nn.Linear(1, 1)

dummy_list = []

def dummy_hook(layer, inputs, outputs):
    dummy_list.append((layer, inputs, outputs))
```

The (forward) **hook function** takes **three arguments**:

- a model (or layer)
- a tensor representing the **inputs** taken by that model (or layer)
- a tensor representing the **outputs** generated by that model (or layer)

So, any function taking three arguments, regardless of their names, can work as a *hook*. In our case (and in many other cases too), we would like to **keep the information** that goes through the hook function.

You should **use a variable (or variables) defined outside the hook function to store values**.

That's the role of the `dummy_list` variable in the snippet above. Our `dummy_hook` function is as basic as it gets: it simply appends a tuple of its three arguments to the `dummy_list` variable defined outside the hook function.

"*How to **hook** the hook function to the model?*"

There is a method for it: `register_forward_hook`, which takes the hook function and returns a **handle**, so we can keep track of the hooks attached to our model.

```
dummy_handle = dummy_model.register_forward_hook(dummy_hook)
dummy_handle
```

Output

```
<torch.utils.hooks.RemovableHandle at 0x7fc9a003e190>
```

Simple enough, right? Let's see it in action:

```
dummy_x = torch.tensor([0.3])
dummy_model.forward(dummy_x)
```

Output

```
tensor([-0.7514], grad_fn=<AddBackward0>)
```

It should add a new tuple to the dummy list, containing a linear layer, an input tensor (0.3), and an output tensor (-0.7514). By the way, your values are going to be different than mine, since we didn't bother to use a seed here.

```
dummy_list
```

Output

```
[]
```

"Empty?! So it is not working?"

GOTCHA! I deliberately used the model's `forward` method here to illustrate something we've discussed much earlier, in Chapter 1:

You should **NOT call the `forward(x)`** method! You should **call the whole model instead**, as in **`model(x)`** to perform a forward pass.

Otherwise, your **hooks won't work**.

Let's do it right this time:

```
dummy_model(dummy_x)
```

Output

```
tensor([-0.7514], grad_fn=<AddBackward0>)
```

```
dummy_list
```

Output

```
[(Linear(in_features=1, out_features=1, bias=True),
  (tensor([0.3000]),),
  tensor([-0.7514], grad_fn=<AddBackward0>))]
```

Now we're talking! Here is the tuple we were expecting! If you **call the model once again**, it will **append yet another tuple** to the list, and so on and so forth. This hook is going to be hooked to our model **until it is explicitly removed** (hence the need to keep the handles). To remove a hook, you can simply call its `remove` method:

```
dummy_handle.remove()
```

And the hook goes bye-bye! But we **did not lose the collected information**, since our variable, `dummy_list` was defined outside the hook function.

Look at the first element of the tuple: it is **an instance of a model (or layer)**. Even if we use a `Sequential` model and **name the layers**, the **names won't make it to the hook function**. So we need to be clever here and make the association ourselves.

Let's get back to our **real model** now. We can get a list of all its named modules using the appropriate method: `named_modules` (what else could it be?!):

```
modules = list(sbs_cnn1.model.named_modules())
modules
```

Output

```
[('', Sequential(
    (conv1): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1))
    (relu1): ReLU()
    (maxp1): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
    (flatten): Flatten()
    (fc1): Linear(in_features=16, out_features=10, bias=True)
    (relu2): ReLU()
    (fc2): Linear(in_features=10, out_features=3, bias=True)
  )),
 ('conv1', Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1))),
 ('relu1', ReLU()),
 ('maxp1',
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)),
 ('flatten', Flatten()),
 ('fc1', Linear(in_features=16, out_features=10, bias=True)),
 ('relu2', ReLU()),
 ('fc2', Linear(in_features=10, out_features=3, bias=True))]
```

The first, unnamed, module is the whole model itself. The other modules are its layers. And those layers are one of the inputs of the hook function. So, we need to be able to **look up the name, given the corresponding layer instance**… if only there was something we could use to easily look up values, right?

```
layer_names = {layer: name for name, layer in modules[1:]}
layer_names
```

Output

```
{Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1)): 'conv1',
 ReLU(): 'relu1',
 MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False): 'maxp1',
 Flatten(): 'flatten',
 Linear(in_features=16, out_features=10, bias=True): 'fc1',
 ReLU(): 'relu2',
 Linear(in_features=10, out_features=3, bias=True): 'fc2'}
```

A dictionary is perfect for that: the hook function will take the layer instance as argument, and look its name up in the dictionary!

OK, it is time to create a real hook function:

```
visualization = {}

def hook_fn(layer, inputs, outputs):
    name = layer_names[layer]
    visualization[name] = outputs.detach().cpu().numpy()
```

It is actually quite simple: it looks up the **name of the layer** and uses it as a **key** to a dictionary **defined outside the hook function**, which will store the **outputs** produced by the hooked layer. The inputs are being ignored in this function.

We can make a list of the layers we'd like to get the outputs from, loop through the list of **named modules**, and **hook our function** to the desired layers, **keeping the handles in another dictionary**:

```
layers_to_hook = ['conv1', 'relu1', 'maxp1', 'flatten',
                  'fc1', 'relu2', 'fc2']

handles = {}

for name, layer in modules:
    if name in layers_to_hook:
        handles[name] = layer.register_forward_hook(hook_fn)
```

Everything is in place now! The only thing left to do is to actually **call the model**, so a forward pass is triggered, the hooks are executed, and the outputs to all these layers are stored in the `visualization` dictionary.

Let's fetch one mini-batch from the validation loader and use the `predict` method of our `StepByStep` class (which will then call the trained model):

```
images_batch, labels_batch = iter(val_loader).next()
logits = sbs_cnn1.predict(images_batch)
```

Now, if everything went well, our `visualization` dictionary should contain **one key for each layer** we hooked a function to:

```
visualization.keys()
```

Output

```
dict_keys(['conv1', 'relu1', 'maxp1', 'flatten', 'fc1', 'relu2',
'fc2'])
```

Bingo! They are all there! But, before checking what's stored inside it, let's **remove the hooks**:

```
for handle in handles.values():
    handle.remove()
handles = {}
```

Make sure to **always remove the hooks** after they served their purpose to avoid unnecessary operations that may slow down your model.

Maybe I got you *hooked* (sorry, I really like puns!), maybe not. Anyway, to make it easier for you to get some layers hooked so you can take a peek at what they are producing, we're appending some methods to our `StepByStep` class: `attach_hooks` and `remove_hooks`.

First, we are creating two dictionaries as attributes, `visualization` and `handles`, which will be the externally defined variables (external to the methods, that is).

The `attach_hooks` method has its own, internal, hook function that is going to store a layer's outputs in the `visualization` attribute. The method handles everything for us: mapping between layer instances and their names, and registering the hook function with the desired layers.

The `remove_hooks` is pretty much exactly the same code, except for the fact that it uses the `handles` attribute now.

StepByStep Method

```
setattr(StepByStep, 'visualization', {})
setattr(StepByStep, 'handles', {})

def attach_hooks(self, layers_to_hook, hook_fn=None):
    # Clear any previous values
    self.visualization = {}
    # Creates the dictionary to map layer objects to their names
    modules = list(self.model.named_modules())
    layer_names = {layer: name for name, layer in modules[1:]}

    if hook_fn is None:
        # Hook function to be attached to the forward pass
        def hook_fn(layer, inputs, outputs):
            # Gets the layer name
            name = layer_names[layer]
            # Detaches outputs
            values = outputs.detach().cpu().numpy()
            # Since the hook function may be called multiple times
            # for example, if we make predictions for multiple
            # mini-batches it concatenates the results
            if self.visualization[name] is None:
                self.visualization[name] = values
            else:
                self.visualization[name] = \
                  np.concatenate([self.visualization[name], values])

    for name, layer in modules:
        # If the layer is in our list
        if name in layers_to_hook:
            # Initializes the corresponding key in the dictionary
            self.visualization[name] = None
            # Register the forward hook and keep the handle
            # in another dict
            self.handles[name] = \
              layer.register_forward_hook(hook_fn)

def remove_hooks(self):
    # Loops through all hooks and removes them
    for handle in self.handles.values():
        handle.remove()
    # Clear the dict, as all hooks have been removed
    self.handles = {}

setattr(StepByStep, 'attach_hooks', attach_hooks)
setattr(StepByStep, 'remove_hooks', remove_hooks)
```

The procedure is fairly straightforward now: give it a list containing the names of the layers to attach hooks to, and you're done!

Hooking It

```
sbs_cnn1.attach_hooks(
  layers_to_hook=['conv1', 'relu1', 'maxp1', 'flatten',
                  'fc1', 'relu2', 'fc2']
)
```

To get the `visualization` attribute filled with values, we still need to make predictions:

Making Predictions (Logits)

```
images_batch, labels_batch = iter(val_loader).next()
logits = sbs_cnn1.predict(images_batch)
```

Don't forget to remove the hooks after you're finished with the predictions. By the way, you **can call `predict` multiple times** and the outputs produced by the hooked layers will be **concatenated**.

Removing Hooks

```
sbs_cnn1.remove_hooks()
```

Before moving on, don't forget the model is producing **logits** as outputs. To get the predicted classes, we can simply take the **index of the largest logit** for each data point:

Making Predictions (Classes)

```
predicted = np.argmax(logits, 1)
predicted
```

Output

```
array([2, 2, 2, 0, 0, 0, 2, 2, 2, 1, 0, 1, 2, 1, 2, 0])
```

We'll use the predicted classes in the next section.

Visualizing Feature Maps

First, let's visualize the first ten images sampled from the validation loader:

```
fig = plot_images(images_batch.squeeze(), labels_batch.squeeze(),
n_plot=10)
```

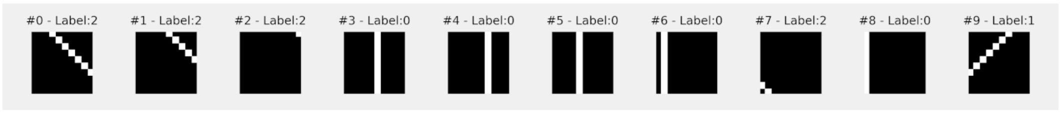


Figure 5.20 - Mini-batch of images

The first part of our model, which we called **featurizer**, has **four layers**: three in a typical convolutional block, and a flattening layer. The **outputs of these layers** are the **feature maps**, which were captured by our hook function when we made predictions for the first mini-batch of the validation loader.

To visualize the feature maps, we can add another method to our class: `visualize_outputs`. This method simply retrieves the captured feature maps from the `visualization` dictionary and uses our `_visualize_tensors` method to plot them:

StepByStep Method

```
def visualize_outputs(self, layers, n_images=10, y=None, yhat=None):
    layers = filter(lambda l: l in self.visualization.keys(),
                    layers)
    layers = list(layers)
    shapes = [self.visualization[layer].shape for layer in layers]
    n_rows = [shape[1] if len(shape) == 4 else 1
              for shape in shapes]
    total_rows = np.sum(n_rows)

    fig, axes = plt.subplots(total_rows, n_images,
                             figsize=(1.5*n_images, 1.5*total_rows))
    axes = np.atleast_2d(axes).reshape(total_rows, n_images)

    # Loops through the layers, one layer per row of subplots
    row = 0
    for i, layer in enumerate(layers):
        start_row = row
        # Takes the produced feature maps for that layer
        output = self.visualization[layer]

        is_vector = len(output.shape) == 2

        for j in range(n_rows[i]):
            StepByStep._visualize_tensors(
                axes[row, :],
                output if is_vector else output[:, j].squeeze(),
                y,
                yhat,
                layer_name=layers[i] \
                           if is_vector \
                           else f'{layers[i]}\nfil#{row-start_row}',
                title='Image' if (row == 0) else None
            )
            row += 1

    for ax in axes.flat:
        ax.label_outer()

    plt.tight_layout()
    return fig

setattr(StepByStep, 'visualize_outputs', visualize_outputs)
```

Then, let's use the method above to plot the **feature maps** for the layers in the **featurizer** part of our model:

```
featurizer_layers = ['conv1', 'relu1', 'maxp1', 'flatten']

with plt.style.context('seaborn-white'):
    fig = sbs_cnn1.visualize_outputs(featurizer_layers)
```

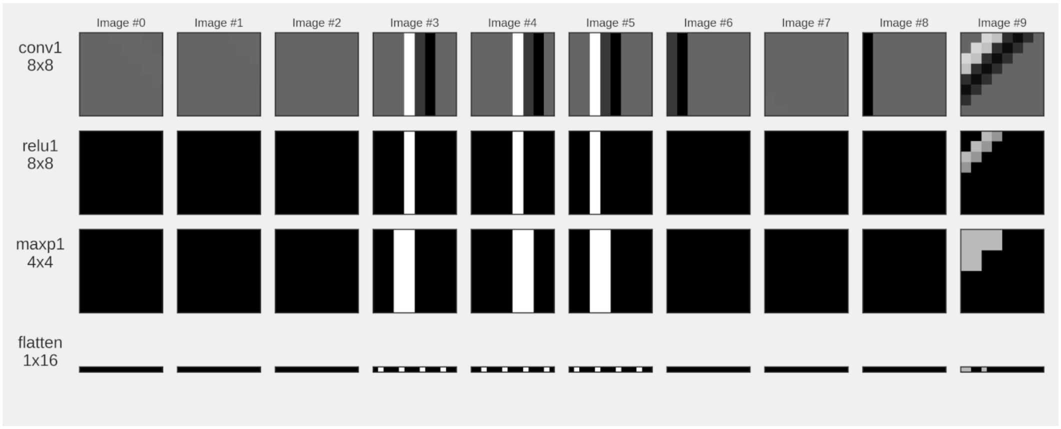


Figure 5.21 - Feature maps (featurizer)

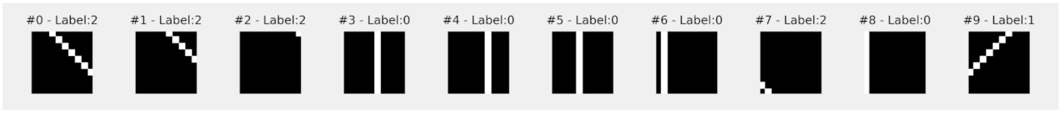


Figure 5.20 - Mini-batch of images (reproduced here for an easier comparison)

Looks cool, right? Even though I've plotted the images in the first four rows with the same size, they have **different dimensions**, as indicated by the row labels on the left. The **shade of gray** is also computed **per row**: the maximum (white) and minimum (black) values were computed across the ten images produced by a given layer, otherwise, some rows would be too dark (the ranges vary a lot from one layer to the next).

What can we learn from these images? First, **convolving the learned filter with the input image** produces some interesting results:

- for **diagonals tilted to the left** (images #0, #1, #2, and #7), the filter seems to **suppress the diagonal** completely
- for **parallel lines** (only verticals in the example above, images #3 to #6, and #8), the filter produces a **striped pattern**, brighter to the left of the original line, darker to its right
- for **diagonals tilted to the right** (only image #9), the filter produces a **thicker line with multiple shades**

Then, the ReLU activation function removes the negative values. Unfortunately, after this operation, images #6 and #8 (parallel vertical lines) had all lines suppressed and seem indistinguishable from images #0, #1, #2, and #7 (diagonals tilted to the left).

Next, max-pooling reduces the dimensions of the images, and they get flattened to represent 16 features.

Now, look at the **flattened features**. That's what the **classifier** will look at to try to split the images into three different classes. For a relatively simple problem like this, we can pretty much *see* the patterns there. Let's see what the classifier layers can make of it.

Visualizing Classifier Layers

The second part of our model, which is aptly called a **classifier**, has the typical structure: a hidden layer (FC1), an activation function, and an output layer (FC2). Let's visualize the **outputs of each and every one of these layers** that were captured by our hook function for the same ten images:

```
classifier_layers = ['fc1', 'relu2', 'fc2']

with plt.style.context('seaborn-white'):
    fig = sbs_cnn1.visualize_outputs(
            classifier_layers, y=labels_batch, yhat=predicted
          )
```

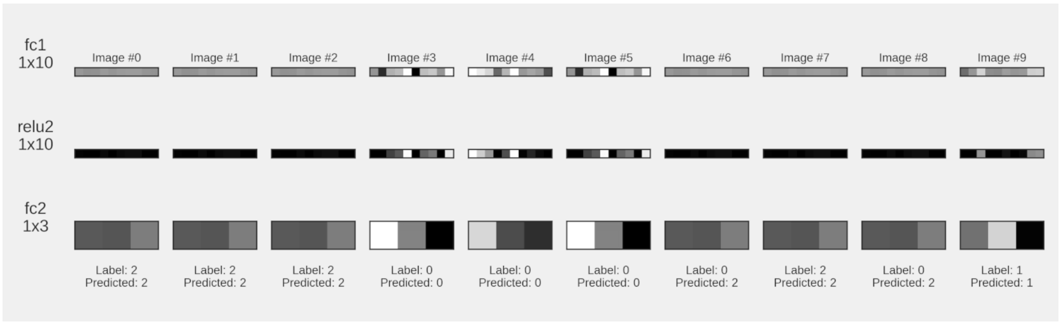


Figure 5.22 - Feature maps (classifier)

The hidden layer performed an **affine transformation** (remember those?), reducing the dimensionality from 16 to 10 dimensions. Next, the activation function, a ReLU, eliminated negative values, resulting in the "*activated*" feature space in the middle row.

Finally, the output layer used these 10 values to compute **three logits**, one for each class. Even without transforming them into probabilities, we know that the **biggest logit wins**. The biggest logit is shown as the **brightest pixel**, so we can tell which class was predicted by looking at the three shades of gray and picking the index of the brightest one.

The classifier got eight out of ten right. It made **wrong predictions for images #6 and #8**. Unsurprisingly, these are the **two images that got their vertical lines suppressed**. The filter doesn't seem to work so well whenever the vertical line is too close to the left edge of the image.

"*How good the model actually is?*"

Good question! Let's check it out.

Accuracy

In Chapter 3, we made predictions using our own `predict` method and used Scikit-Learn's metrics module to evaluate them. Now, let's build a method that takes features (*x*) and labels (*y*), as returned by a data loader, and that takes all necessary steps to produce **two values for each class**: the **number of correct predictions**, and the **number of data points in that class**.

StepByStep Method

```python
def correct(self, x, y, threshold=.5):
    self.model.eval()
    yhat = self.model(x.to(self.device))
    y = y.to(self.device)
    self.model.train()

    # We get the size of the batch and the number of classes
    # (only 1, if it is binary)
    n_samples, n_dims = yhat.shape
    if n_dims > 1:
        # In a multiclass classification, the biggest logit
        # always wins, so we don't bother getting probabilities

        # This is PyTorch's version of argmax,
        # but it returns a tuple: (max value, index of max value)
        _, predicted = torch.max(yhat, 1)
    else:
        n_dims += 1
        # In binary classification, we NEED to check if the
        # last layer is a sigmoid (and then it produces probs)
        if isinstance(self.model, nn.Sequential) and \
           isinstance(self.model[-1], nn.Sigmoid):
            predicted = (yhat > threshold).long()
        # or something else (logits), which we need to convert
        # using a sigmoid
        else:
            predicted = (torch.sigmoid(yhat) > threshold).long()

    # How many samples got classified
    correctly for each class
    result = []
    for c in range(n_dims):
        n_class = (y == c).sum().item()
        n_correct = (predicted[y == c] == c).sum().item()
        result.append((n_correct, n_class))
    return torch.tensor(result)

setattr(StepByStep, 'correct', correct)
```

If the **labels have two or more columns**, it means we're dealing with a **multiclass** classification: the **predicted class** is the one with the **biggest logit**.

If there is a **single column of labels**, that would be a **binary** classification: the **predicted class** will be the **positive class** if the **predicted probability is above a given threshold** (usually 0.5). But there's a *catch* here: if the **last layer** of the model **is not a sigmoid**, we need to **apply it to the logits first** to get the probabilities, and only then compare them with the threshold.

Then, for each possible class, it figures how many predictions match the labels, and appends the result to a tensor. The shape of the resulting tensor will be (*number of classes*, 2), the first column representing correct predictions, the second, the number of data points.

Let's try applying this new method to the first mini-batch of our data loader:

```
sbs_cnn1.correct(images_batch, labels_batch)
```

Output

```
tensor([[5, 7],
        [3, 3],
        [6, 6]])
```

So, there are only two *wrong predictions*, both for class #0 (parallel lines), corresponding to images #6 and #8, as we've already seen in the previous section.

"*What if I want to compute it for **all mini-batches** in a data loader?*"

Loader Apply

On it! That's the role of the static method `loader_apply`: it **applies a function to each mini-batch**, and **stack the results** before applying a reducing function such as sum or mean:

StepByStep Method

```
@staticmethod
def loader_apply(loader, func, reduce='sum'):
    results = [func(x, y) for i, (x, y) in enumerate(loader)]
    results = torch.stack(results, axis=0)

    if reduce == 'sum':
        results = results.sum(axis=0)
    elif reduce == 'mean':
        results = results.float().mean(axis=0)

    return results

setattr(StepByStep, 'loader_apply', loader_apply)
```

Since it is a static method, we can call it from the class itself, passing the loader as its first argument, and a function (or method, in this case) as its second argument. It will call the `correct` method for each mini-batch (as in the example above), and sum all the results up:

```
StepByStep.loader_apply(sbs_cnn1.val_loader, sbs_cnn1.correct)
```

Output

```
tensor([[59, 67],
        [55, 62],
        [71, 71]])
```

Quite simple, right? This method will be very useful for us in the next chapter when we **normalize the images** and thus need to compute the **mean and standard deviation** over all images in the training loader.

From the results above, we see that our model got 185 out of 200 images correctly classified in the validation set, an accuracy of 92.5%! Not bad, not bad at all :-)

Putting It All Together

In this chapter, we focused mostly on the **model configuration** part, adding **convolutional layers** to our model, and defining a different **loss function** to handle the **multiclass** classification problem. We have also added some more methods to our class, such that we can **visualize the filters** learned by our model, attach **hooks** to the model's forward pass, and use the captured results to visualize the corresponding **feature maps**.

Data Preparation

```
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels).long()

# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
```

```python
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]

# We're not doing any data augmentation now
train_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])

# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor,
    transform=train_composer
)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor,
    transform=val_composer
)

# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)

# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16, sampler=sampler
)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Model Configuration

```
torch.manual_seed(13)
model_cnn1 = nn.Sequential()

# Featurizer
# Block 1: 1@10x10 -> n_channels@8x8 -> n_channels@4x4
n_channels = 1
model_cnn1.add_module('conv1', nn.Conv2d(
    in_channels=1, out_channels=n_channels, kernel_size=3
))
model_cnn1.add_module('relu1', nn.ReLU())
model_cnn1.add_module('maxp1', nn.MaxPool2d(kernel_size=2))
# Flattening: n_channels _ 4 _ 4
model_cnn1.add_module('flatten', nn.Flatten())

# Classification
# Hidden Layer
model_cnn1.add_module('fc1',
  nn.Linear(in_features=n_channels*4*4, out_features=10)
)
model_cnn1.add_module('relu2', nn.ReLU())
# Output Layer
model_cnn1.add_module('fc2',
  nn.Linear(in_features=10, out_features=3)
)

lr = 0.1
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn1 = optim.SGD(model_cnn1.parameters(), lr=lr)
```

Model Training

```
sbs_cnn1 = StepByStep(model_cnn1, multi_loss_fn, optimizer_cnn1)
sbs_cnn1.set_loaders(train_loader, val_loader)
sbs_cnn1.train(20)
```

Visualizing Filters

```
fig_filters = sbs_cnn1.visualize_filters('conv1', cmap='gray')
```

Capturing Outputs

```
featurizer_layers = ['conv1', 'relu1', 'maxp1', 'flatten']
classifier_layers = ['fc1', 'relu2', 'fc2']

sbs_cnn1.attach_hooks(
  layers_to_hook=featurizer_layers + classifier_layers
)

images_batch, labels_batch = iter(val_loader).next()
logits = sbs_cnn1.predict(images_batch)
predicted = np.argmax(logits, 1)

sbs_cnn1.remove_hooks()
```

Visualizing Feature Maps

```
with plt.style.context('seaborn-white'):
    fig_maps1 = sbs_cnn1.visualize_outputs(featurizer_layers)
    fig_maps2 = sbs_cnn1.visualize_outputs(
                  classifier_layers, y=labels_batch, yhat=predicted
                )
```

Evaluating Accuracy

```
StepByStep.loader_apply(sbs_cnn1.val_loader, sbs_cnn1.correct)
```

Output

```
tensor([[59, 67],
        [55, 62],
        [71, 71]])
```

Recap

In this chapter, we've introduced convolutions and related concepts and built a convolutional neural network to tackle a multiclass classification problem. This is what we've covered:

- understanding the role of a **kernel/filter** in a convolution
- understanding the role of a **stride** and its impact on the shape of the output
- realizing that there are **as many filters** as **combinations of input and output channels**
- using **padding** to **preserve the shape** of the output
- using **pooling** to **shrink the shape** of the output
- assembling **convolution**, **activation function**, and **pooling** into a **typical convolutional block**
- using a **sequence of convolutional blocks** to pre-process images, **converting them into features**
- (re)building Yann LeCun's **LeNet-5**
- generating a dataset of 1,000 images for a **multiclass classification problem**
- understanding how a **softmax** function transforms **logits** into **probabilities**
- understanding the difference between PyTorch's **negative log-likelihood** and

cross-entropy losses

- highlighting the **importance of choosing the correct combination of last layer and loss function** (again)
- using the loss function to **handle imbalanced datasets**
- building our own **convolutional neural network**, with a **featurizer** made of a **typical convolutional block**, followed by a traditional **classifier with a single hidden layer**
- **visualizing the learned filters**
- understanding and using (forward) **hooks** to **capture the outputs** of intermediate layers of our model
- **removing the hooks** after they served their purpose to not harm the model speed
- using the captured outputs to **visualize feature maps** and understand how the filters learned by the model produce the features that will feed the classifier part
- computing **accuracy** for a **multiclass classification** problem
- creating a **static method** to **apply a function** to all the mini-batches in a **data loader**

Congratulations: you took one big step towards being able to tackle many **computer vision** problems. This chapter introduced the fundamental concepts related to (almost) all things *convolutional*. We still need to add some more tricks to our arsenal, so we make our models even more powerful. In the next chapter, we'll learn about **convolutions over multiple channels**, using **dropout** layers to **regularize a model**, finding a **learning rate**, and the inner workings of **optimizers**.

[85] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter05.ipynb

[86] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter05.ipynb

[87] https://en.wikipedia.org/wiki/Convolution

[88] https://en.wikipedia.org/wiki/Kernel_(image_processing)

[89] https://bit.ly/3sJ7Nn7

[90] https://realpython.com/primer-on-python-decorators/

Chapter 6
Rock, Paper, Scissors

Spoilers

In this chapter, we will:

- **standardize** an image dataset
- train a model to predict **rock**, **paper**, **scissors** poses from hand images
- use **dropout layers** to **regularize** the model
- learn how to **find a learning rate** to train the model
- understand how the **Adam** optimizer uses **adaptive learning rates**
- capture gradients and parameters to **visualize their evolution** during training
- understand how **momentum** and **Nesterov momentum** work
- use **schedulers** to implement **learning rate changes** during training

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 6[91] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[92].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 6's

notebook. If not, just click on `Chapter06.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from PIL import Image
from copy import deepcopy

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, TensorDataset, random_split
from torchvision.transforms import Compose, ToTensor, Normalize, \
ToPILImage, Resize
from torchvision.datasets import ImageFolder
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau, \
MultiStepLR, CyclicLR, LambdaLR

from stepbystep.v2 import StepByStep
from data_generation.rps import download_rps
```

Rock, Paper, Scissors...

☺ ...Lizard, Spock! The "extended" version of the game was displayed in the "The Lizard-Spock Expansion" episode of "The Big Bang Theory" series, and it was developed by Sam Kass and Karen Bryla. To learn more about the extended version, visit Sam Kass' page[93] about the game.

Trivia aside, I guess you're probably a bit *bored* with the image dataset we've been using so far, right? Well, at least, it wasn't MNIST! But it is time to use a **different dataset: ROCK, PAPER, SCISSORS** (unfortunately, no lizard neither Spock).

Rock Paper Scissors Dataset

This dataset was created by Laurence Moroney (lmoroney@gmail.com / laurencemoroney.com) and can be found on his site: Rock Paper Scissors Dataset[94].

The dataset is licensed as Creative Commons (CC BY 2.0). No changes were made to the dataset.

The dataset contains 2,892 images of diverse hands in the typical *rock*, *paper*, and *scissors* poses against a white background. This is a **synthetic dataset** as well since the images were generated using CGI techniques. Each image is 300x300 pixels in size and has four channels (RGBA).

RGBA stands for Red-Green-Blue-Alpha, which is the traditional RGB color model together with an alpha channel indicating how opaque each pixel is. Don't mind the alpha channel, it will be removed later.

The **training set** (2,520 images) can be downloaded here[95] and the **test set** (372 images) can be downloaded here[96]. In the notebook, the datasets will be downloaded and extracted to `rps` and `rps-test-set` folders, respectively.

Here are some examples of its images, one for each pose:

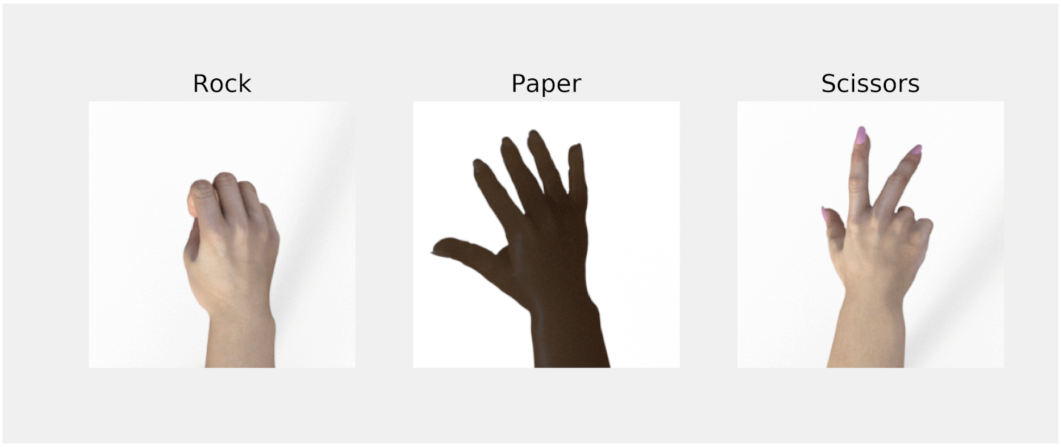


Figure 6.1 - Rock, Paper, Scissors

There are **three classes** once again, so we can use what we learned in Chapter 5.

Data Preparation

The data preparation step will be a bit more demanding this time since we'll be **standardizing the images** (for real this time, no min-max scaling anymore!). Besides, we can use the `ImageFolder` dataset now.

ImageFolder

This is *not* a dataset itself, but a **generic dataset** that you can use with your own images provided that they are properly organized into sub-folders, each sub-folder named after a class and containing the corresponding images.

The "Rock-Paper-Scissors" dataset **is** organized like that: inside the **`rps` folder of the training set**, there are three sub-folders named after the three classes (rock, paper, and scissors).

```
rps/paper/paper01-000.png
rps/paper/paper01-001.png

rps/rock/rock01-000.png
rps/rock/rock01-001.png

rps/scissors/scissors01-000.png
rps/scissors/scissors01-001.png
```

The dataset is also **perfectly balanced**, each sub-folder containing 840 images of its particular class.

The `ImageFolder` dataset requires only the **root folder**, which is the `rps` folder in our case. But it can take another **four optional arguments**:

- `transform`: you know that one already, it tells the dataset which transformations should be applied to each image, like the data augmentation transformations we've seen in previous chapters
- `target_transform`: so far, our targets have always been integers, so this argument wouldn't make sense; but it starts making sense if your target is also an image (for instance, in a segmentation task)
- `loader`: a function that loads an image from a given path, in case you're using weird or atypical formats that cannot be handled by PIL
- `is_valid_file`: a function that checks if a file is corrupt or not

Let's create a dataset then:

Temporary Dataset

```
temp_transform = Compose([Resize(28), ToTensor()])
temp_dataset = ImageFolder(root='rps', transform=temp_transform)
```

We're using only the `transform` optional argument here, and keeping transformations to a minimum.

First, images are **resized to 28x28 pixels** (and automatically transformed to RGB color model by the PIL loader, thus losing the alpha channel), and then **converted to PyTorch's tensors**. Smaller images will make our models faster to train, and more "CPU-friendly". Let's take the first image of the dataset and check its shape and corresponding label:

```
temp_dataset[0][0].shape, temp_dataset[0][1]
```

Output

```
(torch.Size([3, 28, 28]), 0)
```

Perfect!

"Wait, where is the standardization you promised?"

Standardization

To standardize data points, we need to **learn its mean and standard deviation** first. What's the mean pixel value of our rock-paper-scissors images? And standard deviation? To compute them, we need to **load the data**. The good thing is, we have a (temporary) dataset with the resized images already! We're only missing a **data loader**.

Temporary DataLoader

```
temp_loader = DataLoader(temp_dataset, batch_size=16)
```

No need to bother with shuffling, this is **not** the data loader we'll use to train the model anyway. We'll use it to compute statistics only. By the way, we need **statistics for each channel**, as required by the `Normalize` transform.

So, let’s build a function that takes a mini-batch (images and labels), and computes the **mean pixel value and standard deviation per channel of each image**, adding up the results for all images. Better yet, let’s make it a method of our `StepByStep` class too.

StepByStep Method

```
@staticmethod
def statistics_per_channel(images, labels):
    # NCHW
    n_samples, n_channels, n_height, n_weight = images.size()
    # Flatten HW into a single dimension
    flatten_per_channel = images.reshape(n_samples, n_channels, -1)

    # Computes statistics of each image per channel
    # Average pixel value per channel
    # (n_samples, n_channels)
    means = flatten_per_channel.mean(axis=2)
    # Standard deviation of pixel values per channel
    # (n_samples, n_channels)
    stds = flatten_per_channel.std(axis=2)

    # Adds up statistics of all images in a mini-batch
    # (1, n_channels)
    sum_means = means.sum(axis=0)
    sum_stds = stds.sum(axis=0)
    # Makes a tensor of shape (1, n_channels)
    # with the number of samples in the mini-batch
    n_samples = torch.tensor([n_samples]*n_channels).float()

    # Stack the three tensors on top of one another
    # (3, n_channels)
    return torch.stack([n_samples, sum_means, sum_stds], axis=0)

setattr(StepByStep, 'statistics_per_channel',
statistics_per_channel)
```

```
first_images, first_labels = next(iter(temp_loader))
StepByStep.statistics_per_channel(first_images, first_labels)
```

Output

```
tensor([[16.0000, 16.0000, 16.0000],
        [13.8748, 13.3048, 13.1962],
        [ 3.0507,  3.8268,  3.9754]])
```

Applying it to the first mini-batch of images, we get the results above: each **column** represents a **channel**, and the rows are the **number** of data points, the **sum of mean values**, and the **sum of standard deviations**, respectively.

We can leverage the `loader_apply` method we created in the last chapter to get the **sums for the whole dataset**:

```
results = StepByStep.loader_apply(temp_loader,
    StepByStep.statistics_per_channel)
results
```

Output

```
tensor([[2520.0000, 2520.0000, 2520.0000],
        [2142.5359, 2070.0811, 2045.1442],
        [ 526.3024,  633.0677,  669.9554]])
```

So, we can compute the **average mean value** (that sounds weird, I know), and the **average standard deviation**, per channel. Better yet, let's make it a method that takes a data loader and **returns an instance of the `Normalize` transform**, statistics and all:

StepByStep Method

```
@staticmethod
def make_normalizer(loader):
    total_samples, total_means, total_stds = \
      StepByStep.loader_apply(
          loader,
          StepByStep.statistics_per_channel
      )
    norm_mean = total_means / total_samples
    norm_std = total_stds / total_samples
    return Normalize(mean=norm_mean, std=norm_std)

setattr(StepByStep, 'make_normalizer', make_normalizer)
```

IMPORTANT: always use the **training set** to **compute statistics** for standardization!

Now, we can use it to create a transformation that **standardizes** our dataset:

Creating Normalizer Transform

```
normalizer = StepByStep.make_normalizer(temp_loader)
normalizer
```

Output

```
Normalize(mean=tensor([0.8502, 0.8215, 0.8116]),
          std=tensor([0.2089, 0.2512, 0.2659]))
```

Remember that PyTorch converts the pixel values into the [0, 1] range. The average mean value of a pixel, for the red (first) channel, is 0.8502, while its average standard deviation is 0.2089.

In the next chapter, we'll use **pre-computed statistics** to standardize the inputs when using a **pre-trained model**.

The Real Datasets

It's time to build our *real* datasets using the `Normalize` transform with the statistics we learned from the (temporary) training set. The data preparation step looks like this:

Data Preparation

```
composer = Compose([Resize(28),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(
    train_data, batch_size=16, shuffle=True
)
val_loader = DataLoader(val_data, batch_size=16)
```

Even though the second part of the dataset was named `rps-test-set` by its author, we'll be using it as our validation dataset. Since **each dataset**, training and validation, corresponds to a **different folder**, there is no need to split anything.

Next, we use both datasets to create the corresponding data loaders, remembering to **shuffle the training set** now.

Let's take a peek at some images from the *real* training set:

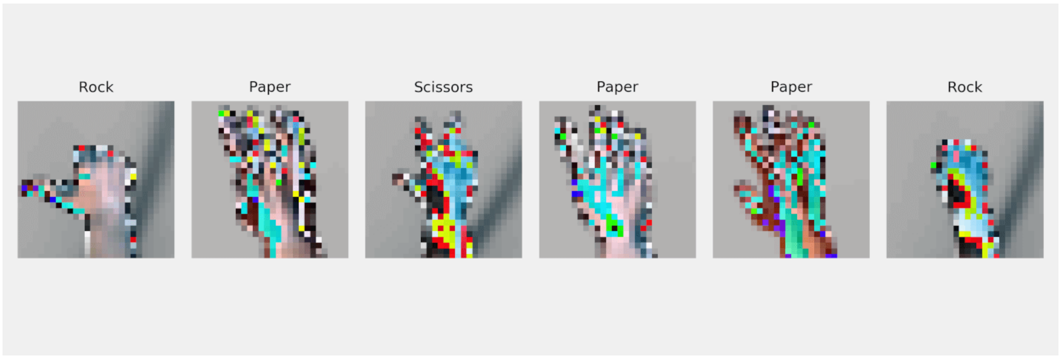


Figure 6.2 - Training set (normalized)

"*What's wrong with the colors?*"

There is nothing wrong with the colors, it is just the **effect of the standardization of the pixel values**. Now that we have colored images, we can take a step back into the convolution world and see how it handles…

Three-Channel Convolutions

Before, there was a single channel image and a single channel filter. Or many filters, but all of them still having a single channel. Now, there is a **three-channel image** and a **three-channel filter**. Or many filters, but all of them still having three channels.

Every **filter** has as many **channels** as the **image** it is convolving.

Convolving a three-channel filter over a three-channel image **still produces a single value**, as depicted in the figure below:

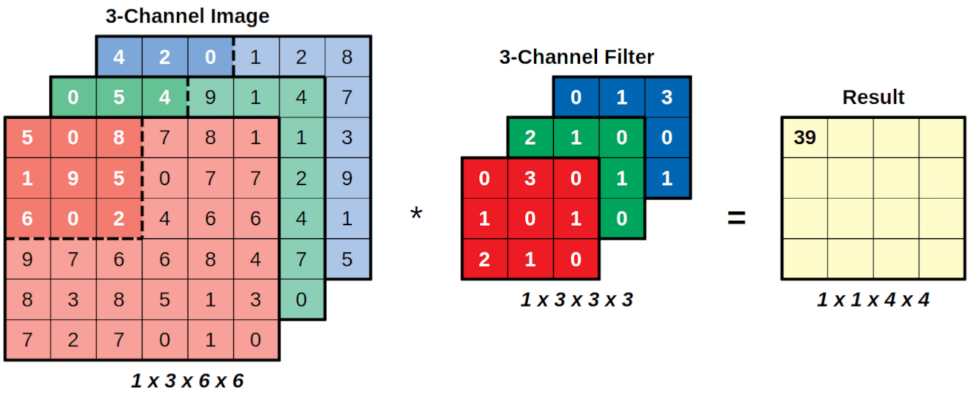


Figure 6.3 - Convolution with multiple channels

We can think of it as performing **three convolutions**, each corresponding to the **element-wise multiplication** of the matching **region/channel** and **filter/channel**, resulting in **three values**, one for each channel. **Adding up the results for each channel** produces the **expected single value**. The figure below should illustrate it better:

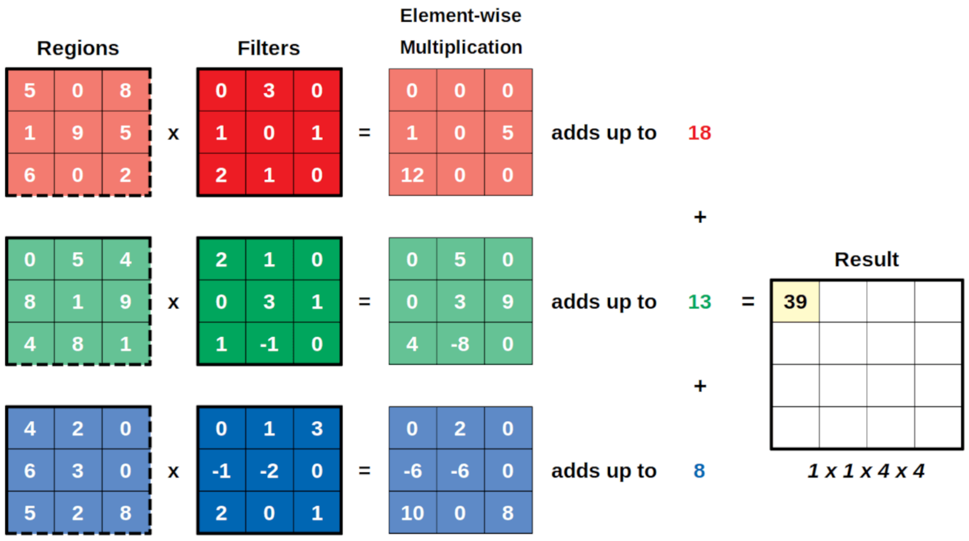


Figure 6.4 - Convolution over each channel

We can also look at it in code if you prefer:

```
regions = np.array([[[[5, 0, 8],
                      [1, 9, 5],
                      [6, 0, 2]],
                     [[0, 5, 4],
                      [8, 1, 9],
                      [4, 8, 1]],
                     [[4, 2, 0],
                      [6, 3, 0],
                      [5, 2, 8]]]])
regions.shape
```

Output

```
(1, 3, 3, 3)
```

```
three_channel_filter = np.array([[[[0, 3, 0],
                                   [1, 0, 1],
                                   [2, 1, 0]],
                                  [[2, 1, 0],
                                   [0, 3, 1],
                                   [1, -1, 0]],
                                  [[0, 1, 3],
                                   [-1, -2, 0],
                                   [2, 0, 1]]]])
three_channel_filter.shape
```

Output

```
(1, 3, 3, 3)
```

```
result = F.conv2d(torch.as_tensor(regions),
                  torch.as_tensor(three_channel_filter))
result, result.shape
```

Output

```
(tensor([[[[39]]]]), torch.Size([1, 1, 1, 1]))
```

"*What if I have **two filters**?*"

Glad you asked! The figure below illustrates the fact that **every filter** has **as many channels** as the image being convolved.

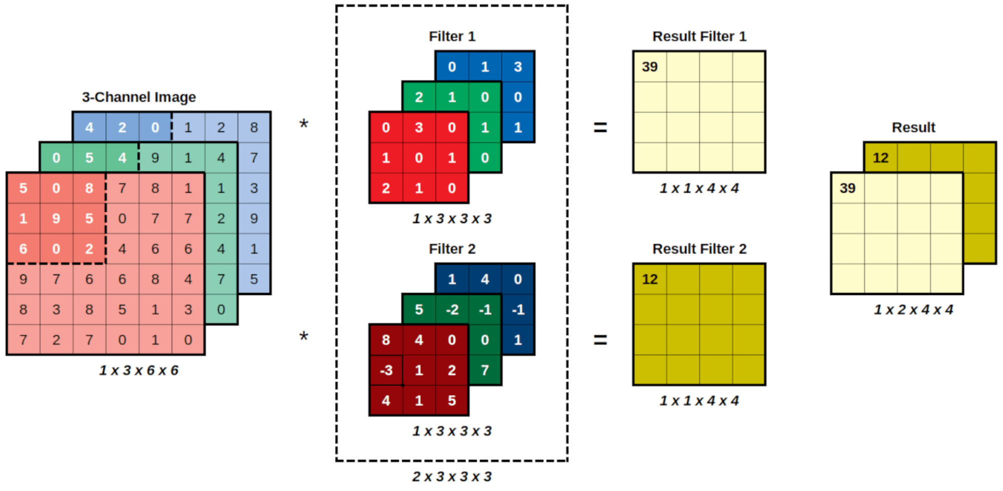


Figure 6.5 - Two filters over three channels

If you have **two filters**, and the input **image has three channels**, each **filter has three channels** as well, and the **output has two channels**.

The convolution produces as many **channels** as there are **filters**.

OK, it is time to develop a…

Fancier Model

Let's leave the `Sequential` model aside for now and build a **model class** once again. This time, our constructor method will take two arguments: `n_filters` and `p`. We'll use `n_filters` as the number of **output channels** for **both convolutional blocks** of our model (yes, there are two now!). And, as you can see from the code below, we'll use `p` as the **probability of dropout**.

Fancier Model (Constructor)

```
class CNN2(nn.Module):
    def __init__(self, n_filters, p=0.0):
        super(CNN2, self).__init__()
        self.n_filters = n_filters
        self.p = p
        # Creates the convolution layers
        self.conv1 = nn.Conv2d(
            in_channels=3,
            out_channels=n_filters,
            kernel_size=3
        )
        self.conv2 = nn.Conv2d(
            in_channels=n_filters,
            out_channels=n_filters,
            kernel_size=3
        )
        # Creates the linear layers
        # Where do this 5 * 5 come from?! Check it below
        self.fc1 = nn.Linear(n_filters * 5 * 5, 50)
        self.fc2 = nn.Linear(50, 3)
        # Creates dropout layers
        self.drop = nn.Dropout(self.p)
```

There are two convolutional layers, and two linear layers, `fc1` (the hidden layer) and `fc2` (the output layer).

*"Where are the layers for **activation functions** and **max-pooling**?"*

Well, the max-pooling layer **doesn't learn anything**, so we can use its **functional form**: `F.max_pool2d`. The same goes for the chosen activation function: `F.relu`.

If you choose the **parametric ReLU (PReLU)**, you **shouldn't use the functional form** since it needs to **learn the coefficient of leakage** (the slope of the negative part).

On the one hand, you keep the model's attributes to a minimum. On the other hand, you don't have layers to *hook* anymore, so you **cannot capture** the output of **activation functions** and **max-pooling operations** anymore.

Let's create our two convolutional blocks in a function aptly named **featurizer**:

Fancier Model (Featurizer)

```
    def featurizer(self, x):
        # First convolutional block
        # 3@28x28 -> n_filters@26x26 -> n_filters@13x13
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        # Second convolutional block
        # n_filters@13x13 -> n_filters@11x11 -> n_filters@5x5
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        # Input dimension (n_filters@5x5)
        # Output dimension (n_filters _ 5 _ 5)
        x = nn.Flatten()(x)
        return x
```

This structure, where an argument `x` is both **input** and **output** of every operation in a sequence, is fairly common. The featurizer produces a feature tensor of size `n_filters` times 25.

The next step is to build the *classifier* using the linear layers, one as a hidden layer, the other as the output layer. But there is **more** to it: there is a **dropout layer before each linear layer**, and it will **drop values with a probability `p`** (the second argument of our constructor method):

Fancier Model (Classifier)

```
    def classifier(self, x):
        # Classifier
        # Hidden Layer
        # Input dimension (n_feature * 5 * 5)
        # Output dimension (50)
        if self.p > 0:
            x = self.drop(x)
        x = self.fc1(x)
        x = F.relu(x)
        # Output Layer
        # Input dimension (50)
        # Output dimension (3)
        if self.p > 0:
            x = self.drop(x)
        x = self.fc2(x)
        return x
```

"*How does dropout work?*"

We'll dive deeper into it in the next section, but we need to *finish* our model class first. What's left to be done? The implementation of the `forward` method:

Fancier Model (Forward)

```
    def forward(self, x):
        x = self.featurizer(x)
        x = self.classifier(x)
        return x
```

It takes the **inputs** (a mini-batch of images, in this case), runs them through the **featurizer** first, and then runs the produced features through the **classifier**, which produces **three logits**, one for each class.

Dropout

Dropout is an important piece of deep learning models. It is used as a **regularizer**, that is, it tries to **prevent overfitting** by **forcing the model** to find **more than one way to achieve the target**.

The general idea behind **regularization** is that, if left unchecked, a model will try to find the "*easy way out*" (can you blame it?!) to achieve the target. What does it mean? It means it may end up **relying on a handful of features** because these features were found to be more relevant in the training set. Maybe they are, maybe they aren't… it could very well be a **statistical fluke**, who knows, right?

To make the model more robust, some of the features are randomly **denied to it**, so it has to achieve the target **in a different way**. It makes training harder but it should result in **better generalization**, that is, the model should perform better when handling **unseen data** (like the data points in the validation set).

The whole thing looks a lot like the **randomization of features** used in **random forests** to perform the splits. Each tree, or even better, each split has access to a **subset of features only**.

"How does this "feature randomization" works in a deep learning model?"

To illustrate it, let's build a sequential model with a single `Dropout` layer:

```
dropping_model = nn.Sequential(nn.Dropout(p=0.5))
```

> ❓ *"Why do I need a model for* ***this****? Can't I use the* ***functional*** *form* `F.dropout` *instead?"*

Yes, a functional dropout would go just fine here, but I wanted to illustrate another point too, so please bear with me. Let's also create some neatly spaced points to make it easier to understand the effect of dropout.

```
spaced_points = torch.linspace(.1, 1.1, 11)
spaced_points
```

Output

```
tensor([0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000,
        0.8000, 0.9000, 1.0000, 1.1000])
```

Next, let's use these points as inputs of our amazingly simple model:

```
torch.manual_seed(44)

dropping_model.train()
output_train = dropping_model(spaced_points)
output_train
```

Output

```
tensor([0.0000, 0.4000, 0.0000, 0.8000, 0.0000, 1.2000, 1.4000,
        1.6000, 1.8000, 0.0000, 2.2000])
```

There are **many things** to notice here:

- the model is in `train` mode (very important, hold on to this!)
- since this model **does not have any weights**, it becomes clear that **dropout drops inputs, not weights**
- it dropped **four elements** only
- the **remaining elements** have **different values** now!

"What's going on here?"

First, **dropping is probabilistic**, so **each input had a 50% chance of being dropped**. In our tiny example, by chance, only four out of ten were actually dropped (hold on to this thought too!).

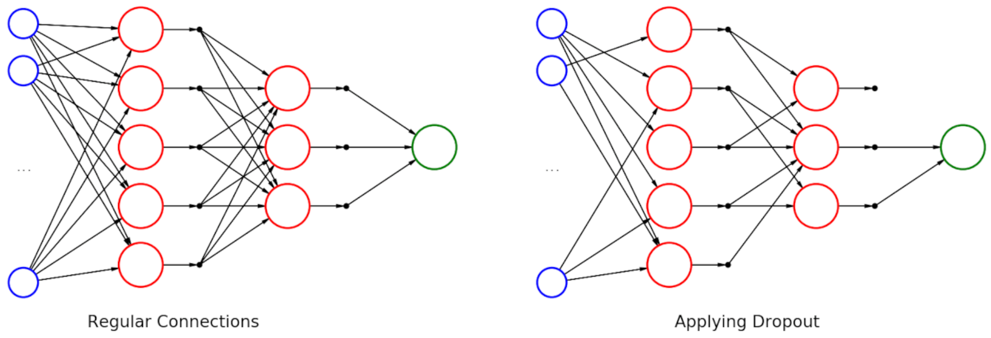


Figure 6.6 - Applying Dropout

Second, the **remaining elements** need to be **proportionally adjusted by a factor of `1/p`**. In our example, that's a factor of two.

```
output_train / spaced_points
```

Output

```
tensor([0., 2., 0., 2., 0., 2., 2., 2., 2., 0., 2.])
```

"Why?"

This adjustment has the purpose of **preserving** (or at least trying to) the **overall level of the outputs** in that particular layer that's "*suffering*" the dropout. So, let's imagine these inputs (after dropping) will feed a **linear layer** and, for educational purposes, that all its **weights are equal to one** (and bias equals zero). As you already know, a linear layer will multiply these weights by the (dropped) inputs and sum them up:

```
F.linear(output_train, weight=torch.ones(11), bias=torch.tensor(0))
```

Output

```
tensor(9.4000)
```

The sum is 9.4. It would have been **half of it** (4.7) without the adjusting factor.

*"OK, so what? Why do I need to **preserve the level of the outputs** anyway?"*

Because **there is no dropping in evaluation mode**! We've talked about it briefly in the past… the dropout is **random** in nature, so it would produce slightly (or maybe not so slightly) **different predictions** for the **same inputs**. You don't want that, that's bad business. So, let's **set our model to `eval` mode** (and that's why I chose to make it a model instead of using functional dropout) and see what happens there:

```
dropping_model.eval()
output_eval = dropping_model(spaced_points)
output_eval
```

Output

```
tensor([0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000,
        0.8000, 0.9000, 1.0000, 1.1000])
```

Pretty *boring*, right? This isn't doing anything!

Finally, an **actual difference in behavior** between `train` and `eval` modes! It was long overdue!

The inputs are just **passing through**. What's the implication of this? Well, that **linear layer** that receives these values is still multiplying them by the weights and summing them up:

```
F.linear(output_eval, weight=torch.ones(11), bias=torch.tensor(0))
```

Output

```
tensor(6.6000)
```

This is the sum of **all inputs** (because all the weights were set to one and no input was dropped). If there was no adjusting factor, the outputs in evaluation and training mode would be substantially different, simply because there would be **more terms to add up** in **evaluation mode**.

*"I am still not convinced… **without adjusting** the output would be **4.7**, which is **closer to 6.6** than the **adjusted 9.4**… what is up?"*

This happened because dropping is **probabilistic**, and only four out of ten elements were actually dropped (that was the thought I asked you to hold on to). The factor **adjusts for the average number of dropped elements**. We set the probability to 50% so, **on average, five elements will be dropped**. By the way, if you change the seed to *45* and re-run the code, it will *actually* drop half of the inputs, and the adjusted output will be 6.4 instead of 9.4.

Instead of setting a different random seed and manually checking which value it produces, let's generate 1,000 scenarios and compute the **sum of the adjusted dropped outputs** to get their distribution:

```
torch.manual_seed(17)
p = 0.5
distrib_outputs = torch.tensor([
  F.linear(F.dropout(spaced_points, p=p),
           weight=torch.ones(11), bias=torch.tensor(0))
  for _ in range(1000)
])
```

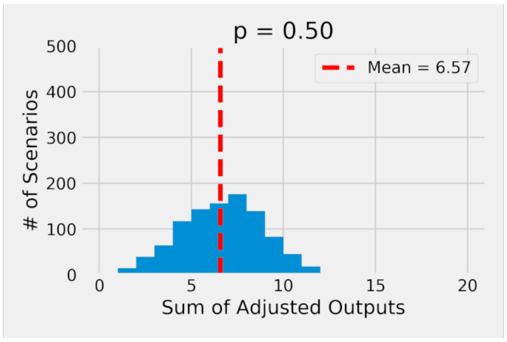


Figure 6.7 - Distribution of outputs

The figure above shows us that, for that set of inputs, the output of our simple linear layer with dropout will **not be exactly 6.6** anymore, but **something between 0 and 12**. The mean value for all scenarios is *quite close* to 6.6, though.

Dropout not only **drops some inputs** but, due to its probabilistic nature, **produces a distribution of outputs**.

In other words, the model needs to learn how to handle a **distribution** of values that is **centered at the value the output would have if there was no dropout**.

Moreover, the choice of the **dropout probability** determines **how spread** the outputs will be:

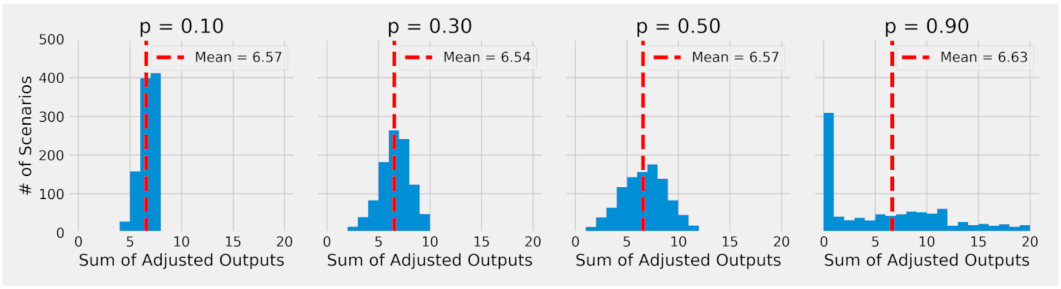


Figure 6.8 - Output distribution for dropout probabilities

On the left, if there is barely any dropout (*p*=0.10), the sum of adjusted outputs is tightly distributed around the mean value. For more **typical dropout probabilities** (like 30% or 50%), the distribution **may** take some more extreme values.

If we go to **extremes**, like a dropout probability of 90%, the distribution gets a *bit* degenerated, I would say… it is pretty much all over the place (and it has a lot of scenarios where *everything gets dropped*, hence the tall bar at zero).

The **variance of the distribution of outputs grows with the dropout probability**.

A higher dropout probability makes it harder for your model to learn - that’s what regularization does.

"*Can I use **dropout** with the **convolutional layers**?*"

Two-Dimensional Dropout

Yes, you can, but not **that** dropout. There is a **specific dropout** to be used with convolutional layers: `nn.Dropout2d`. Its dropout procedure is a bit different, though: instead of dropping individual inputs (which would be pixel values in a given channel), it **drops entire channels/filters**. So, if a convolutional layer produces 10 filters, a two-dimensional dropout with a probability of 50% would drop **five filters** (on average), while the remaining filters would have all their pixel values left untouched.

"Why does it drop entire channels instead of dropping pixels?"

Randomly dropping pixels doesn't do much for regularization because **adjacent pixels are strongly correlated**, that is, they have quite similar values. You can think of it this way: if there are some **dead pixels randomly spread in an image**, the missing pixels can probably be easily filled with the values of the adjacent pixels. On the other hand, if a full channel is dropped (in an RGB image), the **color changes** (good luck figuring the values for the missing channel!).

The figure below illustrates the effect of both, regular and two-dimensional, dropout procedures on an image of our dataset:

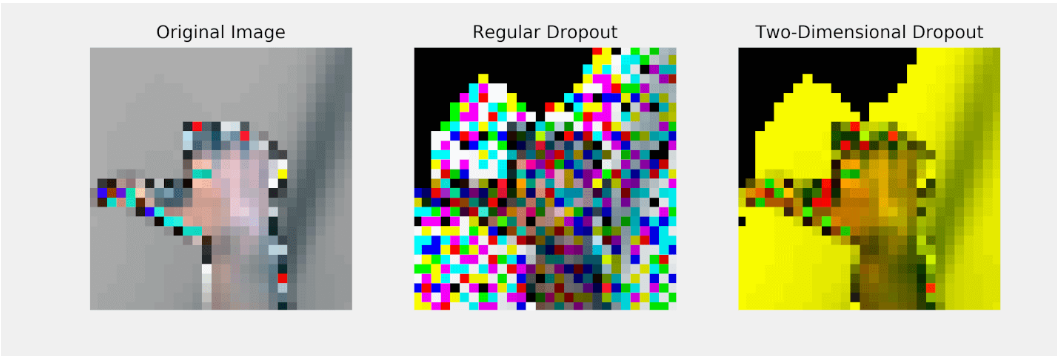


Figure 6.9 - Dropping channels with `Dropout2d`

Sure, in deeper layers, there is no correspondence between channel and color anymore, but each channel still encodes *some feature*. By randomly dropping some channels, two-dimensional dropout achieves the desired regularization.

Now, let's make it *a bit harder* for our model to learn by setting its dropout probability to 30% and observing how it fares…

Model Configuration

The configuration part is short and straightforward: we create a **model**, a **loss function**, and an **optimizer**.

The model will be an instance of our `CNN2` class with **five filters** and a **dropout probability of 30%**. Our dataset has three classes, so we're using a `CrossEntropyLoss` (which will take the **three logits** produced by our model).

Optimizer

Regarding the **optimizer**, let's *ditch* the SGD optimizer and use *Adam* for a change. Stochastic gradient descent is *simple and straightforward*, as we've learned in Chapter 0, but it is also *slow*. So far, the training speed of SGD was not an issue because our problems were quite simple. But, as our models grow a bit more complex, we can benefit from choosing a different optimizer.

Adaptive Moment Estimation (Adam) uses **adaptive learning rates**, computing a learning rate **for each parameter**. Yes, you read it right: **each parameter has a learning rate to call its own**!

If you dig into the `state_dict` of an Adam optimizer, you'll find tensors shaped like the parameters of every layer in your model that Adam will use to compute the corresponding learning rates. True story!

`Adam` is known to achieve good results **fast** and it is likely a safe choice of optimizer. We'll get back to its inner workings in a later section.

Learning Rate

Another thing we need to keep in mind is that `0.1` won't cut as learning rate anymore. Remember what happens when the learning rate is **too big**? The loss doesn't go down or, even worse, goes up! We need to go **smaller**, **much smaller** than that. For this example, let's use **`3e-4`**, the "Karpathy's Constant"[97]. Even though it was meant as a joke, it still is in the right order of magnitude, so let's give it a try.

Model Configuration

```
torch.manual_seed(13)
model_cnn2 = CNN2(n_feature=5, p=0.3)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn2 = optim.Adam(model_cnn2.parameters(), lr=3e-4)
```

We have everything in place to start the…

Model Training

Once again, we use our `StepByStep` class to handle model training for us.

Model Training

```
sbs_cnn2 = StepByStep(model_cnn2, multi_loss_fn, optimizer_cnn2)
sbs_cnn2.set_loaders(train_loader, val_loader)
sbs_cnn2.train(10)
```

You should expect training to **take a while** since this model is more complex than previous ones (6823 parameters against 213 parameters for the last chapter's model). After it finishes, the computed losses should look like this:

```
fig = sbs_cnn2.plot_losses()
```

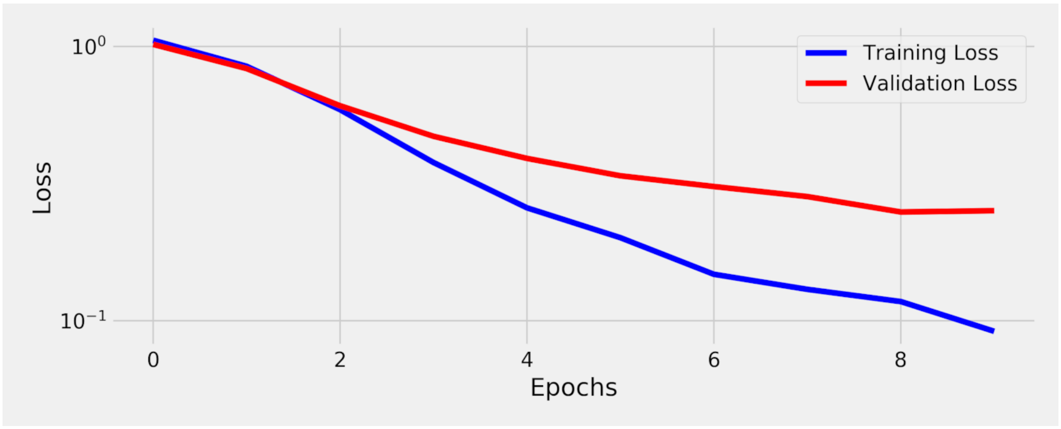


Figure 6.10 - Losses

Accuracy

We can also check the model's accuracy for each class:

```
StepByStep.loader_apply(val_loader, sbs_cnn2.correct)
```

Output

```
tensor([[ 92, 124],
        [106, 124],
        [115, 124]])
```

The model got 313 out of 372 right. That's 84.1% accuracy on the validation set - not bad!

Regularizing Effect

Dropout layers are used for **regularizing**, that is, they should **reduce overfitting** and **improve generalization**. Or so they say :-)

Let's (empirically) verify this claim by *training a model identical in every way BUT the dropout*, and **compare their losses and accuracy**.

```
torch.manual_seed(13)
# Model Configuration
model_cnn2_nodrop = CNN2(n_feature=5, p=0.0)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn2_nodrop = optim.Adam(
    model_cnn2_nodrop.parameters(), lr=3e-4
)
# Model Training
sbs_cnn2_nodrop = StepByStep(
    model_cnn2_nodrop, multi_loss_fn, optimizer_cnn2_nodrop
)
sbs_cnn2_nodrop.set_loaders(train_loader, val_loader)
sbs_cnn2_nodrop.train(10)
```

Then we can plot the losses of the model above (*no dropout*) together with the losses from our previous model (**30% dropout**):

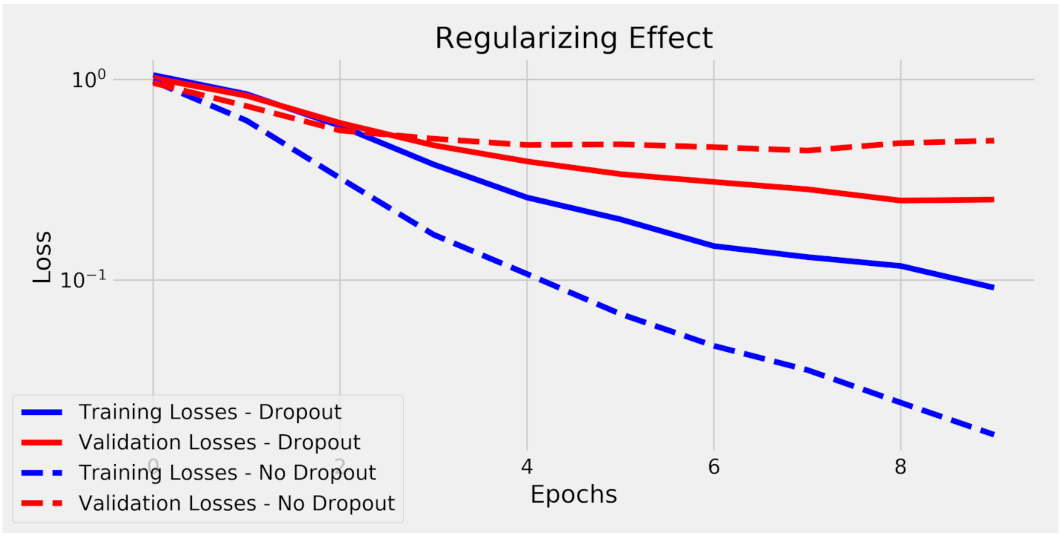


Figure 6.11 - Losses (with and without regularization)

This is actually a very nice depiction of the **regularizing effect of using dropout**:

- training loss is **bigger** with **dropout** - after all, dropout makes training harder

- validation loss is **smaller** with **dropout** - it means that the model is **generalizing better** and achieving a better performance on unseen data, which is the whole point of using a regularization method like dropout

We can also observe this effect by looking at the accuracies for both sets and models. First, the **no dropout** model, which is expected to **overfit to the training data**:

```
print(StepByStep.loader_apply(
          train_loader, sbs_cnn2_nodrop.correct).sum(axis=0),
      StepByStep.loader_apply(
          val_loader, sbs_cnn2_nodrop.correct).sum(axis=0))
```

Output

```
tensor([2518, 2520]) tensor([293, 372])
```

That's 99.92% accuracy on the training set! And 78.76% on the validation set… smells like overfitting!

Then, let's look at the regularized version of the model:

```
print(StepByStep.loader_apply(
          train_loader, sbs_cnn2.correct).sum(axis=0),
      StepByStep.loader_apply(
          val_loader, sbs_cnn2.correct).sum(axis=0))
```

Output

```
tensor([2504, 2520]) tensor([313, 372])
```

That's 99.36% accuracy on the training set - still quite high! But we got 84.13% on the validation set now… a **narrower gap** between training and validation accuracy is always a good sign. You can also try **different probabilities** of dropout and observe how much better (or worse!) the results get.

Visualizing Filters

There are **two** convolutional layers in this model, let's visualize them! For the first one, conv1, we got:

```
model_cnn2.conv1.weight.shape
```

Output

```
torch.Size([5, 3, 3, 3])
```

Its shape indicates it produced **five filters** for each one of the **three input channels** (15 filters in total), and each filter is 3x3 pixels.

```
fig = sbs_cnn2.visualize_filters('conv1')
```

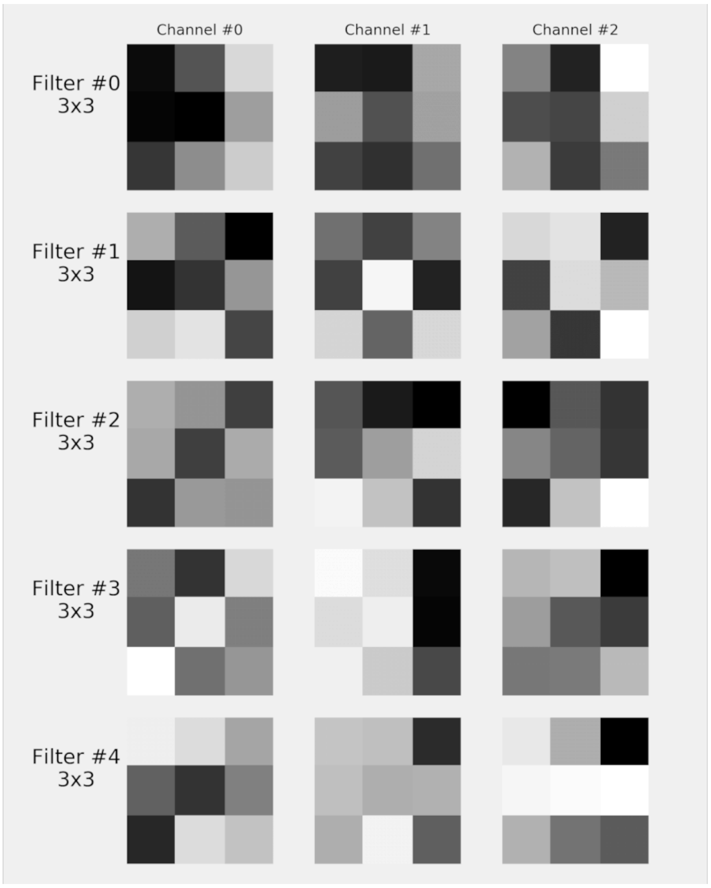


Figure 6.12 - Visualizing filters for `conv1` *layer*

For the second convolutional layer, `conv2`, we got:

```
model_cnn2.conv2.weight.shape
```

Output

```
torch.Size([5, 5, 3, 3])
```

Its shape indicates it produced **five filters** for each one of the **five input channels** (25 filters in total), and each filter is 3x3 pixels.

```python
fig = sbs_cnn2.visualize_filters('conv2')
```

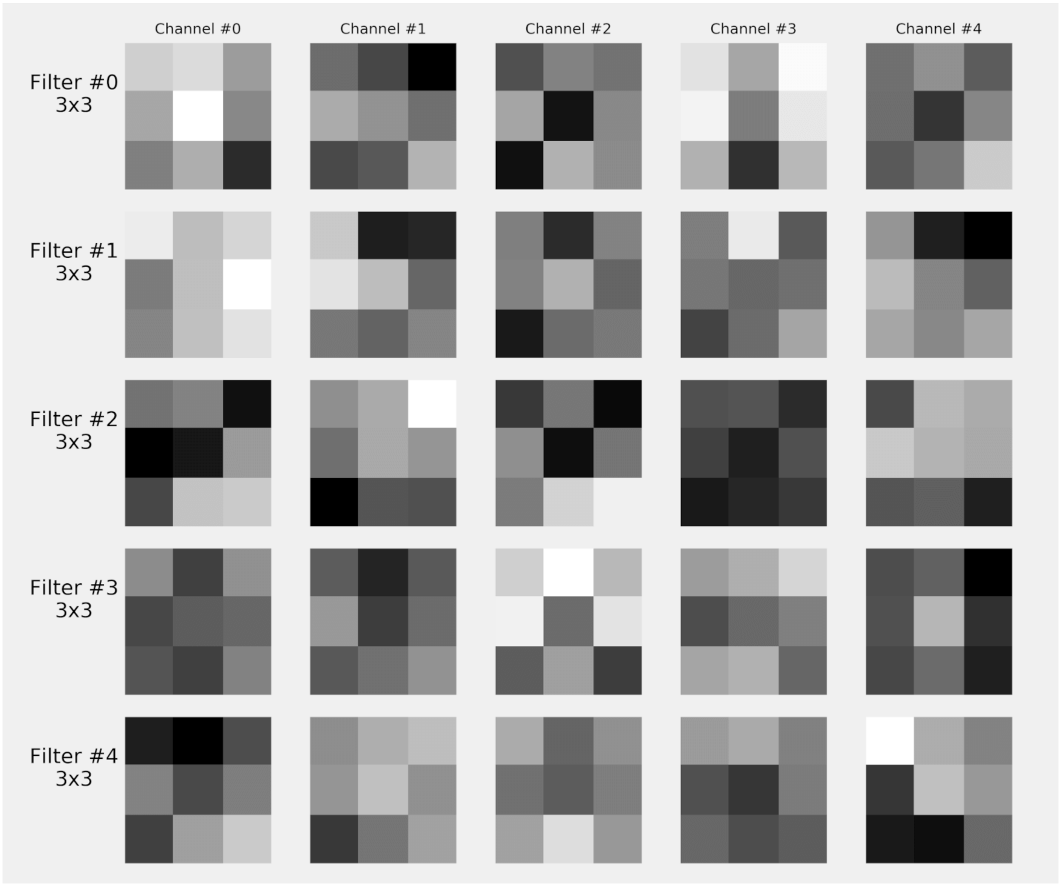


Figure 6.13 - Visualizing filters for `conv2` *layer*

Learning Rates

It is time to have ***"the talk"***... it cannot be postponed any longer, we need to talk about **choosing a learning rate**! It is no secret that the learning rate is **the** most important hyper-parameter of all - it drives the **update of the parameters**, that is, it drives **how fast a model learns** (hence, learning rate).

Choosing a learning rate that works well for a given model (and dataset) is a difficult task, mostly done by **trial-and-error** since there is no analytical way of finding the *optimal* learning rate. One thing we can say for sure is that it should be **less than `1.0`**, and it is likely **bigger than `1e-6`**.

"*Well, that doesn't help much…*"

Indeed, it doesn't. So, let's discuss how we can make it **a bit more specific** than that.

In previous chapters, we used **`0.1`** as the learning rate, which is **kinda big** but worked well for really simple problems. As models grow more complex, though, that value is definitely *too big* for them, and **one order of magnitude smaller (`0.01`)** is a better **starting point**.

"*What if it is **still too big** and the **loss doesn't go down**?*"

That's a real possibility, and one possible way of handling this is to perform a **grid search**, trying **multiple learning rates** over a few epochs each and comparing the evolution of the losses. This is expensive, computationally speaking, since you need to train the model multiple times, but it may still be feasible if your model is not *too* big.

"*How do I choose values for the grid search?*"

It is commonplace to reduce the learning rate by a **factor of 3** or a **factor of 10**. So, your learning rate values could very well be `[0.1, 0.03, 0.01, 3e-3, 1e-3, 3e-4, 1e-4]` (using a factor of 3) or `[0.1, 0.01, 1e-3, 1e-4, 1e-5]` (using a factor of 10). In general, if you plot the learning rates against their corresponding losses, this is what you should expect:

- if the learning rate is **too small**, the model doesn't learn much, and the **loss remains high**
- if the learning rate is **too big**, the model doesn't converge to a solution, and the **loss grows**

- **in between** those two extremes, the **loss should be smaller**, hinting at the **right order of magnitude for the learning rate**

Finding LR

It turns out, you **don’t have to grid search** the learning rate like that. In 2017, Leslie N. Smith published "*Cyclical Learning Rates for Training Neural Networks*"[98] where he outlines a procedure to **quickly find an appropriate range for the initial learning rate** (more on the *cyclical* part of his paper later!). This technique is called **LR Range Test**, and it is quite a simple solution to get a first estimate for the appropriate learning rate.

The general idea is pretty much the same as the grid search: it tries multiple learning rates and logs the corresponding losses. But here comes the difference: **it evaluates the loss over a single mini-batch**, and then changes the learning rate before moving on to the next mini-batch.

This is **computationally cheap** (it is performing ONE training step only for each candidate) and it can be performed **inside the same training loop**.

"*Wait a minute! Wouldn’t the results be affected by the previous training steps performed using different learning rates?*"

Well, technically, yes. But this is not such a big deal: first, we’re looking for a **ballpark estimate** of the learning rate, not a precise value; second, these updates will barely nudge the model from its initial state. It is easier to live with this difference than to reset the model every single time.

First, we need to define the **boundaries** for the test (`start_lr` and `end_lr`), and the **number of iterations** (`num_iter`) to move from one to the other. On top of that, we can choose to change **how** to make the increments: linearly or exponentially. Let’s build a higher-order function that takes all those arguments and returns another function, one that returns the multiplying factor given the current iteration number:

Higher-Order Learning Rate Function Builder

```
def make_lr_fn(start_lr, end_lr, num_iter, step_mode='exp'):
    if step_mode == 'linear':
        factor = (end_lr / start_lr - 1) / num_iter
        def lr_fn(iteration):
            return 1 + iteration * factor
    else:
        factor = (np.log(end_lr) - np.log(start_lr)) / num_iter
        def lr_fn(iteration):
            return np.exp(factor)**iteration
    return lr_fn
```

Now, let's try it out: say we'd like to try **ten different learning rates** between `0.01` and `0.1`, and the increments should be exponential:

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr, num_iter, step_mode='exp')
```

There is a **factor of 10** between the two rates. If we apply this function to a sequence of iteration numbers, from 0 to 10, that's what we get:

```
lr_fn(np.arange(num_iter + 1))
```

Output

```
array([ 1.        ,  1.25892541,  1.58489319,  1.99526231,
        2.51188643,  3.16227766,  3.98107171,  5.01187234,
        6.30957344,  7.94328235, 10.        ])
```

If we **multiply** these values by the **initial learning rate**, we'll get an array of learning rates ranging from `0.01` to `0.1` as expected:

```
start_lr * lr_fn(np.arange(num_iter + 1))
```

Output

```
array([0.01      , 0.01258925, 0.01584893, 0.01995262,
       0.02511886, 0.03162278, 0.03981072, 0.05011872,
       0.06309573, 0.07943282, 0.1       ])
```

*"Cool, but how do I **change the learning rate** of an optimizer?"*

Glad you asked! It turns out, we can assign a **scheduler** to an optimizer, such that it **updates the learning rate** as it goes. We're going to dive deeper into learning rate schedulers in a couple of sections. For now, it suffices to know that we can **make it follow a sequence of values like the one above** using a scheduler that takes a **custom function**. Coincidence? I think not! That's what we'll be using `lr_fn` for:

```
dummy_model = CNN2(n_feature=5, p=0.3)
dummy_optimizer = optim.Adam(dummy_model.parameters(), lr=start_lr)
dummy_scheduler = LambdaLR(dummy_optimizer, lr_lambda=lr_fn)
```

The `LambdaLR` scheduler takes an optimizer and a custom function as arguments and modifies the learning rate of that optimizer accordingly. To make it happen, though, we need to call the **scheduler's `step` method**, but only **after calling the optimizer's own `step` method**:

```
dummy_optimizer.step()
dummy_scheduler.step()
```

After **one step**, the learning rate should have been updated to match the second value in our array (`0.01258925`). Let's double-check it using the scheduler's `get_last_lr` method:

```
dummy_scheduler.get_last_lr()[0]
```

Output

```
0.012589254117941673
```

Perfect! Now let's build the **actual range test**. This is what we're going to do:

- since we'll be updating both **model** and **optimizer**, we need to **store their initial states** so they could be restored in the end
- create both **custom function** and corresponding **scheduler**, just like in the snippets above
- (re)implement a **training loop** over **mini-batches**, so we can **log** the **learning rate** and **loss** at every step
- **restore** model and optimizer states

Moreover, since we're using a **single mini-batch** to evaluate the loss, the resulting values are likely jumping up and down a lot. So, it is better to **smooth the curve** using an **exponentially weighted moving average (EWMA)** (we'll talk about EWMAs in much more detail in the next section) to more easily identify the trend in the values.

This is what the method looks like:

StepByStep Method

```
def lr_range_test(self, data_loader, end_lr, num_iter=100,
                  step_mode='exp', alpha=0.05, ax=None):
    # The test updates both model and optimizer , so we need to
    #  store their initial states to restore them in the end
    previous_states = {
        'model': deepcopy(self.model.state_dict()),
        'optimizer': deepcopy(self.optimizer.state_dict())
    }
    # Retrieves the learning rate set in the optimizer
    start_lr = self.optimizer.state_dict()['param_groups'][0]['lr']

    # Builds a custom function and corresponding scheduler
    lr_fn = make_lr_fn(start_lr, end_lr, num_iter)
    scheduler = LambdaLR(self.optimizer, lr_lambda=lr_fn)

    # Variables for tracking results and iterations
    tracking = {'loss': [], 'lr': []}
    iteration = 0

    # If there are more iterations than mini-batches in the data
    # loader, it will have to loop over it more than once
    while (iteration < num_iter):
        # That's the typical mini-batch inner loop
        for x_batch, y_batch in data_loader:
            x_batch = x_batch.to(self.device)
            y_batch = y_batch.to(self.device)
            # Step 1
            yhat = self.model(x_batch)
            # Step 2
            loss = self.loss_fn(yhat, y_batch)
            # Step 3
            loss.backward()

            # Here we keep track of the losses (smoothed)
            # and the learning rates
            tracking['lr'].append(scheduler.get_last_lr()[0])
            if iteration == 0:
                tracking['loss'].append(loss.item())
            else:
                prev_loss = tracking['loss'][-1]
                smoothed_loss = (alpha * loss.item() +
                                (1-alpha) * prev_loss)
                tracking['loss'].append(smoothed_loss)
            iteration += 1
            # Number of iterations reached
            if iteration == num_iter:
                break

            # Step 4
            self.optimizer.step()
            scheduler.step()
            self.optimizer.zero_grad()

    # Restores the original states
    self.optimizer.load_state_dict(previous_states['optimizer'])
    self.model.load_state_dict(previous_states['model'])

    if ax is None:
        fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    else:
        fig = ax.get_figure()
    ax.plot(tracking['lr'], tracking['loss'])
    if step_mode == 'exp':
        ax.set_xscale('log')
    ax.set_xlabel('Learning Rate')
    ax.set_ylabel('Loss')
    fig.tight_layout()
    return tracking, fig

setattr(StepByStep, 'lr_range_test', lr_range_test)
```

Since the technique is supposed to be applied on an **untrained model**, we're creating a new model (and optimizer) here:

Model Configuration

```
torch.manual_seed(13)
new_model = CNN2(n_feature=5, p=0.3)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)
```

Next, we create an instance of `StepByStep` and call the new method using the **training data loader**, the **upper range** for the learning rate (`end_lr`), and how many iterations we'd like it to try:

Learning Rate Range Test

```
sbs_new = StepByStep(new_model, multi_loss_fn, new_optimizer)
tracking, fig = sbs_new.lr_range_test(
    train_loader, end_lr=1e-1, num_iter=100)
```

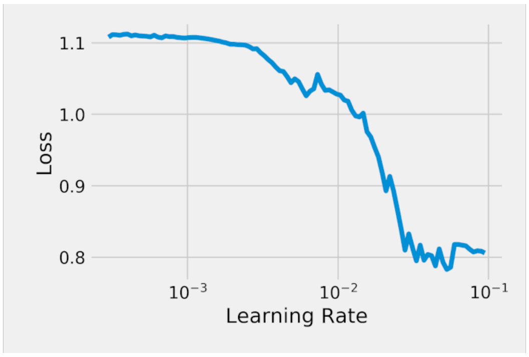


Figure 6.14 - Learning rate finder

There we go, an **"U"-shaped curve**. Apparently, the *Karpathy Constant* (`3e-4`) is **too low** for our model. The **descending part** of the curve is the region we should aim for: something around `0.01`.

It means we could have used a higher learning rate, like `0.005`, to train our model. But this also means we need to **recreate the optimizer** and **update it in `sbs_new`**. First, let's create a method for setting its optimizer:

StepByStep Method

```
def set_optimizer(self, optimizer):
    self.optimizer = optimizer

setattr(StepByStep, 'set_optimizer', set_optimizer)
```

Then, we create and set the new optimizer, and train the model as usual:

Updating LR and Model Training

```
new_optimizer = optim.Adam(new_model.parameters(), lr=0.005)
sbs_new.set_optimizer(new_optimizer)
sbs_new.set_loaders(train_loader, val_loader)
sbs_new.train(10)
```

If you try it out, you'll find that the training loss actually goes down a bit faster (and that the model might be overfitting).

DISCLAIMER: the learning rate finder is surely not magic! Sometimes you'll **not** get the "U"-shaped curve… maybe the initial learning rate (as defined in your optimizer) is too big already, or maybe the `end_lr` is too small. Even if you do, it does not necessarily mean the mid-point of the descending part will give you the fastest learning rate for your model.

"OK, if I manage to choose a good learning rate from the start, am I done with it?"

Sorry, but **NO**… well, it depends, it may be fine for simpler (but real, not toy) problems. The issue here is, for bigger models, the **loss surface** (remember that, from Chapter 0?) becomes **very messy**, and a **learning rate that works well at the start** of model training **may be too big for a later stage** of model training. It means that the learning rate needs to **change** or **adapt**…

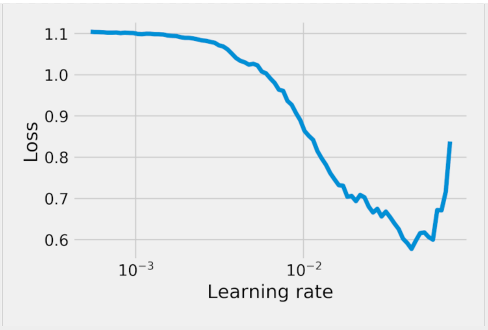
LRFinder

The function we've implemented above is fairly basic. For an implementation with more bells and whistles, check this Python package: torch_lr_finder[99]. I am illustrating its usage here, which is quite similar to what we've done above, but please refer to the documentation for more details.

```
!pip install --quiet torch-lr-finder
from torch_lr_finder import LRFinder
```

Instead of calling a function directly, we need to create an instance of `LRFinder` first, using the typical model configuration objects (model, optimizer, loss function, and the device). Then, we can take the `range_test` method for a spin, providing familiar arguments to it: a data loader, the upper range for the learning rate, and the number of iterations. The `reset` method restores the original states of both model and optimizer.

```
torch.manual_seed(11)
new_model = CNN2(n_feature=5, p=0.3)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)
device = 'cuda' if torch.cuda.is_available() else 'cpu'

lr_finder = LRFinder(
    new_model, new_optimizer, multi_loss_fn, device=device
)
lr_finder.range_test(train_loader, end_lr=1e-1, num_iter=100)
lr_finder.plot(log_lr=True)
lr_finder.reset()
```



Not quite a "U" shape, but we still can tell that something in the ballpark of `1e-2` is a good starting point.

Adaptive Learning Rate

That's what the **Adam** optimizer is actually doing for us… it starts with the learning rate provided as an argument, but it **adapts** the learning rate(s) as it goes, tweaking it in a different way for each parameter in the model. Or **does it**?

Truth to be told, Adam **does not adapt the learning rate**, it really **adapts the gradients**. But, since the parameter update is given by the multiplication of both terms, the learning rate and the gradient, this is a distinction without a difference.

Adam combines the characteristics of two other optimizers: `SGD` (with momentum) and `RMSProp`. Like the former, it uses a **moving average of gradients** instead of gradients themselves (that's the *first moment*, in statistics jargon); like the latter, it **scales the gradients** using a **moving average of squared gradients** (that's the *second moment*, or uncentered variance, in statistics jargon).

But this is not a simple average. It is a **moving average**. And it is not *any* moving average. It is an **exponentially weighted moving average (EWMA)**.

Before diving into EWMAs, though, we need to briefly go over simple moving averages.

Moving Average (MA)

To compute the moving average of a given feature ***x*** over a certain number of ***periods***, we just have to average the values observed over that many time steps (from an initial value observed ***periods+1*** steps ago all the way up to the **current value**):

$$MA_t(periods, x) = \frac{1}{periods}(x_t + x_{t-1} + \ldots + x_{t-periods+1})$$

Equation 6.1 - Simple Moving Average

But, instead of averaging the values themselves, let's compute the **average age of the values**. The **current value** has an **age equals one** unit of time while the **oldest value** in our moving average has an **age equals periods** units of time, so the **average age** is given by the formula below:

$$average\ age_{MA} = \frac{1 + 2 + \cdots + periods}{periods} = \frac{periods + 1}{2}$$

Equation 6.2 - Average age of a MA

For a **five-period moving average**, the **average age** of its values is **three** units of time.

"Why do we care about the average age of the values?"

This may seem a bit silly in the context of a simple moving average, sure. But, as you'll see in the next subsection, an **EWMA does not use the number of periods directly** in its formula: we'll have to rely on the **average age** of its values to estimate its (equivalent) number of periods.

"Why use an EWMA then?"

EWMA

An EWMA is more **practical** to compute than a traditional moving average because it **has only two inputs**: the value of **EWMA in the previous step** and the **current value** of the variable being averaged. There are two ways of representing its formula, using `alpha` or `beta`:

$$
\begin{aligned}
EWMA_t(\alpha, x) &= \alpha \quad x_t + (1-\alpha) EWMA_{t-1}(\alpha, x) \\
EWMA_t(\beta, x) &= (1-\beta) \quad x_t + \beta EWMA_{t-1}(\beta, x)
\end{aligned}
$$

Equation 6.3 - EWMA

The first alternative, using `alpha` as the **weight of the current value** is most common in other fields, like finance. But, for some reason, the `beta` alternative is the one commonly found when the Adam optimizer is discussed.

Let's take the first alternative and **expand the equation** a bit:

$$
\begin{aligned}
EWMA_t(\alpha, x) &= \alpha x_t + (1-\alpha)\ (\alpha x_{t-1} + (1-\alpha)\ EWMA_{t-2}(\alpha, x)) \\
&= \alpha x_t + (1-\alpha)\ \alpha x_{t-1} + (1-\alpha)^2 \alpha x_{t-2} + \dots \\
&= (1-\alpha)^0 \alpha x_{t-0} + (1-\alpha)^1\ \alpha x_{t-1} + (1-\alpha)^2 \alpha x_{t-2} + \dots \\
&= \alpha\ ((1-\alpha)^0\ x_{t-0} + (1-\alpha)^1\ x_{t-1} + (1-\alpha)^2\ x_{t-2} + \dots)
\end{aligned}
$$

Equation 6.4 - EWMA - expanded edition

The first element is taken at face value, but all the remaining elements are **discounted** based on their corresponding **lags**.

"What is a lag?"

It is simply the **distance, in units of time, from the current value**. So, the value of feature *x* **one time unit in the past** is the value of feature *x* **at lag one**.

After working out the expression above, we end up with an expression where each term has an **exponent** depending on the corresponding **number of lags**. We can use this information to make a sum out of it:

$$EWMA_t(\alpha, x) = \alpha \sum_{lag=0}^{T-1} \underbrace{(1-\alpha)^{lag}}_{weight} x_{t-lag}$$

Equation 6.5 - EWMA - lag-based

In the expression above, **T** is the **total number of observed values**. So, an EWMA **takes every value into account**, no matter how far in the past it is. But, due to the **weight** (the discount factor), the **older** a value gets, the **less it contributes** to the sum.

Higher values of `alpha` correspond to **rapidly shrinking weights**, that is, **older values barely make a difference**.

Let's see how the **weights** are distributed over the lags for two averages, an EWMA with `alpha` equals one third and a simple five-period moving average:

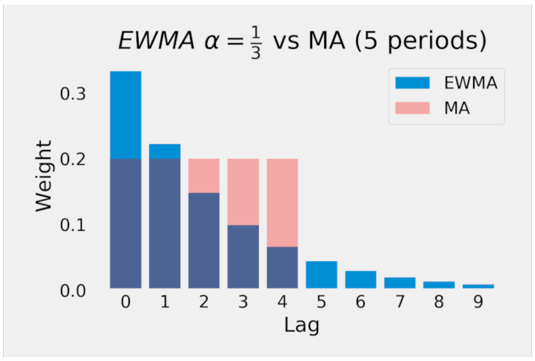


Figure 6.15 - Distribution of Weights over Lags

See the difference? In a simple **moving average** every value has the **same weight**, that is, they contribute equally to the average. But, in an **EWMA**, more **recent** values have **larger weights** than older ones.

It may not seem like it, but the **two averages** above have **something in common**. The **average age** of its values is approximately the same. Cool, right?

So, if the **average age** of the values in a **five-period moving average** is **three**, we should arrive at (approximately) the **same value for the age** of the values in the EWMA above. Let's understand why this is so. Maybe you haven't noticed it yet, but a **lag of zero** corresponds to **an age of one unit of time**, a lag of one corresponds to an age of two units of time, and so on. We can use this information to **compute the average age** of the values in an EWMA:

$$
average\ age_{EWMA} = \alpha \sum_{lag=0}^{T-1} (1-\alpha)^{lag}(lag+1) \approx \frac{1}{\alpha}
$$

Equation 6.6 - Average age of an EWMA

As the total number of observed values (**T**) grows, the **average age** approaches the **inverse of `alpha`**. No, I am not demonstrating this here. Yes, I am showing you a snippet of code that "*proves*" it numerically :-)

You may go bananas with the value of **T**, trying in vain to approach infinity, but 20 periods is more than enough to make a point:

```
alpha = 1/3; T = 20
t = np.arange(1, T + 1)
age = alpha * sum((1 - alpha)**(t - 1) * t)
age
```

Output

```
2.9930832408241015
```

That's **three-ish** enough, right? If you're not convinced, try using 93 periods (or more).

Now that we know how to compute the **average age** of an EWMA **given its `alpha`**, we can figure out **which (simple) moving average** has the **same average age**:

$$average\ age = \frac{periods + 1}{2} = \frac{1}{\alpha} \implies \alpha = \frac{2}{periods + 1};\ periods = \frac{2}{\alpha} - 1$$

Equation 6.7 - Alpha vs Periods

There we go, an easy and straightforward relationship between the **value of `alpha`** and the **number of periods** of a moving average. Guess what happens if you plug the value **one-third** for `alpha`? You get the corresponding number of periods: **five**. An EWMA using `alpha` equals one-third corresponds to a five-period moving average.

It also works the other way around: if we'd like to compute the EWMA equivalent to a 19-period moving average, the corresponding `alpha` would be `0.1`. And, if we're using the EWMA's formula based on `beta`, that would be `0.9`. Similarly, to compute the EWMA equivalent to a 1999-period moving average, `alpha` and `beta` would be `0.001` and `0.999`, respectively.

These choices are **not random** at all: it turns out, **Adam** uses these **two values** for its **betas** (one for the moving average of gradients, the other for the moving average of squared gradients).

In code, the implementation of the `alpha` version of EWMA looks like this:

```
def EWMA(past_value, current_value, alpha):
    return (1- alpha) * past_value + alpha * current_value
```

For computing it over a series of values, given a period, we can define a function like this:

```python
def calc_ewma(values, period):
    alpha = 2 / (period + 1)
    result = []
    for v in values:
        try:
            prev_value = result[-1]
        except IndexError:
            prev_value = 0

        new_value = EWMA(prev_value, v, alpha)
        result.append(new_value)
    return np.array(result)
```

In the `try..except` block, you can see that, if there is no previous value for the EWMA (as in the very first step), it assumes a previous value of zero.

The way the EWMA is constructed has its issues… since it does not need to keep track of all the values inside its period, in its **first steps**, the "average" will be **way off** (or *biased*). For an `alpha=0.1` (corresponding to the 19 periods average), the very first "average" will be exactly the first value divided by ten.

To address this issue, we can compute the **bias-corrected EWMA**:

$$Bias\ Corrected\ EWMA_t(x, \beta) = \frac{1}{1 - \beta^t} EWMA_t(x, \beta)$$

Equation 6.8 - Bias-corrected EWMA

The `beta` in the formula above is the same as before: `1 - alpha`. In code, we can implement the correction factor like this:

```python
def correction(averaged_value, beta, steps):
    return averaged_value / (1 - (beta ** steps))
```

For computing the corrected EWMA over a series of values, we can use a function like this:

```python
def calc_corrected_ewma(values, period):
    ewma = calc_ewma(values, period)

    alpha = 2 / (period + 1)
    beta = 1 - alpha

    result = []
    for step, v in enumerate(ewma):
        adj_value = correction(v, beta, step + 1)
        result.append(adj_value)

    return np.array(result)
```

Let's apply both EWMAs, together with a *regular* moving average, to a sequence of **temperature values** to illustrate the differences:

```python
temperatures = np.array([5, 11, 15, 6, 5, 3, 3, 0, 0, 3, 4, 2, 1,
    -1, -2, 2, 2, -2, -1, -1, 3, 4, -1, 2, 6, 4, 9, 11, 9, -2])

ma_vs_ewma(temperatures, periods=19)
```

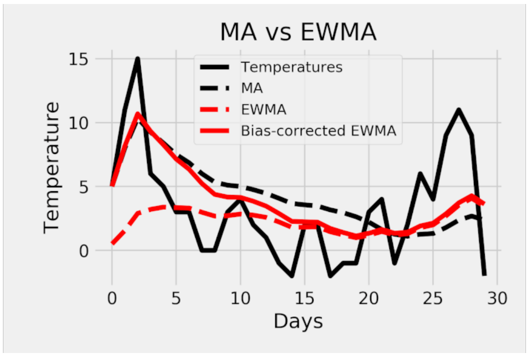


Figure 6.16 - Moving average vs EWMA

As expected, the EWMA without correction (red dashed line) is *way off* at the beginning, while the regular moving average (black dashed line) tracks the actual values much closer. The **corrected EWMA**, though, does a very good job tracking the actual values from the very beginning. Sure enough, after 19 days, both EWMAs are barely distinguishable.

EWMA Meets Gradients

Who cares about temperatures, anyway? Let's apply the EWMAs to our gradients, Adam-style!

For **each parameter**, we compute **two EWMAs**: one for its **gradients**, the other for the **square of its gradients**. Next, we use both values to compute the **adapted gradient** for that parameter:

$$adapted\ gradient_t = \frac{Bias\ Corrected\ EWMA_t(\beta_1, gradients)}{\sqrt{Bias\ Corrected\ EWMA_t(\beta_2, gradients^2)} + \epsilon}$$

Equation 6.9 - Adapted gradient

There they are, Adam's `beta1` and `beta2` parameters! Its default values, `0.9` and `0.999`, correspond to averages of 19 and 1999 periods, respectively.

So, it is a **short term** average for **smoothing the gradients**, and a **very long term** average for **scaling the gradients**. The `epsilon` value in the denominator (usually `1e-8`) is there to prevent numerical issues only.

Once the **adapted gradient** is computed, it **replaces the actual gradient** in the **parameter update**:

$$
\begin{aligned}
SGD: param_t &= param_{t-1} - \eta \ gradient_t \\
Adam: param_t &= param_{t-1} - \eta \ adapted \ gradient_t
\end{aligned}
$$

Equation 6.10 - Parameter update

Clearly, the **learning rate** (the Greek letter *eta*) is left untouched!

Moreover, due to the **scaling**, the **adapted gradient** is likely to be inside the [-3, 3] range most of the time (this is akin to the standardization procedure but without subtracting the mean).

Adam

So, choosing the **Adam** optimizer is an easy and straightforward way to tackle your learning rate needs. Let's take a closer look at PyTorch's `Adam` optimizer and its arguments:

- `params`: model's parameters
- `lr`: learning rate, default value `1e-3`
- `betas`: tuple containing `beta1` and `beta2` for the EWMAs
- `eps`: the `epsilon` (`1e-8`) value in the denominator

The four arguments above should be clear by now. But there are another two we haven't talked about yet:

- `weight_decay`: L2 penalty
- `amsgrad`: if the AMSGrad variant should be used

The first argument, `weight_decay`, introduces a **regularization term** (L2 penalty) to the model's weights. As every regularization procedure, it aims at preventing overfitting by penalizing weights with large values. The term *weight decay* comes from the fact that the regularization actually **increases the gradients** by **adding the weight value multiplied by the weight decay argument**.

*"If it **increases** the gradients, how come it is called weight **decay**?"*

In the **parameter update**, the gradient is multiplied by the learning rate and **subtracted from the weight's previous value**. So, in effect, adding a penalty to the value of the gradients makes the weights smaller. The smaller the weights, the smaller the penalty, thus making further reductions even smaller - in other words - the weights are decaying.

The second argument, `amsgrad`, makes the optimizer compatible with a variant of the same name. In a nutshell, it modifies the formula used to compute *adapted gradients*, ditching the bias correction, and using the peak value of the EWMA of squared gradients instead.

For now, we're sticking with the first four, well-known to us, arguments:

```
optimizer = optim.Adam(model.parameters(), lr=0.1, betas=(0.9,
0.999), eps=1e-8)
```

Visualizing Adapted Gradients

Now, I'd like to give you the chance of **visualizing** the gradients, the EWMAs, and the resulting **adapted gradients**. To make it easier, let's bring back our **simple linear regression** problem from "*Part I*" of this book, and, somewhat nostalgically, let's **perform the training loop** so that we can **log the gradients**.

From now on and until the end of the "*Learning Rates*" section, we'll be ONLY using the **simple linear regression** dataset to illustrate the effects of different parameters on the minimization of the loss. We'll get back to the "*Rock, Paper, Scissors*" dataset in the "*Putting It All Together*" section.

First, we generate the data points again and run the typical data preparation step (building dataset, splitting it, and building data loaders):

Data Generation & Preparation

```
%run -i data_generation/simple_linear_regression.py
%run -i data_preparation/v2.py
```

Then, we go over the model configuration and change the optimizer from SGD to Adam:

Model Configuration

```
torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(1, 1))
optimizer = optim.Adam(model.parameters(), lr=0.1)
loss_fn = nn.MSELoss(reduction='mean')
```

We would be ready to use the `StepByStep` class to *train* our model if it weren't for a minor detail: we still do not have a way of **logging gradients**. So, let's tackle this issue by adding yet another method to our class: `capture_gradients`. Like the `attach_hooks` method, it will take a list of layers that should be monitored for their gradient values.

For each monitored layer, it will go over its parameters and, for those that *require gradients*, it will **create a logging function (`log_fn`)** and **register a hook** for it in the **tensor corresponding to the parameter**.

The logging function simply **appends the gradients to a list** in the dictionary entry corresponding to the layer and parameter names. The dictionary itself, `_gradients`, is an *attribute* of the class (which will be created inside the constructor method, but we're setting it manually using `setattr` for now). The code looks like this:

StepByStep Method

```
setattr(StepByStep, '_gradients', {})

def capture_gradients(self, layers_to_hook):
    if not isinstance(layers_to_hook, list):
        layers_to_hook = [layers_to_hook]

    modules = list(self.model.named_modules())
    self._gradients = {}

    def make_log_fn(name, parm_id):
        def log_fn(grad):
            self._gradients[name][parm_id].append(grad.tolist())
            return None
        return log_fn

    for name, layer in self.model.named_modules():
        if name in layers_to_hook:
            self._gradients.update({name: {}})
            for parm_id, p in layer.named_parameters():
                if p.requires_grad:
                    self._gradients[name].update({parm_id: []})
                    log_fn = make_log_fn(name, parm_id)
                    self.handles[f'{name}.{parm_id}.grad'] = \
                        p.register_hook(log_fn)
    return

setattr(StepByStep, 'capture_gradients', capture_gradients)
```

IMPORTANT: the logging function **must return `None`**, otherwise **the gradients will be modified**, assuming the **returned value**.

The register_hook method registers a **backward hook** to a **tensor for a given parameter**. The **hook function** takes a **gradient as input** and returns either **a modified gradient** or **`None`**. The hook function will be called every time a gradient with respect to that tensor is computed.

Since we're using this function for *logging purposes*, we should leave the gradients alone and return `None`.

"Isn't there a `register_backward_hook` *method? Why aren't we using it?"*

That's a fair question. At the time of writing, this method still has an unsolved issue, so we're following the recommendation of using `register_hook` for individual tensors instead.

Now, we can use the new method to *log gradients* for the `linear` layer of our model, never forgetting to **remove the hooks** after training:

Model Training

```
sbs_adam = StepByStep(model, loss_fn, optimizer)
sbs_adam.set_loaders(train_loader)
sbs_adam.capture_gradients('linear')
sbs_adam.train(10)
sbs_adam.remove_hooks()
```

By the time training is finished, we'll have collected two series of 50 gradients each (each epoch has *five mini-batches*), each series corresponding to a parameter of `linear` (`weight` and `bias`), both of them stored in the `_gradients` attribute of our `StepByStep` instance.

We can use these values to compute the EWMAs and the **adapted gradients actually used by Adam** to update the parameters. Let's do it for the `weight` parameter:

```
gradients = np.array(
  sbs_adam._gradients['linear']['weight']
).squeeze()

corrected_gradients = calc_corrected_ewma(gradients, 19)
corrected_sq_gradients = calc_corrected_ewma(
  np.power(gradients, 2), 1999
)
adapted_gradients = (corrected_gradients /
  (np.sqrt(corrected_sq_gradients) + 1e-8))
```

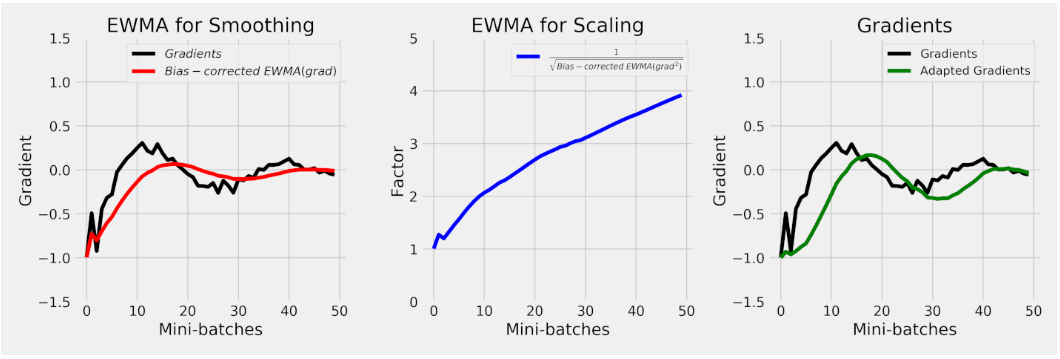


Figure 6.17 - Computing adapted gradients using EWMAs

On the left plot, we see that the **bias-corrected EWMA of gradients** (in red) is **smoothing** the gradients. In the center, the bias-corrected EWMA of squared gradients is used for **scaling the smoothed gradients**. On the right, both EWMAs are combined to compute the **adapted gradients**.

Under the hood, Adam **keeps two values for each parameter**: `exp_avg` and `exp_avg_sq`, representing the (uncorrected) EWMAs for gradients and squared gradients, respectively. We can take a peek at it using the optimizer's `state_dict`:

```
optimizer.state_dict()
```

Output

```
{'state': {140601337662512: {'step': 50,
   'exp_avg': tensor([[-0.0089]], device='cuda:0'),
   'exp_avg_sq': tensor([[0.0032]], device='cuda:0')},
  140601337661632: {'step': 50,
   'exp_avg': tensor([0.0295], device='cuda:0'),
   'exp_avg_sq': tensor([0.0096], device='cuda:0')}},
 'param_groups': [{'lr': 0.1,
   'betas': (0.9, 0.999),
   'eps': 1e-08,
   'weight_decay': 0,
   'amsgrad': False,
   'params': [140601337662512, 140601337661632]}
```

Inside its `state` key, it contains two other dictionaries (with weird numeric keys) representing the different parameters of the model. In our example, the first dictionary (`140614347109072`) corresponds to the `weight` parameter. Since we've logged all the gradients, we should be able to use our `calc_ewma` function to replicate the values contained in the dictionary:

```
(calc_ewma(gradients, 19)[-1],
calc_ewma(np.power(gradients, 2), 1999)[-1])
```

Output

```
(-0.008938403644834258, 0.0031747136253540394)
```

Taking the **last values** of our two uncorrected EWMAs, we **matched** the state of the optimizer (`exp_avg` and `exp_avg_sq`). Cool!

"OK, cool, but how is it better than SGD in practice?"

Fair enough! We've been discussing **how different the parameter update is**, but now it is time to **show how it affects model training**. Let's bring back the **loss surface** we've computed for this linear regression (way back in Chapter 0) and **visualize the path** taken by each optimizer to bring both parameters (closer) to their optimal values.

That would be great, but we're missing *another* minor detail: we also do not have a way of **logging the evolution of parameters**. Guess what we're gonna do about that? Create another method, of course!

The new method, aptly named `capture_parameters` works in a way similar to `capture_gradients`. It keeps a dictionary (`parameters`) as an attribute of the class and **register forward hooks** to the layers we'd like to log the parameters for. The logging function simply loops over the parameters of a given layer and appends its values to the corresponding entry in the dictionary. The registering itself is handled by a method we developed earlier: `attach_hooks`. The code looks like this:

```
setattr(StepByStep, '_parameters', {})

def capture_parameters(self, layers_to_hook):
    if not isinstance(layers_to_hook, list):
        layers_to_hook = [layers_to_hook]

    modules = list(self.model.named_modules())
    layer_names = {layer: name for name, layer in modules}

    self._parameters = {}

    for name, layer in modules:
        if name in layers_to_hook:
            self._parameters.update({name: {}})
            for parm_id, p in layer.named_parameters():
                self._parameters[name].update({parm_id: []})

    def fw_hook_fn(layer, inputs, outputs):
        name = layer_names[layer]
        for parm_id, parameter in layer.named_parameters():
            self._parameters[name][parm_id].append(
                parameter.tolist()
            )

    self.attach_hooks(layers_to_hook, fw_hook_fn)
    return

setattr(StepByStep, 'capture_parameters', capture_parameters)
```

What's next? We need to create two instances of `StepByStep`, each using a different optimizer, set them to capture parameters, and train them for ten epochs. The captured parameters (bias and weight) will draw the following paths (the red dot represents their optimal values):

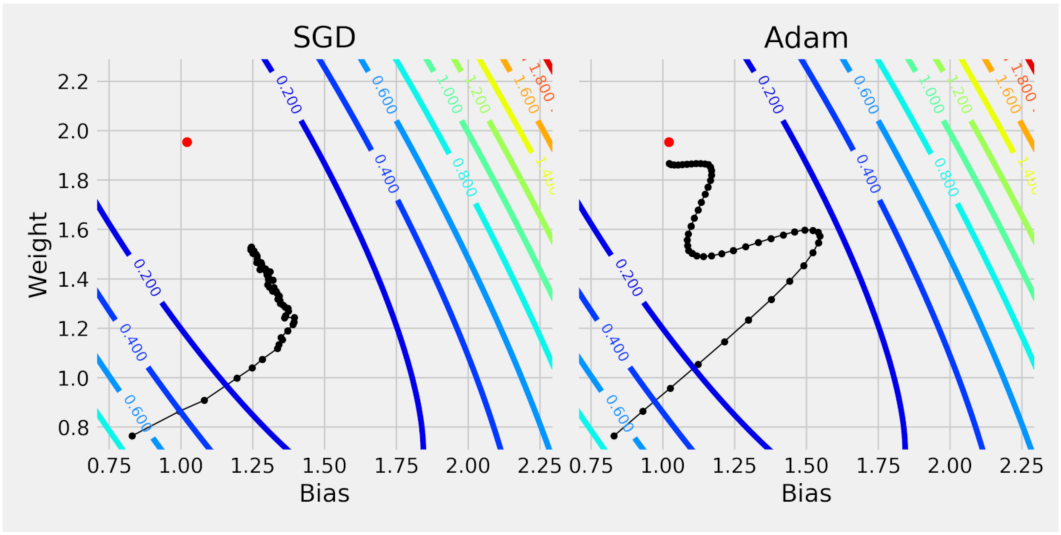


Figure 6.18 - Paths taken by SGD and Adam

On the left plot, the typical well-behaved (and slow) path taken by **simple gradient descent**. You can see it is **wiggling** a bit due to the noise introduced by using **mini-batches**. On the right plot, we see the effect of using the exponentially weighted moving averages: on the one hand, **it is smoother and moves faster**, on the other hand, it **overshoots** and has to **change course back and forth** as it approaches the target. It is **adapting to the loss surface** if you will.

If you like the idea of visualizing (and animating) the paths of optimizers, make sure to check out Louis Tiao's tutorial[100] on the subject.

Talking about losses, we can also compare the trajectories of training and validation losses for each optimizer:

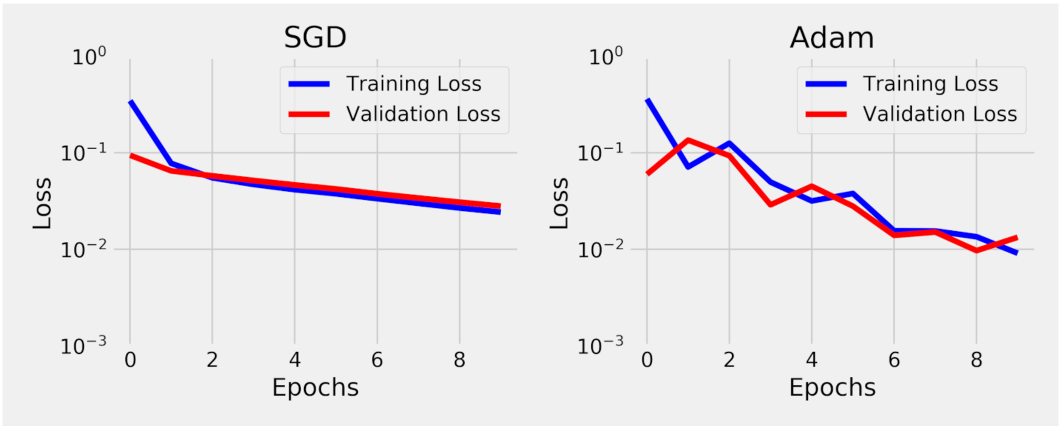


Figure 6.19 - Losses (SGD and Adam)

Remember, the losses are computed at the end of each epoch by averaging the losses of the mini-batches. On the left plot, even if SGD wiggles a bit, we can see that every epoch shows a smaller loss than the previous one. On the right plot, the **overshooting** becomes clearly visible as an **increase in the training loss**. But it is also clear that Adam achieves a **smaller loss** because it got **closer to the optimal value** (the red dot in the previous plot).

In real problems, where it is virtually **impossible to plot the loss surface**, we can look at the **losses** as an "*executive summary*" of what's going on. Training losses will sometimes go up before they go down again and this is expected.

Stochastic Gradient Descent (SGD)

Adaptive learning rates are cool indeed, but good old Stochastic Gradient Descent (SGD) also has a couple of tricks up its sleeve. Let's take a closer look at PyTorch's `SGD` optimizer and its arguments:

- `params`: model's parameters
- `lr`: learning rate
- `weight_decay`: L2 penalty

The three arguments above are already known. But there are three new arguments:

- `momentum`: momentum factor, SGD's own `beta` argument, is the topic of the next section
- `dampening`: dampening factor for momentum
- `nesterov`: enables Nesterov momentum, which is a smarter version of the regular momentum, and it also has its own section

That's the perfect *moment* to dive deeper into *momentum* (sorry, I *really* cannot miss a pun!).

Momentum

One of SGD's tricks is called **momentum**. At first sight, it looks very much like using an EWMA for gradients, but it isn't. Let's compare EWMA's `beta` formulation with momentum's:

$$
\begin{aligned}
EWMA_t &= (1-\beta)\ grad_t + \ \beta \ EWMA_{t-1} \\
Momentum_t &= grad_t + \ \beta \ Momentum_{t-1}
\end{aligned}
$$

Equation 6.11 - Momentum vs EWMA

See the difference? It does not *average* the gradients, it **runs a cumulative sum of "*discounted*" gradients**. In other words, **all past gradients** contribute to the sum, but they are **"*discounted*" more and more as they grow older**. The "*discount*" is driven by the `beta` parameter, and we can also write the formula for momentum like this:

$$
Momentum_t = \beta^0 \ grad_t + \beta^1 \ grad_{t-1} + \beta^2 \ grad_{t-2} + \ldots + \beta^n \ grad_{t-n}
$$

Equation 6.12 - Compounding momentum

The disadvantage of this second formula is that it requires the *full history* of gradients, while the previous one depends only on the gradient's current value and momentum's latest value.

"What about the dampening factor?"

The dampening factor is a way to, well, **dampen the effect of the latest gradient**. Instead of having its full value added, the latest gradient gets its contribution to momentum **reduced by the dampening factor**. So, if the dampening factor is `0.3`, only 70% of the latest gradient gets added to momentum. Its formula is given by:

$$Momentum_t = (1 - damp)\ \ grad_t + \beta\ \ Momentum_{t-1}$$

Equation 6.13 - Momentum with dampening factor

If the **dampening factor equals the momentum factor (`beta`)**, it becomes a true EWMA!

Similarly to Adam, SGD with momentum **keeps the value of momentum for each parameter**. The `beta` parameter is stored there as well (`momentum`). We can take a peek at it using the optimizer's `state_dict`:

```
{'state': {139863047119488: {'momentum_buffer': tensor([[-
0.0053]])},
  139863047119168: {'momentum_buffer': tensor([-0.1568])}},
 'param_groups': [{'lr': 0.1,
   'momentum': 0.9,
   'dampening': 0,
   'weight_decay': 0,
   'nesterov': False,
   'params': [139863047119488, 139863047119168]}]}
```

Even though old gradients slowly fade away, contributing less and less to the sum, **very recent gradients** are taken **almost at their face value** (assuming a typical value of `0.9` for `beta` and *no dampening*). This means that, given a **sequence of all positive (or all negative) gradients**, their sum, that is, the **momentum is going up really fast (in absolute value)**. A large momentum gets translated into a **large update** since **momentum replaces gradients in the parameter update**:

$$
\begin{aligned}
SGD: param_t &= param_{t-1} - \eta\ gradient_t \\
Adam: param_t &= param_{t-1} - \eta\ adapted\ gradient_t \\
SGD + Mom: param_t &= param_{t-1} - \eta\ Momentum_t
\end{aligned}
$$

Equation 6.14 - Parameter update

This behavior can be easily visualized in the **path taken by SGD with momentum**:

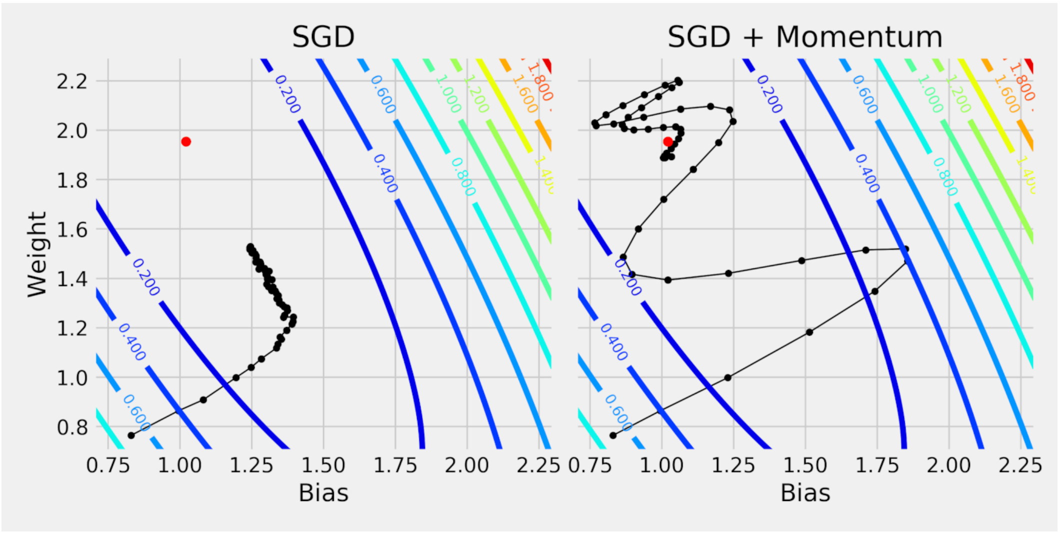


Figure 6.20 - Paths taken by SGD (with, and without, momentum)

Like the Adam optimizer, SGD with momentum also **moves faster** and **overshoots**. But it does seem to get **carried away** with it, so much so that it **gets past the target** and has to **backtrack** to approach it from a different direction.

The analogy for the momentum update is that of a **ball rolling down a hill**: it picks up so much speed that it ends up climbing the opposite side of the valley, only to roll back down again with a little bit less speed, doing this back and forth over and over again until eventually reaching the bottom.

"Isn't Adam better than this already?"

Yes, and no. Adam is indeed faster to converge to *a* minimum, but not necessarily a *good* one. In a simple linear regression, there is a **global minimum** corresponding to the **optimal value of the parameters**. This is **not the case of deep learning models**: there are **many minima** (plural of minimum), and **some are better than others** (corresponding to lower losses). So, Adam will find one of these minima and move there fast, perhaps overlooking better alternatives in the neighborhood.

Momentum may seem a bit *sloppy* at first, but it may be **combined with a learning rate scheduler** (more on that shortly!) to **better explore the loss surface** in hopes of finding a better quality minimum than Adam.

Both alternatives, **Adam** and **SGD with momentum** (especially when combined with a learning rate scheduler), are commonly used.

But, if a ball running downhill seems a bit too much out of control for your taste, maybe you'll like its variant better…

Nesterov

Nesterov Accelerated Gradient (NAG), or Nesterov for short, is a clever variant of SGD with momentum. Let's say we're computing **momentum** for two consecutive steps (t and t+1):

$$
\begin{aligned}
&step\ t: &Mo_t = grad_t + \beta\ \ Mo_{t-1} \\
&step\ t+1: \ \ Mo_{t+1} = grad_{t+1} + \beta Mo_t
\end{aligned}
$$

Equation 6.15 - Nesterov momentum

In the **current step** (t), we use the **current gradient** (t), and the **momentum from**

the previous step (t-1) to compute the **current momentum**. So far, nothing new.

In the **next step** (t+1), we'll use the **next gradient** (t+1) and the **momentum we've just computed for the current step** (t) to compute the **next momentum**. Again, nothing new.

What if I ask you to **compute momentum one step ahead**?

*"Can you tell me **momentum at step** **t+1** while you're **still at step** **t**?"*

*"Of course I can't, I do not know the **gradient at step** **t+1**!"* you say, puzzled at my bizarre question. Fair enough. So I ask you yet another question:

*"What's your best guess for the **gradient at step** **t+1**?"*

I hope you answered, "*the **gradient at step** **t***". If you do not know the future value of a variable, the naive estimate is its current value. So, let's go with it, Nesterov-style!

In a nutshell, what NAG does is **trying to compute momentum one step ahead** since it is only missing one value and it has a good (naive) guess for it. It is as if it was computing momentum **between two steps**:

$$
\begin{aligned}
&step\ t: && && Mo_t = grad_t + \beta\ \ Mo_{t-1} \\
&step\ t: && Nesterov_t = grad_t && +\beta\ \ Mo_t \\
&step\ t+1: && Mo_{t+1} = grad_{t+1} && +\beta\ \ Mo_t
\end{aligned}
$$

Equation 6.16 - Looking ahead

Once Nesterov's momentum is computed, it **replaces the gradient in the parameter update**, just like regular momentum:

$$
\begin{aligned}
SGD + Momentum :param_t &= param_{t-1} - \eta \ Momentum_t \\
SGD + Nesterov :param_t &= param_{t-1} - \eta \ Nesterov_t
\end{aligned}
$$

Equation 6.17 - Parameter update

But, Nesterov actually uses momentum, so we can expand its expression like this:

$$
\begin{aligned}
param_t &= param_{t-1} - \eta \ Nesterov_t \\
&= param_{t-1} - \eta \ (grad_t + \beta \ Momentum_t) \\
&= param_{t-1} - \eta \ grad_t - \beta \ \eta \ Momentum_t
\end{aligned}
$$

Equation 6.18 - Parameter update (expanded)

"Why did you do this? What's the purpose of making the formula more complicated?"

You'll understand why in a minute :-)

Flavors of SGD

Let's compare the three flavors of SGD, vanilla (regular), momentum, and Nesterov, when it comes to the way they perform the **parameter update**:

$$
\begin{aligned}
SGD :param_t &= param_{t-1} & -\eta \ grad_t & & \\
SGD + Momentum :param_t &= param_{t-1} & & - & \eta \ Momentum_t \\
SGD + Nesterov :param_t &= param_{t-1} & -\eta \ grad_t & -\beta & \eta \ Momentum_t
\end{aligned}
$$

Equation 6.19 - Flavors of parameter update

That's why I expanded Nesterov's expression in the last section: it is easier to compare the updates this way! First, there is **regular SGD**, which uses the **gradient and nothing else**. Then, there is **momentum**, which uses a **"*discounted*" sum of past gradients** (the momentum). Finally, there is **Nesterov**, which **combines both** (with a small tweak).

How different are the updates? Let’s check it out! The plots below show the **update term** (before multiplying it by the learning rate) for the `weight` parameter of our linear regression:

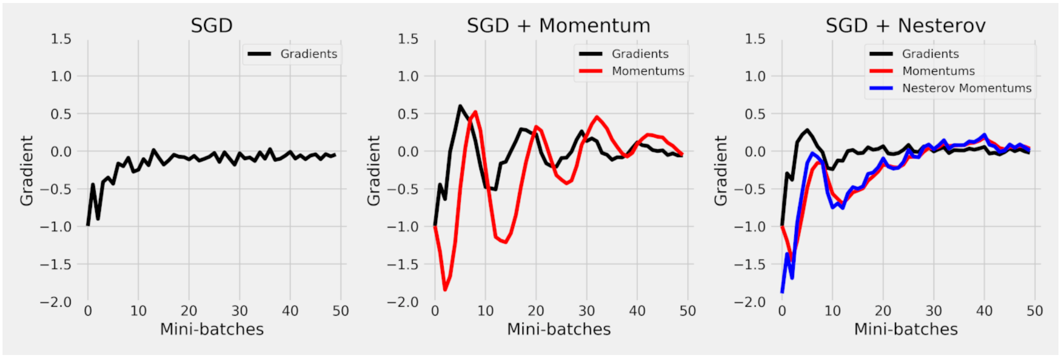


Figure 6.21 - Update terms corresponding to SGD flavors

Does the **shape of the update term for SGD with momentum** ring a bell? The oscillating pattern was already depicted in the **path taken by SGD with momentum** while optimizing the two parameters: when it **overshoots**, it has to **reverse direction**, and by repeatedly doing that, these oscillations are produced.

Nesterov momentum seems to do a better job: the **look-ahead** has the effect of **dampening the oscillations** (please do not confuse this effect with the actual *dampening* argument). Sure, the idea *is* to look ahead to avoid going too far, but could you have told me the difference between the two plots beforehand? Me neither! Well, I am *assuming* you replied "*no*" to this question, and that’s why I thought it was a good idea to illustrate the patterns above.

"How come the black lines are different in these plots? Isn’t the ***underlying gradient*** *supposed to be the same?"*

The gradient is indeed *computed the same way* in all three flavors, but since the **update terms are different**, the gradients are **computed at different locations of the loss surface**. This becomes clear when we look at the paths taken by each one of the flavors:

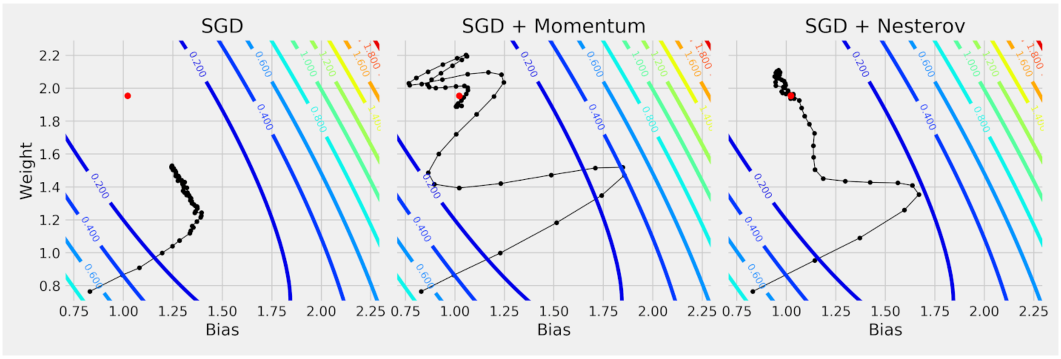


Figure 6.22 - Path taken by each SGD flavor

Take the *third point* in the lower left part of the black line, for instance: its location is quite different in each one of the plots and thus are the corresponding gradients.

The two plots on the left are already known to us. The new plot in town is the one to the right. The **dampening of the oscillations** is abundantly clear, but Nesterov's momentum **still gets past its target** and has to **backtrack** a little to approach it from the opposite direction. And let me remind you that this is **one of the easiest loss surfaces of all!**

Talking about losses, let's take a peek at their trajectories:

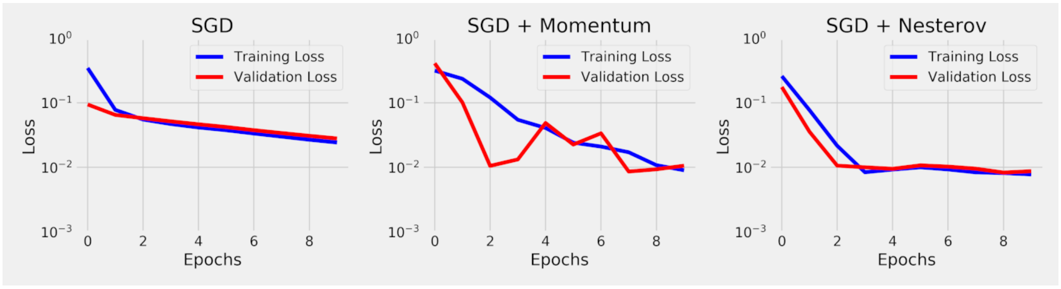


Figure 6.23 - Losses for each SGD flavor

The plot on the left is there just for comparison, it is the same as before. The one on the right is quite straightforward too, depicting the fact that Nesterov's momentum quickly found its way to a lower loss and slowly approached the optimal value.

The plot in the middle is a bit more intriguing: even though **regular momentum** produced a path with **wild swings** over the loss surface (each black dot corresponds to a mini-batch), its **loss trajectory** oscillates less than Adam's. This is an artifact of this simple linear regression problem (namely, the bowl-shaped loss surface), and should not be taken as representative of typical behavior.

If you're not convinced by *momentum*, either regular or Nesterov, let's add something else to the mix…

Learning Rate Schedulers

It is also possible to **schedule** the changes in the **learning rate** as training goes, instead of adapting the gradients. Say you'd like to **reduce the learning rate by one order of magnitude** (that is, multiplying it by `0.1`) **every T epochs**, such that training is **faster at the beginning** and **slows down** after a while to try avoiding convergence problems.

That's what a **learning rate scheduler** does: it **updates the learning rate of the optimizer**.

So, it should be no surprise that one of the scheduler's arguments is the optimizer itself. The learning rate set for the optimizer will be the initial learning rate of the scheduler. As an example, let's take the simplest of the schedulers: `StepLR`, which simply **multiplies the learning rate by a factor `gamma` every `step_size` epochs**.

In the code below, we create a dummy optimizer, which is "updating" some fake parameter with an initial learning rate of `0.01`. The dummy scheduler, an instance of `StepLR`, will multiply that learning rate by `0.1` every two epochs.

```
dummy_optimizer = optim.SGD([nn.Parameter(torch.randn(1))], lr=0.01)
dummy_scheduler = StepLR(dummy_optimizer, step_size=2, gamma=0.1)
```

The **scheduler** has a `step` method just like the optimizer.

You should call the scheduler’s `step` method **after** calling the optimizer’s `step` method.

Inside the training loop, it will look like this:

```
for epoch in range(4):
    # trainin loop code goes here

    print(dummy_scheduler.get_last_lr())
    # First call optimizer's step
    dummy_optimizer.step()
    # Then call scheduler's step
    dummy_scheduler.step()

    dummy_optimizer.zero_grad()
```

Output

```
[0.01]
[0.01]
[0.001]
[0.001]
```

As expected, it kept the initial learning rate for two epochs and then multiplied it by 0.1, resulting in a learning rate of `0.001` for another two epochs. In a nutshell, that’s how a learning rate scheduler works.

"*Does every scheduler **shrink** the learning rate?*"

Not really, no. It used to be standard procedure to *shrink* the learning rate as you train the model, but this idea was then challenged by **cyclical learning rates** (that’s the "*cyclical*" part of that paper!). There are many different flavors of scheduling, as you can see. And many of them are available in PyTorch.

We're dividing them into three groups: schedulers that update the learning rate **every T epochs** (even if T=1), like in the example above; the scheduler that only updates the learning rate when the **validation loss seems to be stuck**; and schedulers that update the learning rate after **every mini-batch**.

Epoch Schedulers

These schedulers will have their **`step` method** called at the **end of every epoch**. But each one has its own rules for updating the learning rate.

- **StepLR**: multiplies the learning rate by a factor `gamma` every `step_size` epochs
- **MultiStepLR**: multiplies the learning rate by a factor `gamma` at the epochs indicated in the list of `milestones`
- **ExponentialLR**: multiplies the learning rate by a factor `gamma` every epoch, no exceptions
- **LambdaLR**: takes your own customized function that should take the epoch as an argument and return the corresponding multiplicative factor (with respect to the **initial learning rate**)
- **CosineAnnealingLR**: it uses a technique called cosine annealing to update the learning rate, but we're not delving into details here

We can use `LambdaLR` to mimic the behavior of the `StepLR` scheduler defined above:

```
dummy_optimizer = optim.SGD([nn.Parameter(torch.randn(1))], lr=0.01)
dummy_scheduler = LambdaLR(
    dummy_optimizer, lr_lambda=lambda epoch: 0.1 ** (epoch//2)
)
# The scheduler above is equivalent to this one
# dummy_scheduler = StepLR(dummy_optimizer, step_size=2, gamma=0.1)
```

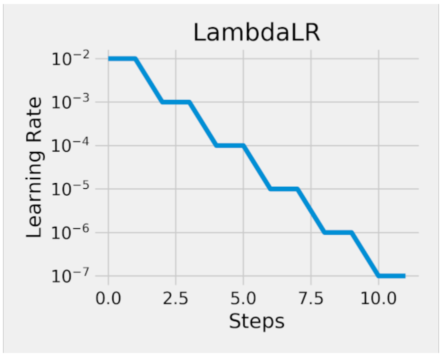


Figure 6.24 - Evolution of learning rate (epoch scheduler)

Validation Loss Scheduler

The `ReduceLROnPlateau` scheduler should **also** have its `step` method called at the end of every epoch, but it has its own group here because it **does not follow a predefined schedule**. Ironic, right?

The `step` method takes the **validation loss** as an argument, and the scheduler can be configured to **tolerate a lack of improvement in the loss** (to a threshold, of course) up to a given number of epochs (the aptly named `patience` argument). After the scheduler runs out of patience, it updates the learning rate, multiplying it by the `factor` argument (for the schedulers listed in the last section, this factor was named `gamma`).

To illustrate its behavior, let's assume the validation loss remains at the same value (whatever that is) for 12 epochs in a row. What would our scheduler do?

```
dummy_optimizer = optim.SGD([nn.Parameter(torch.randn(1))], lr=0.01)
dummy_scheduler = ReduceLROnPlateau(
    dummy_optimizer, patience=4, factor=0.1
)
```

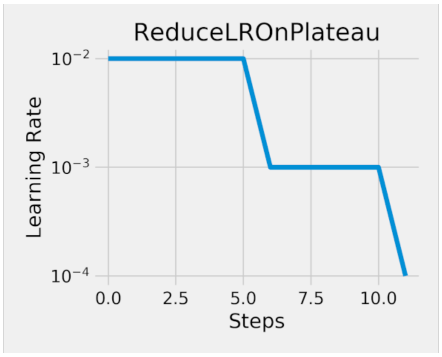


Figure 6.25 - Evolution of learning rate (validation loss scheduler)

Its patience is **four epochs**, so after four epochs observing the same loss, it is hanging by a thread. Then comes the **fifth epoch** and **no change**: "*that's it, the learning rate must go down*", you can almost hear it saying :-) So, in the **sixth epoch**, the optimizer is already using the newly updated learning rate. If nothing changes for four more epochs, it goes down again, as shown in the figure above.

Schedulers in StepByStep - Part I

If we want to incorporate learning rate schedulers into our training loop, we need to make some changes to our `StepByStep` class. Since schedulers are definitely *optional*, we need to add a **method** to allow the user to **set a scheduler** (similarly to what we did with TensorBoard integration). Moreover, we need to define some **attributes**: one for the scheduler itself, and a boolean variable to distinguish if it is an epoch or a mini-batch scheduler.

StepByStep Method

```
setattr(StepByStep, 'scheduler', None)
setattr(StepByStep, 'is_batch_lr_scheduler', False)

def set_lr_scheduler(self, scheduler):
    # Makes sure the scheduler in the argument is assigned to the
    # optimizer we're using in this class
    if scheduler.optimizer == self.optimizer:
        self.scheduler = scheduler
        if (isinstance(scheduler, optim.lr_scheduler.CyclicLR) or
            isinstance(scheduler, optim.lr_scheduler.OneCycleLR) or
            isinstance(scheduler,
                optim.lr_scheduler.CosineAnnealingWarmRestarts)):
            self.is_batch_lr_scheduler = True
        else:
            self.is_batch_lr_scheduler = False

setattr(StepByStep, 'set_lr_scheduler', set_lr_scheduler)
```

Next, we create a protected method that invokes the `step` method of the scheduler and appends the current learning rate to an attribute, so we can keep track of its evolution.

StepByStep Method

```
setattr(StepByStep, 'learning_rates', [])

def _epoch_schedulers(self, val_loss):
    if self.scheduler:
        if not self.is_batch_lr_scheduler:
            if isinstance(self.scheduler,
                  torch.optim.lr_scheduler.ReduceLROnPlateau):
                self.scheduler.step(val_loss)
            else:
                self.scheduler.step()

            current_lr = list(
              map(lambda d: d['lr'],
                  self.scheduler.optimizer.state_dict()\
                  ['param_groups'])
            )
            self.learning_rates.append(current_lr)

setattr(StepByStep, '_epoch_schedulers', _epoch_schedulers)
```

And then we modify the `train` method to include a call to the protected method defined above. It should come *after the validation* inner loop.

StepByStep Method

```
def train(self, n_epochs, seed=42):
    # To ensure reproducibility of the training process
    self.set_seed(seed)

    for epoch in range(n_epochs):
        # Keeps track of the numbers of epochs
        # by updating the corresponding attribute
        self.total_epochs += 1
```

```
            # inner loop
            # Performs training using mini-batches
            loss = self._mini_batch(validation=False)
            self.losses.append(loss)

            # VALIDATION
            # no gradients in validation!
            with torch.no_grad():
                # Performs evaluation using mini-batches
                val_loss = self._mini_batch(validation=True)
                self.val_losses.append(val_loss)

            self._epoch_schedulers(val_loss)   ①

            # If a SummaryWriter has been set...
            if self.writer:
                scalars = {'training': loss}
                if val_loss is not None:
                    scalars.update({'validation': val_loss})
                # Records both losses for each epoch under tag "loss"
                self.writer.add_scalars(main_tag='loss',
                                        tag_scalar_dict=scalars,
                                        global_step=epoch)

        if self.writer:
            # Closes the writer
            self.writer.close()

setattr(StepByStep, 'train', train)
```

① Calls the learning rate scheduler at the end of every epoch

Mini-Batch Schedulers

These schedulers will have their **`step`** **method** called at the **end of every mini-batch**. They are all **cyclical** schedulers.

- **CyclicLR**: it cycles between `base_lr` and `max_lr` (so it disregards the initial learning rate set in the optimizer), taking `step_size_up` updates to go from base to max learning rate, and `step_size_down` updates to go back. This behavior corresponds to `mode=triangular`. Additionally, it is possible to *shrink* the amplitude using different modes: `triangular2` will **halve** the amplitude after each cycle while `exp_range` will exponentially shrink the amplitude using `gamma` as base and the number of the cycle as the exponent.

A typical choice of value for `max_lr` is the learning rate found using the LR Range Test.

- **OneCycleLR**: it uses a method called annealing to update the learning rate from its initial value up to a defined maximum learning rate (`max_lr`) and then down to a much smaller learning rate over a `total_steps` number of updates, thus performing a single cycle
- **CosineAnnealingWarmRestarts**: it uses cosine annealing[101] to update the learning rate, but we're not delving into details here, except for the fact that this particular scheduler requires the **epoch number** (including the **fractional part** corresponding to the number of mini-batches over the length of the data loader) as an **argument** of its `step` method

Let's try `CyclicLR` in different modes for a range of learning rates between `1e-4` and `1e-3`, two steps in each direction.

```
dummy_parm = [nn.Parameter(torch.randn(1))]
dummy_optimizer = optim.SGD(dummy_parm, lr=0.01)

dummy_scheduler1 = CyclicLR(dummy_optimizer, base_lr=1e-4,
  max_lr=1e-3, step_size_up=2, mode='triangular')
dummy_scheduler2 = CyclicLR(dummy_optimizer, base_lr=1e-4,
  max_lr=1e-3, step_size_up=2, mode='triangular2')
dummy_scheduler3 = CyclicLR(dummy_optimizer, base_lr=1e-4,
  max_lr=1e-3, step_size_up=2, mode='exp_range', gamma=np.sqrt(.5))
```

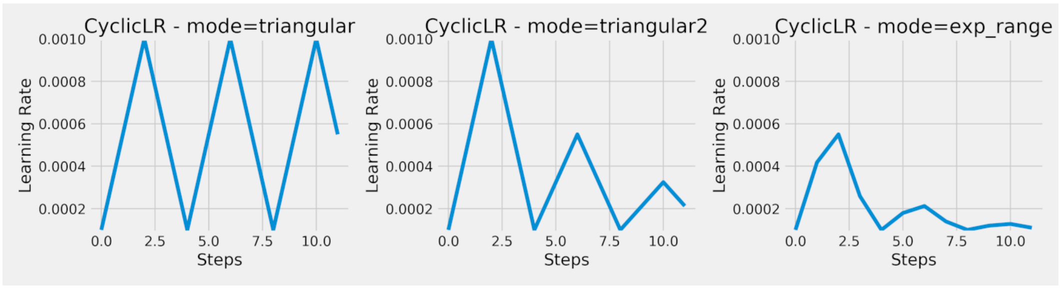


Figure 6.26 - Evolution of learning rate (cyclical scheduler)

By the way, two steps means it would complete a full cycle every four mini-batch updates - that's completely unreasonable - and only used here to illustrate the behavior.

In practice, a cycle should encompass between **two and ten epochs** (according to Leslie N. Smith's paper), so you need to figure how many mini-batches your training set contains (that's the **length of the data loader**) and multiply it by the desired number of epochs in a cycle to get the total number of steps in a cycle.

In our example, the train loader has 158 mini-batches, so if we want the learning rate to **cycle over five epochs**, the full cycle should have 790 steps, and thus `step_size_up` should be half that value (395).

Schedulers in StepByStep - Part II

We need to make some more changes to handle **mini-batch schedulers**. Similarly to "*Part I*" above, we need to create a protected method that handles the `step` method of this group of schedulers.

StepByStep Method

```
def _mini_batch_schedulers(self, frac_epoch):
    if self.scheduler:
        if self.is_batch_lr_scheduler:
            if isinstance(self.scheduler,
              torch.optim.lr_scheduler.CosineAnnealingWarmRestarts):
                self.scheduler.step(self.total_epochs + frac_epoch)
            else:
                self.scheduler.step()

            current_lr = list(
              map(lambda d: d['lr'],
                  self.scheduler.optimizer.state_dict()\
                  ['param_groups'])
            )
            self.learning_rates.append(current_lr)

setattr(StepByStep, '_mini_batch_schedulers',
_mini_batch_schedulers)
```

And then we modify the `mini_batch` method to include a call to the protected method defined above. It should be called at the end of the loop, but **only during training**.

StepByStep Method

```
def _mini_batch(self, validation=False):
    # The mini-batch can be used with both loaders
    # The argument `validation` defines which loader and
    # corresponding step function is going to be used
    if validation:
        data_loader = self.val_loader
        step = self.val_step
    else:
        data_loader = self.train_loader
        step = self.train_step

    if data_loader is None:
        return None
    n_batches = len(data_loader)
    # Once the data loader and step function, this is the same
    # mini-batch loop we had before
    mini_batch_losses = []
    for i, (x_batch, y_batch) in enumerate(data_loader):
        x_batch = x_batch.to(self.device)
        y_batch = y_batch.to(self.device)

        mini_batch_loss = step(x_batch, y_batch)
        mini_batch_losses.append(mini_batch_loss)

        if not validation:                          ①
            self._mini_batch_schedulers(i / n_batches) ②

    loss = np.mean(mini_batch_losses)
    return loss

setattr(StepByStep, '_mini_batch', _mini_batch)
```

① Only during training!

② Calls the learning rate scheduler at the end of every mini-batch update

Scheduler Paths

Before trying a couple of schedulers out, let's run an LR Range test on our model:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(1, 1))
loss_fn = nn.MSELoss(reduction='mean')

nesterov = False
optimizer = optim.SGD(
  model.parameters(), lr=1e-3, momentum=0.9, nesterov=nesterov
)

tracking = lr_range_test(model, loss_fn, optimizer, device,
                         train_loader, end_lr=1, num_iter=100)
```

We're starting really small (`lr=1e-3`) and testing all the way up to `1.0` (`end_lr`) using exponential increments. The results suggest a **learning rate somewhere between `0.01` and `0.1`** (corresponding to the center of the descending part of the curve). We know for a fact that a learning rate of `0.1` works. A more conservative choice of value would be `0.025`, for instance, since it is a midpoint in the descending part of the curve.

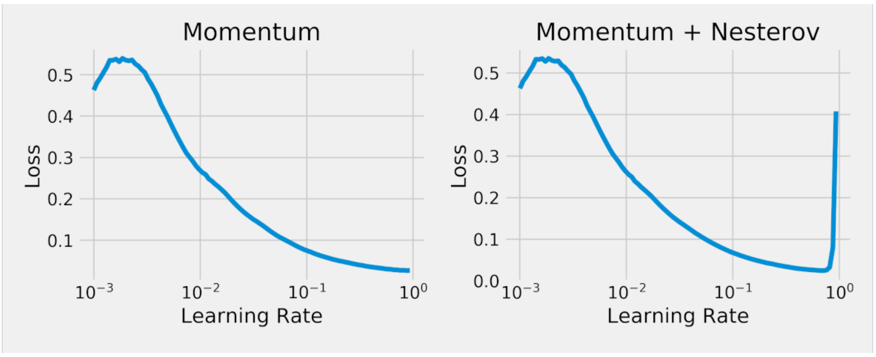


Figure 6.27 - Learning rate finder

Let’s be bold! First, we define the optimizer with our choice for initial learning rate (`0.1`):

```
nesterov = False
optimizer = optim.SGD(
    model.parameters(), lr=0.1, momentum=0.9, nesterov=nesterov
)
```

Then, we can pick a scheduler to bring the learning rate all the way down to `0.025`. If we choose a *step scheduler*, we can cut the learning rate in half (`gamma=0.5`) every four epochs. If we choose a *cyclical scheduler*, we can oscillate the learning rate between the two extremes every four epochs (20 mini-batches, 10 up, 10 down).

```
step_scheduler = StepLR(optimizer, step_size=4, gamma=0.5)
cyclic_scheduler = CyclicLR(
    optimizer, base_lr=0.025, max_lr=0.1,
    step_size_up=10, mode='triangular2'
)
```

Applying each scheduler to SGD with momentum, and to SGD with Nesterov’s momentum, we obtain the following paths:

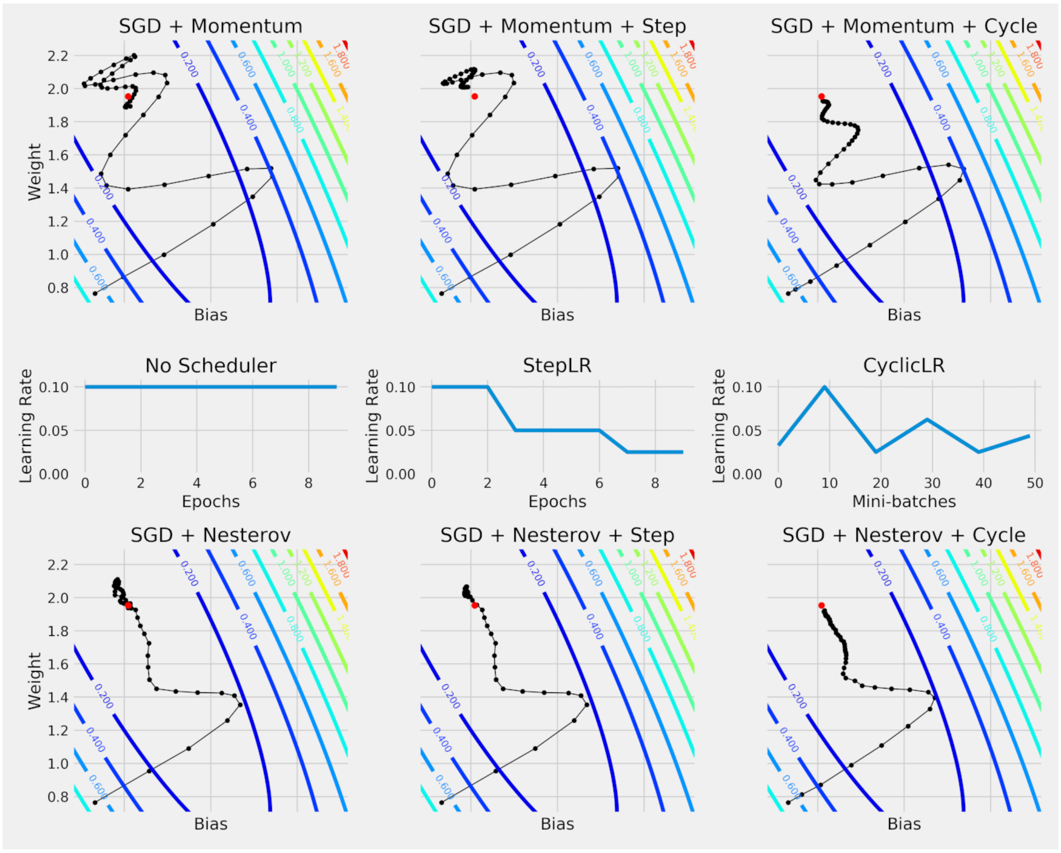


Figure 6.28 - Paths taken by SGD combining momentum and scheduler

Adding a scheduler to the mix seems to have helped the optimizer to achieve a more stable path towards the minimum.

The general idea of using a scheduler is to allow the optimizer to alternate between **exploring the loss surface** (high learning rate phase) and **targeting a minimum** (low learning rate phase).

What about the impact of the scheduler on loss trajectories? Let's check it out:

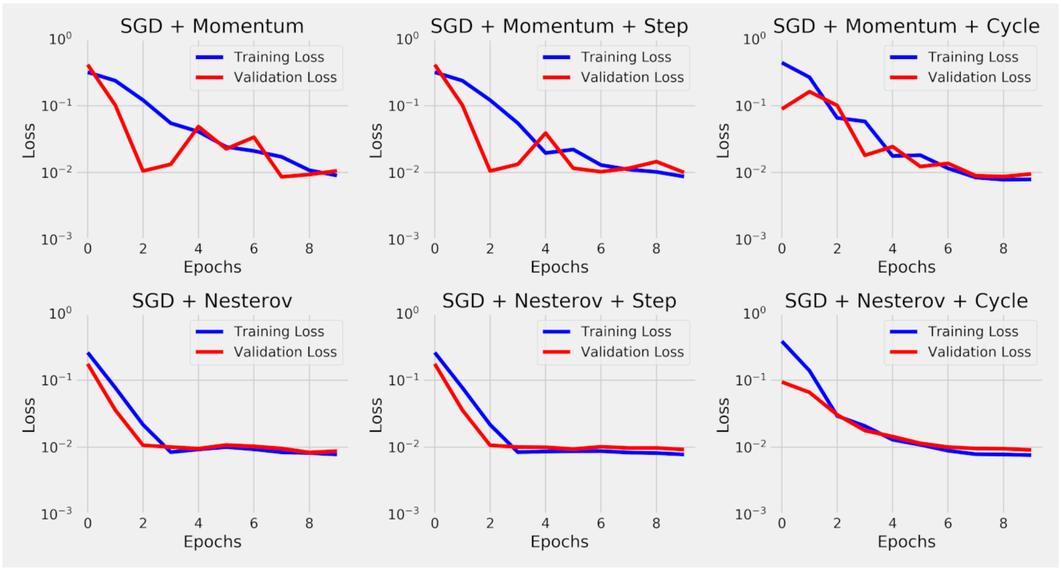


Figure 6.29 - Losses for SGD combining momentum and scheduler

It is definitely harder to tell the difference between curves in the same row, except for the combination of Nesterov's momentum and cyclical scheduler, which produced a smoother reduction in the training loss.

Adaptive vs Cycling

Although *adaptive learning rates* are considered competitors of *cyclical learning rates*, this does not prevent you from combining them both and **cycle learning rates** while using **Adam**. While Adam adapts the gradients using its EWMAs, the cycling policy modifies the learning rate itself, so they can work together indeed.

There is **much more** to learn about in the topic of **learning rates**: this section is meant to be a short introduction to the topic only.

Putting It All Together

In this chapter, we were all over the place: data preparation, model configuration, and model training, a little bit of everything. Starting with a brand new dataset, "*Rock*, *Paper*, *Scissors*", we built a method for **standardizing** the images (for real this time) using a temporary data loader. Next, we developed a fancier model including **dropout** layers for regularization. Then we turned our focus to the training part, diving deeper into **learning rates**, **optimizers**, and **schedulers**. We implemented many methods: for finding a learning rate, for capturing gradients and parameters, and for updating the learning rate using a scheduler.

Data Preparation

```
# Loads temporary dataset to build normalizer
temp_transform = Compose([Resize(28), ToTensor()])
temp_dataset = ImageFolder(root='rps', transform=temp_transform)
temp_loader = DataLoader(temp_dataset, batch_size=16)
normalizer = StepByStep.make_normalizer(temp_loader)

# Builds transformation, datasets and data loaders
composer = Compose([Resize(28),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(train_data, batch_size=16, shuffle=
True)
val_loader = DataLoader(val_data, batch_size=16)
```

In the model configuration part, we can use **SGD with Nesterov's momentum** and a **higher dropout probability** to increase regularization:

Model Configuration

```python
torch.manual_seed(13)
model_cnn3 = CNN2(n_feature=5, p=0.5)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn3 = optim.SGD(
    model_cnn3.parameters(), lr=1e-3, momentum=0.9, nesterov=True
)
```

Before the actual training, we can run an LR Range Test:

Learning Rate Range Test

```python
sbs_cnn3 = StepByStep(model_cnn3, multi_loss_fn, optimizer_cnn3)
tracking, fig = sbs_cnn3.lr_range_test(
    train_loader, end_lr=2e-1, num_iter=100
)
```

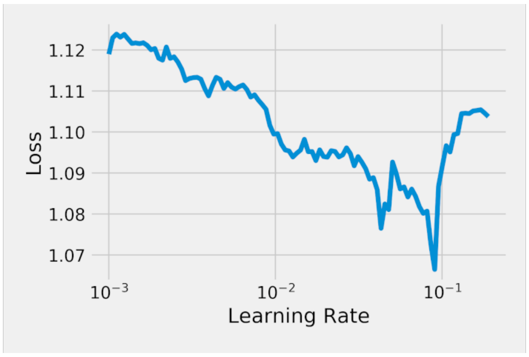


Figure 6.30 - Learning rate finder

The test suggests a learning rate around `0.01`, so we re-create the optimizer and set it in our `StepByStep` instance.

We can also use the suggested learning rate as the **upper range** of a **cyclical scheduler**. For its step size, we can use the length of the data loader, so the learning rate goes all the way up in one epoch, and all the way down in the next - a two-epoch cycle.

Updating Learning Rate

```
optimizer_cnn3 = optim.SGD(
    model_cnn3.parameters(), lr=0.03, momentum=0.9, nesterov=True
)
sbs_cnn3.set_optimizer(optimizer_cnn3)

scheduler = CyclicLR(
    optimizer_cnn3, base_lr=1e-3, max_lr=0.01,
    step_size_up=len(train_loader), mode='triangular2'
)
sbs_cnn3.set_lr_scheduler(scheduler)
```

After doing this, it is training as usual:

Model Training

```
sbs_cnn3.set_loaders(train_loader, val_loader)
sbs_cnn3.train(10)
```

```
fig = sbs_cnn3.plot_losses()
```

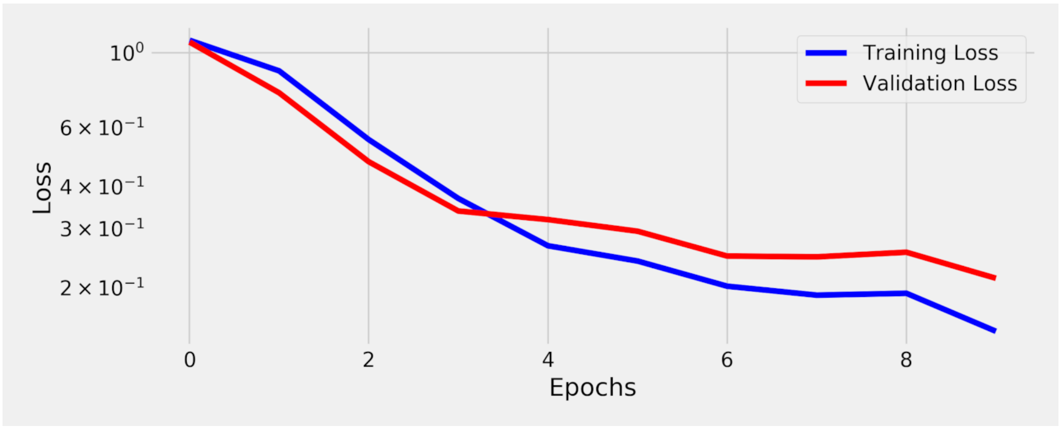


Figure 6.31 - Losses

Evaluation

```
print(StepByStep.loader_apply(
          train_loader, sbs_cnn3.correct).sum(axis=0),
      StepByStep.loader_apply(
          val_loader, sbs_cnn3.correct).sum(axis=0))
```

Output

```
tensor([2511, 2520]) tensor([336, 372])
```

Looking good! Smaller losses, 99.64% training accuracy, and 90.32% validation accuracy.

Recap

In this chapter, we've introduced dropout layers for regularization and focused on the inner workings of different optimizers and the role of the learning rate in the process. This is what we've covered:

- **computing channel statistics** using a temporary data loader to build a `Normalizer` transform

- using the `Normalizer` to **standardize** an image dataset
- understanding how **convolutions over multiple channels** work
- building a fancier model with **two typical convolutional blocks** and **dropout layers**
- understanding how the **dropout probability** generates a **distribution of outputs**
- observing the **effect of `train` and `eval` modes** in dropout layers
- visualizing the **regularizing effect** of **dropout layers**
- using **learning rate range test** to find an interval of learning rate candidates
- computing **bias-corrected exponentially weighted moving averages** of both **gradients** and **squared gradients** to implement **adaptive learning rates** like the **Adam optimizer**
- capturing **gradients** using `register_hook` on tensors of learnable parameters
- capturing **parameters** using the previously implemented `attach_hooks` method
- **visualizing the path** taken by different optimizers for updating parameters
- understanding how **momentum** is computed and its **effect on the parameter update**
- (re)discovering the clever **look-ahead trick** implemented by **Nesterov's momentum**
- learning about different types of **schedulers**: epoch, validation loss, and mini-batch
- including **learning rate schedulers** in the training loop
- **visualizing** the impact of a **scheduler** on the **path** taken for updating parameters

Congratulations! You have just learned the tools commonly used for training deep learning models: **adaptive learning rates**, **momentum**, and **learning rate schedulers**. Far from being an exhaustive lesson on this topic, the general idea is to give you a good understanding of the basic building blocks. You have also learned how **dropout** can be used to **reduce overfitting** and, consequently, **improving generalization**.

In the next chapter, we'll learn about **transfer learning** to leverage the power of **pre-trained models**, and we'll go over some key components of popular architectures, like **1x1 convolutions**, **batch normalization** layers, and **residual connections**.

[91] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter06.ipynb

[92] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter06.ipynb

[93] http://www.samkass.com/theories/RPSSL.html

[94] http://www.laurencemoroney.com/rock-paper-scissors-dataset/

[95] https://storage.googleapis.com/laurencemoroney-blog.appspot.com/rps.zip

[96] https://storage.googleapis.com/laurencemoroney-blog.appspot.com/rps-test-set.zip

[97] https://twitter.com/karpathy/status/801621764144971776

[98] https://arxiv.org/abs/1506.01186

[99] https://pypi.org/project/torch-lr-finder/

[100] http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/

[101] https://paperswithcode.com/method/cosine-annealing

Chapter 7
Transfer Learning

Spoilers

In this chapter, we will:

- learn about **ImageNet** and popular models like AlexNet, VGG, Inception, and ResNet
- use **transfer learning** to classify images from the "*Rock*, *Paper*, *Scissors*" dataset
- load **pre-trained models** for **fine-tuning** and **feature extraction**
- understand the role of **auxiliary classifiers** in very deep architectures
- use **1x1 convolutions** as a **dimension reduction layer**
- build an **inception module**
- understand how **batch normalization** impacts model training in many ways
- understand the purpose of **residual (skip) connections** and build a **residual block**

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 7[102] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[103].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 7's notebook. If not, just click on `Chapter07.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from PIL import Image

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset, random_split, \
    TensorDataset
from torchvision.transforms import Compose, ToTensor, Normalize, \
    Resize, ToPILImage, CenterCrop, RandomResizedCrop
from torchvision.datasets import ImageFolder
from torchvision.models import alexnet, resnet18, inception_v3
from torchvision.models.alexnet import model_urls
from torchvision.models.utils import load_state_dict_from_url

from stepbystep.v3 import StepByStep
```

Transfer Learning

In the last chapter, I called the model **fancier** just because it had not one, but **two convolutional blocks**, and **dropout** layers as well. Truth to be told, this is *far from fancy*… really fancy models have **tens of convolutional blocks** and other neat architectural tricks that make them **really powerful**. They have **many million parameters** and require not only **humongous amounts of data** but also **thousands of (expensive) GPU-hours** for training.

I don't know about you, but I have neither! So, what's left to do? **Transfer learning to the rescue!**

The idea is quite simple: first, some big tech company, which has access to virtually infinite amounts of data and computing power, develops and **trains a huge model** for their own purpose. Next, once it is trained, its **architecture and the corresponding trained weights** (the **pre-trained model**) are released. Finally, everyone else can **use these weights as a starting point** and **fine-tune it further for a different (but similar) purpose**.

That's **transfer learning** in a nutshell. It started with computer vision models and…

ImageNet

> **ImageNet** is an image database organized according to the WordNet[104] hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently we have an average of over five hundred images per node. We hope ImageNet will become a useful resource for researchers, educators, students and all of you who share our passion for pictures.
>
> Source: ImageNet[105]

ImageNet is a comprehensive database of images spanning 27 high-level categories, more than 20,000 sub-categories, and more than 14 million images (check its statistics here[106]). The images themselves **cannot be downloaded from**

its website because ImageNet does not own the copyright of these images. It **does provide the URLs for all images**, though.

As you can probably guess, classifying these images was a monumental task in the early 2010s. No wonder they created a **challenge**…

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

> The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale.
>
> Source: ILSVRC[107]

The ILSVRC ran for eight years, from 2010 to 2017. Many architectures we take for granted today were developed to tackle this challenge: AlexNet, VGG, Inception, ResNet, and more. We're focusing on the years of 2012, 2014, and 2015 only.

ILSVRC-2012

The 2012 edition[108] of the ILSVRC is probably the most popular of them all. Its winner, the architecture dubbed **AlexNet**, represented a milestone for image classification, sharply reducing the classification error. The training data had 1.2 million images belonging to 1,000 categories (it is actually a subset of the ImageNet dataset).

AlexNet (SuperVision Team)

The architecture was developed by the SuperVision team, composed of Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton from the University of Toronto (now you know why it's called AlexNet). Here is their model's description:

> Our model is a large, deep convolutional neural network trained on raw RGB pixel values. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three globally-connected layers with a final 1000-way softmax. It was trained on two NVIDIA GPUs for about a week. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of convolutional nets. To reduce overfitting in the globally-connected layers we employed hidden-unit "dropout", a recently-developed regularization method that proved to be very effective.
>
> Source: Results (ILSVRC2012)[109]

You should be able to recognize all the elements in the description: five typical convolutional blocks (convolution, activation function, and max-pooling) corresponding to the "featurizer" part of the model, three hidden (linear) layers combined with dropout layers corresponding to the "classifier" part of the model, and the softmax output layer typical of multiclass classification problems.

It is pretty much the **fancier model** from Chapter 6 but **on steroids**! We'll be using AlexNet to demonstrate **how to use a pre-trained model**. In case you're interested to learn more about AlexNet, their paper is called *"ImageNet Classification with Deep Convolutional Neural Networks"*[110].

ILSVRC-2014

The 2014 edition[111] gave rise to two common household names when it comes to architectures for computer vision problems: **VGG** and **Inception**. The training data had 1.2 million images belonging to 1,000 categories, just like the 2012 edition.

VGG

The architecture developed by Karen Simonyan and Andrew Zisserman from the Oxford *Vision Geometry Group* (VGG) is pretty much an even larger or, better yet, deeper model than AlexNet (and now you know the origin of yet another architecture name). Their goal is crystal clear in their model's description:

> ...we explore the effect of the **convolutional network (ConvNet) depth** on its accuracy.
>
> Source: Results (ILSVRC2014)[112]

VGG models are **massive**, so we're not paying much attention to them here. If you want to learn more about it, its paper is called "*Very Deep Convolutional Networks for Large-Scale Image Recognition*"[113].

Inception (GoogLeNet Team)

The Inception architecture is probably the one with the best meme of all: "*We need to go deeper*". The authors, Christian Szegedy, et al., like the VGG team, also wanted to train a deeper model. But they came up with a clever way of doing it (highlights are mine):

> Additional **dimension reduction layers** based on embedding learning intuition allow us to **increase both the depth and the width of the network** significantly without incurring significant computational overhead.
>
> Source: Results (ILSVRC2014)[114]

If you want to learn more about it, the paper is called "*Going Deeper with Convolutions*"[115].

"*What are these **dimension reduction layers**?*"

No worries, we'll get back to it in the "*Inception Modules*" section.

ILSVRC-2015

The 2015 edition[116] popularized **residual connections** in the aptly named architecture: **Res**(idual) **Net**(work). The training data used in the competition remained unchanged.

ResNet (MSRA Team)

The trick developed by Kaiming He, et al. was to add **residual connections**, or **shortcuts**, to a very deep architecture.

> We train neural networks with depth of over 150 layers. We propose a "deep residual learning" framework that eases the optimization and convergence of extremely deep networks.
>
> Source: Results (ILSVRC2015)[117]

In a nutshell, it allows the network to more easily learn the **identity function**. We'll also get back to it in the "*Residual Connections*" section later in this chapter. If you want to learn more about it, the paper is called "*Deep Residual Learning for Image Recognition*"[118].

By the way, Kaiming He also has an initialization scheme named after him - sometimes referred to as "*He initialization*", sometimes referred to as "*Kaiming initialization*" - and we'll learn about those later in the next chapter.

Imagenette

If you are looking for a smaller, and more manageable dataset that's ImageNet-like, Imagenette is for you! Developed by Jeremy Howard, from fast.ai, it is a subset of 10 easily classified classes from Imagenet.

You can find it here: https://github.com/fastai/imagenette.

Comparing Architectures

Now that you're familiar with some of the popular architectures (many of them readily available as Torchvision's models), let's compare their performances (Top-1 accuracy %), number of operations in a *single* forward pass (billions), and sizes (in millions of parameters). The figure below is very illustrative in this sense:

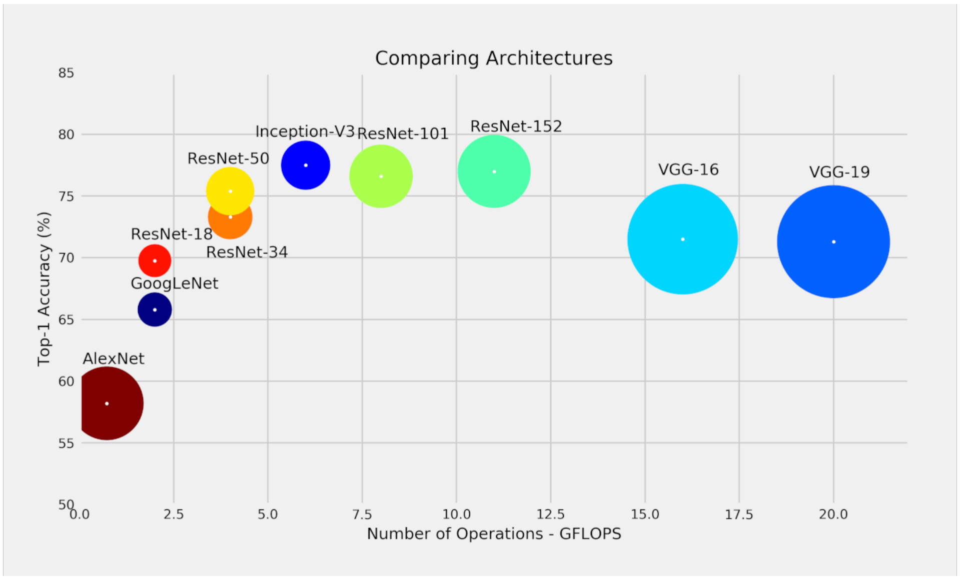


Figure 7.1 - Comparing architectures (size proportional to number of parameters)

Source: Data for accuracy and GFLOPs estimates obtained from this report[119], number of parameters (proportional to the size of the circles) obtained from Torchvision's models. For a more detailed analysis, see Canziani, A., Culurciello, E., Paszke, A. "An Analysis of Deep Neural Network Models for Practical Applications"[120] (2017).

See how **massive** the VGG models are, both in size and in the number of operations required to deliver a **single prediction**? On the other hand, check **Inception-V3** and **ResNet-50**'s positions in the plot: they would give more bang for your buck. The former has a slightly higher performance, and the latter is slightly faster.

These are the models you're likely using for transfer learning: **Inception** and **ResNet**.

On the bottom left, there is AlexNet. It was miles ahead of anything else in 2012, but it is not competitive at all anymore.

"If AlexNet is not competitive, why are you using it to illustrate transfer learning?"

A fair point indeed. The reason is, its architectural elements are already familiar to you, thus making it easier for me to explain **how we're modifying it** to fit our purposes.

Transfer Learning in Practice

In Chapter 6, we created our own model to classify images in the "*Rock, Paper, Scissors*" dataset. We'll use the **same dataset** here but, instead of creating another model, we'll take AlexNet for a spin!

It all starts with **loading a pre-trained model**, which can be easily done using Torchvision's library of models. There, we find `AlexNet`, a PyTorch model that implements the architecture designed by Alex Krizhevsky et al, and `alexnet`, a helper method that creates an instance of `AlexNet` and, optionally, downloads and loads its pre-trained weights.

Pre-Trained Model

We'll start by creating an instance of `AlexNet` without loading its pre-trained weights just yet:

Loading AlexNet architecture

```
alex = alexnet(pretrained=False)
print(alex)
```

Output

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding
=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding
=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

AlexNet's architecture has three main elements: `features`, `avgpool`, and `classifier`. The first and last are *nested* sequential models. The featurizer contains five typical convolutional blocks, and the classifier has two hidden layers using 50% dropout. You're already familiar with all of this, except for **the element in the middle**.

Adaptive Pooling

The element in the middle, `AdaptiveAvgPool2d` (`F.adaptive_avg_pool2d` in its functional form), is a **special kind of pooling**: instead of requiring the kernel size (and stride), it requires the **desired output size**. In other words, **whatever the image size** it gets as input, it will **return a tensor with the desired size**.

"*What's so special about this?*"

It gives you the freedom to use images of different sizes as inputs! We've seen that convolutions and traditional max-pooling, in general, *shrink* the image's dimensions. But the **classifier part** of the model **expects an input of a determined size**. That meant that the **input image had to be of a determined size** such that, at the end of the shrinkage process, it **matched** what the classifier part was expecting. The **adaptive pooling guarantees the output size**, so it can easily match the classifier expectations regarding input size.

Let's say we have two dummy tensors representing feature maps produced by the "featurizer part of the model. The **feature maps have different dimensions** (32x32 and 12x12). Applying **adaptive pooling** to both of them ensures that both **outputs have the same dimensions** (6x6):

```
result1 = F.adaptive_avg_pool2d(
    torch.randn(16, 32, 32), output_size=(6, 6)
)
result2 = F.adaptive_avg_pool2d(
    torch.randn(16, 12, 12), output_size=(6, 6)
)
result1.shape, result2.shape
```

Output

```
(torch.Size([16, 6, 6]), torch.Size([16, 6, 6]))
```

Cool, right? We got the architecture already, but our model is **untrained** (it still has random weights). Let's fix that by…

Loading Weights

First things first: to load the weights into our model, we need to **retrieve them**. Sure, an easy way of retrieving them would be to set the argument `pretrained=True` while creating the model. But you can also **download the weights** from a given URL, which gives you the flexibility to use pre-trained weights from wherever you want!

PyTorch offers the `load_state_dict_from_url` method: it will retrieve the weights from the specified URL and, optionally, save them to a specified folder (`model_dir` argument).

"Great, but what's the URL for AlexNet's weights?"

You can get the URL from the `model_urls` variable in `torchvision.models.alexnet`:

URL for AlexNet's pre-trained weights

```
url = model_urls['alexnet']
url
```

Output

```
'https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth'
```

Of course, it doesn't make any sense to do this manually for models in PyTorch's library. But, assuming you're using pre-trained weights from a third-party, you'd be able to load them like this:

Loading pre-trained weights

```
state_dict = load_state_dict_from_url(
    url, model_dir='pretrained', progress=True
)
```

Output

```
Downloading: "https://download.pytorch.org/models/alexnet-owt-
4df8aa71.pth" to ./pretrained/alexnet-owt-4df8aa71.pth
```

From now on, it works **as if we had saved a model to disk**. To load the model's state dictionary, we can use its `load_state_dict` method:

Loading model

```
alex.load_state_dict(state_dict)
```

Output

```
<All keys matched successfully>
```

There we go! We have a fully trained AlexNet to play with! Now what?

Model Freezing

In most cases, you **don't** want to continue training the whole model. I mean, in theory, you *could* pick it up where it was left off by the original authors, and *resume training* using your own dataset. That's a lot of work, and you'd need a *lot of data* to make any kind of meaningful progress. There *must* be a better way! Of course, there is: we can **freeze the model**.

Freezing the model means it **won't learn anymore**, that is, **its parameters/weights will not be updated anymore**.

What best characterizes a tensor representing a **learnable parameter**? It **requires gradients**. So, if we'd like to make them stop learning anything, we need to change exactly that:

Helper Method #6 - Model freezing

```
def freeze_model(model):
    for parameter in model.parameters():
        parameter.requires_grad = False
```

Output

```
freeze_model(alex)
```

The function above loops over **all parameters** of a given model and **freezes them**.

"If the model is frozen, how I am supposed to train it for my own purpose?"

Excellent question! We have to *unfreeze* a small part of the model or, better yet, **replace a small part of the model**. We'll be replacing the…

Top of the Model

The "*top*" of the model is loosely defined as the **last layer(s)** of the model, usually belonging to its **classifier** part. The **"featurizer"** part is usually left untouched since we're trying to leverage the model's ability to **generate features** for us. Let's inspect AlexNet's classifier once again:

```
print(alex.classifier)
```

Output

```
Sequential(
  (0): Dropout(p=0.5, inplace=False)
  (1): Linear(in_features=9216, out_features=4096, bias=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=4096, out_features=4096, bias=True)
  (5): ReLU(inplace=True)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```

It has two hidden layers and one output layer. The output layer produces 1,000 logits, one for each class in the ILSVRC challenge. But, unless you are playing with the dataset used for the challenge, **you'd have your own classes to compute logits for**.

In our "*Rock*, *Paper*, *Scissors*" dataset, we have **three classes**. So, we need to **replace the output layer accordingly**:

Replacing the "top" of the model

```
alex.classifier[6] = nn.Linear(4096, 3)
```

The following diagram may help you visualize what's happening:

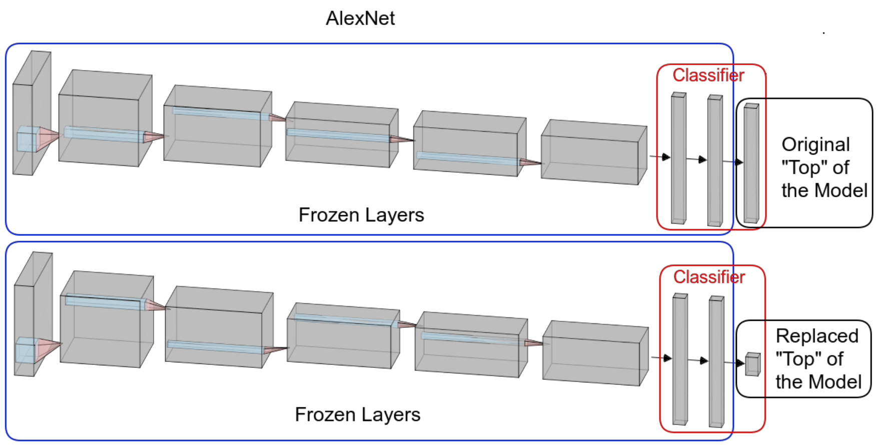


Figure 7.2 - AlexNet

Source: Generated using Alexander Lenail's NN-SVG[121] and adapted by the author

Notice that the **number of input features remains the same** since it still takes the output from the hidden layer that precedes it. The **new output layer requires gradients by default**, but we can double-check it:

```
for name, param in alex.named_parameters():
    if param.requires_grad == True:
        print(name)
```

Output

```
classifier.6.weight
classifier.6.bias
```

Great, the **only layer** that will be **learning** anything is our brand new **output layer** (`classifier.6`), the **top of the model**.

> *"What about **unfreezing** some of the hidden layers?"*

That's also a possibility… in this case, it is like **resuming training** for the hidden layers, while **learning from scratch** for the output layer. You'd probably have to have **more data** to pull this off, though.

> *"Could I have changed the **whole classifier** instead of just the output layer?"*

Sure thing! It would be *possible* to have a **different architecture** for the classifier part, as long as it takes the 9,216 input features produced by the first part of AlexNet, and outputs as many logits as necessary for the task at hand. In this case, the whole classifier would be learning from scratch, and you'd need **even more data** to pull it off.

> The **more layers you unfreeze or replace**, the **more data** you'll need to fine-tune the model.

We're sticking with the **simplest approach** here, that is, **replacing the output layer only**.

> Technically speaking, we're only **fine-tuning a model** if we **do not freeze pre-trained weights**, that is, the whole model will be (slightly) updated. Since we are **freezing everything but the last layer**, we are actually using the pre-trained model for **feature extraction** only.

> *"What if I use a **different model**? Which layer should I replace then?"*

The table below covers some of the most common models you may use for transfer learning. It lists the **expected size of the input images**, the **classifier layer to be**

replaced, and the **appropriate replacement**, given the number of classes for the task at hand (three in our case):

Model	Size	Classifier Layer(s)	Replacement Layer(s)
AlexNet	224	`model.classifier[6]`	`nn.Linear(4096,num_classes)`
VGG	224	`model.classifier[6]`	`nn.Linear(4096,num_classes)`
InceptionV3	299	`model.fc`	`nn.Linear(2048,num_classes)`
		`model.AuxLogits.fc`	`nn.Linear(768,num_classes)`
ResNet	224	`model.fc`	`nn.Linear(512,num_classes)`
DenseNet	224	`model.classifier`	`nn.Linear(1024,num_classes)`
SqueezeNet	224	`model.classifier[1]`	`nn.Conv2d(512,num_classes, kernel_size=1,stride=1)`

"*Why there are **two layers** for the **Inception V3** model?*"

The Inception model is a **special case** because it has **auxiliary classifiers**. We'll discuss them later in this chapter.

Model Configuration

What's missing in the model configuration? A loss function and an optimizer. A multiclass classification problem, when the model produces logits, requires `CrossEntropyLoss` as the loss function. For the optimizer, let's use Adam with the "*Karpathy Constant*" (`3e-4`) as learning rate.

Model Configuration

```
torch.manual_seed(17)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_alex = optim.Adam(alex.parameters(), lr=3e-4)
```

Cool, the model configuration is taken care of, we can turn our attention to the…

Data Preparation

This step is quite similar to what we've done in the previous chapter (we're still using the "*Rock*, *Paper*, *Scissors*" dataset), except for one key difference: we will **use different parameters for standardizing** the images.

> *"So we're not computing statistics for the images in our dataset anymore?"*

Nope!

> *"Why not?"*

Since we're using a **pre-trained model**, we need to use the **standardization parameters** used to train the original model. In other words, we need to use the **statistics** of the **original dataset** used to train that model. For AlexNet (and pretty much every computer vision pre-trained model), these statistics were computed on the ILSVRC dataset.

You can find these values in PyTorch's documentation for pre-trained models:

ImageNet Statistics

All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`. You can use the following transform:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

So the data preparation step for the "*Rock, Paper, Scissors*" dataset looks like this now:

Data Preparation

```
normalizer = Normalize(mean=[0.485, 0.456, 0.406],
                       std=[0.229, 0.224, 0.225])

composer = Compose([Resize(256),
                    CenterCrop(224),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
val_loader = DataLoader(val_data, batch_size=16)
```

Model Training

We have everything set to train the top layer of our modified version of AlexNet:

Model Training

```
sbs_alex = StepByStep(alex, multi_loss_fn, optimizer_alex)
sbs_alex.set_loaders(train_loader, val_loader)
sbs_alex.train(1)
```

You probably noticed it took some seconds (and a lot more if you're running on a CPU) to run the code above, even though it is training for **one single epoch**.

*"How come? Most of the model is **frozen**, there is only one measly layer to train..."*

You’re right, there is only one measly layer to *compute gradients for* and to *update its parameters*, but the **forward pass** still uses the **whole model**. So, every single image (out of 2,520 in our training set) will have its features computed using more than **61 million parameters**! No wonder it is taking some time! By the way, **only 12,291 parameters are trainable**.

If you’re thinking "*there must be a better way…*", you’re absolutely right - that’s the topic of the next section.

But, first, let’s see **how effective transfer learning is** by **evaluating** our model after training it over **one epoch only**:

```
StepByStep.loader_apply(val_loader, sbs_alex.correct)
```

Output

```
tensor([[111, 124],
        [124, 124],
        [124, 124]])
```

That’s 96.51% accuracy in the validation set (it is 99.33% for the training set, in case you’re wondering). Even if it is taking some time to train, these results are pretty good!

Generating a Dataset of Features

We’ve just realized that, from the time it takes to train the last layer of our model over **one single** epoch, **most of it was spent in the forward pass**. Now, imagine if we wanted to train it over **ten epochs**: not only the model would spend most of its time performing the forward pass but, **even worse**, it would perform the **same operations ten times over**.

Since all layers but the last are frozen, the **output of the second to last layer is always the same**.

That’s assuming you’re **not doing data augmentation**, of course.

That’s a huge waste of your time, energy, and money (if you’re paying for cloud computing).

"*What can we do about it?*"

Well, since the **frozen layers** are simply **generating features** that will be the input of the **trainable layers**, why not treat the frozen layers as such? We could do it in four easy steps:

- keep **only the frozen layer**s in the model
- **run the whole dataset through it** and collect **its outputs as a dataset of features**
- train a **separate model** (that corresponds to the "*top*" of the original model) using the **dataset of features**
- **attach the trained model** to the **top of the frozen layers**

This way we’re effectively **splitting the feature extraction and actual training phases**, thus avoiding the overhead of generating features over and over again for every single forward pass.

To keep **only the frozen layers**, we need to get rid of the "*top*" of the original model. But, since we also want to **attach our new layer to the whole model after training**, it is a better idea to simply **replace the "*top*" layer with an identity layer instead of removing it**:

"Removing" the Top Layer

```
alex.classifier[6] = nn.Identity()
print(alex.classifier)
```

Output

```
Sequential(
  (0): Dropout(p=0.5, inplace=False)
  (1): Linear(in_features=9216, out_features=4096, bias=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=4096, out_features=4096, bias=True)
  (5): ReLU(inplace=True)
  (6): Identity()
)
```

This way, the last effective layer still is **`classifier.5`**, which will produce the features we're interested in. We have a **feature extractor** in our hands now! Let's use it to pre-process our dataset.

The function below loops over the mini-batches from a data loader, sends them through our feature extractor model, combines the outputs with the corresponding labels, and returns a `TensorDataset`:

Helper Method #7

```
def preprocessed_dataset(model, loader, device=None):
    if device is None:
        device = next(model.parameters()).device

    features = None
    labels = None

    for i, (x, y) in enumerate(loader):
        model.eval()
        output = model(x.to(device))
        if i == 0:
            features = output.detach().cpu()
            labels = y.cpu()
        else:
            features = torch.cat([features, output.detach().cpu()])
            labels = torch.cat([labels, y.cpu()])

    dataset = TensorDataset(features, labels)
    return dataset
```

We can use it to pre-process our datasets:

Data Preparation (1)

```
train_preproc = preprocessed_dataset(alex, train_loader)
val_preproc = preprocessed_dataset(alex, val_loader)
```

There we go, we have `TensorDatasets` containing tensors for **features** generated by AlexNet for each and every image, as well as for the corresponding **labels**.

IMPORTANT: this pre-processing step assumes **no data augmentation**. If you want to perform data augmentation you will need to train the top of the model while it is still attached to the rest of the model since the features produced by the frozen layers will be slightly different every time due to the augmentation itself.

We can also **save these tensors to disk**:

```
torch.save(train_preproc.tensors, 'rps_preproc.pth')
torch.save(val_preproc.tensors, 'rps_val_preproc.pth')
```

So they can be used to build datasets later:

```
x, y = torch.load('rps_preproc.pth')
train_preproc = TensorDataset(x, y)
val_preproc = TensorDataset(*torch.load('rps_val_preproc.pth'))
```

The last step of the data preparation, as usual, is the creation of the data loader:

Data Preparation (2)

```
train_preproc_loader = DataLoader(
    train_preproc, batch_size=16, shuffle=True
)
val_preproc_loader = DataLoader(val_preproc, batch_size=16)
```

Bye, bye, costly and repetitive forward passes! We can now focus on training our…

Top Model

The model has only one layer, which matches the one we used in the "*Top of the Model*" subsection. The rest of the model configuration part remains unchanged:

Model Configuration - Top Model

```
torch.manual_seed(17)
top_model = nn.Sequential(nn.Linear(4096, 3))
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_top = optim.Adam(top_model.parameters(), lr=3e-4)
```

Next, we create another `StepByStep` instance to train the model above using the **pre-processed dataset**. Since it is a tiny model, we can afford to train it over **ten epochs, instead of only one**:

Model Training - Top Model

```
sbs_top = StepByStep(top_model, multi_loss_fn, optimizer_top)
sbs_top.set_loaders(train_preproc_loader, val_preproc_loader)
sbs_top.train(10)
```

See? That was **blazing fast**!

Now we can **attach the trained model** to the top of the full (frozen) model:

Replacing the Top Layer

```
sbs_alex.model.classifier[6] = top_model
print(sbs_alex.model.classifier)
```

Output

```
Sequential(
  (0): Dropout(p=0.5, inplace=False)
  (1): Linear(in_features=9216, out_features=4096, bias=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=4096, out_features=4096, bias=True)
  (5): ReLU(inplace=True)
  (6): Sequential(
    (0): Linear(in_features=4096, out_features=3, bias=True)
  )
)
```

The sixth element of the classifier part corresponds to our small trained model. Let's see how it performs on the validation set.

We're using the **full model** again, so we should use the **original dataset instead of the pre-processed one**.

```
StepByStep.loader_apply(val_loader, sbs_alex.correct)
```

Output

```
tensor([[109, 124],
        [124, 124],
        [124, 124]])
```

It is **almost the same result** as before. The model is probably overfitting, but it doesn't matter because the purpose of this exercise was to show you how to **use transfer learning** and how you can **pre-process your dataset to speed up model training**.

AlexNet was fun to work with, but it is time to move on. In the next sections, we'll focus on **new architectural elements** that are part of **Inception** and **ResNet** models.

Auxiliary Classifiers (Side-Heads)

The first version of the **Inception** model (depicted in the figure below) introduced **auxiliary classifiers**, that is, **side-heads** attached to intermediate parts of the model that would also try to perform classification, **independently** from the typical **main classifier** at the very end of the network:

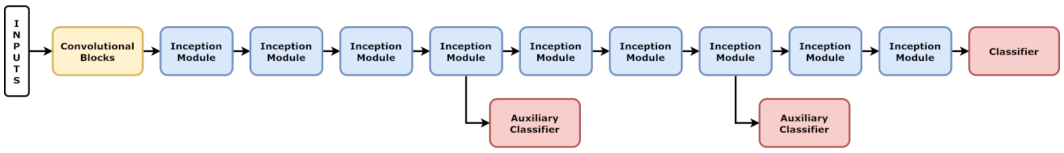


Figure 7.3 - Inception model: simplified diagram

The **cross-entropy loss** was also computed independently for **each one of the three classifiers** and **added together** to the total loss (although auxiliary losses were multiplied by a factor of 0.3). The auxiliary classifiers (and losses) were used during training time only. During the **evaluation** phase, **only the logits produced by the main classifier were considered**.

The technique was originally developed to mitigate the **vanishing gradients** problem (more on that in the next chapter), but it was later found that the **auxiliary classifiers** are more likely to have a **regularizer effect** instead[122].

The third version of the Inception model (`inception_v3`), available as a pre-trained model in PyTorch, has only **one auxiliary classifier** instead of two, but we still need to **make some adjustments** if we're using this model for transfer learning.

First, we **load** the pre-trained model, **freeze** its layers, and **replace the layers** for both **main and auxiliary classifiers**:

Loading Pre-trained Inception V3 and Replacing Top Layers

```
model = inception_v3(pretrained=True)
freeze_model(model)

torch.manual_seed(42)
model.AuxLogits.fc = nn.Linear(768, 3)
model.fc = nn.Linear(2048, 3)
```

Unfortunately, we cannot use the standard cross-entropy loss because the Inception model **outputs two tensors**, one for each classifier (although it is possible to force it to return only the main classifier by setting its `aux_logits` argument to `False`). But we can create a simple **function** that can handle **multiple outputs**, computing the **corresponding losses** and returning their total:

Helper Method #8 - Inception Loss with Side-heads

```
def inception_loss(outputs, labels):
    try:
        main, aux = outputs
    except ValueError:
        main = outputs
        aux = None
        loss_aux = 0

    multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
    loss_main = multi_loss_fn(main, labels)
    if aux is not None:
        loss_aux = multi_loss_fn(aux, labels)
    return loss_main + 0.4 * loss_aux
```

The auxiliary loss, in this case, is multiplied by a factor of 0.4 before being added to the main loss. Now, we're only missing an optimizer:

Model Configuration

```
optimizer_model = optim.Adam(model.parameters(), lr=3e-4)
sbs_incep = StepByStep(model, inception_loss, optimizer_model)
```

"Wait, aren't we pre-processing the dataset this time?"

Unfortunately, no. The `preprocessed_dataset` cannot handle multiple outputs. Instead of making the process too convoluted in order to handle the peculiarities of the Inception model, I am sticking with the simpler (yet slower) way of training the last layer while it is still attached to the rest of the model.

The Inception model is also different from the others in its expected input size: 299 instead of 224. So, we need to recreate the data loaders accordingly:

Data Preparation

```
normalizer = Normalize(mean=[0.485, 0.456, 0.406],
                       std=[0.229, 0.224, 0.225])

composer = Compose([Resize(299),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
val_loader = DataLoader(val_data, batch_size=16)
```

We're ready, let's train our model for a single epoch and evaluate the result:

Model Training

```
sbs_incep.set_loaders(train_loader, val_loader)
sbs_incep.train(1)
```

```
StepByStep.loader_apply(val_loader, sbs_incep.correct)
```

Output

```
tensor([[108, 124],
        [116, 124],
        [108, 124]])
```

It achieved an accuracy of 89.25% on the validation set. Not bad!

There is more to the Inception model than auxiliary classifiers, though. Let's check other architectural elements it contains.

1x1 Convolutions

This particular architectural element is *not exactly new*, but it is a somewhat *special case* of an already known element. So far, the **smallest kernel** used in a convolutional layer had a size **three-by-three**. These kernels performed an **element-wise multiplication** and then **added up the resulting elements** to produce a **single value** for each region they were applied to. So far, nothing new.

The idea of a kernel of size **one-by-one** is somewhat counterintuitive at first. For a single channel, this kernel is only **scaling the values** of its input and nothing else. That seems hardly useful…

But everything changes if you have **multiple channels**! Remember the three-channel convolutions from Chapter 6? A filter has **as many channels as its input**. It means that **each channel will be scaled independently** and the **results will be**

added up, resulting in **one channel as output** (per filter).

A 1x1 convolution can be used to **reduce the number of channels**, that is, it may work as a **dimension reduction layer**.

An image is worth a thousand words, so let's visualize this:

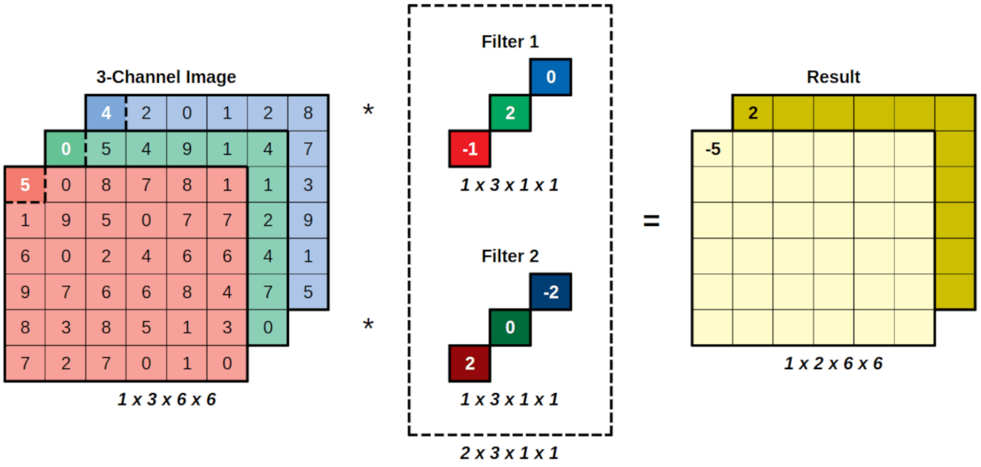


Figure 7.4 - 1x1 Convolution

The input is an RGB image and there are **two filters**, each filter has **three 1x1 kernels**, one for each channel of the input. What are these filters actually doing? Let's check it out:

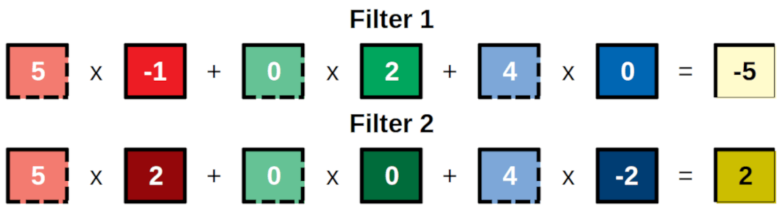


Figure 7.5 - 1x1 Convolution

Maybe it is even more clear if it is presented as a formula:

$$\begin{aligned}
Filter\ 1: &\quad -1R + 2G \quad +0B \\
Filter\ 2: &\quad 2R + 0G \quad -2B
\end{aligned}$$

Equation 7.1 - Filter arithmetic

A filter using a **1x1 convolution** corresponds to a **weighted average of the input channels**.

In other words, a 1x1 convolution is a **linear combination of the input channels**, computed **pixel by pixel**.

There is another way to get a **linear combination of the inputs**: a linear layer, also referred to as a *fully connected layer*. Performing a 1x1 convolution is akin to **applying a linear layer to each individual pixel over its channels**.

This is the reason why a **1x1 convolution** is said to be **equivalent to a fully connected (linear) layer**.

In the example above, each one of the two filters produces a different linear combination of the RGB channels. Does this ring any bells? In Chapter 6, we've seen that **grayscale images** can be computed using a **linear combination of the red, green, and blue channels** of colored images. So, we can **convert an image to grayscale** using a **1x1 convolution**!

```
scissors = Image.open('rps/scissors/scissors01-001.png')
image = ToTensor()(scissors)[:3, :, :].view(1, 3, 300, 300)

weights = torch.tensor([0.2126, 0.7152, 0.0722]).view(1, 3, 1, 1)
convolved = F.conv2d(input=image, weight=weights)

converted = ToPILImage()(convolved[0])
grayscale = scissors.convert('L')
```

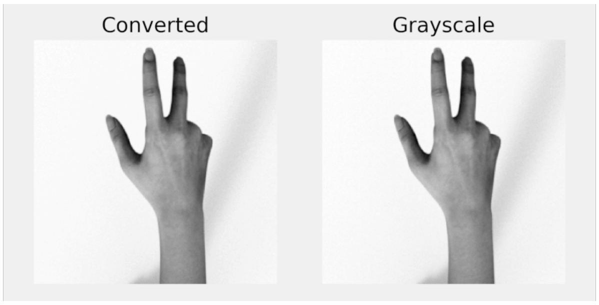


Figure 7.6 - Convolution vs Conversion

See? They are the same… or are they? If you have a *really sharp eye*, maybe you are able to notice a subtle difference between the two shades of gray. It doesn't have anything to do with the use of convolutions, though… it turns out, PIL uses slightly different weights for converting RGB into grayscale.

The weights used by PIL are 0.299 for red, 0.587 for blue, and 0.114 for green, the "*ITU-R 601-2 luma transform*". Our weights were different because we used the colorimetric conversion to grayscale. If you want to learn more about it, check Wikipedia's article on Grayscale[123].

You can think of converting colored images to grayscale as **reducing the dimensions of the image** since the size of the output is one-third of the input size (one instead of three channels). This translates into **having three times fewer parameters** in the layer that is receiving it as its own input, and it allows the networks to **grow deeper** (and wider).

"*We need to go deeper.*"

Dom Cobb

Inception Modules

Memes aside, let's talk about the inception module. There are *many versions* of it, as it evolved over time, but we're focusing on the first two only: the *regular* version, and the one with **dimension reductions**. Both versions are depicted in the figure below:

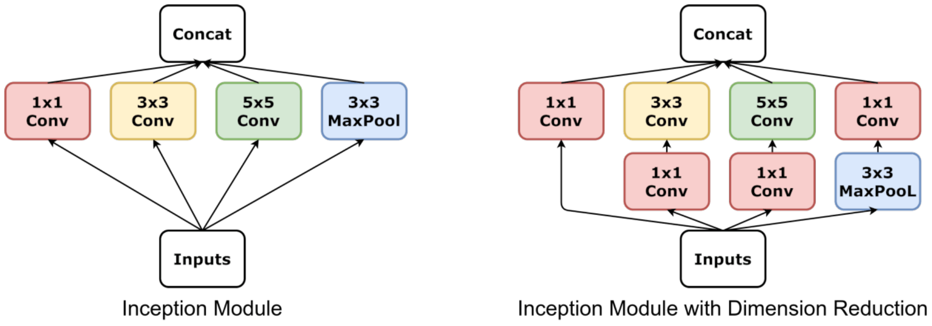


Figure 7.7 - Inception modules

In the regular version, the 1x1 convolution is **not** used as a dimension reduction layer. Each one of the **convolution branches** is producing a **given number of channels**, and the **max-pooling** branch produces **as many channels as it receives as input**. In the end, **all channels are stacked (concatenated) together**.

*"Honestly, I am a bit confused with **channels** and **filters**… are they the same or not?! How about **kernels**?"*

This is a bit confusing indeed… let's try to organize our thoughts about filters, kernels, and channels.

This is what we have seen so far:

1. every **filter** has as many **channels** as the **image** it is convolving (input)
2. each **channel** of the **filter/kernel** is a **small square matrix** (that is being convolved over the corresponding input channel)
3. a convolution produces as many **channels** as there are **filters** (output)

For example, we may have an **RGB image as input (three channels)** and use **two filters** (each having **three channels of its own** to match the input (1)) to **produce two channels** as output (3).

But things can get very messy if we start using "filter" to designate other things:

- **each channel of a filter/kernel** is often referred to as a **"*filter*"** itself
- **each channel of the output** produced by convolving the filter over the input image is often referred to as a **"*filter*"** too

In the example, we would then have **six "*filters*"** (instead of two filters with three channels each), and we would produce **two "*filters*"** as output (instead of two channels). This is confusing!

We're avoiding these messy definitions here, so we're using **channel concatenation (or even better, stacking)** instead of the confusing "*filter concatenation*".

Having cleared that out, we can return to the inception module itself. As we've seen, the **regular version** is simply **stacking the output channels of convolutions with different filter/kernel sizes**. What about the **other** version?

It also does that, but the inception module with **dimension reductions** uses **1x1 convolutions** to:

- **reduce the number of input channels** for both **3x3 and 5x5 convolution branches**
- **reduce the number of output channels** for the **max-pooling branch**

The 3x3 and 5x5 convolution branches **may still output many channels** (one for each filter), but **each filter is convolving a reduced number of input channels**.

You can think of the RGB to grayscale conversion: instead of using three-channel convolutions (as in Chapter 6) for colored images, it would use a single-channel filter (as in Chapter 5) for the grayscale (dimension reduced) image. For a **3x3 filter/kernel**, it means using **nine parameters instead of twenty-seven**. We can definitely go deeper!

Let's see how the inception module looks like in code:

```
class Inception(nn.Module):
    def __init__(self, in_channels):
        super(Inception, self).__init__()
        # in_channels@HxW -> 2@HxW
        self.branch1x1_1 = nn.Conv2d(in_channels,2,kernel_size=1)
        # in_channels@HxW -> 2@HxW -> 3@HxW
        self.branch5x5_1 = nn.Conv2d(in_channels,2,kernel_size=1) ①
        self.branch5x5_2 = nn.Conv2d(2, 3, kernel_size=5, padding=2)
        # in_channels@HxW -> 2@HxW -> 3@HxW
        self.branch3x3_1 = nn.Conv2d(in_channels,2,kernel_size=1) ①
        self.branch3x3_2 = nn.Conv2d(2, 3, kernel_size=3, padding=1)
        # in_channels@HxW -> in_channels@HxW -> 1@HxW
        self.branch_pool_1 = nn.AvgPool2d(
            kernel_size=3, stride=1, padding=1
        )
        self.branch_pool_2 = nn.Conv2d(                           ①
            in_channels, 2, kernel_size=1
        )

    def forward(self, x):
        # Produces 2 channels
        branch1x1 = self.branch1x1_1(x)
        # Produces 3 channels
        branch5x5 = self.branch5x5_1(x)                           ①
        branch5x5 = self.branch5x5_2(branch5x5)
        # Produces 3 channels
        branch3x3 = self.branch3x3_1(x)                           ①
        branch3x3 = self.branch3x3_2(branch3x3)
        # Produces 2 channels
        branch_pool = self.branch_pool_1(x)
        branch_pool = self.branch_pool_2(branch_pool)             ①
        # Concatenates all channels together (10)
        outputs = torch.cat([branch1x1, branch5x5,
                             branch3x3, branch_pool], 1)          ②
        return outputs
```

① Dimension reduction with 1x1 convolution

② Stacking/concatenating channels

The constructor method defines the seven elements used by four branches (you can identify each one of them in the figure above). In this example, I've configured all 1x1 convolutions to produce **two channels** each, but it is not required that they all have the same number of output channels. The same applies to both 3x3 and 5x5 convolution branches: although I've configured them both to produce the same number of channels (three) each, this is not a requirement.

❗ It **is** required, though, that **all branches** produce outputs with **matching height and width**. This means that the **padding** must be adjusted according to the kernel size in order to output the correct dimensions.

The forward method feeds the input `x` to each one of the four branches, and then it uses `torch.cat` to **concatenate the resulting channels** along the corresponding dimension (according to PyTorch's NCHW shape convention). This concatenation would fail if the height and width of the outputs did not match across the different branches.

What if we run our example image (scissors, in the color version) through the inception module?

```
inception = Inception(in_channels=3)
output = inception(image)
output.shape
```

Output

```
torch.Size([1, 10, 300, 300])
```

There we go, the output has the expected **10 channels**.

As you can see, the idea behind the inception module is actually quite simple. Later versions had slightly different architectures (like switching the 5x5 convolution by two 3x3 convolutions in a row, but the overall structure remains. There is **one more thing**, though…

If you check Inception's code in PyTorch, you'll find out that it **does not use `nn.Conv2d` directly**, but something called `BasicConv2d` instead (reproduced below):

```
class BasicConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, **kwargs):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(
            in_channels, out_channels, bias=False, **kwargs
        )
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)
```

Sure, its main component still is `nn.Conv2d`, but it also applies a ReLU activation function to the final output. More importantly, though, it calls `nn.BatchNorm2d` between the two.

"*What is that?*"

That is…

Batch Normalization

The batch normalization layer is a very important component of many modern architectures. Although its inner workings are not exactly complex (you'll see that in the following paragraphs), its impact on model training certainly is complex. From its placement (before or after an activation function) to the way its behavior is impacted by the mini-batch size, I try to briefly address the main discussion points in asides along the main text. This is meant to bring you up to speed on this topic, but it is by no means a comprehensive take on it.

We've briefly talked about the need for **normalization layers** in Chapter 4 to prevent (or mitigate) an issue commonly called "*internal covariate shift*", which is just *fancy* for **different distributions of activation values in different layers**. In general, we would like to have all layers producing **activation values with similar distributions**, ideally with **zero mean and unit standard deviation**.

Does it sound familiar? That's what we did with our **features** way back in Chapter 0, we **standardized** them.

Now, **batch normalization** will be doing something very similar to it, but with some important differences:

- instead of standardizing features, the inputs to the model as a whole, batch normalization **standardizes the activation values of a layer**, that is, the inputs to the following layer, so that they have **zero mean and unit standard deviation**
- instead of computing statistics (mean and standard deviation) for the whole training set, batch normalization **computes statistics for each mini-batch**
- batch normalization may perform an optional **affine transformation** to the standardized output, that is, scaling it and adding a constant to it (in this case, both scaling factor and constant are parameters learned by the model)

Before or After

There is a **very** common question regarding batch normalization:

"*Should I place the batch norm layer **before** or **after** the **activation function**?*"

From what I said above, that batch normalization standardizes the **activation values** of a layer, the only logical conclusion is that the **batch normalization layer comes AFTER the activation function**. It makes sense, right? Placing an activation function after normalizing would completely modify the inputs to the next layer and defeat the purpose of the batch normalization.

Or would it?

Some people argue that **it is OK to place the batch normalization layer BEFORE the activation function**. In fact, if you look at the Inception module, it is **exactly like that**. On the one hand, the outputs aren't going to have zero mean and unit standard deviation (a ReLU would chop the negative values off, for instance). On the other hand, the same should happen in every layer using batch normalization placed like that, so distributions are still similar across different layers.

So, there is no easy answer to this question.

For a **mini-batch of *n* data points**, given one particular **feature x**, batch normalization will first compute the statistics for that mini-batch:

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\sigma(X) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{X})^2}$$

Equation 7.2 - Mean and standard deviation

Then, it will use these statistics to **standardize each data point in the mini-batch**:

$$standardized\ x_i = \frac{x_i - \overline{X}}{\sigma(X) + \epsilon}$$

Equation 7.3 - Standardization

So far, this is **pretty much the same** as the standardization of features, except for the **epsilon** term added to the denominator to make it numerically stable (its typical value is `1e-5`).

Since the batch normalization layer is meant to produce a **zero mean** output, it makes the **`bias` in the layer that precedes it is totally redundant**. It would be a waste of computation to learn a bias that will be immediately removed by the following layer.

So, it is best practice to **set `bias=False` in the preceding layer** (you can check it out on the code for `BasicConv2d` in the previous section).

The actual difference is the **optional affine transformation** at the end:

$$batch\ normed\ x_i = b + w\ standardized\ x_i$$

Equation 7.4 - Batch normalization

If you choose **not to perform an affine transformation**, it will **automatically set parameters *b* and *w* to zero and one**, respectively. Although I've chosen the familiar *b* and *w* to represent these parameters, so it becomes even more clear there is nothing special to this transformation, you'll find them represented as *beta* and *gamma*, respectively, in the literature. Moreover, the terms may appear in a different order, like this:

$$batch\ normed\ x_i = standardized\ x_i\ \gamma + \beta$$

Equation 7.5 - Batch normalization with affine transformation

We're leaving the affine transformation aside, and focusing on a different aspect of the batch normalization: it does not only compute statistics for each mini-batch, but it also keeps track of…

Running Statistics

Since batch normalization computes statistics on mini-batches, and **mini-batches contain a small number of points**, these **statistics are likely to fluctuate a lot**. The smaller the mini-batches, the more the statistics will fluctuate. But, more importantly, **which statistics should it use for unseen data** (like the data points in the validation set)?

During the evaluation phase (or when the model is already trained and deployed), there are **no mini-batches**. It is perfectly natural to feed the model a **single input** to get its prediction. Clearly, there are no statistics for a single data point: it is its own mean, and the variance is zero. How can you standardize that? You can't! So, I repeat the question:

Affine Transformations and the Internal Covariate Shift (ICS)

If batch normalization was developed to mitigate the ICS (remember, this is just fancy for different distributions of activation values in different layers) by producing outputs with **zero mean** and **unit standard deviation**, how can an **affine transformation** possibly fit into this?

Well, in theory, **it can't**… if the normalization layer can learn any affine transformation, its outputs may have **any mean** and **any standard deviation**. So much for producing similar distributions across different layers to mitigate the internal covariate shift!

Nonetheless, the batch normalization layer in PyTorch **performs an affine transformation by default**.

If you look at the Inception module, it uses PyTorch's default. So, its batch norm layer not only performs an affine transformation but also is placed before the activation function. Clearly, this isn't mitigating ICS at all! But it is still successfully used in many model architectures, like Inception. How come?

Truth to be told, mitigating the ICS was the original motivation behind batch normalization, but it was later found that this technique actually **improves model training for a different reason**[124]. That's a plot twist!

It all boils down to **making the loss surface smoother**. We've actually already seen the effect of using standardization on the loss surface in Chapter 0: it became more bowl-shaped, thus making it easier for gradient descent to find the minimum. Can you imagine that in a thousand-dimension feature space? No?! Me neither! But hold on to this thought because we'll get back to it in the "*Residual Connections*" section.

"Which statistics should the model use when applying batch normalization to unseen data?"

What about keeping track of running statistics (that is, moving averages of the statistics)? It is a good way of **smoothing the fluctuations**. Besides, every data point will have a chance to contribute to the statistics. That's what batch normalization does.

Let's see it in action using code… we'll use a dummy dataset with 200 random data points and two features:

```
torch.manual_seed(23)
dummy_points = torch.randn((200, 2)) + torch.rand((200, 2)) * 2
dummy_labels = torch.randint(2, (200, 1))
dummy_dataset = TensorDataset(dummy_points, dummy_labels)
dummy_loader = DataLoader(
    dummy_dataset, batch_size=64, shuffle=True
)
```

A mini-batch of size 64 is small enough to have fluctuating statistics and big enough for plotting decent histograms.

Let's fetch three mini-batches, and plot histograms corresponding to each feature in the first mini-batch:

```
iterator = iter(dummy_loader)
batch1 = next(iterator)
batch2 = next(iterator)
batch3 = next(iterator)
```

Batch Normalization, Mini-Batch Size, and Regularization

It is said that batch normalization enforces a **lower limit** on **mini-batch size**.

The problem is the natural **fluctuation of the statistics**. As mentioned above, the smaller the mini-batches, the more the statistics will fluctuate. If they are **too small**, their statistics may **significantly diverge** from the **overall statistics for the whole training set**, thus **impacting negatively** the training of the model.

There is also the possibility of using **batch renormalization** (yes, that's a thing!) for those cases where it's impossible to have larger mini-batches (due to hardware constraints, for example) to prevent the issue above. This technique is beyond the scope of this book, though.

The flip side is that the **fluctuation of the statistics** is actually **injecting randomness** in the training process, thus having a **regularizing effect** and **impacting positively** the training of the model.

In Chapter 6, we've discussed another regularization procedure: dropout. Its way of injecting randomness was zeroing some of the inputs, such that its output would also vary slightly, or fluctuate.

Since both batch normalization and dropout layers have a regularizing effect, **combining both layers may actually harm the model performance**.

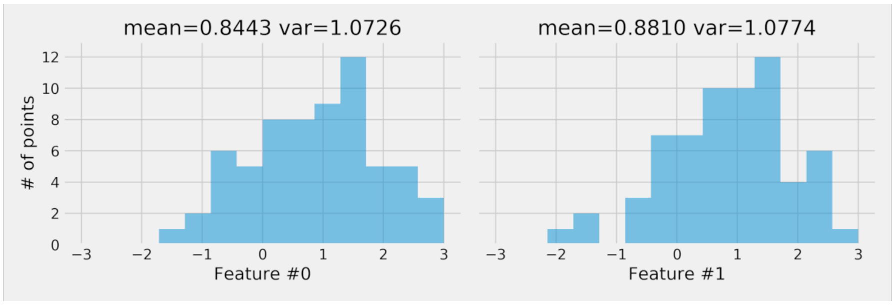


Figure 7.8 - Before Batch Normalization

```
mean1, var1 = batch1[0].mean(axis=0), batch1[0].var(axis=0)
mean1, var1
```

Output

```
(tensor([0.8443, 0.8810]), tensor([1.0726, 1.0774]))
```

These features can surely benefit from some standardization. We'll use `nn.BatchNorm1d` to accomplish it:

```
batch_normalizer = nn.BatchNorm1d(
    num_features=2, affine=False, momentum=None
)
batch_normalizer.state_dict()
```

Output

```
OrderedDict([('running_mean', tensor([0., 0.])),
             ('running_var', tensor([1., 1.])),
             ('num_batches_tracked', tensor(0))])
```

The `num_features` argument should match the dimension of the inputs. To keep matters simple, we **won't be using the affine transformation** (`affine=False`), and

neither the momentum (more on that later in this section).

The `state_dict` of the batch normalizer tells us the initial values for both running mean and variance, as well as the number of batches it has already used to compute the running statistics. Let's see what happens to them after we **normalize our first mini-batch**:

```
normed1 = batch_normalizer(batch1[0])
batch_normalizer.state_dict()
```

Output

```
OrderedDict([('running_mean', tensor([0.8443, 0.8810])),
             ('running_var', tensor([1.0726, 1.0774])),
             ('num_batches_tracked', tensor(1))])
```

Great, it matches the statistics we've computed before. The resulting values should be **standardized** by now, right? Let's double-check it:

```
normed1.mean(axis=0), normed1.var(axis=0)
```

Output

```
(tensor([ 3.3528e-08, -9.3132e-09]), tensor([1.0159, 1.0159]))
```

*"This looks a bit **off**... shouldn't the variance be **exactly** one?"*

Yes, and no. I confess I find this a bit annoying too... the **running variance** is **unbiased**, but the actual standardization of the data points of a mini-batch uses a **biased variance**.

"What's the difference between the two?"

The difference lies in the denominator only:

$$Biased\ Var(X) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{X})^2$$

$$Var(X) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{X})^2$$

Equation 7.6 - Biased variance

This was actually implemented like that by design. We're not discussing the reasoning here but, if you'd like to double-check the variance of the standardized mini-batch, you can use the following:

```
normed1.var(axis=0, unbiased=False)
```

Output

```
tensor([1.0000, 1.0000])
```

That's more like it! We can also plot the histograms once again to more easily visualize the effect of batch normalization:

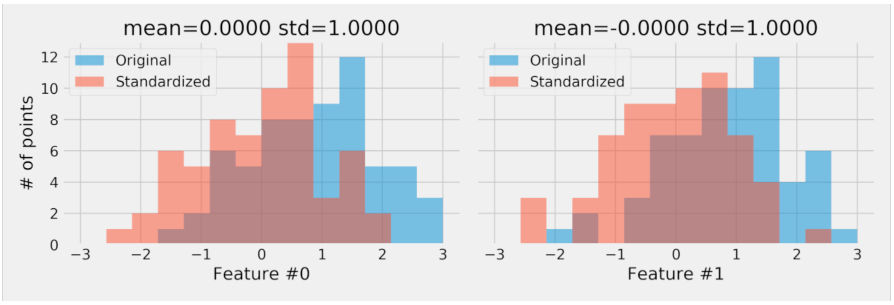


Figure 7.9 - After Batch Normalization

Event though batch normalization achieved an output with **zero mean and unit**

standard deviation, the overall distribution of the output is still mostly determined by the distribution of the inputs.

Batch normalization won't turn it into a **normal distribution**.

If we feed the second mini-batch to the batch normalizer, it will update its running statistics accordingly:

```
normed2 = batch_normalizer(batch2[0])
batch_normalizer.state_dict()
```

Output

```
OrderedDict([('running_mean', tensor([0.9070, 1.0931])),
             ('running_var', tensor([1.2592, 0.9192])),
             ('num_batches_tracked', tensor(2))])
```

Both running mean and running variance are simple averages over the mini-batches:

```
mean2, var2 = batch2[0].mean(axis=0), batch2[0].var(axis=0)
running_mean, running_var = (mean1 + mean2) / 2, (var1 + var2) / 2
running_mean, running_var
```

Output

```
(tensor([0.9070, 1.0931]), tensor([1.2592, 0.9192]))
```

Now, let's pretend we have **finished training** (even though we don't have an actual model), and we're using the third mini-batch for **evaluation**.

Evaluation Phase

Just like dropout, batch normalization also exhibits **different behaviors** depending on the **mode**: `train()` or `eval()`. We've already seen what it does during the training phase. We've also realized that it **doesn't make sense to compute statistics** for any data that isn't training data.

So, in the **evaluation phase**, it will use the **running statistics** computed during training to **standardize** the new data (the third mini-batch, in our small example):

```
batch_normalizer.eval()
normed3 = batch_normalizer(batch3[0])
normed3.mean(axis=0), normed3.var(axis=0, unbiased=False)
```

Output

```
(tensor([ 0.1590, -0.0970]), tensor([1.0134, 1.4166]))
```

"Is it a bit off again?"

Actually, no… since it is **standardizing unseen data** using statistics computed on training data, the results above are expected. The **mean will be around zero** and the **standard deviation will be around one**.

Momentum

There is an alternative way of computing running statistics: instead of using a simple average, it uses an **exponentially weighted moving average** of the statistics.

The naming convention, though, is **very unfortunate**: the **alpha parameter** of the EWMA was named **momentum**, adding to the confusion. There is even a note in PyTorch's documentation warning about this:

> "*This* `momentum` *argument is different from one used in optimizer classes and the conventional notion of momentum.*"[125]

The bottom line is: ignore the confusing naming convention and think of the "*momentum*" argument as the **alpha** parameter of a regular EWMA.

The documentation also uses **x** to refer to a particular statistic when introducing the mathematical formula of the "*momentum*", which does not help at all.

So, to make it abundantly clear what is being computed, I present you the formulas below:

$$\begin{array}{rcccc} EWMA_t(\alpha, x) = & \alpha x_t & +(1-\alpha) & EWMA_{t-1}(\alpha, x) \\ running\ stat_t = & "momentum"\ stat_t & +(1-"momentum") & running\ stat_{t-1} \end{array}$$

Equation 7.7 - Running statistic

Let's try it out in practice:

```
batch_normalizer_mom = nn.BatchNorm1d(
    num_features=2, affine=False, momentum=0.1
)
batch_normalizer_mom.state_dict()
```

Output

```
OrderedDict([('running_mean', tensor([0., 0.])),
             ('running_var', tensor([1., 1.])),
             ('num_batches_tracked', tensor(0))])
```

Initial values are zero and one, respectively, for running mean and running variance. These will be the running statistics at time `t-1`. What happens if we run the first mini-batch through it?

```
normed1_mom = batch_normalizer_mom(batch1[0])
batch_normalizer_mom.state_dict()
```

Output

```
OrderedDict([('running_mean', tensor([0.0844, 0.0881])),
             ('running_var', tensor([1.0073, 1.0077])),
             ('num_batches_tracked', tensor(1))])
```

The running statistics barely budged since the mini-batch statistics were multiplied by the "*momentum*" argument. We can easily verify the results for the running means:

```
running_mean = torch.zeros((1, 2))
running_mean = 0.1 * batch1[0].mean(axis=0) + \
               (1 - 0.1) * running_mean
running_mean
```

Output

```
tensor([[0.0844, 0.0881]])
```

"*Very well, but we have only used* `BatchNorm1d` *so far, and the inception module actually used* `BatchNorm2d`..."

Glad you brought that up!

BatchNorm2d

The difference between the one-dimension and the **two-dimensions batch normalization** is actually quite simple: the former standardized features (columns), while the latter **standardizes channels (pixels)**.

This is easier to see in code:

```
torch.manual_seed(39)
dummy_images = torch.rand((200, 3, 10, 10))
dummy_labels = torch.randint(2, (200, 1))
dummy_dataset = TensorDataset(dummy_images, dummy_labels)
dummy_loader = DataLoader(
    dummy_dataset, batch_size=64, shuffle=True
)

iterator = iter(dummy_loader)
batch1 = next(iterator)
batch1[0].shape
```

Output

```
torch.Size([64, 3, 10, 10])
```

The code above creates a dummy dataset of 200 colored (three-channel) images of size 10x10 pixels and then retrieves the first mini-batch. The mini-batch has the expected NCHW shape.

The **batch normalization is done over the C dimension**, so it will compute statistics using the remaining dimensions: N, H, and W (`axis=[0, 2, 3]`), representing **all pixels of a given channel from every image in the mini-batch**.

The `nn.BatchNorm2d` layer has the same arguments as its one-dimension counterpart, but its **`num_features` argument** must match the **number of channels of the input** instead:

```
batch_normalizer = nn.BatchNorm2d(
    num_features=3, affine=False, momentum=None
)
normed1 = batch_normalizer(batch1[0])
print(normed1.mean(axis=[0, 2, 3]),
      normed1.var(axis=[0, 2, 3], unbiased=False))
```

Output

```
(tensor([ 2.3171e-08,  3.4217e-08, -2.9616e-09]),
 tensor([0.9999, 0.9999, 0.9999]))
```

As expected, **each channel** in the output has its **pixel values with zero mean and unit standard deviation**.

Other Normalizations

Batch normalization is certainly the most popular kind of normalization, but not the only one. If you check PyTorch's documentation on Normalization Layers, you'll see many alternatives, like the `SyncBatchNorm`, for instance. But, just like the batch renormalization technique, they are beyond the scope of this book.

Small Summary

This was probably the **most challenging section** in this book so far. At the same time, it goes over a *lot* of information and only scratches the surface of this topic. So, I am organizing a small summary of the main points we've addressed:

- during **training** time, it **computes statistics** (mean and variance) for each individual **mini-batch** and uses these statistics to produce **standardized outputs**
- the **fluctuations in the statistics**, from one mini-batch to the next, introduce **randomness** into the process thus having a **regularizing effect**

- due to the regularizing effect of batch normalization, it **may not work well if combined with other regularization techniques** (like dropout)
- during **evaluation** time, it uses a **(smoothed) average of the statistic**s computed during training
- its original motivation was addressing the so-called "*internal covariate shift*" by producing similar distributions across different layers, but it was later found that it actually **improves model training by making the loss surface smoother**
- the batch normalization **may be placed either before or after the activation function**, there is no "*right*" or "*wrong*" way
- the **layer preceding the batch normalization layer should have its `bias=False` set** to avoid useless computation
- even though batch normalization works for a different reason than initially thought, addressing the "*internal covariate shift*" may still bring benefits, like solving the **vanishing gradients problem**, one of the topics of the next chapter.

So, we've learned that **batch normalization** speeds up training by making the **loss surface smoother**. It turns out, there is yet another technique that works along these lines…

Residual Connections

The idea of a **residual connection** is quite simple, actually: after passing the input through a layer and activation function, the **input itself is added to the result**. That's it! Simple, elegant, and effective.

"*Why would I want to add the input to the result?*"

Learning the Identity

Neural networks and their nonlinearities (activation functions) are great! We've seen in the "*Bonus*" chapter how models manage to **twist and turn** the feature space so much so that **classes can be separated by a straight line** in the transformed feature space. But **nonlinearities** are both **a blessing and a curse**: they make it extremely hard for a model to learn the **identity function**.

To illustrate this, let's start with a dummy dataset containing 100 random data points with a single feature. But this feature isn't simply a feature, it is also the label. The data preparation is fairly straightforward:

```
torch.manual_seed(23)
dummy_points = torch.randn((100, 1))
dummy_dataset = TensorDataset(dummy_points, dummy_points)
dummy_loader = DataLoader(
    dummy_dataset, batch_size=16, shuffle=True
)
```

If we were using a simple **linear model**, that would be a **no-brainer**, right? The model would just **keep the input as it is** (multiplying it by one - the weight- and adding zero to it - the bias). But what happens if we **introduce a nonlinearity**? Let's configure the model and train it to see what happens:

```
class Dummy(nn.Module):
    def __init__(self):
        super(Dummy, self).__init__()
        self.linear = nn.Linear(1, 1)
        self.activation = nn.ReLU()

    def forward(self, x):
        out = self.linear(x)
        out = self.activation(out)
        return out
```

```
torch.manual_seed(555)
dummy_model = Dummy()
dummy_loss_fn = nn.MSELoss()
dummy_optimizer = optim.SGD(dummy_model.parameters(), lr=0.1)
```

```
dummy_sbs = StepByStep(dummy_model, dummy_loss_fn, dummy_optimizer)
dummy_sbs.set_loaders(dummy_loader)
dummy_sbs.train(200)
```

If we compare the actual labels with the model's predictions, we'll see that it **failed to learn the identity function**:

```
np.concatenate([dummy_points[:5].numpy(),
                dummy_sbs.predict(dummy_points)[:5]], axis=1)
```

Output

```
array([[-0.9012059 ,  0.        ],
       [ 0.56559485,  0.56559485],
       [-0.48822638,  0.        ],
       [ 0.75069577,  0.75069577],
       [ 0.58925384,  0.58925384], dtype=float32)
```

No surprises here, right? Since the ReLU can only return positive values, it will never be able to produce the points with negative values.

"Wait, that doesn't look right… where is the ***output*** *layer?"*

OK, you caught me… I suppressed the output layer on purpose to make a point here. Please bear with me a little bit longer while I add a **residual connection** to the model:

```
class DummyResidual(nn.Module):
    def __init__(self):
        super(DummyResidual, self).__init__()
        self.linear = nn.Linear(1, 1)
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x                    ①
        out = self.linear(x)
        out = self.activation(out)
        out += identity                 ①
        return out
```

① Adding the output to the result

Guess what happens if we replace the `Dummy` model with the `DummyResidual` model and retrain it?

```
np.concatenate([dummy_points[:5].numpy(),
                dummy_sbs.predict(dummy_points)[:5]], axis=1)
```

Output

```
array([[-0.9012059 , -0.9012059 ],
       [ 0.56559485,  0.56559485],
       [-0.48822638, -0.48822638],
       [ 0.75069577,  0.75069577],
       [ 0.58925384,  0.58925384]], dtype=float32)
```

It looks like the model actually learned the identity function… or did it? Let's check its parameters:

```
dummy_model.state_dict()
```

Output

```
OrderedDict([('linear.weight', tensor([[0.1488]], device='cuda:0')),
             ('linear.bias', tensor([-0.3326], device='cuda:0'))])
```

For an input value equals to zero, the output of the linear layer will be `-0.3326` which, in turn, will be chopped off by the ReLU activation. Now I have a question for you:

"*Which input values produce outputs* ***bigger than zero****?*"

The answer: input values above `2.2352 (=0.3326/0.1488)` will produce positive outputs which, in turn, will get past by the ReLU activation. But I have another question for you:

"*Guess what's the* ***highest*** *input value in our dataset?*"

Close enough! I am assuming you answered `2.2352`, but it is just a little bit less than that:

```
dummy_points.max()
```

Output

```
tensor(2.2347)
```

"*So what? Does it actually* ***mean*** *anything?*"

It means the **model learned to stay out of the way** of the inputs! Now that the model has the ability to use the raw inputs directly, its linear layer learned to produce only negative values, so its nonlinearity doesn't interfere with the outputs. Cool, right?

The Power of Shortcuts

The residual connection works as a **shortcut**, enabling the model to **skip** the nonlinearities when it pays off to do so (if it yields a smaller loss). For this reason, residual connections are also known as **skip connections**.

"I'm still not convinced… what's the big deal about this?"

The big deal is, these shortcuts **make the loss surface smoother**, so gradient descent has an easier job finding a minimum. Don't take my word for it, go and check the beautiful loss landscape visualizations produced by Li et al. in their paper *"Visualizing the Loss Landscape of Neural Nets"*[126].

Awesome, right? These are projections of a multi-dimensional loss surface for the ResNet model, with and without skip connections. Guess which one is easier to train? :-)

If you're curious to see more landscapes like these, make sure to check their website: Visualizing the Loss Landscape of Neural Nets[127].

Another advantage of these **shortcuts** is that they provide a **shorter path for the gradients** to travel back to the initial layers, thus preventing the **vanishing gradients** problem.

Residual Blocks

We're finally ready to tackle the main component of the ResNet model (the top performer of ILSVRC-2015), the residual block:

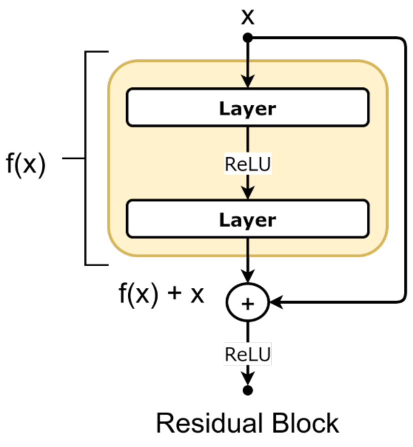


Figure 7.10 - Residual Block

The residual block isn't so different from our own `DummyResidual` model, except for the fact that the residual block has **two consecutive weight layers** and a **ReLU activation at the end**. Moreover, it may have **more than two consecutive weight layers**, and the weight layers do **not necessarily** need to be **linear**.

For image classification, it makes much more sense to use **convolutional layers** instead, right? Right! And why not throw some **batch normalization** layers in the mix? Sure! The residual block looks like this now:

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels, out_channels,
            kernel_size=3, padding=1, stride=stride,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(
            out_channels, out_channels,
            kernel_size=3, padding=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.downsample = None
        if out_channels != in_channels:
            self.downsample = nn.Conv2d(
                in_channels, out_channels,
                kernel_size=1, stride=stride
            )

    def forward(self, x):
        identity = x
        # First "weight layer" + activation
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        # Second "weight layer"
        out = self.conv2(out)
        out = self.bn2(out)
        # What is that?!
        if self.downsample is not None:
            identity = self.downsample(identity)
        # Adding inputs before activation
        out += identity
        out = self.relu(out)

        return out
```

It should be pretty clear, except for one small detail: it **may** be necessary to **downsample** the input.

"Why is that?"

To **add up two images**, they **must have the same dimensions**, not only the height and the width, but also the number of channels (adding up is *not* the same as stacking up channels!). That poses a problem for the residual block, since the **number of output channels** of the last **convolutional layer** may be **different than the number of channels in the input**.

If only there was an operation that took the original input and generated an **output with a different number of channels**…. do you know any?

"*What about a **convolutional layer**?*"

Bingo! We can use **yet another convolutional layer** to produce an input (now modified) that has a **matching number of channels**, so it can be added to the main output.

"*But then it is **not** the original input anymore, is it?*"

Not really, no, because it will be modified by the downsampling convolutional layer. But, even though it goes somewhat against the idea of learning the identity function, the idea of a **shortcut** still stands.

Finally, to illustrate the effect of the skip connection on an image, I've passed one of the images from the "*Rock, Paper, Scissors*" dataset through a randomly initialized residual block (three channels in and out, no downsampling), with and without a skip connection. These are the results:

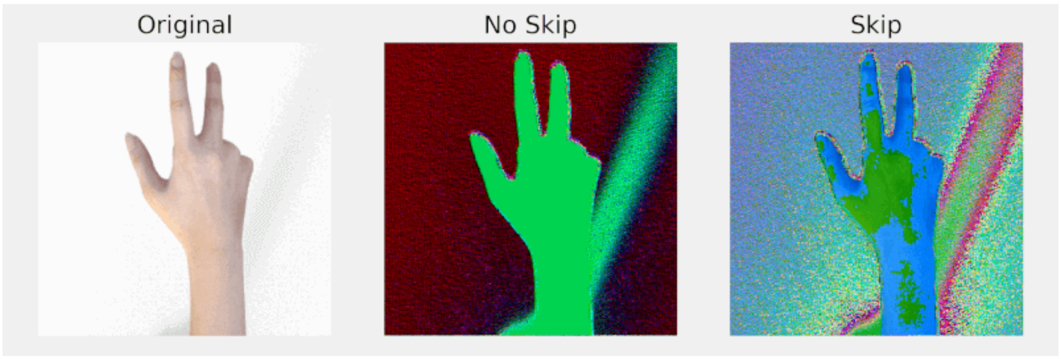


Figure 7.11 - Skip connection in action

On the one hand (that's a good pun, c'mon!), if there are no skip connections some information may be lost, like the different shades on the back of the hand. On the other hand (sorry!), skip connections **may** help to preserve that information.

That's the general idea behind the ResNet model. Of course, the whole architecture is more complex than that, stacking many different residual blocks, and adding some more bells and whistles to the mix. We're not going into any more details here, but the pre-trained models can be easily used for transfer learning, just like we did with the AlexNet model.

Putting It All Together

In this chapter we've gone through the necessary steps to use **transfer learning** with **pre-trained models** for computer vision tasks: using **ImageNet statistics** for pre-processing the inputs, **freezing layers** (or not), **replacing the top layer**, and optionally speeding up training by **generating features and training the top of the model independently**.

Data Preparation

```
# ImageNet statistics
normalizer = Normalize(mean=[0.485, 0.456, 0.406],
                       std=[0.229, 0.224, 0.225])

composer = Compose([Resize(256),
                    CenterCrop(224),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
val_loader = DataLoader(val_data, batch_size=16)
```

This time, we'll use the smallest version of the ResNet model (resnet18) and either fine-tune it or use it as a feature extractor only.

Fine-Tuning

Model Configuration (1)

```
model = resnet18(pretrained=True)
torch.manual_seed(42)
model.fc = nn.Linear(512, 3)
```

There is **no freezing** since fine-tuning entails the training of all the weights, not only those from the top layer.

Model Configuration (2)

```
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_model = optim.Adam(model.parameters(), lr=3e-4)
```

Model Training

```
sbs_transfer = StepByStep(model, multi_loss_fn, optimizer_model)
sbs_transfer.set_loaders(train_loader, val_loader)
sbs_transfer.train(1)
```

Let's see what the model can accomplish after training for a single epoch:

Evaluation

```
StepByStep.loader_apply(val_loader, sbs_transfer.correct)
```

Output

```
tensor([[124, 124],
        [124, 124],
        [124, 124]])
```

Perfect score!

If we **had frozen the layers** in the model above, it would have been a case of **feature extraction suitable for data augmentation** since we would be training the top layer while attached to the rest of the model.

Feature Extraction

In the model that follows, we're **modifying the model** (replacing the top layer with an **identity** layer) to generate a **dataset of features** first and then using it to train the real top layer independently.

Model Configuration (1)

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = resnet18(pretrained=True).to(device)
model.fc = nn.Identity()
freeze_model(model)
```

Data Preparation - Preprocessing

```
train_preproc = preprocessed_dataset(model, train_loader)
val_preproc = preprocessed_dataset(model, val_loader)
train_preproc_loader = DataLoader(
    train_preproc, batch_size=16, shuffle=True
)
val_preproc_loader = DataLoader(val_preproc, batch_size=16)
```

Once the dataset of features and its corresponding loaders are ready, we only need to **create a model corresponding to the top layer** and train it in the usual way:

Model Configuration - Top Model

```
torch.manual_seed(42)
top_model = nn.Sequential(nn.Linear(512, 3))
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_top = optim.Adam(top_model.parameters(), lr=3e-4)
```

Model Training - Top Model

```
sbs_top = StepByStep(top_model, multi_loss_fn, optimizer_top)
sbs_top.set_loaders(train_preproc_loader, val_preproc_loader)
sbs_top.train(10)
```

We surely can evaluate the model as it is using the same data loaders (containing pre-processed features):

Evaluation - Top Model

```
StepByStep.loader_apply(val_preproc_loader, sbs_top.correct)
```

Output

```
tensor([[ 98, 124],
        [124, 124],
        [104, 124]])
```

But, if we want to **try it out on the original dataset** (containing the images), we need to **attach the top layer back**:

Replacing Top Layer

```
model.fc = top_model
sbs_temp = StepByStep(model, None, None)
```

We can still create a separate instance of `StepByStep` for the full model to be able to call its `predict` or `correct` methods (in this case, both loss function and optimizers are set to `None` since we won't be training the model anymore):

Evaluation

```
StepByStep.loader_apply(val_loader, sbs_temp.correct)
```

Output

```
tensor([[ 98, 124],
        [124, 124],
        [104, 124]])
```

We got the same results, as expected.

Recap

In this chapter, we've learned about the **ImageNet** Large Scale Visual Recognition Challenge (ILSVRC) and the many model architectures developed to tackle it (AlexNet, VGG, Inception, and ResNet). We used its pre-trained weights to perform transfer learning and either fine-tune or extract features for our own classification task instead. Moreover, we took a quick tour of the inner workings of many architectural elements built-in these models. This is what we've covered:

- learning about **transfer learning** (last pun in this chapter, I promise!)
- learning about **ImageNet**, ILSVRC, and the most popular architectures develop to tackle it

- **comparing** the size, speed, and performance of these architectures
- loading the AlexNet model
- loading the model's **pre-trained weights**
- **freezing the layers** of the model
- **replacing the "*top*" layer** of the model
- understanding the difference between **fine-tuning** and **feature extraction**
- using **ImageNet statistics** to pre-process the images
- generating a **dataset of features** using the frozen model
- training an independent model and **attaching it** to the original model
- understanding the role of **auxiliary classifiers** in very deep architectures
- building a **loss function** that handles auxiliary classifiers too
- training the **top layer of an Inception V3** model
- using **1x1 convolutions** as a **dimension reduction layer**
- building an **inception module**
- understanding what a **batch normalization** layer does
- discussing where to **place** the batch normalization layer, before or after an activation function
- understanding the impact of **mini-batch size** on batch normalization statistics
- understanding the **regularizer effect** of batch normalization
- observing the **effect of `train` and `eval` modes** in batch normalization layers
- understanding what a **residual/skip connection** is
- understanding the **effect of skip connections** on the **loss surface**
- building a **residual block**
- **fine-tuning** and **extracting features** using a ResNet18 model

Congratulations! You have just finished the fourth and last chapter of Part II (not counting the "*Extra*" chapter)! You are now familiar with the **most important tools and techniques** for handling **computer vision** problems. Although there will always be a lot to learn since the field is very dynamic and new techniques are being constantly developed, I believe that having a **good grasp on how these building blocks work** should make it easier for you to further explore and keep learning on your own.

In the next part, we'll shift our focus to **sequences** and a whole new class of models: **recurrent neural networks** and their variants.

[102] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter07.ipynb

[103] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter07.ipynb

[104] http://wordnet.princeton.edu

[105] http://www.image-net.org/

[106] http://image-net.org/about-stats

[107] http://www.image-net.org/challenges/LSVRC/

[108] http://image-net.org/challenges/LSVRC/2012/

[109] http://image-net.org/challenges/LSVRC/2012/results

[110] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[111] http://image-net.org/challenges/LSVRC/2014/

[112] http://image-net.org/challenges/LSVRC/2014/results

[113] https://arxiv.org/abs/1409.1556

[114] http://image-net.org/challenges/LSVRC/2014/results

[115] https://arxiv.org/abs/1409.4842

[116] http://image-net.org/challenges/LSVRC/2015/

[117] http://image-net.org/challenges/LSVRC/2015/results

[118] https://arxiv.org/abs/1512.03385

[119] https://github.com/albanie/convnet-burden

[120] https://arxiv.org/abs/1605.07678

[121] http://alexlenail.me/NN-SVG/

[122] https://arxiv.org/abs/1512.00567v3

[123] https://en.wikipedia.org/wiki/Grayscale

[124] https://arxiv.org/abs/1805.11604

[125] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html

[126] https://arxiv.org/abs/1712.09913

[127] https://www.cs.umd.edu/~tomg/projects/landscapes/

Extra Chapter
Vanishing and Exploding Gradients

Spoilers

In this chapter, we will:

- tackle the **vanishing gradients** problem using **initialization schemes**
- understand the effect of **batch normalization** on the vanishing gradients
- tackle the **exploding gradients** problem using **gradient clipping**
- **clip gradients** in different ways: element-wise, using its norm, and using hooks
- understand the difference between clipping gradients **after** or **during** backpropagation

Jupyter Notebook

The Jupyter notebook corresponding to Chapter Extra[128] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[129].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter Extra's notebook. If not, just click on `ChapterExtra.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import torch
import torch.optim as optim
import torch.nn as nn
from sklearn.datasets import make_regression

from torch.utils.data import DataLoader, TensorDataset
from stepbystep.v3 import StepByStep

from data_generation.ball import load_data
```

Vanishing and Exploding Gradients

In this *extra* chapter, we're discussing **gradients** once again. The gradients, together with the learning rate, are what makes the model **tick**, better yet, **learn**. We've discussed both of them in quite some detail in Chapter 6, but we always assumed that the gradients were *well-behaved*, as long as our learning rate was sensible. Unfortunately, this is not necessarily true, and sometimes the **gradients may go awry**: they can either **vanish** or **explode**. Either way, we need to rein them in, so let's see how we can accomplish that.

Vanishing Gradients

Do you remember how can we tell PyTorch to compute gradients for us? It starts with the **loss value**, followed by a call to the `backward` method which works its way **back** up to the first layer. That's backpropagation in a nutshell. It works fine for models with a few hidden layers, but as models grow deeper, the **gradients** computed for the weights in the **initial layers** become **smaller and smaller**. That's the so-called **vanishing gradients** problem, and it has always been a major obstacle for training deeper models.

"Why is it so bad?"

If gradients vanish, that is, if they are close to **zero**, **updating the weights** will **barely change them**. In other words, the **model is not learning anything**, it got **stuck**.

"Why does it happen?"

We can blame it on the (in)famous "*internal covariate shift*". But, instead of discussing it, let me illustrate it.

Ball Dataset and Block Model

Let’s use a dataset of **1,000 random points** drawn from a **ten-dimensional ball** (this seems *fancier* than it actually is, you can think of it as a dataset with 1,000 points with ten features each) such that each feature has **zero mean** and **unit standard deviation**. In this dataset, points situated *within half of the radius* of the ball are labeled as *negative cases*, while the remaining points are labeled *positive cases*. It is a familiar binary classification task.

Data Generation

```
X, y = load_data(n_points=1000, n_dims=10)
```

Next, we can use these data points to create a dataset and a data loader (no mini-batches this time):

Data Preparation

```
ball_dataset = TensorDataset(
    torch.as_tensor(X).float(), torch.as_tensor(y).float()
)
ball_loader = DataLoader(ball_dataset, batch_size=len(X))
```

The data preparation part is done. What about the model configuration? To illustrate the vanishing gradients problem, we need a **deeper model** than the ones we've built so far. Let's call it the **"block"** model: it is a block of **several hidden layers** (and activations function) stacked together, every layer containing the same number of hidden units (neurons).

Instead of building the model manually, I've created a function that allows us to configure a model like that. Its main arguments are the number of features, the number of layers, the number of hidden units per layer, the activation function to be placed after each hidden layer, and if it should add a batch normalization layer after every activation function or not:

Model Building

```
torch.manual_seed(11)
n_features = X.shape[1]
n_layers = 5
hidden_units = 100
activation_fn = nn.ReLU
model = build_model(
    n_features, n_layers, hidden_units, activation_fn, use_bn=False
)
```

Let's check the model out:

```
print(model)
```

Output

```
Sequential(
  (h1): Linear(in_features=10, out_features=100, bias=True)
  (a1): ReLU()
  (h2): Linear(in_features=100, out_features=100, bias=True)
  (a2): ReLU()
  (h3): Linear(in_features=100, out_features=100, bias=True)
  (a3): ReLU()
  (h4): Linear(in_features=100, out_features=100, bias=True)
  (a4): ReLU()
  (h5): Linear(in_features=100, out_features=100, bias=True)
  (a5): ReLU()
  (o): Linear(in_features=100, out_features=1, bias=True)
)
```

Exactly as expected! The layers are labeled sequentially, from one up to the number of layers, and have prefixes according to their roles: `h` for linear layers, `a` for activation functions, `bn` for batch normalization layers, and `o` for the last (output) layer.

We're only missing a loss function and an optimizer, and we're done with the model configuration part too:

Model Configuration

```
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=1e-2)
```

Weights, Activations, and Gradients

To visualize what's happening with the weights, the activation values, and the gradients, we need to **capture** these values first. Luckily, we already have the appropriate methods for these tasks: `capture_parameters`, `capture_gradients`, and `attach_hooks`, respectively. We only need to create an instance of our

`StepByStep` class, configure these methods to capture these values for the corresponding layers, and train it for a single epoch:

Model Training

```
hidden_layers = [f'h{i}' for i in range(1, n_layers + 1)]
activation_layers = [f'a{i}' for i in range(1, n_layers + 1)]

sbs = StepByStep(model, loss_fn, optimizer)
sbs.set_loaders(ball_loader)
sbs.capture_parameters(hidden_layers)
sbs.capture_gradients(hidden_layers)
sbs.attach_hooks(activation_layers)
sbs.train(1)
```

Since we're not using mini-batches this time, training the model for one epoch will use **all data points** to:

- perform one **forward pass**, thus capturing the **initial weights** and generating **activation values**
- perform one **backward pass**, thus computing the **gradients**

To make matters even easier, I've also created a function that takes a **data loader** and a **model** as arguments and returns the captured values after training it for a single epoch. This way, you can **experiment** with different models too!

```
parms, gradients, activations = get_plot_data(
    train_loader=ball_loader, model=model
)
```

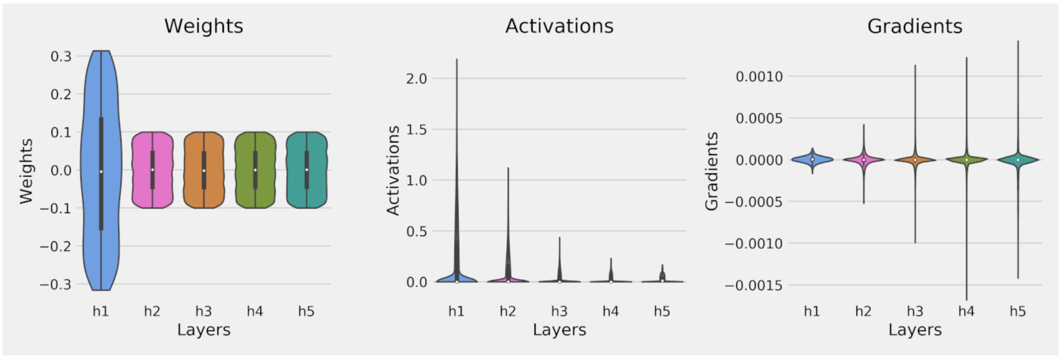


Figure E.1 - Vanishing gradients

On the left-most plot, we can see that the initial weights in each layer are uniformly distributed, but the first hidden layer has a much wider range. This is a consequence of the default initialization scheme used by PyTorch's linear layer, but we're not delving into these details here.

The **activation** values are clearly **shrinking** as data moves from one layer to the next. Conversely, the **gradients** are **larger** in the **last layer**, and **shrink** as the gradient descent algorithm works its way back to the first layer. That's a simple and straightforward example of **vanishing gradients**.

Gradients can also be **exploding** instead of vanishing. In this case, activation values grow larger and larger as data moves from one layer to the next, and gradients are smaller in the last layer, growing as we move back up to the first layer.

This phenomenon is less common, though, and can be more easily handled using a technique called **gradient clipping** that simply caps the absolute value of the gradients. We'll get back to that later.

"*How can we prevent the vanishing gradients problem?*"

If we manage to get the **distribution of activation values similar across all layers**, we may have a shot at it. But, to achieve that, we need to **tweak the variance of the**

weights. If done properly, the **initial distribution of the weights** may lead to a more **consistent distribution** of activation values across layers.

If you haven't noticed already, keeping **similar distributions of activation values across all layers** is exactly what **batch normalization** was doing.

So, if you're using batch normalization, vanishing gradients are likely not an issue. But, before batch normalization layers became a thing, there was another way of tackling the problem, which is the topic of the next section.

Initialization Schemes

An initialization scheme is a **clever way of tweaking the initial distribution** of the weights. It is all about choosing the **best standard deviation** to use for drawing random weights from a normal or uniform distribution. In this section, we'll briefly discuss two of the most traditional schemes, Xavier (Glorot) and Kaiming (He), and how to manually initialize weights in PyTorch. For a more detailed explanation of the inner workings of these initialization schemes, please check my post: Hyper-parameters in Action! Part II — Weight Initializers[130].

The Xavier (Glorot) initialization scheme was developed by Xavier Glorot and Yoshua Bengio and it is meant to be used with the **hyperbolic-tangent (Tanh) activation function**. It is referred to as either **Xavier** or **Glorot** initialization, depending on the context. In PyTorch, it is available both as `nn.init.xavier_uniform` and `nn.init.xavier_normal`.

The Kaiming (He) initialization scheme was developed by Kaiming He (yes, the same guy from the ResNet architecture) et al. and it is meant to be used with the **rectified linear unit (ReLU) activation function**. It is referred to as either **Kaiming** or **He** initialization, depending on the context. In PyTorch, it is available both as `nn.init.kaiming_uniform` and `nn.init.kaiming_normal`.

*"Should I use **uniform** or **normal** distribution?"*

It shouldn't make much of a difference, but using the **uniform** distribution usually delivers slightly better results than the alternative.

*"Do I **have to** manually initialize the weights?"*

Not necessarily, no. If you're using **transfer learning**, for instance, this is pretty much **not an issue** because most of the model would be already trained, and a bad initialization of the trainable part should have little to no impact on model training. Besides, as we'll see in a short while, using **batch normalization** layers makes your model much more **forgiving when it comes to a bad initialization** of the weights.

"What about PyTorch's defaults? Can't I simply trust them?"

Trust, but verify. Each PyTorch layer has its own default initialization of the weights in the `reset_parameters` method. For instance, the `Linear` layer is initialized using Kaiming (He) scheme drawn from a uniform distribution:

```
# nn.Linear.reset_parameters()
def reset_parameters(self) -> None:
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in)
        init.uniform_(self.bias, -bound, bound)
```

Moreover, it also initializes the biases based on the "*fan-in*", which is simply the number of units in the preceding layer.

IMPORTANT: Every default initialization has its own assumptions and, in this particular case, it is assumed (in the `reset_parameters` method) that the `Linear` layer will be followed by a **leaky ReLU** (the default value for the `nonlinearity` argument in the Kaiming initialization) with a **negative slope** equals the **square root of five** (the "a" argument in the Kaiming initialization).

If your model does not follow these assumptions, you may run into problems. For instance, our model used a regular ReLU instead of a leaky one, so the default initialization scheme was off and we ended up with vanishing gradients.

"*How am I supposed to know that?*"

Unfortunately, there is no easy way around it. You may inspect a layer's `reset_parameters` method and figure its assumptions from the code (like we just did) or, if you are **training a deeper model from scratch**, it is probably best to **initialize the layers manually**, so you have total control over the process.

Don't worry **too** much about initialization schemes just now! This is a somewhat advanced topic already, but I thought it was worth introducing it after going over batch normalization. As I mentioned before, you're likely using transfer learning with deeper models anyway.

"*What if I **really** want to try initializing the weights myself, how can I do it?*"

Let's go over a simple example. Let's say you'd like to initialize all linear layers using the Kaiming uniform scheme with the proper nonlinearity function for the weights and setting all the biases to zeros. You'll have to **build a function that takes a layer as its argument**, like that:

Weight Initialization

```
def weights_init(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight, nonlinearity='relu')
        if m.bias is not None:
            nn.init.zeros_(m.bias)
```

The function may set both `weight` and `bias` attributes of the layer passed as argument. Notice that both methods from `nn.init` used to initialize the attributes have **an underscore** at the end, so they are making changes **in-place**.

To actually **apply** the initialization scheme to your model, we can simply call the `apply` method of your model, and it will recursively apply the initialization function to all its internal layers:

```
with torch.no_grad():
    model.apply(weights_init)
```

You should also use the `no_grad` context manager while initializing/modifying the weights and biases of your model.

To illustrate the effect of a proper initialization, I've plotted activation values and gradients for three different configurations of the "block" model: sigmoid activation with normal initialization, Tanh activation with Xavier uniform initialization, and ReLU activation with Kaiming uniform initialization:

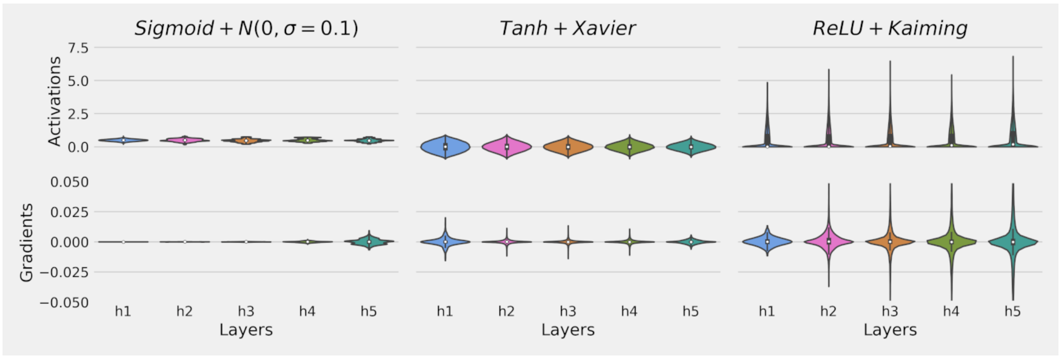


Figure E.2 - The effect of initializations

Well, using a sigmoid function as an activation function in deep models is just hopeless. But, for the other two, it should be clear that the **correct initialization of the weights** resulted in more **stable distributions** of activation values and gradients **across all the layers**.

Even though initialization schemes are definitely a clever approach to the vanishing gradients problem, their usefulness *vanishes* (pun intended!) when batch normalization layers are added to the model.

Batch Normalization

Since batch normalization layers are supposed to produce similar distributions of activation values (and gradients) across the layers, we have to ask ourselves:

"*Can we get away with a bad initialization?*"

Yes, we can! Let's compare activation values and gradients for yet another three different configurations of the "block" model: ReLU activation and normal initialization, ReLU activation and Kaiming uniform initialization, and ReLU activation and normal initialization **followed by batch normalization layer**:

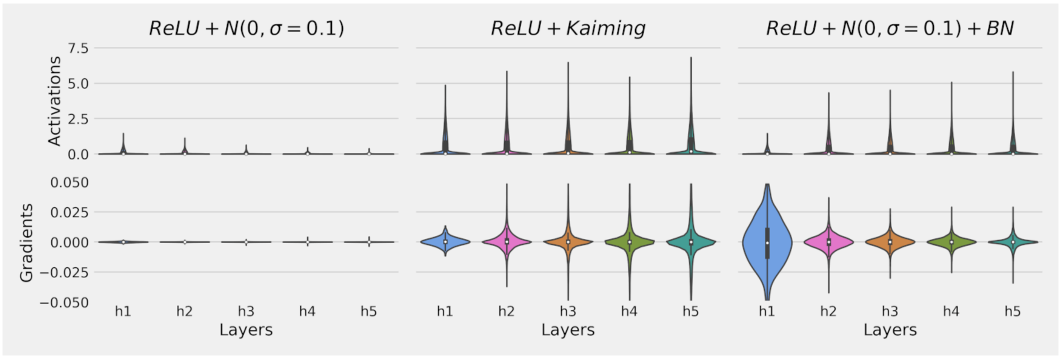


Figure E.3 - The effect of batch normalization

The left-most plot shows us the result of a bad initialization scheme: vanished gradients. The center plot shows us the result of a proper initialization scheme. Finally, the right-most plot shows us that **batch normalization can indeed compensate for a bad initialization**.

Not all *bad* gradients *vanish*, though… some *bad* gradients **explode**!

Exploding Gradients

The *root* of the problem is the same: deeper and deeper models. If the model has a couple of layers only, one **large gradient** won't do any harm. But, if there are *many layers*, the gradients may end up **growing uncontrollably**. That's the so-called **exploding gradients** problem and it's fairly easy to spot: just look for `NaN` values in the **loss**. If that's the case, it means that the gradients grew *so large* that they **cannot be properly represented anymore**.

"*Why does it happen?*"

There may be a *compounding effect* (think of raising a relatively small number (e.g. 1.5) to a large power (e.g. 20)), especially in **recurrent neural networks** (the topic of Chapter 8) since the **same weights** are **used repeatedly** along a sequence. But, there are *other* reasons as well: the **learning rate** may be **too high**, or the **target variable** (in a regression problem) may have a **large range of values**. Let me illustrate it.

Data Generation & Preparation

Let's use Scikit-Learn's `make_regression` to generate a dataset of **1,000 points** with **ten features** each, and a little bit of noise:

Data Generation & Preparation

```
X_reg, y_reg = make_regression(
    n_samples=1000, n_features=10, noise=0.1, random_state=42
)
X_reg = torch.as_tensor(X_reg).float()
y_reg = torch.as_tensor(y_reg).float().view(-1, 1)

dataset = TensorDataset(X_reg, y_reg)
train_loader = DataLoader(
    dataset=dataset, batch_size=32, shuffle=True
)
```

Even though we cannot *plot* a 10-dimensional regression, we can *still* visualize the **distribution** of both **features** and **target** values:

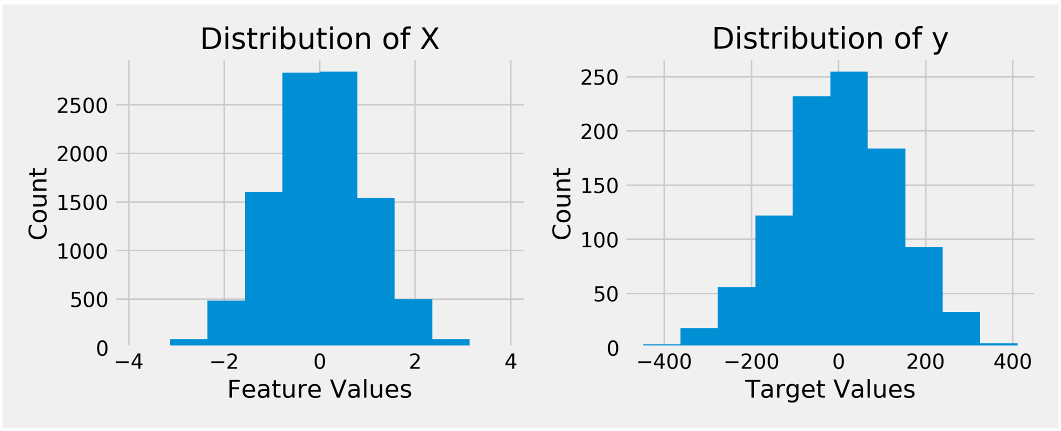


Figure E.4 - Distributions of feature and target values

It's all good and fine with our **feature values** since they are inside a *typical standardized range* (-3, 3). The **target values**, though, are on a very different scale, from -400 to 400. If the target variable represents a *monetary value*, for example, these ranges are fairly common. Sure, we could *standardize* the target value as well, but that would *ruin* the example of **exploding gradients** :-)

Model Configuration & Training

We can build a fairly simple model to tackle this regression problem: a network with **one hidden layer** with 15 units, a ReLU as activation function, and an output layer:

```
torch.manual_seed(11)
model = nn.Sequential()
model.add_module('fc1', nn.Linear(10, 15))
model.add_module('act', nn.ReLU())
model.add_module('fc2', nn.Linear(15, 1))
optimizer = optim.SGD(model.parameters(), lr=0.01)
loss_fn = nn.MSELoss()
```

Before training it, let's set our instance of the `StepByStep` class to **capture the gradients** for the weights in the **hidden layer** (`fc1`):

```
sbs_reg = StepByStep(model, loss_fn, optimizer)
sbs_reg.set_loaders(train_loader)
sbs_reg.capture_gradients(['fc1'])
sbs_reg.train(2)
```

It turns out, **two epochs** are already enough to get **exploding gradients**. Yay! Well, maybe not "yay", but you know what I mean, right?

Let's look at the losses:

```
sbs_reg.losses
```

Output

```
[16985.014358520508, nan]
```

Bingo! There is the NaN value we were looking for. The phenomenon is not called *exploding losses*, but *exploding gradients*, so let's look for NaNs there too. Since there are 150 weights in the hidden layer, and 32 mini-batches per epoch (resulting in 64 gradient computations over two epochs), it's easier to look at the **average gradient** used for updating the parameters:

```
grads = np.array(sbs_reg._gradients['fc1']['weight'])
print(grads.mean(axis=(1, 2)))
```

Output

```
[ 1.58988627e+00 -2.41313894e+00  1.61042006e+00  4.27530414e+00
  2.00876453e+01 -5.46269826e+01  4.76936617e+01 -6.68976169e+01
  4.89202255e+00 -5.48839445e+00 -8.80165010e+00  2.42120121e+01
 -1.95470126e+01 -5.61713082e+00  4.16399702e+01  2.09703794e-01
  9.78054642e+00  8.47080885e+00 -4.37233462e+01 -1.22754592e+01
 -1.05804357e+01  6.17669332e+00 -3.27032627e+00  3.43037068e+01
  6.90878444e+00  1.15130024e+01  8.64732616e+00 -3.04457552e+01
 -3.79791490e+01  1.57137556e+01  1.43945687e+01  8.90063342e-01
 -3.60141261e-01  9.18566430e+00 -7.91019879e+00  1.92959307e+00
 -6.55456380e+00 -1.66785977e+00 -4.13915831e+01  2.03403218e+01
 -5.39869087e+02 -2.33201361e+09  3.74779743e+26             nan
             nan             nan             nan             nan
             nan             nan             nan             nan
             nan             nan             nan             nan
             nan             nan             nan             nan
             nan             nan             nan             nan]
```

The first NaN shows up at the 44th update, but the **explosion** started at the 41st update: the average gradient goes from hundreds (1e+02) to **billions** (1e+09) in one step, to **gazillions** (1e+26) or whatever this is called in the next, to a full-blown NaN.

"How do we fix it?"

On the one hand, we *could* standardize the target value or try a smaller learning rate (like 0.001). On the other hand, we can simply **clip the gradients**.

Gradient Clipping

Gradient clipping is quite simple: you **pick a value** and **clip gradients** higher (in absolute terms) than the value you picked. That's it. Actually, there is one more small detail: you can pick a value for **each and every gradient** or a value for the **norm of all gradients**. To illustrate both mechanisms, let's randomly generate some **parameters**:

```
torch.manual_seed(42)
parm = nn.Parameter(torch.randn(2, 1))
fake_grads = torch.tensor([[2.5], [.8]])
```

We're also generating the **fake gradients** above so we can *manually* set them as if they were the computed gradients of our random parameters. We'll use these gradients to illustrate two different ways of *clipping* them.

Value Clipping

This is the most straightforward way: it *clips* gradients element-wise so they stay inside the [-clip_value, +clip_value] range. We can use PyTorch's nn.utils.clip_grad_value_ to **clip gradients in-place**:

```
parm.grad = fake_grads.clone()
#Gradient Value Clipping
nn.utils.clip_grad_value_(parm, clip_value=1.0)
parm.grad.view(-1,)
```

Output

```
tensor([1.0000, 0.8000])
```

The first gradient got *clipped*, the other one kept its original value. It doesn't get any simpler than that.

Now, pause for a moment, and think of the gradients above as the **steps** gradient descent is taking along **two different dimensions** to navigate the **loss surface** towards (some) minimum value. What happens if we **clip** some of these steps? We're actually **changing** directions on our path towards the minimum. The figure below illustrates both vectors, original and clipped:

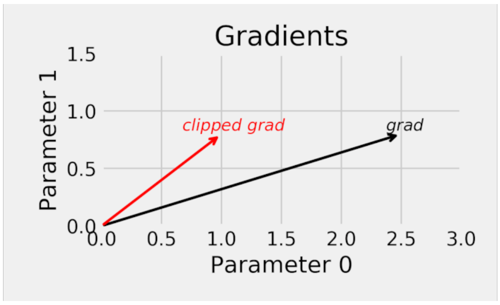


Figure E.5 - Gradients: before and after clipping by value

By **clipping values**, we're modifying the gradients in such a way that, not only the **step is smaller**, but it is in a **different direction**. Is this a problem? No, not necessarily. Can we avoid changing directions? Yes, we can, that's what **norm clipping** is good for.

Backward Hooks

As we've seen in Chapter 6, the register_hook method registers a **backward hook** to a **tensor for a given parameter**. The **hook function** takes a **gradient as input** and it may return **a modified, or *clipped*, gradient**. The hook function will be called every time a gradient with respect to that tensor is computed, that is, it can **clip gradients during backpropagation**, unlike the other methods.

The code below attaches hooks to all parameters of the model, thus performing **gradient clipping** on the fly:

```python
def clip_backprop(model, clip_value):
    handles = []
    for p in model.parameters():
        if p.requires_grad:
            func = lambda grad: torch.clamp(grad,
                                            -clip_value,
                                            clip_value)
            handle = p.register_hook(func)
            handles.append(handle)
    return handles
```

Do not forget that you should **remove the hooks** using `handle.remove()` after you're done with them.

Norm Clipping (or Gradient Scaling)

While value clipping was an element-wise operation, **norm clipping** computes the **norm for all gradients together** as if they were concatenated into a single vector. If (and only *if*) the **norm exceeds the clipping value**, the **gradients are scaled down** to match the desired norm, otherwise they keep their values. We can use PyTorch's nn.utils.clip_grad_norm_ to **scale gradients in-place**:

```
parm.grad = fake_grads.clone()
# Gradient Norm Clipping
nn.utils.clip_grad_norm_(parm, max_norm=1.0, norm_type=2)
fake_grads.norm(), parm.grad.view(-1,), parm.grad.norm()
```

Output

```
(tensor(2.6249), tensor([0.9524, 0.3048]), tensor(1.0000))
```

The norm of our fake gradients *was* `2.6249`, and we're **clipping the norm** at `1.0`, so the gradients get scaled by a factor of `0.3810`.

Clipping the norm **preserves the direction** of the gradient vector:

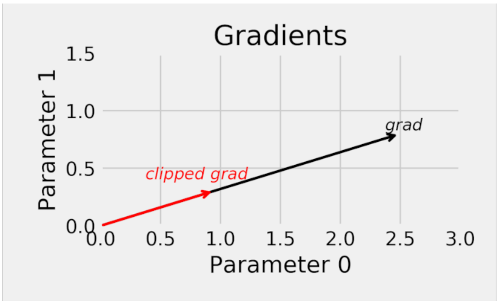


Figure E.6 - Gradients: before and after clipping by norm

"A couple of questions… first, which one is better?"

On the one hand, **norm clipping** maintains the **balance** between the updates of all parameters since it's only **scaling the norm** and **preserving the direction**. On the other hand, **value clipping** is faster, and the fact that it *slightly changes* the direction of the gradient vector does not seem to have any harmful impact on performance. So, you're probably OK using **value clipping**.

"Second, which clip value should I use?"

That’s *trickier* to answer… the **clip value** is a hyper-parameter that can be *fine-tuned* like any other. You can start with a value like ten, and work your way down if the gradients keep exploding.

*"Finally, how do I **actually** do it in practice?"*

Glad you asked! We’re creating some more methods in our `StepByStep` class to handle both kinds of clipping, and modifying the `_make_train_step` method to account for them. Gradient clipping must happen **after gradients are computed** (`loss.backward()`), and **before updating the parameters** (`optimizer.step()`):

StepByStep Method

```python
setattr(StepByStep, 'clipping', None)

def set_clip_grad_value(self, clip_value):
    self.clipping = lambda: nn.utils.clip_grad_value_(
        self.model.parameters(), clip_value=clip_value
    )

def set_clip_grad_norm(self, max_norm, norm_type=2):
    self.clipping = lambda: nn.utils.clip_grad_norm_(
        self.model.parameters(), max_norm, norm_type
    )

def remove_clip(self):
    self.clipping = None

def _make_train_step(self):
    # This method does not need ARGS... it can refer to
    # the attributes: self.model, self.loss_fn and self.optimizer

    # Builds function that performs a step in the train loop
    def perform_train_step(x, y):
        # Sets model to TRAIN mode
        self.model.train()
        # Step 1 - Computes model's predicted output - forward pass
        yhat = self.model(x)
        # Step 2 - Computes the loss
        loss = self.loss_fn(yhat, y)
        # Step 3 - Computes gradients
        loss.backward()

        if callable(self.clipping):       ①
            self.clipping()               ①

        # Step 4 - Updates parameters
        self.optimizer.step()
        self.optimizer.zero_grad()

        # Returns the loss
        return loss.item()

    # Returns the function that will be called inside the train loop
    return perform_train_step

setattr(StepByStep, 'set_clip_grad_value', set_clip_grad_value)
setattr(StepByStep, 'set_clip_grad_norm', set_clip_grad_norm)
setattr(StepByStep, 'remove_clip', remove_clip)
setattr(StepByStep, '_make_train_step', _make_train_step)
```

① Gradient clipping after computing gradients and before updating parameters

The gradient clipping methods above work just fine for most models, but they are of *little use* for **recurrent neural networks** (we'll discuss them in Chapter 8), which require gradients to be clipped **during backpropagation**. Fortunately, we can use the **backward hooks** code to accomplish that:

StepByStep Method

```
def set_clip_backprop(self, clip_value):
    if self.clipping is None:
        self.clipping = []
    for p in self.model.parameters():
        if p.requires_grad:
          func = lambda grad: torch.clamp(grad,
                                          -clip_value,
                                          clip_value)
          handle = p.register_hook(func)
          self.clipping.append(handle)

def remove_clip(self):
    if isinstance(self.clipping, list):
        for handle in self.clipping:
            handle.remove()
    self.clipping = None

setattr(StepByStep, 'set_clip_backprop', set_clip_backprop)
setattr(StepByStep, 'remove_clip', remove_clip)
```

The method above will attach hooks to all parameters of the model and perform gradient clipping on the fly. We also adjusted the `remove_clip` method to remove any handles associated with the hooks.

Model Configuration & Training

Let's use the method below to **initialize the weights** so we can **reset the parameters** of our model that had its gradients exploded:

```
def weights_init(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight, nonlinearity='relu')
        if m.bias is not None:
            nn.init.zeros_(m.bias)
```

Moreover, let's use a **ten times larger learning rate**, after all, we're in full control of the gradients now:

```
torch.manual_seed(42)
with torch.no_grad():
    model.apply(weights_init)

optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Before training it, let's use `clip_grad_value` to make sure no gradients are ever above `1.0`:

```
sbs_reg_clip = StepByStep(model, loss_fn, optimizer)
sbs_reg_clip.set_loaders(train_loader)
sbs_reg_clip.set_clip_grad_value(1.0)
sbs_reg_clip.capture_gradients(['fc1'])
sbs_reg_clip.train(10)
sbs_reg_clip.remove_clip()
sbs_reg_clip.remove_hooks()
```

```
fig = sbs_reg_clip.plot_losses()
```

No more exploding gradients, it seems. The loss is being minimized even after choosing a much larger learning rate to train the model.

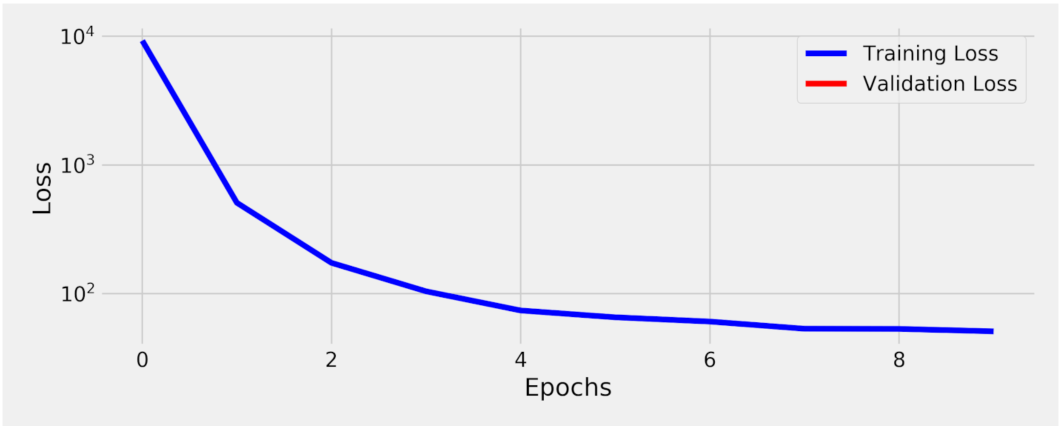


Figure E.7 - Losses - clipping by value

What about taking a look at the **average gradients** once again (there 320 updates now, so we're looking at the extremes only):

```
avg_grad = np.array(sbs_reg_clip._gradients['fc1']['weight'])
.mean(axis=(1, 2))
avg_grad.min(), avg_grad.max()
```

Output

```
(-24.69288555463155, 14.385948762893676)
```

"How come these (absolute) values are much larger than our clipping value?"

These are the *computed gradients*, that is, **before clipping**. Left unchecked, these gradients would have caused **large updates** which, in turn, would have resulted in **even larger gradients**, and so on and so forth. Explosion, basically. But these values were all *clipped* before being used in the parameter update, so all went well with the model training.

It is possible to take a *more aggressive approach* and **clip** the gradients at the origin using the **backward hooks** we discussed before.

Clipping with Hooks

First, we reset the parameters once again:

```
torch.manual_seed(42)
with torch.no_grad():
    model.apply(weights_init)
```

Now, we use `set_clip_backprop` to clip the gradients **during backpropagation** using hooks:

```
sbs_reg_clip_hook = StepByStep(model, loss_fn, optimizer)
sbs_reg_clip_hook.set_loaders(train_loader)
sbs_reg_clip_hook.set_clip_backprop(1.0)
sbs_reg_clip_hook.capture_gradients(['fc1'])
sbs_reg_clip_hook.train(10)
sbs_reg_clip_hook.remove_clip()
sbs_reg_clip_hook.remove_hooks()
```

```
fig = sbs_reg_clip_hook.plot_losses()
```

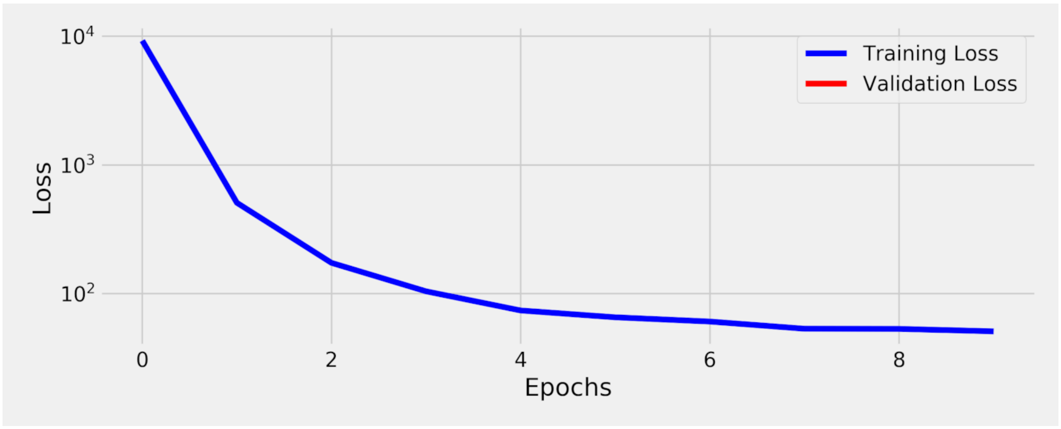


Figure E.8 - Losses - clipping by value with hooks

The loss is, once again, well-behaved. At first sight, there doesn't seem to be *any difference*…

Or does it? Let's compare the distributions of the *computed gradients* over the whole training loop for both methods:

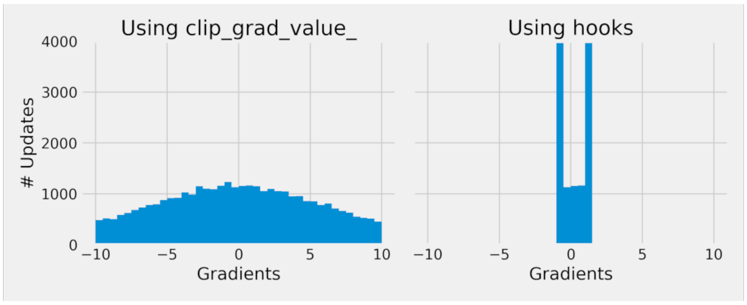


Figure E.9 - Distributions of gradients during training

Well, **that's** a big difference! On the left plot, the gradients were computed as usual and only clipped before the parameter update to prevent the compounding effect that led to the explosion of the gradients. On the right plot, **no gradients** are ever **above the clip value** (in absolute terms).

Keep in mind that, even though the choice of clipping method does not seem to have an impact on the overall loss of our simple model, this **won't** hold true for **recurrent neural networks**, and you **should use hooks for clipping gradients** in that case.

Recap

This *extra* chapter is much shorter than the others, and its purpose was to illustrate some simple techniques to take back control of **gradients gone wild**. Therefore, we're skipping the "*Putting It All Together*" section this time. We used two simple datasets, together with two simple models, to show the signs of both **vanishing** and **exploding** gradients. The former was addressed with different **initialization schemes** and, optionally, **batch normalization**, while the latter was addressed by **clipping the gradients** in different ways. This is what we've covered:

- visualizing the **vanishing gradients** problem in deeper models
- using a function to **initialize the weights** of a model
- visualizing the **effect of initialization schemes** on the gradients
- realizing that **batch normalization** can compensate for **bad initializations**
- understanding the **exploding gradients** problem
- using **gradient clipping** to address the exploding gradients problem
- visualizing the difference between **value clipping** and **norm clipping**
- using **backward hooks** to perform gradient clipping **during backpropagation**
- visualizing the difference between clipping **after** and **during** backpropagation

After this small detour into *gradient land*, that's the end of Part II, for real this time.

[128] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/ChapterExtra.ipynb

[129] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/ChapterExtra.ipynb

[130] https://bit.ly/3eCfJ4h

Part III
Sequences

Chapter 8
Sequences

Spoilers

In this chapter, we will:

- learn about the characteristics of **sequential data** and generate our own
- understand the **inner workings** of **recurrent layers**
- build and train models to perform **classification** of **sequences**
- understand the importance of the **hidden state** as the **representation of a sequence**
- visualize the **journey of a hidden state** from beginning to end of a sequence
- preprocess **variable-length sequences** using **padding** and **packing** techniques, and the **collate function**
- learn how **1D convolutions** can be used on sequential data

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 8[131] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[132].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 8's notebook. If not, just click on `Chapter08.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, random_split, \
    TensorDataset
from torch.nn.utils import rnn as rnn_utils

from data_generation.square_sequences import generate_sequences
from stepbystep.v4 import StepByStep
```

Sequences

In this third part of the book, we'll dive into a new kind of input: **sequences**! So far, each data point was considered in and on itself, that is, each data point had a label to call its own. An image of a hand was classified as "*rock*", "*paper*", or "*scissors*" based on its pixel values alone, without paying any attention to other images' pixel values. This **won't** be the case anymore.

In sequence problems, an **ordered sequence of data points shares a single label** - emphasis on **being ordered**.

*"Why is **ordered** so important?"*

If the data points **aren't ordered**, even if they share a single label, they are **not a sequence**, but a **collection** of data points.

Let's think of a slightly contrived example: greyscale images with *shuffled* pixels. Each pixel has a single value, but **a pixel alone doesn't have a label**. It is the **collection of shuffled pixels**, the shuffled image, that **has a label**: a duck, a dog, or a cat (labeled before shuffling the pixels, of course).

Before shuffling, the pixels were **ordered**, that is, they *had an underlying two-dimensional structure*. This structure can be exploited by the convolutional neural networks: the kernel moving around the image looks at a pixel in the center and all its neighbors in both dimensions, height, and width.

If the underlying structure has a **single dimension**, though, that's a **sequence**. This particular structure can be exploited by **recurrent neural networks** and their many variants, as well as **1D convolutional neural networks**, and they constitute the subject of this chapter.

There are two main types of **sequence problems**: **time series** and **natural language processing (NLP)**. We'll start by generating a synthetic dataset and use it to illustrate the **inner workings** of **recurrent neural networks**, **encoder-decoder** models, **attention mechanisms**, and even **transformers**! Only then we'll get to the natural language processing part.

I chose to follow this *sequence* of topics (yeah, another pun!) because I find it much easier to develop intuition (and to produce meaningful visualizations) while working with a two-dimensional dataset, as opposed to 100-dimensional word embeddings.

Data Generation

Our data points are **two-dimensional**, so they can be visualized as an image, and **ordered**, so they are a **sequence**. We'll be **drawing squares**! Each square, as depicted in the figure below, has **four corners** (duh!), each corner being assigned a **letter** and a **color**. The bottom-left corner is **A** and **gray**, the bottom-right one is **D** and **red**, and so on.

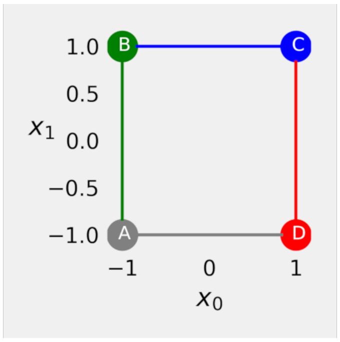


Figure 8.1 - Our colored square

The **coordinates** (x_0, x_1) of the four corners are our **data points**. The "*perfect*" square has the coordinates depicted in the figure above: `A=(-1, -1)`, `B=(-1, 1)`, `C=(1, 1)`, and `D=(1,-1)`. Sure enough, we'll generate a dataset full of **noisy squares**, each having its corners *around* these perfect coordinates.

Now, we need to give each **sequence of data points (corners)** a **label**: assuming you can draw a square **starting at any corner**, and draw it **without lifting your pencil** at any time, you can choose to draw it **clockwise** or **counterclockwise**. These are our labels:

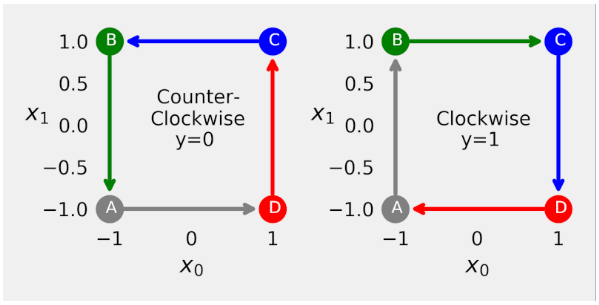


Figure 8.2 - Drawing directions

Since there are four corners to start from, and there are two directions to follow, there are effectively eight possible sequences:

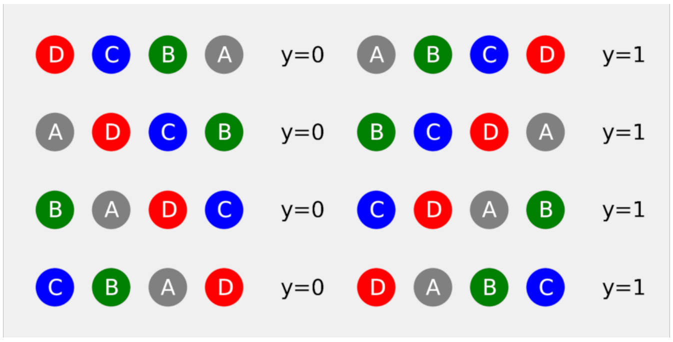


Figure 8.3 - Possible sequences of corners

Our task is to **classify the sequence** of corners: **is it drawn in a clockwise direction**? A familiar binary classification problem!

Let's generate 128 random noisy squares:

Data Generation

```
points, directions = generate_sequences(n=128, seed=13)
```

And then let's visualize the first ten squares:

```
fig = plot_data(points, directions)
```

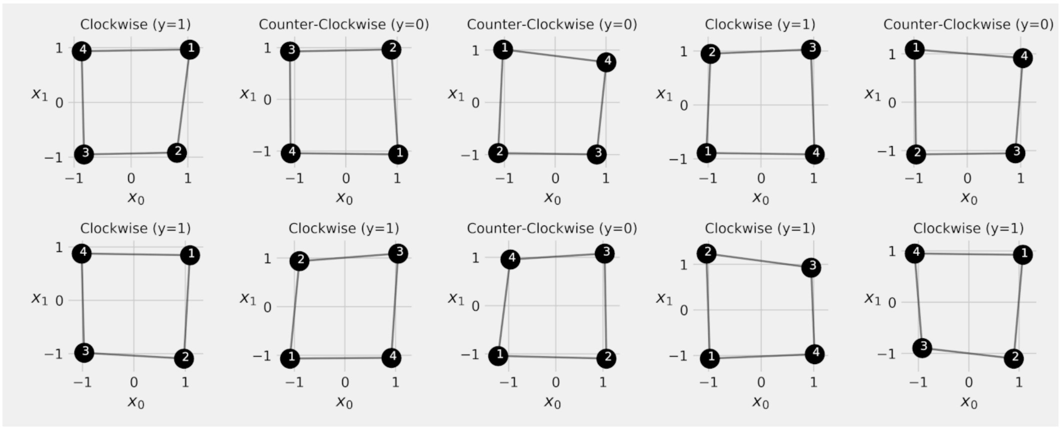


Figure 8.4 - Sequence dataset

The corners show the **order** in which they were drawn. In the first square, the drawing **started at the top-right corner** (corresponding to the *blue C* corner) and followed a **clockwise direction** (corresponding to the *CDAB* sequence).

In the next chapter, we'll use **the first two corners** to predict the **other two**, so the model will need to learn **not only the direction but also the coordinates**. We'll build a **sequence-to-sequence** model which uses one sequence to predict another.

For now, we're sticking to **classifying the direction**, given all the **four data points** of a given square. But, first, we need to introduce…

Recurrent Neural Networks (RNNs)

Recurrent neural networks are perfectly suited for **sequence** problems since they take advantage of the **underlying structure** of the data, namely, the **order of the data points.** We'll see in great detail **how the data points**, sequentially presented to a recurrent neural network, **modify the RNNs internal (hidden) state**, which will ultimately be a **representation of the full sequence**.

SPOILER ALERT: Recurrent neural networks are all about **producing a hidden state that best represents a sequence**.

"*But what **is** a hidden state anyway?*"

Excellent question! A hidden state is simply a **vector**. The size of the vector is up to you. Really. You need to specify the number of **hidden dimensions**, which means specifying the size of the vector that represents a hidden state. Let's create a **two-dimensional hidden state** just for kicks:

```
hidden_state = torch.zeros(2)
hidden_state
```

Output

```
tensor([0., 0.])
```

That's actually a fine example of an **initial hidden state**, as we'll see shortly. But, before diving deep into the journey of a hidden state through an RNN, let's take a look at its top-level representation:

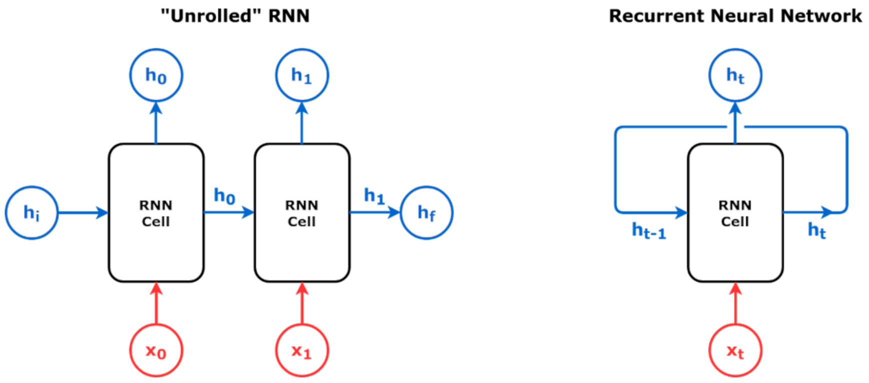


Figure 8.5 - Top-level representation of an RNN

If you've ever seen figures representing RNNs, you probably bumped into one of the two versions depicted above: "*unrolled*" or not. Let's start with the **unrolled** one: it shows a **sequence of two data points** being fed to an RNN. We can describe the flow of information inside an RNN in **five easy steps**:

1. there is an **initial hidden state** (h_i) that represents the **state of the empty sequence** and is usually **initialized with zeros** (like the one we created above)
2. an RNN cell takes **two inputs**: the **hidden state** representing the state of the sequence *so far*, and **a data point from the sequence** (like the coordinates of one of the corners from a given square)
3. the two inputs are used to **produce a new hidden state** (h_0 for the first data point), representing the **updated state of the sequence** now that a new point was presented to it
4. the new hidden state is **both** the **output of the current step** and **one of the**

inputs of the next step

5. if there is **yet another data point in the sequence**, it goes back to **Step #2**; if not, the **last hidden state** (*h_1* in the figure above) is also the **final hidden state** (*h_f*) of the whole RNN

Since the **final hidden state** is a representation of the **full sequence**, that's what we're going to use as **features** for our **classifier**.

In a way, that's not so different from the way we used CNNs: there, we'd run the pixels through multiple convolutional blocks (convolutional layer + pooling + activation) and flatten them into a vector at the end to use them as features for a classifier.

Here, we run a sequence of data points through RNN cells and use the final hidden state (also a vector) as features for a classifier.

There is a **fundamental difference** between CNNs and RNNs, though: while there are several **different convolutional layers**, each learning its own filters, the **RNN cell is one and the same**. In this sense, the "*unrolled*" representation is misleading: it definitely *looks like* each input is being fed to a different RNN cell, but that's *not* the case.

There is only **one cell**, which will learn a particular set of weights and biases, and which will **transform the inputs exactly the same way in every step of the sequence**. Don't worry if this doesn't completely make sense to you just yet, I promise it will become more clear soon, especially in the "*Journey of a Hidden State*" section.

RNN Cell

Let's take a look at some of the *internals* of an RNN cell:

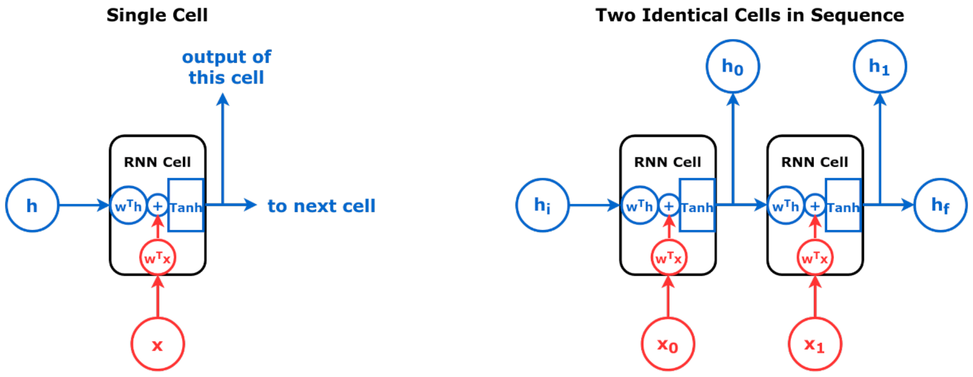


Figure 8.6 - Internals of an RNN cell

On the left, we have a **single RNN cell**. It has **three main components**:

- a **linear layer** to transform the **hidden state** (in blue)
- a **linear layer** to transform the **data point from the sequence** (in red)
- an **activation function**, usually the hyperbolic tangent (Tanh), which is applied to the **sum of both transformed inputs**

We can also represent them as equations:

$$
\begin{array}{lcccccc}
RNN: & t_h & = & W_{hh} & h_{t-1} & + & b_{hh} \\
 & t_x & = & W_{ih} & x_t & + & b_{ih} \\
 & h_t & = & tanh & (t_h & + & t_x)
\end{array}
$$

Equation 8.1 - RNN

I chose to split the equation into smaller, colored, parts to highlight the fact that these are simple linear layers producing both a **transformed hidden state** (t_h) and a **transformed data point** (t_x). The updated hidden (h_t) state is both the output of this particular cell and one of the inputs of the "*next*" cell.

But there is **no other cell**, really, it is just the **same cell over and over again**, as depicted on the right side of the figure above. So, in the second step of the sequence, the updated hidden state will run through the **very same linear layer** the initial hidden state ran through. The same goes for the second data point. Considering this, the **not "*unrolled*"** ("*rolled*" doesn't sound right!) representation is a better characterization of the internal structure of an RNN.

Let's dive **deeper** into the internals of an RNN cell and look it at the **neuron level**:

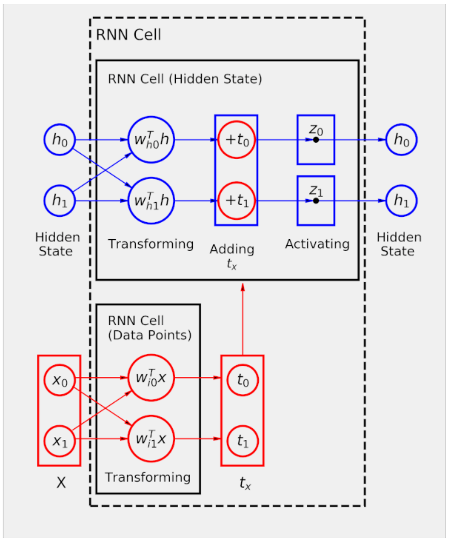


Figure 8.7 - RNN cell at neuron level

Since one can choose the number of hidden dimensions, I chose **two dimensions** simply because I want to be able to easily **visualize the results**. Hence, **two blue neurons** are transforming the hidden state.

The number of **red neurons** will **necessarily** be the **same** as the chosen **number of hidden dimensions** since both transformed outputs need to be added together. But it **doesn't mean** the **data points** must have the **same number of dimensions**.

Coincidentally, our data points have **two coordinates**, but even if we had 25 dimensions, these 25 features would **still** be mapped into **two dimensions** by the two red neurons.

The only operation left is the activation function, most likely the hyperbolic tangent, which will produce the updated hidden state.

"*Why hyperbolic tangent? Isn't ReLU a better activation function?*"

The hyperbolic tangent has a "*competitive advantage*" here since it maps the feature space to clearly defined boundaries: the interval (-1, 1). This guarantees that, at every step of the sequence, the **hidden state is always within these boundaries**. Given that we have **only one linear layer to transform the hidden state**, regardless of which step of the sequence it is being used in, it is definitely convenient to have its values within a predictable range. We'll get back to it in the "*Journey of a Hidden State*" section.

Now, let's see how an RNN cell works in code: we'll create one using PyTorch's own `nn.RNNCell` and **disassemble** it into its components to manually reproduce all the steps involved in **updating the hidden state**. To create a cell we need to tell it the `input_size` (number of features in our data points) and the `hidden_size` (the size of the vector representing the hidden state). It is also possible to tell it not to add biases, and to use a ReLU instead of Tanh, but we're sticking to the defaults.

```
n_features = 2
hidden_dim = 2

torch.manual_seed(19)
rnn_cell = nn.RNNCell(input_size=n_features, hidden_size=hidden_dim)
rnn_state = rnn_cell.state_dict()
rnn_state
```

Output

```
OrderedDict([('weight_ih', tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh', tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih', tensor([0.4954, 0.6533])),
             ('bias_hh', tensor([-0.3565, -0.2904]))])
```

The `weight_ih` and `bias_ih` (`i` stands for inputs - the data) tensors correspond to the **red neurons** in Figure 8.7. The `weight_hh` and `bias_hh` (`h` stands for hidden) tensors, to the **blue neurons**. We can use these weights to create **two linear layers**:

```
linear_input = nn.Linear(n_features, hidden_dim)
linear_hidden = nn.Linear(hidden_dim, hidden_dim)

with torch.no_grad():
    linear_input.weight = nn.Parameter(rnn_state['weight_ih'])
    linear_input.bias = nn.Parameter(rnn_state['bias_ih'])
    linear_hidden.weight = nn.Parameter(rnn_state['weight_hh'])
    linear_hidden.bias = nn.Parameter(rnn_state['bias_hh'])
```

Now, let's work our way through the mechanics of the RNN cell! It all starts with the **initial hidden state** representing the **empty sequence**:

```
initial_hidden = torch.zeros(1, hidden_dim)
initial_hidden
```

Output

```
tensor([[0., 0.]])
```

Then we use the **two blue neurons**, the `linear_hidden` layer, to **transform the**

hidden state:

```
th = linear_hidden(initial_hidden)
th
```

Output

```
tensor([[-0.3565, -0.2904]], grad_fn=<AddmmBackward>)
```

Cool! Now, let's take look at a **sequence of data points** from our dataset:

```
X = torch.as_tensor(points[0]).float()
X
```

Output

```
tensor([[ 1.0349,  0.9661],
        [ 0.8055, -0.9169],
        [-0.8251, -0.9499],
        [-0.8670,  0.9342]])
```

As expected, four data points, two coordinates each. The first data point, `[1.0349, 0.9661]`, corresponding to the top-right corner of the square, is going to be transformed by the `linear_input` layers (the **two red neurons**):

```
tx = linear_input(X[0:1])
tx
```

Output

```
tensor([[0.7712, 1.4310]], grad_fn=<AddmmBackward>)
```

There we go, we got both t_x and t_h. Let's add them together:

```
adding = th + tx
adding
```

Output

```
tensor([[0.4146, 1.1405]], grad_fn=<AddBackward0>)
```

The **effect of adding t_x is similar to the effect of adding the bias**: it is effectively **translating** the transformed hidden state to the right (by `0.7712`) and up (by `1.4310`).

Finally, the hyperbolic tangent activation function "*compresses*" the feature space back into the `(-1, 1)` interval:

```
torch.tanh(adding)
```

Output

```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward>)
```

That's the **updated hidden state**!

Now, let's make a quick sanity check, feeding the same input to the original RNN cell:

```
rnn_cell(X[0:1])
```

Output

```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward>)
```

Great, the values match.

We can also **visualize** this sequence of operations, assuming that every hidden space "*lives*" in a feature space delimited by the boundaries given by the hyperbolic tangent. So, the initial hidden state `(0, 0)` sits at the center of this feature space, depicted in the left-most plot in the figure below:

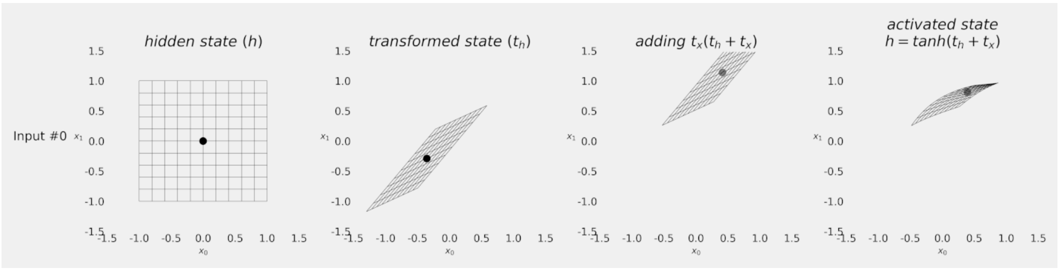


Figure 8.8 - Evolution of the hidden state

The **transformed hidden state** (the output of `linear_hidden`) is depicted in the second plot: it went through an **affine transformation**. The point in the center corresponds to t_h. In the third plot, we can see the effect of **adding t_x** (the output of `linear_input`): the whole feature space was **translated** to the right and up. And then, in the right-most plot, the **hyperbolic tangent** works its magic and brings the whole feature space **back to the `(-1, 1)` range**. That was the **first step** in the journey of a hidden state. We'll do it once again, using the full sequence, after training a model.

I guess it is time to feed the **full sequence** to the RNN cell, right? You may be tempted to do it like this:

```
# WRONG!
rnn_cell(X)
```

Output

```
tensor([[ 0.3924,  0.8146],
        [ 0.7864,  0.5266],
        [-0.0047, -0.2897],
        [-0.6817,  0.1109]], grad_fn=<TanhBackward>)
```

This is **wrong**! Remember, the RNN cell has **two inputs**: one hidden state and one data point.

"Where is the hidden state then?"

That's exactly the problem! If not provided, it defaults to the zeros corresponding to the initial hidden state. So, the call above is **not** processing **four steps of a sequence**, but processing the **first step of what it is assuming to be four sequences**.

To effectively use the RNN cell in a sequence, we need to **loop over the data points** and **provide the updated hidden state at each step**:

```
hidden = torch.zeros(1, hidden_dim)
for i in range(X.shape[0]):
    out = rnn_cell(X[i:i+1], hidden)
    print(out)
    hidden = out
```

Output

```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward>)
tensor([[ 0.4347, -0.0481]], grad_fn=<TanhBackward>)
tensor([[-0.1521, -0.3367]], grad_fn=<TanhBackward>)
tensor([[-0.5297,  0.3551]], grad_fn=<TanhBackward>)
```

Now we're talking! The last hidden state, `(-0.5297, 0.3551)`, is the representation of the full sequence.

Figure 8.9 depicts how the loop above looks like at the neuron level. In it, you can easily see what I call "*the journey of a hidden state*": it is **transformed**, **translated** (adding the input), and **activated** many times over. Moreover, you can also see that the **data points are independently transformed** - the model will learn the best way to transform them. We'll get back to it after training a model.

At this point, you may be thinking:

"Looping over the data points in a sequence?! That looks like a lot of work!"

And you're absolutely right! Instead of an RNN cell, we can use a fully-fledged…

RNN Layer

The `nn.RNN` layer takes care of the hidden state handling for us, no matter how long the input sequence is. This is the layer we'll be actually using in the model. We've been through the inner workings of its cells, but the fully-fledged RNN offers **many more options** (stacked and/or bidirectional layers, for instance) and **one tricky thing regarding the shapes of inputs and outputs** (yes, shapes are a kinda *recurrent* problem - pun intended!).

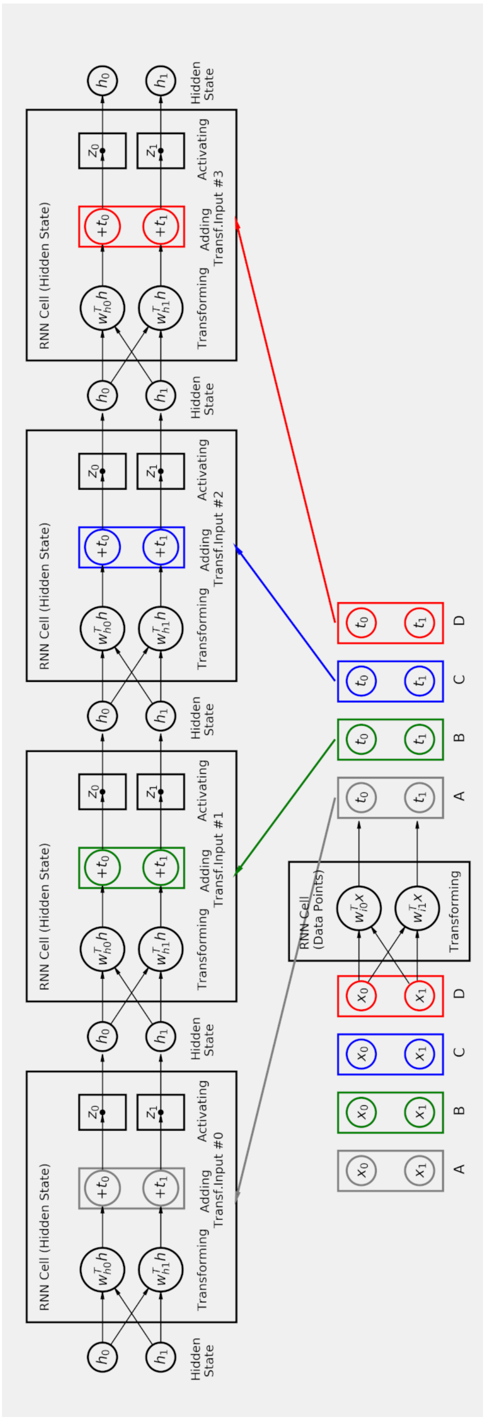


Figure 8.9 - Multiple cells in sequence

Let’s take a look at the RNN’s arguments:

- `input_size`: it is the number of features in each data point of the sequence
- `hidden_size`: it is the number of hidden dimensions you want to use
- `bias`: just like any other layer, it includes the bias in the equations
- `nonlinearity`: by default, it uses the hyperbolic tangent, but you can change it to ReLU if you want

The four arguments above are exactly the same as in the RNN cell. So we can easily create a fully-fledged RNN like that:

```
n_features = 2
hidden_dim = 2

torch.manual_seed(19)
rnn = nn.RNN(input_size=n_features, hidden_size=hidden_dim)
rnn.state_dict()
```

Output

```
OrderedDict([('weight_ih_l0', tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh_l0', tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih_l0', tensor([0.4954, 0.6533])),
             ('bias_hh_l0', tensor([-0.3565, -0.2904]))])
```

Since the seed is exactly the same, you’ll notice that the **weights** and **biases** have **exactly the same values** as our former RNN cell. The only difference is in the **parameters' names**: now they all have an `_l0` suffix to indicate they belong to the **first "*layer*"**.

*"What do you mean by **layer**? Isn’t the RNN itself a layer?"*

Yes, the RNN itself can be a layer in our model. But it may have its **own internal layers**! You can configure those with those extra three arguments:

- `num_layers`: the RNN we've been using so far has *one* layer (the default value), but if you use **more than one**, you'll be creating a **stacked RNN**, which we'll see in its own section
- `bidirectional`: so far, our RNNs have been handling sequences in the **left-to-right direction** (the default), but if you set this argument to `True`, you'll be creating a **bidirectional RNN**, which we'll also see in its own section
- `dropout`: introduces RNN's **own dropout layer between its internal layers**, so it only makes sense if you're using a **stacked RNN**

And I saved the best (actually, the worst) for last:

- `batch_first`: the documentation says "*if* `True`, *then the input and output tensors are provided as (batch, seq, feature)*", which makes you think that you only need to set it to `True` and it will turn everything into your nice and familiar tensors where different batches are concatenated together as its first dimension - and you'd be sorely mistaken

"*Why? What's wrong with that?*"

The problem is, you need to read the documentation very literally: **only** the input and output tensors are going to be batch first, **the hidden state will never be batch first**. This behavior *may* bring a lot of complications you need to be aware of.

Shapes

Before going through an example, let's take a look at the expected inputs and outputs of our RNN:

- Inputs:
  - the **input tensor** containing the **sequence** you want to run through the RNN

        - the default shape is **sequence-first**, that is, **(sequence length, batch size, number of features)** which we’re abbreviating to **(L, N, F)**
        - but if you choose `batch_first`, it will *flip* the first two dimensions and then it will expect an **(N, L, F)** shape, which is what you’re likely getting from a data loader
        - by the way, the input can also be a **packed sequence** - we’ll get back to that in a later section
    - the **initial hidden state**, which defaults to zeros if not provided
        - a simple RNN will have a hidden state tensor with shape **(1, N, H)**
        - a stacked RNN (more on that in the next section) will have a hidden state tensor with shape **(number of stacked layers, N, H)**
        - a bidirectional RNN (more on that later) will have a hidden state tensor with shape **(2*number of stacked layers, N, H)**
- Outputs:
    - the **output tensor** contains the **hidden states** corresponding to the outputs of its RNN cells for **all steps in the sequence**
        - a simple RNN will have an output tensor with shape **(L, N, H)**
        - a bidirectional RNN (more on that later) will have an output tensor with shape **(L, N, 2*H)**
        - if you choose `batch_first`, it will *flip* the first two dimensions and then it will produce outputs with shape**(N, L, H)**
    - the **final hidden state** corresponding to the representation of the full sequence and its shape follows the same rules as the initial hidden state

Let’s illustrate the difference in shapes by creating a "*batch*" containing **three sequences**, each sequence having **four data points** (corners), each data point having **two coordinates**. Its shape is **(3, 4, 2)** and it is an example of a *batch-first* tensor **(N, L, F)**, like a mini-batch you’d get from a data loader:

```
batch = torch.as_tensor(points[:3]).float()
batch.shape
```

Output

```
torch.Size([3, 4, 2])
```

Since RNN’s use **sequence-first** by default, we *could* explicitly change the shape of the batch using `permute` to *flip* the first two dimensions:

```
permuted_batch = batch.permute(1, 0, 2)
permuted_batch.shape
```

Output

```
torch.Size([4, 3, 2])
```

Now the data is in an "*RNN-friendly*" shape and we can run it through a regular RNN to get two *sequence-first* tensors back:

```
torch.manual_seed(19)
rnn = nn.RNN(input_size=n_features, hidden_size=hidden_dim)
out, final_hidden = rnn(permuted_batch)
out.shape, final_hidden.shape
```

Output

```
(torch.Size([4, 3, 2]), torch.Size([1, 3, 2]))
```

For simple RNNs, the **last element of the output IS the final hidden state**!

```
(out[-1] == final_hidden).all()
```

Output

```
tensor(True)
```

Once we're done with the RNN we can turn the data **back to our familiar batch-first shape**:

```
batch_hidden = final_hidden.permute(1, 0, 2)
batch.shape
```

Output

```
torch.Size([3, 1, 2])
```

That seems a lot of work, though. Alternatively, we could set RNN's `batch_first` argument to `True` so we can use the batch above without any modifications:

```
torch.manual_seed(19)
rnn_batch_first = nn.RNN(input_size=n_features,
                         hidden_size=hidden_dim,
                         batch_first=True)
out, final_hidden = rnn_batch_first(batch)
out.shape, final_hidden.shape
```

Output

```
(torch.Size([3, 4, 2]), torch.Size([1, 3, 2]))
```

But then you get these **two distinct shapes as a result**: *batch-first* (N, L, H) for the output and *sequence-first* (1, N, H) for the final hidden state.

On the one hand, this can lead to confusion. On the other hand, most of the time we **won't** be handling the hidden state, and we'll handle the **batch-first output** instead. So, we can stick with **batch-first** for now and, when it comes the time we *have* to handle the hidden state, I will highlight the difference in shapes once again.

In a nutshell, the **RNN's default behavior** is to handle tensors having the shape **(L, N, H)** for hidden states and **(L, N, F)** for sequences of data points. Datasets and **data loaders**, unless customized otherwise, will produce **data points** in the shape **(N, L, F)**.

To address this difference, we'll be using the `batch_first` argument to turn both **inputs and outputs** into this **familiar batch-first shape**.

Stacked RNN

First, take **one RNN** and feed it a **sequence of data points**. Next, take **another RNN** and feed it the **sequence of outputs produced by the first RNN**. There you go, you got a **stacked RNN** where each one of the RNNs is considered a "*layer*" of the stacked one. The figure below depicts a stacked RNN with two "*layers*":

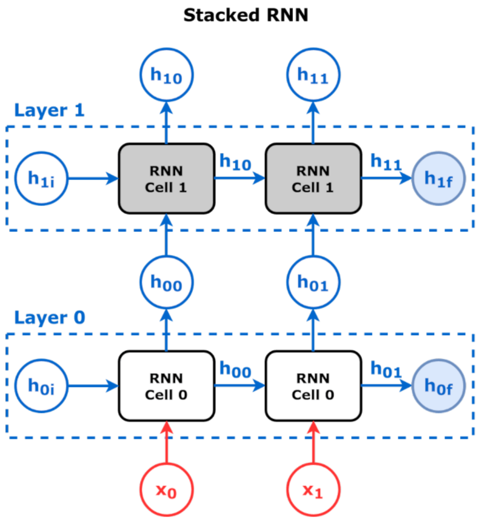


Figure 8.10 - Stacked RNN with two layers

"Two layers only? It doesn't seem much..."

It may not seem like it but a two-layer stacked RNN is already computationally expensive since not only one cell depends on the previous one but also one "*layer*" depends on the other.

Each "*layer*" starts with its own initial hidden state and produces **its own final hidden state** too. The **output of the stacked RNN**, that is, the hidden states at each step of the sequence are **those of the top-most layer**.

Let's create a stacked RNN with two layers:

```
torch.manual_seed(19)
rnn_stacked = nn.RNN(input_size=2, hidden_size=2,
                     num_layers=2, batch_first=True)
state = rnn_stacked.state_dict()
state
```

Output

```
OrderedDict([('weight_ih_l0', tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh_l0', tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih_l0', tensor([0.4954, 0.6533])),
             ('bias_hh_l0', tensor([-0.3565, -0.2904])),
             ('weight_ih_l1', tensor([[-0.6701, -0.5811],
                      [-0.0170, -0.5856]])),
             ('weight_hh_l1', tensor([[ 0.1159, -0.6978],
                      [ 0.3241, -0.0983]])),
             ('bias_ih_l1', tensor([-0.3163, -0.2153])),
             ('bias_hh_l1', tensor([ 0.0722, -0.3242]))])
```

From its state dictionary, we can see it has **two groups** of weights and biases, one for each layer, each layer indicated by its corresponding suffix (`_l0` and `_l1`).

Now, let's create **two simple RNNs**, and use the weights and biases above to set their weights accordingly. Each RNN will behave as one of the layers from the stacked one:

```
rnn_layer0 = nn.RNN(input_size=2, hidden_size=2, batch_first=True)
rnn_layer1 = nn.RNN(input_size=2, hidden_size=2, batch_first=True)

rnn_layer0.load_state_dict(dict(list(state.items())[:4]))
rnn_layer1.load_state_dict(dict([(k[:-1]+'0', v)
                                 for k, v in
                                 list(state.items())[4:]]))
```

Output

```
<All keys matched successfully>
```

Now, let's make **a batch containing one sequence** from our synthetic dataset (thus having shape `(N=1, L=4, F=2)`):

```
x = torch.as_tensor(points[0:1]).float()
```

The RNN representing the first layer takes the sequence of data points as usual:

```
out0, h0 = rnn_layer0(x)
```

It produces the expected two outputs: a sequence of hidden states (`out0`) and the final hidden state (`h0`) for this layer.

Next, it uses the **sequence of hidden states as inputs for the next layer**:

```
out1, h1 = rnn_layer1(out0)
```

The second layer produces the expected two outputs again: another sequence of hidden states (`out1`) and the final hidden state (`h1`) for this layer.

The **overall output of the stacked RNN** must have two elements as well:

- a **sequence of hidden states**, those produced by the **last layer** (`out1`)
- the **concatenation** of **final hidden states** of **all layers**

```
out1, torch.cat([h0, h1])
```

Output

```
(tensor([[[-0.7533, -0.7711],
          [-0.0566, -0.5960],
          [ 0.4324, -0.2908],
          [ 0.1563, -0.5152]]], grad_fn=<TransposeBackward1>),
 tensor([[[-0.5297,  0.3551]],

         [[ 0.1563, -0.5152]]], grad_fn=<CatBackward>))
```

Done! We've replicated the inner workings of a stacked RNN using two simple RNNs. You can double-check the results by feeding the sequence of data points to the actual stacked RNN itself:

```
out, hidden = rnn_stacked(x)
out, hidden
```

And you'll get exactly the same results.

For **stacked** RNNs, the **last element of the output is the final hidden state of the LAST LAYER**! But, since we're using a `batch_first` layer, we need to *permute* the hidden state's dimensions to *batch-first* as well:

```
(out[:, -1] == hidden.permute(1, 0, 2)[:, -1]).all()
```

Output

```
tensor(True)
```

Bidirectional RNN

First, take **one RNN** and feed it a **sequence of data points**. Next, take **another RNN** and feed it the **sequence of data points in reversed order**. There you go, you got a **bidirectional RNN** where each one of the RNNs is considered a "*direction*" of the bidirectional one. The figure below depicts a bidirectional RNN:

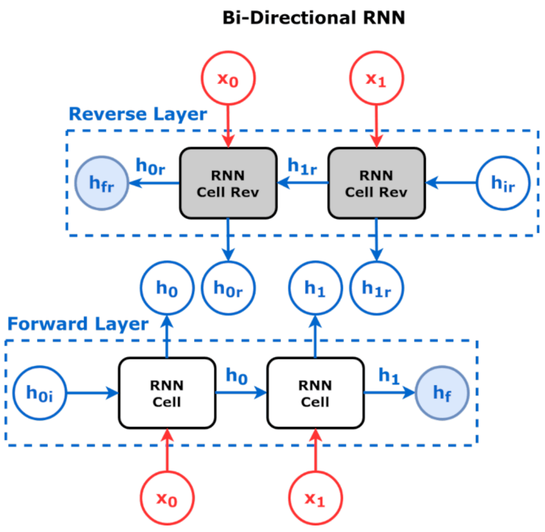


Figure 8.11 - Bidirectional RNN

Each "*layer*" starts with its own initial hidden state and produces **its own final hidden state** too. But, unlike the stacked version, it **keeps both sequences of hidden states** produced at each step. Moreover, it also **reverses back** the sequence of hidden states produced by the reverse layer to make both sequences match (h_0 with h_{0r},h_1 with h_{1r}, and so on).

"*Why would you need a bidirectional RNN?*"

The **reverse layer** allows the network to look at "*future*" information in a given sequence, thus better describing the **context** in which the elements of the sequence are. This is particularly important in Natural Language Processing tasks, where the role of a given word sometimes may only be ascertained by the word that *follows* it. These relationships would never be captured by a unidirectional

RNN.

Let's create a bidirectional RNN:

```
torch.manual_seed(19)
rnn_bidirect = nn.RNN(input_size=2, hidden_size=2,
                      bidirectional=True, batch_first=True)
state = rnn_bidirect.state_dict()
state
```

Output

```
OrderedDict([('weight_ih_l0', tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh_l0', tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih_l0', tensor([0.4954, 0.6533])),
             ('bias_hh_l0', tensor([-0.3565, -0.2904])),
             ('weight_ih_l0_reverse', tensor([[-0.6701, -0.5811],
                      [-0.0170, -0.5856]])),
             ('weight_hh_l0_reverse', tensor([[ 0.1159, -0.6978],
                      [ 0.3241, -0.0983]])),
             ('bias_ih_l0_reverse', tensor([-0.3163, -0.2153])),
             ('bias_hh_l0_reverse', tensor([ 0.0722, -0.3242]))])
```

From its state dictionary, we can see it has **two groups** of weights and biases, one for each layer, each layer indicated by its corresponding suffix (`_l0` and `_l0_reverse`).

Once again, let's create **two simple RNNs**, and use the weights and biases above to set their weights accordingly. Each RNN will behave as one of the layers from the bidirectional one:

```
rnn_forward = nn.RNN(input_size=2, hidden_size=2, batch_first=True)
rnn_reverse = nn.RNN(input_size=2, hidden_size=2, batch_first=True)

rnn_forward.load_state_dict(dict(list(state.items())[:4]))
rnn_reverse.load_state_dict(dict([(k[:-8], v)
                                  for k, v in
                                  list(state.items())[4:]]))
```

Output

```
<All keys matched successfully>
```

We'll be using the same single-sequence batch from before, but we also need it **in reverse**. We can use PyTorch's flip to reverse the dimension corresponding to the sequence (L):

```
x_rev = torch.flip(x, dims=[1]) #N, L, F
x_rev
```

Output

```
tensor([[[-0.8670,  0.9342],
         [-0.8251, -0.9499],
         [ 0.8055, -0.9169],
         [ 1.0349,  0.9661]]])
```

Since there is **no dependency** between the two layers, we just need to feed each layer its corresponding sequence (regular and reversed) and remember to **reverse back** the **sequence of hidden states**.

```
out, h = rnn_forward(x)
out_rev, h_rev = rnn_reverse(x_rev)
out_rev_back = torch.flip(out_rev, dims=[1])
```

The **overall output of the bidirectional RNN** must have two elements as well:

- a **concatenation** side-by-side of **both sequences of hidden states** (`out` and `out_rev_back`)
- the **concatenation** of **final hidden states** of **both layers**

```
torch.cat([out, out_rev_back], dim=2), torch.cat([h, h_rev])
```

Output

```
(tensor([[[ 0.3924,  0.8146, -0.9355, -0.8353],
          [ 0.4347, -0.0481, -0.1766,  0.2596],
          [-0.1521, -0.3367,  0.8829,  0.0425],
          [-0.5297,  0.3551, -0.2032, -0.7901]]], grad_fn
=<CatBackward>),
 tensor([[[-0.5297,  0.3551]],

         [[-0.9355, -0.8353]]], grad_fn=<CatBackward>))
```

Done! We've replicated the inner workings of a bidirectional RNN using two simple RNNs. You can double-check the results by feeding the sequence of data points to the actual bidirectional RNN itself:

```
out, hidden = rnn_bidirect(x)
```

And, once again, you'll get the very same results.

For **bidirectional** RNNs, the **last element of the output ISN'T the final hidden state**! Once again, since we're using a `batch_first` layer, we need to *permute* the hidden state's dimensions to *batch-first* as well:

```
out[:, -1] == hidden.permute(1, 0, 2).view(1, -1)
```

Output

```
tensor([[ True,  True, False, False]])
```

Bidirectional RNNs are **different** because the **final hidden state** corresponds to the **last element in the sequence** for the **forward layer** and to the **first element in the sequence** for the **reverse layer**. The **output**, on the other hand, is **aligned to sequence**, hence the difference.

Square Model

It is finally time to build a **model** to classify the direction in which the square was drawn: clockwise or counterclockwise. Let's put into practice what we've learned so far and use a **simple RNN** to obtain the **final hidden state** that **represents the full sequence** and use it to train a **classifier layer** which is, once again, the same as a **logistic regression**.

"*There can be only one… hidden state.*"

Connor MacLeod

Data Generation

If you hadn't noticed yet, we only have a *training set*. But, since our data is synthetic anyway, let's simply **generate new data** which, by definition, wasn't seen by the model and therefore qualifies as validation or test data (just make sure to pick a different seed for the generation):

Data Generation

```
test_points, test_directions = generate_sequences(seed=19)
```

Data Preparation

There is nothing special about it: typical data preparation using a tensor dataset and data loaders that will yield batches of sequences with shape `(N=16, L=4, F=2)`.

Data Preparation

```
train_data = TensorDataset(
    torch.as_tensor(points).float(),
    torch.as_tensor(directions).view(-1, 1).float()
)
test_data = TensorDataset(
    torch.as_tensor(test_points).float(),
    torch.as_tensor(test_directions).view(-1, 1).float()
)
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
test_loader = DataLoader(test_data, batch_size=16)
```

Model Configuration

The main structure behind the `SquareModel` is fairly simple: a simple **RNN layer** followed by a **linear layer** that works as a classifier producing logits. Then, in the `forward` method, the linear layer takes the **last output** of the recurrent layer as its input.

Model Configuration

```
class SquareModel(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple RNN
        self.basic_rnn = nn.RNN(self.n_features,
                                self.hidden_dim,
                                batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

*"Why are we taking the **last output** instead of the **final hidden state**? Aren't they the **same**?"*

They are the same in most cases, yes, but they **are different** if you're using **bidirectional RNNs**. By using the **last output**, we're ensuring that the code will work for all sorts of RNN: simple, stacked, *and* bidirectional. Besides, we want to *avoid* handling the hidden state anyway because it's always in *sequence-first* shape.

In the next chapter we'll be using the **full output**, that is, the **full sequence of hidden states** for encoder-decoder models.

Next, we create an instance of the model, the corresponding loss function for a binary classification problem, and an optimizer:

Model Configuration

```
torch.manual_seed(21)
model = SquareModel(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

Then, we train our `SquareModel` over 100 epochs, as usual, visualize the losses, and evaluate its accuracy on the test data:

Model Training

```
sbs_rnn = StepByStep(model, loss, optimizer)
sbs_rnn.set_loaders(train_loader, test_loader)
sbs_rnn.train(100)
```

```
fig = sbs_rnn.plot_losses()
```

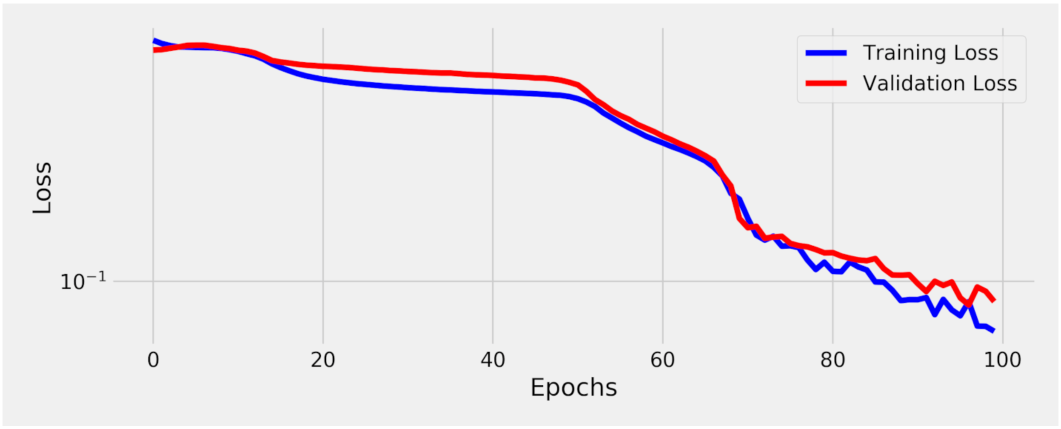


Figure 8.12 - Losses - `SquareModel`

```
StepByStep.loader_apply(test_loader, sbs_rnn.correct)
```

Output

```
tensor([[50, 53],
        [75, 75]])
```

Our simple model hit 97.65% accuracy on the test data. Very good but, then again, this is a toy dataset.

Now, for the *fun* part :-)

Visualizing the Model

In this section, we're going to thoroughly explore **how** the model managed to successfully classify the sequences. We'll see:

- how the model **transformed the inputs**
- how the **classifier separates the final hidden states**
- what the **sequence of hidden states** looks like

- the **journey of a hidden state** through *every* **transformation**, **translation**, and **activation**

Buckle up!

Transformed Inputs

While the hidden state is sequentially transformed, we've already seen (in Figure 8.9) that the **data points are independently transformed**, that is, every data point (corner) goes through **the same affine transformation**. This means we can simply use the parameters `weights_ih_l0` and `bias_ih_l0` learned by our model to see what's happening to the inputs (data points) before they are added up to the transformed hidden state:

```
state = model.basic_rnn.state_dict()
state['weight_ih_l0'], state['bias_ih_l0']
```

Output

```
(tensor([[-0.5873, -2.5140],
         [-1.6189, -0.4233]], device='cuda:0'),
 tensor([0.8272, 0.9219], device='cuda:0'))
```

"How does it look like?"

Let's visualize the transformed "perfect" square:

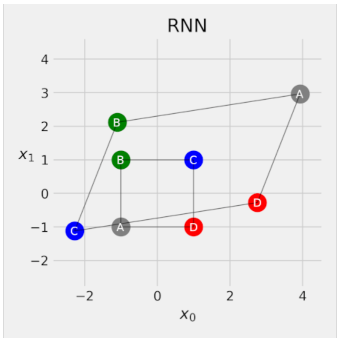


Figure 8.13 - Transformed inputs (corners)

Our `SquareModel` learned that it needs to **scale**, **shear**, **flip** and **translate** the inputs (corners) before adding each one of them to the transformed hidden state at every step.

Hidden States

Remember, there are eight possible sequences (Figure 8.3) since it is possible to start at any of the four corners, and move in either clockwise or counterclockwise direction. Each corner was assigned a **color** (as in the figure above), and since **clockwise** is the **positive class**, it is represented by a "**+**" sign.

If we use the "*perfect*" square as input to our trained model, that's what the **final hidden states** look like for each one of the eight sequences:

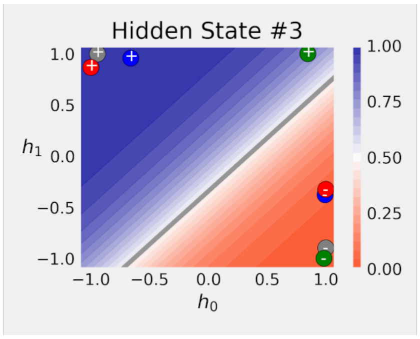


Figure 8.14 - Final hidden states for eight sequences of the "perfect" square

For **clockwise** movement, the final hidden states are situated in the upper-left region, while for the **counterclockwise** movement brought the final hidden states to the lower-right corner. The decision boundary, as expected from a logistic regression, is a straight line. The point closest to the decision boundary, that is, the one the model is less confident about, corresponds to the sequence starting at the *B corner* (green) and moving clockwise (+).

"*What about the **other hidden states** for the **actual sequences**?*"

Let's visualize them as well. In the figure below, **clockwise** sequences are represented by **blue points** and **counterclockwise** sequences, by **red points**:

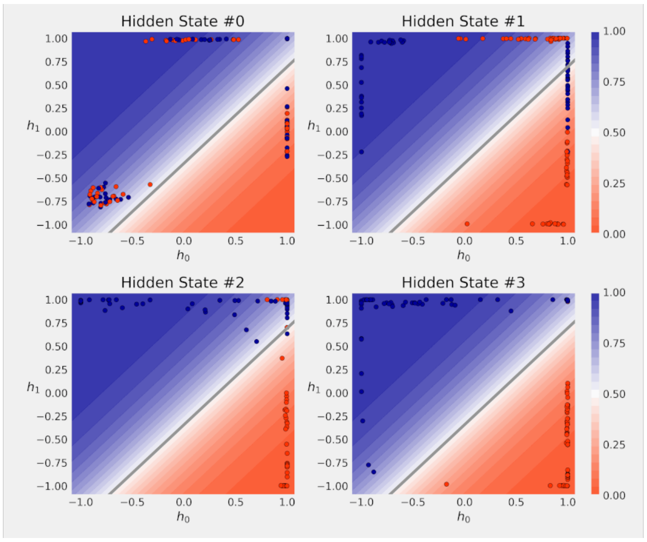


Figure 8.15 - Sequence of hidden states

We can see that the model already achieves *some separation* after "*seeing*" **two data points (corners)**, corresponding to "*Hidden State #1*". After "*seeing*" the third corner, most of the sequences are already correctly classified and, after observing all corners, it gets every *noisy square* right.

"*Can we pick **one sequence** and observe its **hidden state** from its initial to its final values?*"

Sure we can!

The Journey of a Hidden State

Let’s use the **ABCD** sequence of the "*perfect*" square for this. The initial hidden state is `(0, 0)` by default and it is colored black. Every time a new data point (corner) is going to be used in the computation, the affected hidden state is colored accordingly (gray, green, blue, and red, in order).

The figure below **tracks the progress of the hidden state** over **every operation performed inside the RNN**.

The **first column** has the hidden state that’s an **input** for the RNN cell at a given step; the **second column** has the **transformed hidden state**; the **third**, the **translated hidden state** (by adding the transformed input); and the **last**, the **activated hidden state**.

There are **four rows**, one for each data point (corner) in our sequence. The initial hidden state of each row is the activated state of the previous row, so it starts at the initial hidden state of the whole sequence `(0, 0)` and, after processing the gray, green, blue, and red corners, ends at the final hidden state, the red dot close to `(-1, 1)` in the last plot.

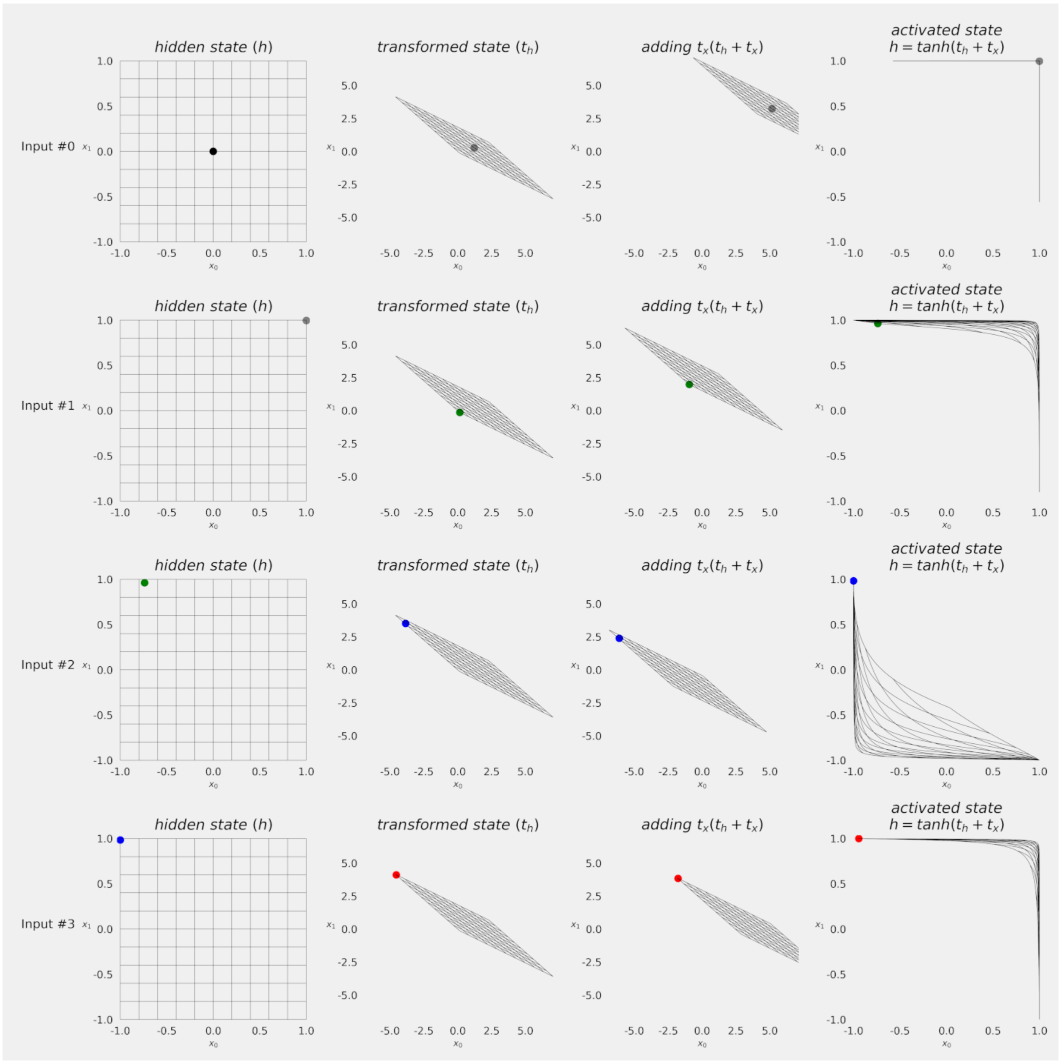


Figure 8.16 - Transforming the hidden state

If we **connect all the hidden state's positions** throughout the whole sequence and **color the path** following the assigned colors for each corner, we get to visualize everything in a single plot in the end:

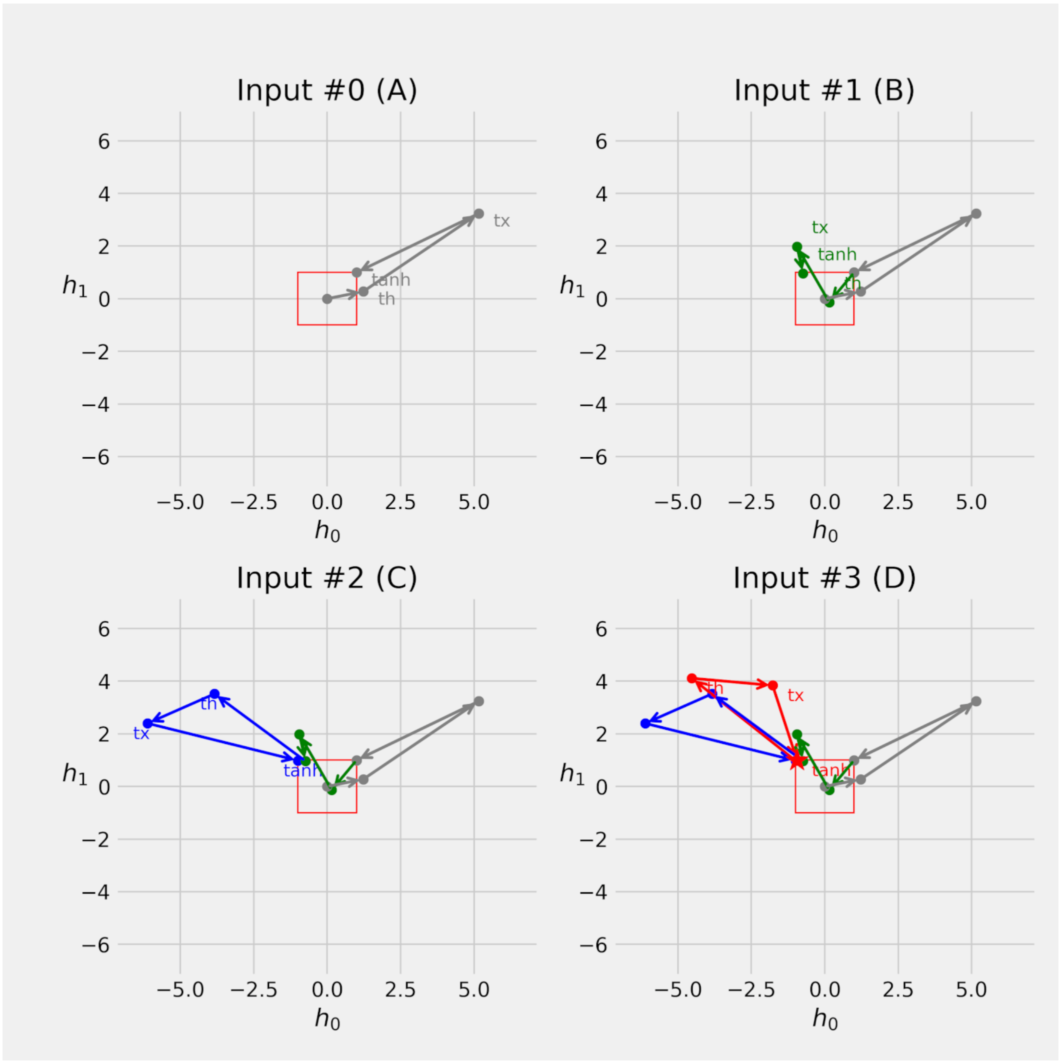


Figure 8.17 - The path of the hidden state

The red square at the center shows the `[-1, 1]` bound given by the hyperbolic-tangent activation. Every hidden state (the last point of a given color (corner)) will be inside or at the edge of the red square. The final position is depicted by a star.

Can We Do Better?

There are a couple of questions I'd like to raise:

- what if the **previous hidden state contains more information** than the **newly computed one**?
- what if **the data point adds more information** than the **previous hidden state**?

Since the RNN cell has both of them (t_h and t_x) on the same footing and **simply adds them up**, there is no way to address the two questions above. To do so, we need something different, we need…

Gated Recurrent Units (GRUs)

Gated Recurrent Units, or GRUs for short, are the answer to those two questions! Let's see how they do it by tackling one problem at a time. What if, instead of simply computing a **new hidden state** and going with it, we try a **weighted average** of both hidden states, **old and new**?

$$h_{new} = tanh(t_h + t_x)$$
$$h' = h_{new} * (1 - z) + h_{old} * z$$

Equation 8.2 - Weighted average of old and new hidden states

The new **parameter z** controls **how much weight** it should give to the **old hidden state**. OK, the first question was addressed, and we can recover the typical RNN behavior simply by **setting *z* to zero**.

Now, what if, instead of computing the new hidden state by simply **adding up t_h and t_x**, we try **scaling t_h** first?

$$h_{new} = tanh(r * t_h + t_x)$$

Equation 8.3 - Scaling the old hidden state

The new **parameter r** controls **how much we keep** from the **old hidden state** before adding the transformed input. For **low values of *r***, the **relative importance**

of the data point is increased, thus addressing the second question. Moreover, we can recover the typical RNN behavior simply by **setting *r* to one**.

Next, we **combine these two changes** into a single expression:

$$h' = tanh(r * t_h + t_x) * (1 - z) + h * z$$

Equation 8.4 - Hidden state, the GRU way

And we've (re)**invented the Gated Recurrent Unit** cell on our own :-)

By the way, the two new parameters ***r*** and ***z*** are called **gates**, respectively, **reset** and **update** gates. Both of them must produce values **between zero and one** thus allowing **only a fraction of the original values** to go through.

Every gate **produces a vector of values** (each value between zero and one) with a **size** corresponding to the **number of hidden dimensions**. For **two** hidden dimensions, a gate may have values like **`[0.52, 0.87]`** for example.

Since gates produce vectors, operations involving them are **element-wise multiplications**.

GRU Cell

If we place both expressions next to one another we can more easily see that the **RNN is a special case of the GRU** (for `r`=1 and `z`=`0`):

$$
\begin{aligned}
RNN : h' &= tanh(t_h + t_x) \\
GRU : h' &= \underbrace{\underbrace{tanh(r * t_{hn} + t_{xn})}_{n} * (1 - z) + h * z}_{weighted\ average\ of\ n\ and\ h}
\end{aligned}
$$

Equation 8.5 - RNN vs GRU

"OK, I see it… but where do ***r*** *and* ***z*** *come from?"*

Well, that's a deep learning book, so the **only right answer** to "*where do **something** come from*" is a **neural network**! Just kidding… or am I? Actually, we'll train **both gates** using a structure that is **pretty much an RNN cell**, except for the fact that it uses a **sigmoid** activation function:

$$
\begin{aligned}
r(eset\ gate) &= \sigma(t_{hr} + t_{xr}) \\
z(update\ gate) &= \sigma(t_{hz} + t_{xz}) \\
n &= tanh(r * t_{hn} + t_{xn})
\end{aligned}
$$

Equation 8.6 - Gates (r and z) and candidate hidden state (n)

Every **gate** worth of its name will use a **sigmoid activation function** to produce gate-compatible values **between zero and one**.

Moreover, since all components of a GRU (*n*, *r*, and *z*) share a similar structure, it should be no surprise that its corresponding transformations (t_h and t_x) are also similarly computed:

$$
\begin{array}{rcccccc}
r\ (hidden) & : t_{hr} & = & W_{hr} & h & + & b_{hr} \\
r\ (input) & : t_{xr} & = & W_{ir} & x & + & b_{ir} \\
z\ (hidden) & : t_{hz} & = & W_{hz} & h & + & b_{hz} \\
z\ (input) & : t_{xz} & = & W_{iz} & x & + & b_{iz} \\
n\ (hidden) & : t_{hn} & = & W_{hn} & h & + & b_{hn} \\
n\ (input) & : t_{xn} & = & W_{in} & x & + & b_{in}
\end{array}
$$

Equation 8.7 - Transformations of a GRU

See? They all follow the same logic! Actually, let's *literally* **see** how all these components are connected together in the following diagram:

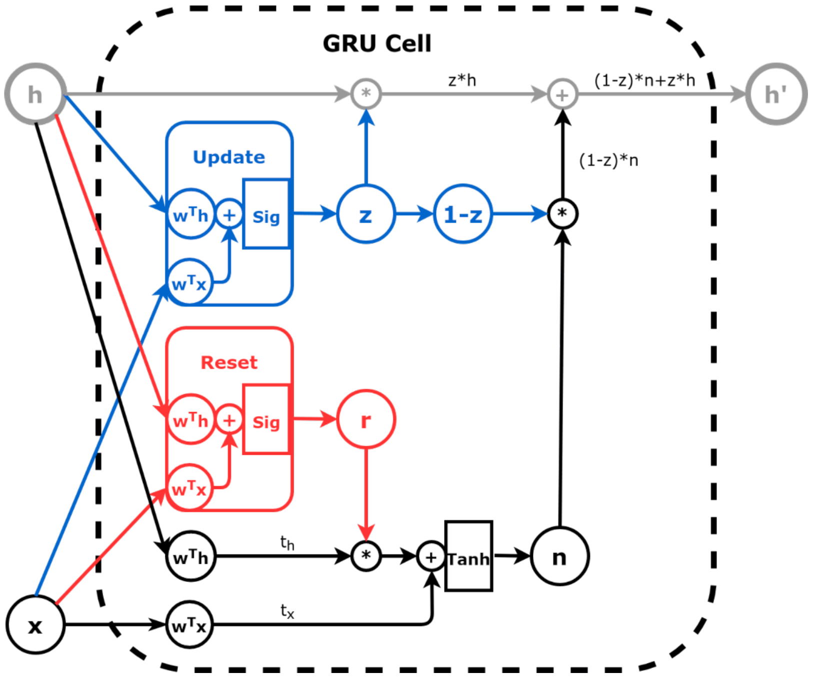


Figure 8.18 - Internals of a GRU cell

The gates are following the same color convention I used in the equations: **red** for the **reset gate** (*r*) and **blue** for the **update gate** (*z*). The path of the **(new) candidate hidden state** (*n*) is drawn in **black** and it joins the **(old) hidden state** (*h*), drawn in **gray**, to produce the **actual new hidden state** (*h'*).

To really understand the flow of information inside the GRU cell, I suggest you try these exercises:

- first, learn to look *past* (or literally ignore) the **internals of the gates**: both ***r*** and ***z*** are simply **values between zero and one** (for each hidden dimension)
- pretend `r=1`; can you see that the **resulting n** is equivalent to the output of a **simple RNN**?
- keep `r=1` and now pretend `z=0`; can you see that the **new hidden state h'** is equivalent to the output of a **simple RNN**?
- now pretend `z=1`; can you see that the **new hidden state *h'*** is simply a **copy of the old hidden state** (in other words, the data (*x*) does not have any effect)?
- if you **decrease *r* all the way to zero**, the **resulting *n*** is **less and less influenced** by the **old hidden state**
- if you **decrease *z* all the way to zero**, the **new hidden state *h'*** is **closer and closer** to ***n***
- for `r=0` and `z=0`, the cell becomes equivalent to a **linear layer** followed by a **Tanh** activation function (in other words, the old hidden state (*h*) does not have any effect)

Now, let's see how a GRU cell works in code: we'll create one using PyTorch's own `nn.GRUCell` and **disassemble** it into its components to manually reproduce all the steps involved in **updating the hidden state**. To create a cell we need to tell it the `input_size` (number of features in our data points) and the `hidden_size` (the size of the vector representing the hidden state), exactly the same as in the RNN cell. The nonlinearity is fixed, though, as the hyperbolic tangent.

```
n_features = 2
hidden_dim = 2

torch.manual_seed(17)
gru_cell = nn.GRUCell(input_size=n_features, hidden_size=hidden_dim)
gru_state = gru_cell.state_dict()
gru_state
```

Output

```
OrderedDict([('weight_ih', tensor([[-0.0930,  0.0497],
                                   [ 0.4670, -0.5319],
                                   [-0.6656,  0.0699],
                                   [-0.1662,  0.0654],
                                   [-0.0449, -0.6828],
                                   [-0.6769, -0.1889]])),
             ('weight_hh', tensor([[-0.4167, -0.4352],
                                   [-0.2060, -0.3989],
                                   [-0.7070, -0.5083],
                                   [ 0.1418,  0.0930],
                                   [-0.5729, -0.5700],
                                   [-0.1818, -0.6691]])),
             ('bias_ih',
              tensor([-0.4316,  0.4019,  0.1222, -0.4647, -0.5578,
0.4493])),
             ('bias_hh',
              tensor([-0.6800,  0.4422, -0.3559, -0.0279,  0.6553,
0.2918]))])
```

*"Wait... there is something definitely **weird** with these shapes..."*

Yeah, you're right... instead of returning **separate weights** for each one of GRU cell's components (*r*, *z*, and *n*), the `state_dict` returns the **concatenated weights and biases**.

```
Wx, bx = gru_state['weight_ih'], gru_state['bias_ih']
Wh, bh = gru_state['weight_hh'], gru_state['bias_hh']

print(Wx.shape, Wh.shape)
print(bx.shape, bh.shape)
```

Output

```
torch.Size([6, 2]) torch.Size([6, 2])
torch.Size([6]) torch.Size([6])
```

The shape is **(3*hidden_dim, n_features)** for `weight_ih`, **(3*hidden_dim, hidden_dim)** for `weight_hh`, and simply **(3*hidden_dim)** for both biases.

For `Wx` and `bx` in the state dictionary above, we can split the values like this:

$$W_{xr} = \begin{cases} -0.0930, & 0.0497, \\ 0.4670, -0.5319, \end{cases}$$

$$W_{xz} = \begin{cases} -0.6656, & 0.0699, \\ -0.1662, & 0.0654, \end{cases}$$

$$W_{xn} = \begin{cases} -0.0449, -0.6828, \\ -0.6769, -0.1889 \end{cases}$$

$$\underbrace{-0.4316, 0.4019}_{b_{xr}}, \underbrace{0.1222, -0.4647}_{b_{xz}}, \underbrace{-0.5578, 0.4493}_{b_{xn}}$$

Equation 8.8 - Splitting tensors into their r, z, and n components

In code, we can use `split` to get tensors for each one of the components:

```
Wxr, Wxz, Wxn = Wx.split(hidden_dim, dim=0)
bxr, bxz, bxn = bx.split(hidden_dim, dim=0)

Whr, Whz, Whn = Wh.split(hidden_dim, dim=0)
bhr, bhz, bhn = bh.split(hidden_dim, dim=0)

Wxr, bxr
```

Output

```
(tensor([[-0.0930,  0.0497],
         [ 0.4670, -0.5319]]), tensor([-0.4316,  0.4019]))
```

Next, let's use the weights and biases to create the corresponding linear layers:

```
def linear_layers(Wx, bx, Wh, bh):
    hidden_dim, n_features = Wx.size()
    lin_input = nn.Linear(n_features, hidden_dim)
    lin_input.load_state_dict({'weight': Wx, 'bias': bx})
    lin_hidden = nn.Linear(hidden_dim, hidden_dim)
    lin_hidden.load_state_dict({'weight': Wh, 'bias': bh})
    return lin_hidden, lin_input

# reset gate - red
r_hidden, r_input = linear_layers(Wxr, bxr, Whr, bhr)
# update gate - blue
z_hidden, z_input = linear_layers(Wxz, bxz, Whz, bhz)
# candidate state - black
n_hidden, n_input = linear_layers(Wxn, bxn, Whn, bhn)
```

Then, let's use these layers to create functions that replicate **both gates (*r* and *z*)** and the **candidate hidden state (*n*)**:

```
def reset_gate(h, x):
    thr = r_hidden(h)
    txr = r_input(x)
    r = torch.sigmoid(thr + txr)
    return r  # red

def update_gate(h, x):
    thz = z_hidden(h)
    txz = z_input(x)
    z = torch.sigmoid(thz + txz)
    return z  # blue

def candidate_n(h, x, r):
    thn = n_hidden(h)
    txn = n_input(x)
    n = torch.tanh(r * thn + txn)
    return n  # black
```

Cool, all the transformations and activations are handled by the functions above. This means we can replicate the mechanics of a GRU cell at its component level (*r*, *z*, and *n*). We also need an **initial hidden state** and the **first data point (corner)** of a sequence:

```
initial_hidden = torch.zeros(1, hidden_dim)
X = torch.as_tensor(points[0]).float()
first_corner = X[0:1]
```

We use both values to get the output from the **reset gate (*r*)**:

```
r = reset_gate(initial_hidden, first_corner)
r
```

Output

```
tensor([[0.2387, 0.6928]], grad_fn=<SigmoidBackward>)
```

Let's pause for a moment here. First, the reset gate **returns a tensor of size two** because we have **two hidden dimensions**. Second, the **two values may be different** (duh, I know!). What does it mean?

The **reset gate** may **scale each hidden dimension independently**. It can completely suppress the values from one of the hidden dimensions while letting the other pass unchallenged. In geometrical terms, it means that the **hidden space may shrink in one direction while stretching in the other**. We'll visualize it shortly in the journey of a (gated) hidden state.

The reset gate is an input for the **candidate hidden state (*n*)**:

```
n = candidate_n(initial_hidden, first_corner, r)
n
```

Output

```
tensor([[-0.8032, -0.2275]], grad_fn=<TanhBackward>)
```

That *would* be the end of it and that would be the new hidden state if it wasn't for the **update gate (*z*)**:

```
z = update_gate(initial_hidden, first_corner)
z
```

Output

```
tensor([[0.2984, 0.3540]], grad_fn=<SigmoidBackward>)
```

Another short pause here… the update gate is telling us to keep **29.84% of the first** and **35.40% of the second dimensions of the initial hidden state**. The remaining **60.16%** and **64.6%**, respectively, are coming from the **candidate hidden state (*n*)**. So, the **new hidden state (`h_prime`)** is computed accordingly:

```
h_prime = n*(1-z) + initial_hidden*z
h_prime
```

Output

```
tensor([[-0.5635, -0.1470]], grad_fn=<AddBackward0>)
```

Now, let's make a quick sanity check, feeding the same input to the original GRU cell:

```
gru_cell(first_corner)
```

Output

```
tensor([[-0.5635, -0.1470]], grad_fn=<AddBackward0>)
```

Perfect match!

But, then again, you're likely not inclined to loop over the sequence yourself while using a GRU cell, right? You probably want to use a fully-fledged…

GRU Layer

The `nn.GRU` layer takes care of the hidden state handling for us, no matter how long the input sequence is. We’ve been through that once with the RNN layer. The arguments, inputs, and outputs are **almost** exactly the same for both of them, except for **one small difference**: you **cannot** choose a different activation function anymore. That’s it.

And yes, you can create **stacked GRUs** and **bidirectional GRUs** as well. The logic doesn’t change a bit - the only difference is that you’ll be using a **fancier GRU cell** instead of the basic RNN cell.

So, let’s go straight to **creating a model** using a Gated Recurring Unit.

Square Model II - The Quickening

This model is pretty much the same as the original "Square Model", except for **one difference**: its recurrent neural network is not a plain RNN anymore, but a GRU. Everything else stays exactly the same.

```
class SquareModelGRU(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModelGRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple GRU
        self.basic_rnn = nn.GRU(self.n_features,
                                self.hidden_dim,
                                batch_first=True)          ①
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

① The ONLY change in the code: from `nn.RNN` to `nn.GRU`

We'll be using the same data loaders once again, so we're going directly to the model configuration and training.

Model Configuration & Training

Model Configuration

```
torch.manual_seed(21)
model = SquareModelGRU(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_gru = StepByStep(model, loss, optimizer)
sbs_gru.set_loaders(train_loader, test_loader)
sbs_gru.train(100)
```

```
fig = sbs_gru.plot_losses()
```

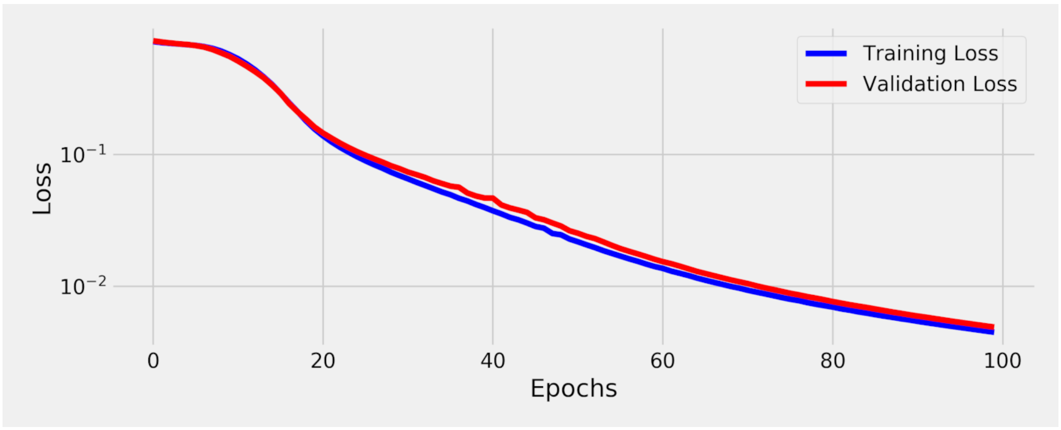


Figure 8.19 - Losses - `SquareModelGRU`

Cool, the loss decreased **much quicker** now, and all it takes is switching from RNN to GRU.

☺

"*The sensation you feel is the* ***quickening***."

Ramirez

```
StepByStep.loader_apply(test_loader, sbs_gru.correct)
```

Output

```
tensor([[53, 53],
        [75, 75]])
```

That's 100% accuracy! Let's try to **visualize** the effect of the GRU architecture on the classification of the hidden states.

Visualizing the Model

Hidden States

Once again, if we use the "*perfect*" square as input to our newly trained model, that's what the **final hidden states** look like for each one of the eight sequences (plotted sided-by-side with the previous model for easier comparison):

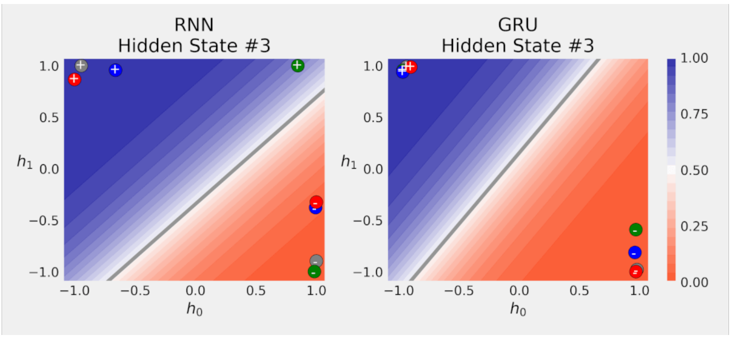


Figure 8.20 - Final hidden states for eight sequences of the "perfect" square

The GRU model achieves a **better separation** of the sequences than its RNN counterpart. What about the actual sequences?

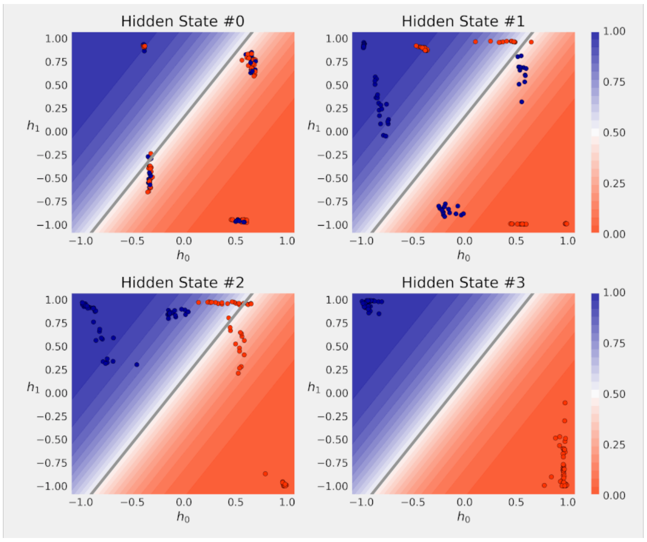


Figure 8.21 - Sequence of hidden states

Like the RNN, the GRU also achieves increasingly better separation as it sees more data points. It is interesting to notice that there are **four distinct groups** of sequences, each corresponding to a **starting corner**.

The Journey of a Gated Hidden State

Once again, we're going to track the **journey** of a hidden state using the **ABCD** sequence of the "*perfect*" square. The initial hidden state is `(0, 0)` by default and it is colored black. Every time a new data point (corner) is going to be used in the computation, the affected hidden state is colored accordingly (gray, green, blue, and red, in order).

Figure 8.22 **tracks the progress of the hidden state** over **every operation performed inside the GRU**.

The **first column** has the hidden state that's an **input** for the GRU cell at a given step: the **third**, **sixth** and **last** columns correspond to the new operations performed by the GRU. The **third** column shows the **gated hidden state**; the **sixth**, the **gated (candidate) hidden state**; and the **last**, the **weighted average of old and candidate hidden states**.

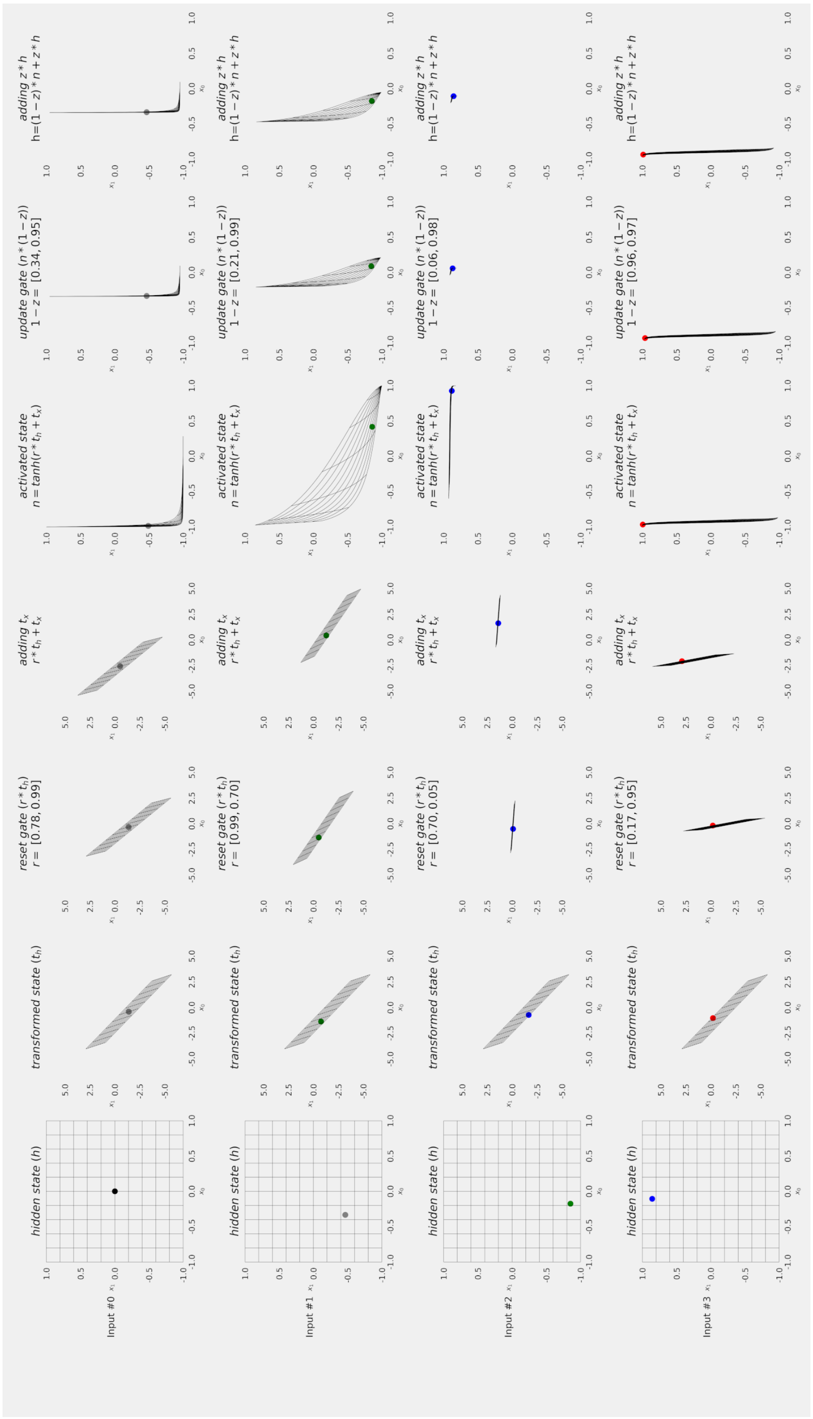


Figure 8.22 - Transforming the hidden state

I'd like to draw your attention to the **third column** in particular: it clearly shows the **effect of a gate**, the reset gate in this case, over the **feature space**. Since a gate has a **distinct value for each dimension**, each dimension will **shrink differently** (it can *only* shrink because values are always between zero and one). In the **third row**, for example, the first dimension gets multiplied by `0.70` while the second dimension gets multiplied by only `0.05`, making the resulting feature space *really* small.

Can We Do Better?

The Gated Recurrent Unit is definitely an improvement over the regular RNN, but there are a couple of points I'd like to raise:

- using the reset gate **inside** the hyperbolic tangent seems "*weird*" (not a scientific argument at all, I know)
- the **best thing** about the **hidden state** is that it is **bounded** by the hyperbolic tangent - it guarantees the next cell will get the hidden state in the same range
- the **worst thing** about the **hidden state** is that it is **bounded** by the hyperbolic tangent - it constrains the values the hidden state can take and, along with them, the corresponding **gradients**
- since we **cannot have the cake** and **eat it too** when it comes to the **hidden state being bounded**, what is preventing us from **using two hidden states in the same cell**?

Yes, let's try that - two hidden states are surely better than one, right?

By the way - I know that GRUs were invented a long time *AFTER* the development of LSTMs, but I've decided to present them in order of increasing complexity. Please don't take the "*story*" I'm telling too literally - it is just a way to facilitate learning.

Long Short-Term Memory (LSTM)

Long Short-Term Memory, or LSTMs for short, use **two states** instead of one.

Besides the regular **hidden state (*h*)**, which is **bounded** by the hyperbolic tangent, as usual, they introduce a second **cell state (*c*)** as well, which is **unbounded**.

So, let's work through the points raised in the last section. First, let's keep it simple and use a **regular RNN** to generate a **candidate hidden state (*g*)**:

$$g = tanh(t_{hg} + t_{xg})$$

Equation 8.9 - LSTM - candidate hidden state

Then, let's turn our attention to the **cell state (*c*)**. What about computing the **new cell state (*c'*)** using a **weighted sum** of the **old cell state** and the **candidate hidden state (*g*)**?

$$c' = g * i + c * f$$

Equation 8.10 - LSTM - new cell state

*"What about **i** and **f**? What are they?"*

They are **gates**, of course :-) The **input (*i*)** gate and the **forget (*f*)** gate. Now, we're only missing the **new hidden state (*h'*)**: if the **cell state** is unbounded, what about making it **bounded** again?

$$h' = tanh(c') * o$$

Equation 8.11 - LSTM - new hidden state

Can you *guess* what that ***o*** is? It is **yet another gate**, the **output (*o*)** gate.

The **cell state** corresponds to the **long-term memory** while the **hidden state**, to the **short-term memory**.

That's it, we've (re)**invented the Long Short-Term Memory** cell on our own!

LSTM Cell

If we place the three expressions next to each another we can more easily see the differences between them:

$$
\begin{aligned}
RNN: h' &= \quad tanh(t_h + t_x) \\
GRU: h' &= tanh(r * t_{hn} + t_{xn}) * \quad (1-z) + \quad h * z \\
LSTM: c' &= \underbrace{tanh(t_{hg} + t_{xg})}_{g} * \quad i + \quad c * f \\
h' &= tanh(c') * \quad o
\end{aligned}
$$

Equation 8.12 - RNN vs GRU vs LSTM

They are not *that* different, to be honest. Sure, the complexity is growing a bit, but it all boils down to finding different ways of **adding up hidden states**, both **old and new**, using **gates.**

The gates themselves always follow the same structure:

$$
\begin{aligned}
i(nput\ gate) &= \sigma(t_{hi} + t_{xi}) \\
f(orget\ gate) &= \sigma(t_{hf} + t_{xf}) \\
o(utput\ gate) &= \sigma(t_{ho} + t_{xo}) \\
g &= tanh(t_{hg} + t_{xg})
\end{aligned}
$$

Equation 8.13 - LSTM's gates

And the transformations used inside the gates and cells **also** follow the same structure:

$$\begin{aligned}
i\ (hidden): t_{hi} &= W_{hi} & h + && b_{hi} \\
i\ (input): t_{xi} &= W_{ii} & x + && b_{ii} \\
f\ (hidden): t_{hf} &= W_{hf} & h + && b_{hf} \\
f\ (input): t_{xf} &= W_{if} & x + && b_{if} \\
g\ (hidden): t_{hg} &= W_{hg} & h + && b_{hg} \\
g\ (input): t_{xg} &= W_{ig} & x + && b_{ig} \\
o\ (hidden): t_{ho} &= W_{ho} & h + && b_{ho} \\
o\ (input): t_{xo} &= W_{io} & x + && b_{io}
\end{aligned}$$

Equation 8.14 - Gates' internal transformations

Now, let’s **visualize** the internals of the LSTM cell:

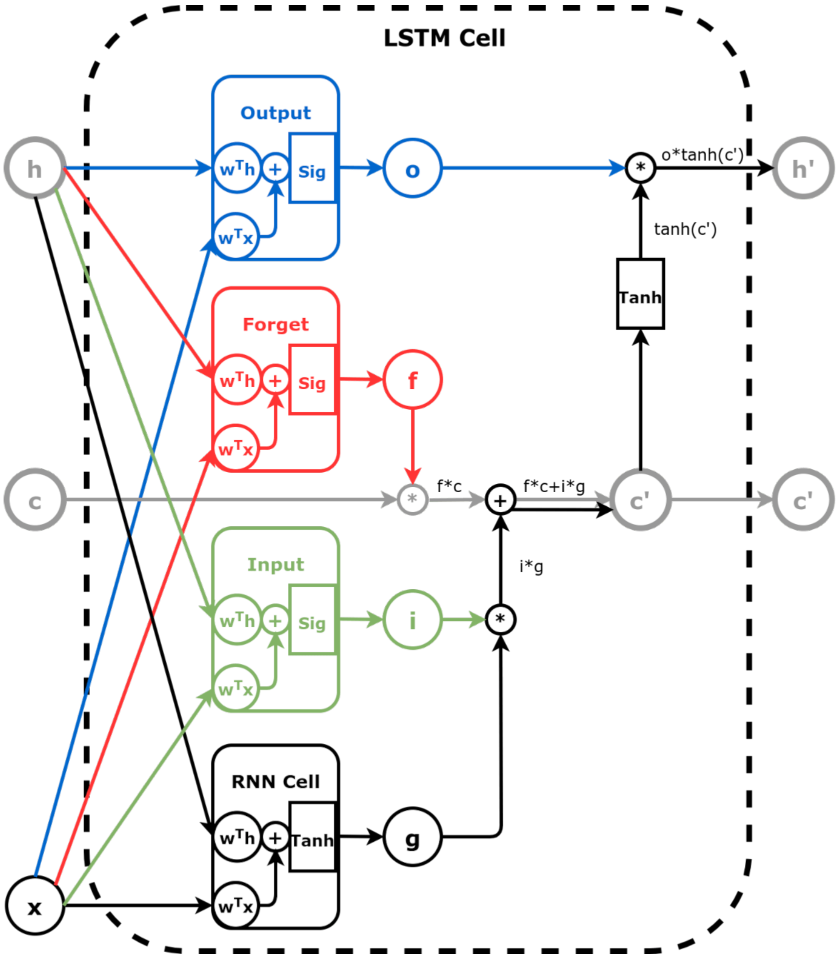


Figure 8.23 - Internals of an LSTM cell

The gates are following the same color convention I used in the equations: **red** for the **forget gate** (*f*), **blue** for the **output gate** (*o*), and **green** for the **input gate (*i*)**. The path of the **(new) candidate hidden state** (*g*) is drawn in **black** and it joins the **(old) cell state** (*c*), drawn in **gray**, to produce **both** the **new cell state (*c'*)** and the **actual new hidden state** (*h'*).

To really understand the flow of information inside the LSTM cell, I suggest you try these exercises:

- first, learn to look *past* (or literally ignore) the **internals of the gates**: ***o***, ***f***, and ***i*** are simply **values between zero and one** (for each dimension)

- pretend `i=1` and `f=0` - can you see that the **new cell state *c'*** is equivalent to the output of a **simple RNN**?
- pretend `i=0` and `f=1` - can you see that the **new cell state *c'*** is simply a **copy of the old cell state** (in other words, the data (*x*) does not have any effect)?
- if you **decrease** `o` **all the way to zero**, the **new hidden state *h'*** is going to be **zero** as well

There is **yet another** important **difference** between the two states, hidden and cell: the **cell state** is computed using **two multiplications and one addition only**. No hyperbolic tangent!

"*So what? What's wrong with the Tanh?*"

There is nothing wrong with it, but its **gradients get very small really fast** (as we've seen in Chapter 4). This can become a problem of **vanishing gradients** for longer sequences. But the **cell state does not suffer from this issue**: it is like a "*highway for gradients*" if you will :-) We're not getting into details about the gradient computation in LSTMs, though.

Now, let's see how an LSTM cell works in code: we'll create one using PyTorch's own `nn.LSTMCell` and **disassemble** it into its components to manually reproduce all the steps involved in **updating the hidden state**. To create a cell we need to tell it the `input_size` (number of features in our data points) and the `hidden_size` (the size of the vector representing the hidden state), exactly the same as in the other two cells. The nonlinearity is, once again, fixed as the hyperbolic tangent.

```
n_features = 2
hidden_dim = 2

torch.manual_seed(17)
lstm_cell = nn.LSTMCell(input_size=n_features,
                        hidden_size=hidden_dim)
lstm_state = lstm_cell.state_dict()
lstm_state
```

Output

```
OrderedDict([('weight_ih', tensor([[-0.0930,  0.0497],
                                   [ 0.4670, -0.5319],
                                   [-0.6656,  0.0699],
                                   [-0.1662,  0.0654],
                                   [-0.0449, -0.6828],
                                   [-0.6769, -0.1889],
                                   [-0.4167, -0.4352],
                                   [-0.2060, -0.3989]])),
             ('weight_hh', tensor([[-0.7070, -0.5083],
                                   [ 0.1418,  0.0930],
                                   [-0.5729, -0.5700],
                                   [-0.1818, -0.6691],
                                   [-0.4316,  0.4019],
                                   [ 0.1222, -0.4647],
                                   [-0.5578,  0.4493],
                                   [-0.6800,  0.4422]])),
             ('bias_ih',
              tensor([-0.3559, -0.0279,  0.6553,  0.2918,  0.4007,
0.3262, -0.0778, -0.3002])),
             ('bias_hh',
              tensor([-0.3991, -0.3200,  0.3483, -0.2604, -0.1582,
0.5558,  0.5761, -0.3919]))])
```

Guess what? Same weird shapes once again but this time there are **four** components instead of **three**. You already know the drill: split the weights and biases using `split` and create linear layers using our `linear_layers` function.

```
Wx, bx = lstm_state['weight_ih'], lstm_state['bias_ih']
Wh, bh = lstm_state['weight_hh'], lstm_state['bias_hh']

# Split weights and biases for data points
Wxi, Wxf, Wxg, Wxo = Wx.split(hidden_dim, dim=0)
bxi, bxf, bxg, bxo = bx.split(hidden_dim, dim=0)
# Split weights and biases for hidden state
Whi, Whf, Whg, Who = Wh.split(hidden_dim, dim=0)
bhi, bhf, bhg, bho = bh.split(hidden_dim, dim=0)

# Creates linear layers for the components
# input gate - green
i_hidden, i_input = linear_layers(Wxi, bxi, Whi, bhi)
# forget gate - red
f_hidden, f_input = linear_layers(Wxf, bxf, Whf, bhf)
 # output gate - blue
o_hidden, o_input = linear_layers(Wxo, bxo, Who, bho)
```

"Wait... isn't there a component missing? You mentioned ***four*** *of them... where are the linear layers for* ***g****?"*

Good catch! It turns out we **don't need** linear layers for ***g*** because it is an **RNN cell** on its own! We can simply use `load_state_dict` to create the corresponding cell:

```
g_cell = nn.RNNCell(n_features, hidden_dim) # black
g_cell.load_state_dict({'weight_ih': Wxg, 'bias_ih': bxg,
                        'weight_hh': Whg, 'bias_hh': bhg})
```

Output

```
<All keys matched successfully>
```

That was easy, right? For the other components, since they are **gates**, we need to create functions for them:

```
def forget_gate(h, x):
    thf = f_hidden(h)
    txf = f_input(x)
    f = torch.sigmoid(thf + txf)
    return f  # red

def output_gate(h, x):
    tho = o_hidden(h)
    txo = o_input(x)
    o = torch.sigmoid(tho + txo)
    return o  # blue

def input_gate(h, x):
    thi = i_hidden(h)
    txi = i_input(x)
    i = torch.sigmoid(thi + txi)
    return i  # green
```

It is all set - we can replicate the mechanics of an LSTM cell at its component level (*f*, *o*, *i*, and *g*) now. We also need an **initial hidden state**, an **initial cell state**, and the **first data point (corner)** of a sequence:

```
initial_hidden = torch.zeros(1, hidden_dim)
initial_cell = torch.zeros(1, hidden_dim)

X = torch.as_tensor(points[0]).float()
first_corner = X[0:1]
```

Then, we start by computing the **gated input** using both the RNN cell (*g*) and its corresponding gate (*i*):

```
g = g_cell(first_corner)
i = input_gate(initial_hidden, first_corner)
gated_input = g * i
gated_input
```

Output

```
tensor([[-0.1340, -0.0004]], grad_fn=<MulBackward0>)
```

Next, we compute the **gated cell state** using the **old cell state** (*c*) and its corresponding gate, the forget (*f*) gate:

```
f = forget_gate(initial_hidden, first_corner)
gated_cell = initial_cell * f
gated_cell
```

Output

```
tensor([[0., 0.]], grad_fn=<MulBackward0>)
```

Well, that's kinda boring… since the old cell state is the **initial cell state** for the first data point in a sequence, gated or not, it will be a bunch of zeros…

The new, updated, **cell state** (*c'*) is simply the **sum** of both **gated input** and **gated cell state**:

```
c_prime = gated_cell + gated_input
c_prime
```

Output

```
tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>)
```

The only thing missing is "*converting*" the cell state to a **new hidden state** (*h'*) using the hyperbolic tangent and the output (*o*) gate:

```
o = output_gate(initial_hidden, first_corner)
h_prime = o * torch.tanh(c_prime)
h_prime
```

Output

```
tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>)
```

The LSTM cell must return **both states**, hidden and cell, in that order, as a tuple:

```
(h_prime, c_prime)
```

Output

```
(tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>),
 tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>))
```

That's it - wasn't that bad, right? The formulation of the LSTM may seem scary at first sight, especially if you bump into a huge sequence of equations using *all weights and biases* at once, but it *doesn't have to be that way*.

Finally, let's make a quick sanity check, feeding the same input to the original LSTM cell:

```
lstm_cell(first_corner)
```

Output

```
(tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>),
 tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>))
```

And we're done with cells. I guess you know what comes next…

LSTM Layer

The `nn.LSTM` layer takes care of the hidden and cell states handling for us, no matter how long the input sequence is. We've been through that once with the RNN layer and then again with the GRU layer. The arguments, inputs, and outputs of the LSTM are **almost** exactly the same as the GRU, except for the fact that, as you already know, **LSTMs return two states (hidden and cell) with the same shape** instead of one. By the way, you can create **stacked LSTMs** and **bidirectional LSTMs** too.

So, let's go straight to **creating a model** using a Long Short-Term Memory.

Square Model III - The Sorcerer

This model is pretty much the same as the original "Square Model", except for **two differences**: its recurrent neural network is not a plain RNN anymore, but an LSTM, and it produces two states as output instead of one. Everything else stays exactly the same.

Model Configuration

```
class SquareModelLSTM(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModelLSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        self.cell = None                                      ②
        # Simple LSTM
        self.basic_rnn = nn.LSTM(self.n_features,
                                 self.hidden_dim,
                                 batch_first=True)            ①
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        # final cell state is (1, N, H)
        batch_first_output, (self.hidden, self.cell) = \
                                           self.basic_rnn(X)   ②

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

① First change: from RNN to LSTM

② Second change: including the **cell state** as output

Model Configuration & Training

Model Configuration

```
torch.manual_seed(21)
model = SquareModelLSTM(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_lstm = StepByStep(model, loss, optimizer)
sbs_lstm.set_loaders(train_loader, test_loader)
sbs_lstm.train(100)
```

```
fig = sbs_lstm.plot_losses()
```

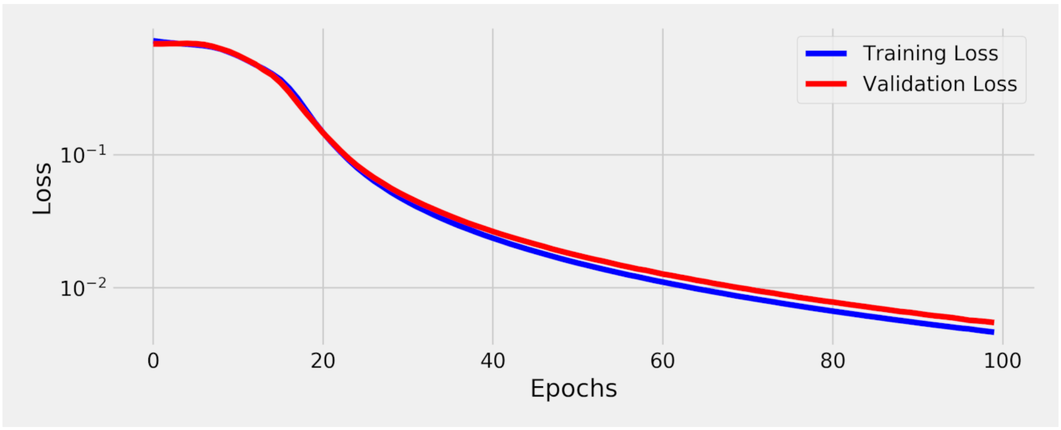


Figure 8.24 - Losses - `SquareModelLSTM`

```
StepByStep.loader_apply(test_loader, sbs_lstm.correct)
```

Output

```
tensor([[53, 53],
        [75, 75]])
```

And that’s 100% accuracy again!

Visualizing the Hidden States

Once again, if we use the "*perfect*" square as input to our latest trained model, that’s what the **final hidden states** look like for each one of the eight sequences (plotted sided-by-side with the previous models for easier comparison):

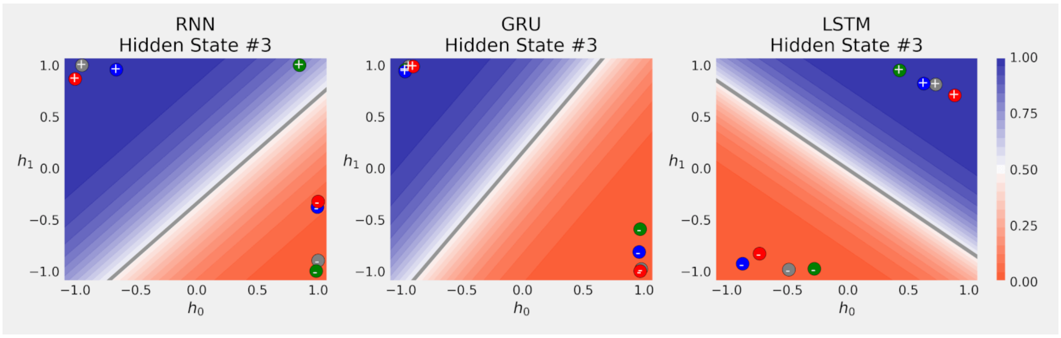


Figure 8.25 - Final hidden states for eight sequences of the "perfect" square

The LSTM model achieves a difference that’s *not necessarily better* than the GRU. What about the actual sequences?

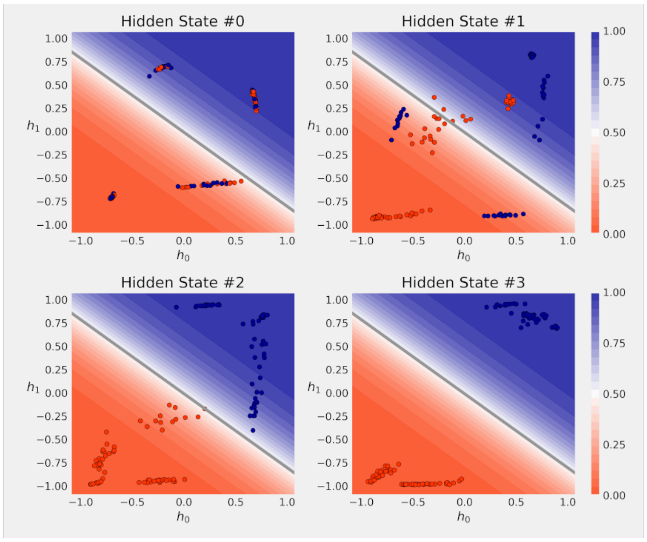


Figure 8.26 - Sequence of hidden states

Like the GRU, the LSTM also presents **four distinct groups** of sequences corresponding to the different starting corners. Moreover, it is able to classify most sequences correctly after seeing **only three** points.

Variable-Length Sequences

So far we've been working with full, regular, sequences of four data points each, and that's nice. But what do you do if you get **variable-length sequences**, like the ones below:

```
x0 = points[0]      # 4 data points
x1 = points[1][2:]  # 2 data points
x2 = points[2][1:]  # 3 data points

x0.shape, x1.shape, x2.shape
```

Output

```
((4, 2), (2, 2), (3, 2))
```

The answer: you **pad** them!

> *"Could you please remind me again what **padding** is?"*

Sure! Padding means **stuffing with zeros**. We've seen *padding* in Chapter 5 already: we used it to *stuff* an image with zeros around it in order to preserve its original size after being convolved.

Padding

Now we'll **stuff sequences with zeros** so they all have **matching sizes**. Simple enough, right?

> *"OK, it is simple, but **why** are we doing it?"*

We need to pad the sequences because we **cannot create a tensor** out of a list of **elements with different sizes**:

```
all_seqs = [s0, s1, s2]
torch.as_tensor(all_seqs)
```

Output

```
-----------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-154-9b17f363443c> in <module>
----> 1 torch.as_tensor([x0, x1, x2])

ValueError: expected sequence of length 4 at dim 1 (got 2)
```

We can use PyTorch's `nn.utils.rnn.pad_sequence` to perform the padding for us. It takes as arguments a **list of sequences**, a **padding value** (default is zero), and the option to make the result **batch first**. Let's give it a try:

```
seq_tensors = [torch.as_tensor(seq).float() for seq in all_seqs]
padded = rnn_utils.pad_sequence(seq_tensors, batch_first=True)
padded
```

Output

```
tensor([[[ 1.0349,  0.9661],
         [ 0.8055, -0.9169],
         [-0.8251, -0.9499],
         [-0.8670,  0.9342]],

        [[-1.0911,  0.9254],
         [-1.0771, -1.0414],
         [ 0.0000,  0.0000],
         [ 0.0000,  0.0000]],

        [[-1.1247, -0.9683],
         [ 0.8182, -0.9944],
         [ 1.0081,  0.7680],
         [ 0.0000,  0.0000]]])
```

Both 2nd and 3rd sequences were shorter than the first, so they got **padded** accordingly to **match the length of the longest sequence**.

Now we can proceed as usual and feed the **padded sequences** to an RNN and look at the results:

```
torch.manual_seed(11)
rnn = nn.RNN(2, 2, batch_first=True)
```

```
output_padded, hidden_padded = rnn(padded)
output_padded
```

Output

```
tensor([[[-0.6388,  0.8505],
         [-0.4215,  0.8979],
         [ 0.3792,  0.3432],
         [ 0.3161, -0.1675]],

        [[ 0.2911, -0.1811],
         [ 0.3051,  0.7055],
         [ 0.0052,  0.5819],
         [-0.0642,  0.6012]],

        [[ 0.3385,  0.5927],
         [-0.3875,  0.9422],
         [-0.4832,  0.6595],
         [-0.1007,  0.5349]]], grad_fn=<PermuteBackward>)
```

Since the sequences were padded to **four data points** each, we got **four hidden states** for each sequence as output.

"*How come the **hidden states** for the **padded points** are **different** from the hidden state of the **last real data point**?*"

Even though each padded point is just a bunch of zeros, it doesn't mean it won't change the hidden state. The hidden state itself gets transformed and, even if the padded point is full of zeros, its corresponding transformation may include a bias term that gets added nonetheless. This isn't *necessarily* a problem but, if you don't like padded points modifying your hidden state, you can prevent that by **packing the sequence** instead.

Before moving on to packed sequences, though, let's just check the (permuted, batch-first) final hidden state:

```
hidden_padded.permute(1, 0, 2)
```

Output

```
tensor([[[ 0.3161, -0.1675]],

        [[-0.0642,  0.6012]],

        [[-0.1007,  0.5349]]], grad_fn=<PermuteBackward>)
```

Packing

Packing works like a **concatenation** of sequences: instead of padding them to have equal-length elements, it **lines the sequences up**, one after the other and it **keeps track of the lengths**, so it knows **the indices corresponding to the start of each sequence**.

Let's work through an example using PyTorch's `nn.utils.rnn.pack_sequence`. First, it takes a **list of tensors** as input. If your list **is not sorted by decreasing sequence length**, you'll need to set its `enforce_sorted` argument to `False`.

Sorting the sequences by their lengths is **only** necessary if you're planning on **exporting your model** using the **ONNX** format, which allows you to import the model in different frameworks.

```
packed = rnn_utils.pack_sequence(seq_tensors, enforce_sorted=False)
packed
```

Output

```
PackedSequence(data=tensor([[ 1.0349,  0.9661],
        [-1.1247, -0.9683],
        [-1.0911,  0.9254],
        [ 0.8055, -0.9169],
        [ 0.8182, -0.9944],
        [-1.0771, -1.0414],
        [-0.8251, -0.9499],
        [ 1.0081,  0.7680],
        [-0.8670,  0.9342]]), batch_sizes=tensor([3, 3, 2, 1]),
sorted_indices=tensor([0, 2, 1]), unsorted_indices=tensor([0, 2,
1]))
```

The output is a bit *cryptic*, to say the least. Let’s decipher it, piece-by-piece, starting with the `unsorted_indices_tensor`. Even though we didn’t sort the list ourselves, PyTorch did it internally, and it found out that the **longest sequence** is the **first** (four data points, index 0), followed by the third (three data points, index 2), and by the shortest one (two data points, index 1).

Once the sequences are listed in order of decreasing length, like in Figure 8.27, the **number of sequences** that are **at least *t* steps long** (corresponding to the number of columns in the figure above) is given by the `batch_sizes` attribute:

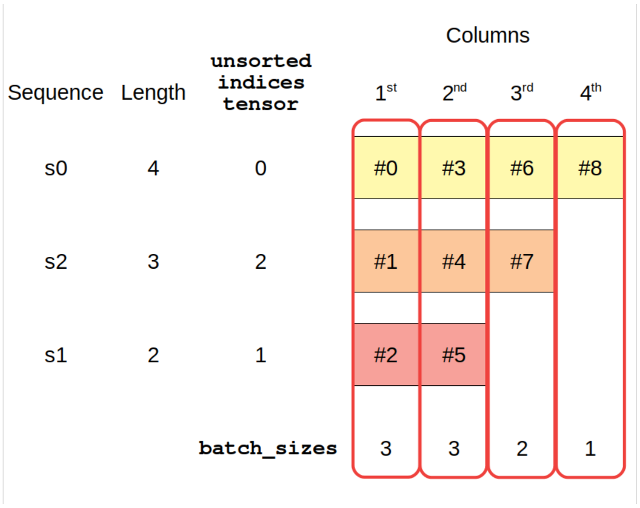


Figure 8.27 - Packing sequences

For example, the `batch_size` for the **third** column is **two** because **two sequences** have **at least three data points**. Then, it goes through the data points in the same **column-wise fashion**, from top-to-bottom, and from left-to-right, to assign the data points to the corresponding **indices** in the data tensor.

Finally, it uses these indices to **assemble the `data` tensor**:

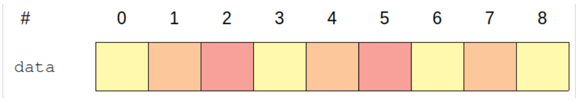


Figure 8.28 - Packed data points

Thus, to retrieve the values of the original sequences, we need to slice the `data` tensor accordingly. For example, we can retrieve the first sequence from the `data` tensor by reading the values from its corresponding indices: 0, 3, 6, and 8.

```
(packed.data[[0, 3, 6, 8]] == seq_tensors[0]).all()
```

Output

```
tensor(True)
```

Once the sequence is properly **packed**, we can feed it directly to an RNN:

```
output_packed, hidden_packed = rnn(packed)
output_packed, hidden_packed
```

Output

```
(PackedSequence(data=tensor([[-0.6388,  0.8505],
         [ 0.3385,  0.5927],
         [ 0.2911, -0.1811],
         [-0.4215,  0.8979],
         [-0.3875,  0.9422],
         [ 0.3051,  0.7055],
         [ 0.3792,  0.3432],
         [-0.4832,  0.6595],
         [ 0.3161, -0.1675]], grad_fn=<CatBackward>), batch_sizes
=tensor([3, 3, 2, 1]), sorted_indices=tensor([0, 2, 1]),
unsorted_indices=tensor([0, 2, 1])),
 tensor([[[ 0.3161, -0.1675],
          [ 0.3051,  0.7055],
          [-0.4832,  0.6595]]], grad_fn=<IndexSelectBackward>))
```

If the **input is packed**, the **output tensor is packed too**, but the **hidden state is not**.

Let's compare both final hidden states, from **padded** and **packed** sequences:

```
hidden_packed == hidden_padded
```

Output

```
tensor([[[ True,  True],
         [False, False],
         [False, False]]])
```

From three sequences, only one matches. Well, this shouldn't be a surprise, after all, we're **packing sequences to avoid updating the hidden state with padded inputs**.

"Cool, so I can use the permuted hidden state, right?"

Well, it depends:

- yes, if you're using networks that are **not bidirectional** - the final hidden state does match the last output
- no, if you're using a **bidirectional** network - you're **only** getting the **properly aligned hidden states** in the **last output**, so you'll need to **unpack it**

To unpack the actual **sequence of hidden states** for the shortest sequence, for example, we *could* get the corresponding indices from the `data` tensor in the packed output:

```
output_packed.data[[2, 5]] # x1 sequence
```

Output

```
tensor([[ 0.2911, -0.1811],
        [ 0.3051,  0.7055]], grad_fn=<IndexBackward>)
```

But that would be *extremely annoying* so, no, you *don't have to.*

Unpacking (to padded)

You can **unpack a sequence** using PyTorch's `nn.utils.rnn.pad_packed_sequence`. The name does not help, I know, I would rather call it `unpack_sequence_to_padded` instead. Anyway, we can use it to transform our **packed output** into a **regular, yet padded, output**:

```
output_unpacked, seq_sizes = \
    rnn_utils.pad_packed_sequence(output_packed, batch_first=True)
output_unpacked, seq_sizes
```

Output

```
(tensor([[[-0.6388,  0.8505],
          [-0.4215,  0.8979],
          [ 0.3792,  0.3432],
          [ 0.3161, -0.1675]],

         [[ 0.2911, -0.1811],
          [ 0.3051,  0.7055],
          [ 0.0000,  0.0000],
          [ 0.0000,  0.0000]],

         [[ 0.3385,  0.5927],
          [-0.3875,  0.9422],
          [-0.4832,  0.6595],
          [ 0.0000,  0.0000]]], grad_fn=<IndexSelectBackward>),
 tensor([4, 2, 3]))
```

It returns both the **padded sequences** and the **original sizes**, which will be useful as well.

*"Problem solved then? Can I take the **last output** now?"*

Almost there… if you take the last output the same way we did before, you'll **still get some padded zeros** back:

```
output_unpacked[:, -1]
```

Output

```
tensor([[ 0.3161, -0.1675],
        [ 0.0000,  0.0000],
        [ 0.0000,  0.0000]], grad_fn=<SelectBackward>)
```

So, to **actually get the last output** we need to use some **fancy indexing** and the information about **original sizes** returned by `pad_packed_sequence`:

```
seq_idx = torch.arange(seq_sizes.size(0))
output_unpacked[seq_idx, seq_sizes-1]
```

Output

```
tensor([[ 0.3161, -0.1675],
        [ 0.3051,  0.7055],
        [-0.4832,  0.6595]], grad_fn=<IndexBackward>)
```

And we finally have the **last output** for each *packed sequence*, even if we're using a bidirectional network.

Packing (from padded)

You can also convert an *already padded* sequence into a **packed sequence** using PyTorch's `nn.utils.rnn.pack_padded_sequence`. Since the sequence is already padded, though, we need to compute the **original sizes** ourselves:

```
len_seqs = [len(seq) for seq in all_seqs]
len_seqs
```

Output

```
[4, 2, 3]
```

And then pass them as an argument:

```
packed = rnn_utils.pack_padded_sequence(padded, len_seqs,
                                        enforce_sorted=False,
                                        batch_first=True)
packed
```

Output

```
PackedSequence(data=tensor([[ 1.0349,  0.9661],
        [-1.1247, -0.9683],
        [-1.0911,  0.9254],
        [ 0.8055, -0.9169],
        [ 0.8182, -0.9944],
        [-1.0771, -1.0414],
        [-0.8251, -0.9499],
        [ 1.0081,  0.7680],
        [-0.8670,  0.9342]]), batch_sizes=tensor([3, 3, 2, 1]),
sorted_indices=tensor([0, 2, 1]), unsorted_indices=tensor([0, 2,
1]))
```

Variable-Length Dataset

Let’s create a dataset with **variable-length sequences** and train a model using it:

```
var_points, var_directions = generate_sequences(variable_len=True)
var_points[:2]
```

Output

```
[array([[ 1.12636495,  1.1570899 ],
        [ 0.87384513, -1.00750892],
        [-0.9149893 , -1.09150317],
        [-1.0867348 ,  1.07731667]]), array([[ 0.92250954,
-0.89887678],
        [ 1.0941646 ,  0.92300589]])]
```

Data Preparation

We simply **cannot use a `TensorDataset`** because we cannot create a tensor out of a list of **elements with different sizes**.

So, we build a **custom dataset** that **makes a tensor out of each sequence** and, when prompted for a given item, returns the corresponding tensor and associated label:

Data Preparation

```
class CustomDataset(Dataset):
    def __init__(self, x, y):
        self.x = [torch.as_tensor(s).float() for s in x]
        self.y = torch.as_tensor(y).float().view(-1, 1)

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

train_var_data = CustomDataset(var_points, var_directions)
```

But this is *not enough*… if we create a **data loader** for our custom dataset and try to **retrieve a mini-batch** out of it, it will **raise an error**:

Data Preparation

```
train_var_loader = DataLoader(train_var_data, batch_size=16,
shuffle=True)
next(iter(train_var_loader))
```

Output

```
---------------------------------------------------------------------
RuntimeError                          Traceback (most recent call last)
<ipython-input-34-596b8081f8d1> in <module>
      1 train_var_loader = DataLoader(train_var_data, batch_size=16,
shuffle=True)
----> 2 next(iter(train_var_loader))
...
RuntimeError: stack expects each tensor to be equal size, but got [
3, 2] at entry 0 and [4, 2] at entry 2
```

It turns out, the data loader is **trying to `stack` together** the sequences which, as we know, have **different sizes** and thus **cannot be stacked together**.

We *could* simply **pad** all the sequences and move on with a **`TensorDataset` and regular data loader**. But, in that case, the final hidden states would be affected by the padded data points, as we've already discussed.

We can do *better* than that: we can **pack** our mini-batches using a **collate function**.

Collate Function

The *collate function* takes a **list of tuples** (sampled from a dataset using its `__get_item__`) and **collates** them together **into a batch** that's being returned by the data loader. It gives you the ability to **manipulate the sampled data points** in any way you want to make them into a **mini-batch**.

In our case, we'd like to get all sequences (the first item in every tuple) and *pack them*. Besides, we can get all labels (the second item in every tuple) and make them into a tensor that's in the correct shape for our binary classification task:

Data Preparation

```
def pack_collate(batch):
    X = [item[0] for item in batch]
    y = [item[1] for item in batch]
    X_pack = rnn_utils.pack_sequence(X, enforce_sorted=False)

    return X_pack, torch.as_tensor(y).view(-1, 1)
```

Let's see the function in action by creating a dummy batch of two elements and applying the function to it:

```
# list of tuples returned by the dataset
dummy_batch = [train_var_data[0], train_var_data[1]]
dummy_x, dummy_y = pack_collate(dummy_batch)
dummy_x
```

Output

```
PackedSequence(data=tensor([[ 1.1264,  1.1571],
        [ 0.9225, -0.8989],
        [ 0.8738, -1.0075],
        [ 1.0942,  0.9230],
        [-0.9150, -1.0915],
        [-1.0867,  1.0773]]), batch_sizes=tensor([2, 2, 1, 1]),
sorted_indices=tensor([0, 1]), unsorted_indices=tensor([0, 1]))
```

Two sequences of different sizes go in, one packed sequence comes out. Now we can create a **data loader** that **uses our collate function**:

Data Preparation

```
train_var_loader = DataLoader(train_var_data,
                              batch_size=16,
                              shuffle=True,
                              collate_fn=pack_collate)
x_batch, y_batch = next(iter(train_var_loader))
```

And now every batch coming out of our data loader has a packed sequence.

"Do I have to change the model too?"

Square Model IV - Packed

There are *some* changes we need to make to the model. Let's illustrate them by creating a model that uses a **bidirectional LSTM** and expects **packed sequences as inputs**.

First, since `X` is a packed sequence now, it means that the **output is packed**, and therefore we need to **unpack it** to a **padded output**.

Once it is unpacked, we can get the **last output** using the **fancier indexing** to get the last (actual) element of the padded sequences. Moreover, using a bidirectional LSTM means that the output for each sequence has an **(N, 1, 2*H)** shape.

Model Configuration

```
class SquareModelPacked(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModelPacked, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        self.cell = None
        # Simple LSTM
        self.basic_rnn = nn.LSTM(self.n_features,
                                 self.hidden_dim,
                                 bidirectional=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(2 * self.hidden_dim,
                                    self.n_outputs)         ③

    def forward(self, X):
        # X is a PACKED sequence now
        # final hidden state is (2, N, H) - bidirectional
        # final cell state is (2, N, H) - bidirectional
        rnn_out, (self.hidden, self.cell) = self.basic_rnn(X)
        # unpack the output (N, L, 2*H)
        batch_first_output, seq_sizes = \
            rnn_utils.pad_packed_sequence(rnn_out,
                                          batch_first=True)   ①

        # only last item in sequence (N, 1, 2*H)
        seq_idx = torch.arange(seq_sizes.size(0))
        last_output = batch_first_output[seq_idx, seq_sizes-1]②
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

① Unpacking the output

② Fancy indexing to retrieve the last output of a padded sequence

③ Two hidden states are concatenated side-by-side in a bidirectional network

Model Configuration & Training

We can use our data loader that outputs packed sequences (`train_var_loader`) to feed our `SquareModelPacked` model and train it in the usual way:

Model Configuration

```
torch.manual_seed(21)
model = SquareModelPacked(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_packed = StepByStep(model, loss, optimizer)
sbs_packed.set_loaders(train_var_loader)
sbs_packed.train(100)
```

```
fig = sbs_packed.plot_losses()
```

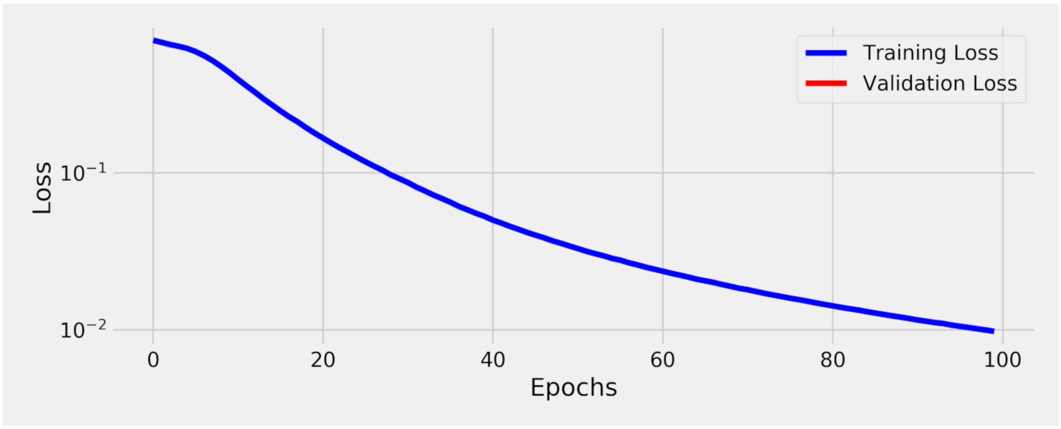


Figure 8.29 - Losses - `SquareModelPacked`

```
StepByStep.loader_apply(train_var_loader, sbs_packed.correct)
```

Output

```
tensor([[66, 66],
        [62, 62]])
```

1D Convolutions

In Chapter 5, we learned about **convolutions**, their **kernels** and **filters**, and how to perform a **convolution** by **repeatedly applying a filter** to a **moving region** over the image. Those were **2D convolutions**, though, meaning that the **filter was moving in two dimensions**, both along the width (left to right) and the height (top to bottom) of the image.

Guess what **1D convolutions** do? They **move the filter** in **one dimension**, from left to right. The filter works like a **moving window**, performing a **weighted sum of the values in the region it has moved over**. Let's use a sequence of temperature values over 13 days as an example:

```
temperatures = np.array([5, 11, 15, 6, 5, 3, 3, 0, 0, 3, 4, 2, 1])
```

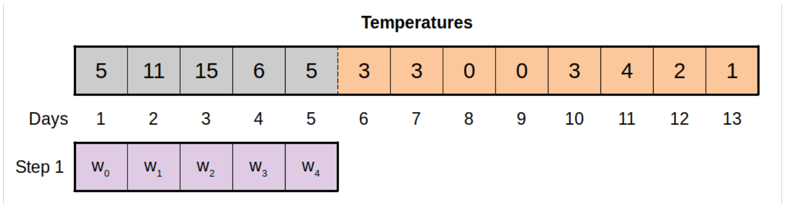


Figure 8.30 - Moving window over series of temperatures

Then, let's use a **window (filter)** of size **five**, like in the figure above. In its first step, the window is over days one to five. In the next step, since it can only move to the right, it will be over days two to six. By the way, the **size of our movement to the right** is, once again, the **stride**.

Now, let's assign the same value (`0.2`) for every **weight** in our **filter** and use PyTorch's `F.conv1d` to **convolve the filter with our sequence** (don't mind the shape just yet, we'll get back to it in the next section):

```
size = 5
weight = torch.ones(size) * 0.2
F.conv1d(torch.as_tensor(temperatures).float().view(1, 1, -1),
         weight=weight.view(1, 1, -1))
```

Output

```
tensor([[[8.4000, 8.0000, 6.4000, 3.4000, 2.2000,
          1.8000, 2.0000, 1.8000, 2.0000]]])
```

Does it look familiar? That's a **moving average**, just like those we used in Chapter 6.

"Does it mean every 1D convolution is a moving average?"

Well, kinda… in the functional form above we had to provide the weights but, as expected, the corresponding **module** (`nn.Conv1d`), will **learn the weights** itself. Since there is no requirement that the weights must add up to one, it won't be a moving average but a **moving weighted sum**.

Moreover, it is very unlikely we'll use it over a **single feature** like in the example above. Things get **more interesting** as we include **more features** to be convolved with the **filter**, which brings us to the next topic…

Shapes

The *shapes* topic, **one more time**, I know… unfortunately, there is no escape from it. In Chapter 4 we discussed the **NCHW** shape for images:

- **N** stands for the **N**umber of images (in a mini-batch, for instance)
- **C** stands for the number of **C**hannels (or **filters**) in each image
- **H** stands for each image's **H**eight
- **W** stands for each image's **W**idth

For **sequences**, though, the shape should be **NCL**:

- **N** stands for the **N**umber of sequences (in a mini-batch, for instance)
- **C** stands for the number of **C**hannels (or **filters**) in each element of the sequence
- **L** stands for the **L**ength of each sequence

"*Wait, where is the **number of features** in it?*"

Good point! Since 1D convolutions only move along the sequence, **each feature is considered an input channel**. So, you can think of the shape as **NFL** or **N(C/F)L** if you like.

"*Really?! Yet **another** shape for input sequences?*"

Unfortunately, yes. But I've built this small table to help you wrap your head around the different shape conventions while working with sequences as inputs:

	Shape	Use Case
Batch-first	N, L, F	Typical shape; RNNs with `batch_first=True`
RNN-friendly	L, N, F	Default for RNNs (`batch_first=False`)
Sequence-last	N, F, L	Default for 1D convolutions

Having (hopefully) cleared that out, let's use **permute** to get our sequences in the appropriate shape:

```
seqs = torch.as_tensor(points).float() # N, L, F
seqs_length_last = seqs.permute(0, 2, 1)
seqs_length_last.shape # N, F=C, L
```

Output

```
torch.Size([128, 2, 4])
```

Multiple Features or Channels

Our sequences of corners have **two coordinates**, that is, **two features**. These will be considered (input) **channels** as far as the **1D convolution** is concerned, so we create the convolutional layer accordingly:

```
torch.manual_seed(17)
conv_seq = nn.Conv1d(in_channels=2, out_channels=1,
                     kernel_size=2, bias=False)
conv_seq.weight, conv_seq.weight.shape
```

Output

```
(Parameter containing:
 tensor([[[-0.0658,  0.0351],
          [ 0.3302, -0.3761]]], requires_grad=True),
torch.Size([1, 2, 2]))
```

We're using only **one output channel**, so there is only **one filter**, which will produce **one output value** for each region the window moves over. Since the **kernel size** is **two**, each window will move over **two corners**. Any **two corners make an edge** of the square, so there will be **one output for each edge**. This information will be useful for visualizing what the model is actually doing.

Since each channel (feature) will be multiplied, element-wise, by its corresponding weights in the filter, and **all values for all channels are added up** to produce a **single value** anyway, I've chosen to represent the sequence (and the filter) as if it were two-dimensional in the figure below:

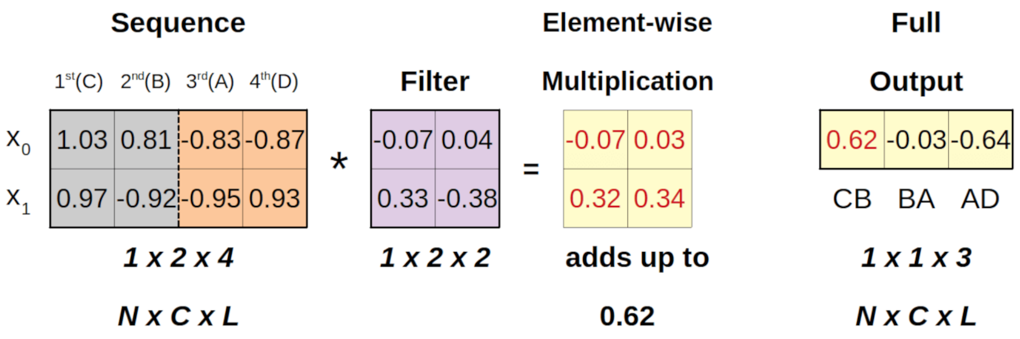


Figure 8.31 - Applying filter over a sequence

Our first sequence corresponds to corners CBAD and, for the first region (in gray, corresponding to the CB edge), it results in an output of `0.6241`. Let's use our convolutional layer to get all outputs:

```
conv_seq(seqs_length_last[0:1])
```

Output

```
tensor([[[ 0.6241, -0.0274, -0.6412]]], grad_fn=<SqueezeBackward1>)
```

The resulting **shape** is given by the formula below, where *l* is the *length of the sequence*, *f* is the *filter size*, *p* is the *padding*, and *s* is the *stride*:

$$l_i * f = \frac{(l_i + 2p) - f}{s} + 1$$

Equation 8.15 - Resulting shape

If any of the resulting dimensions is not an integer, it must be **rounded down**.

Dilation

There is yet another operation that can be used with convolutions in any number of dimensions, but that we haven't discussed yet: **dilation**. The general idea is quite simple: instead of a **contiguous** kernel, it uses a **dilated kernel**. A **dilation of size two**, for instance, means that the kernel uses **every other element** (be it a pixel or a feature value in a sequence).

"*Why would I want to do that?*"

In a nutshell, the idea is to capture long-term properties of a sequence (like seasonality in a time series, for example) or integrate information from different scales in an image (local and global context). We're not delving deeper than explaining the mechanism itself, though.

In our example, a **kernel of size two** (so it goes over **two values** in the sequence) with a **dilation of two** (so it **skips every other value** in the sequence) works like this:

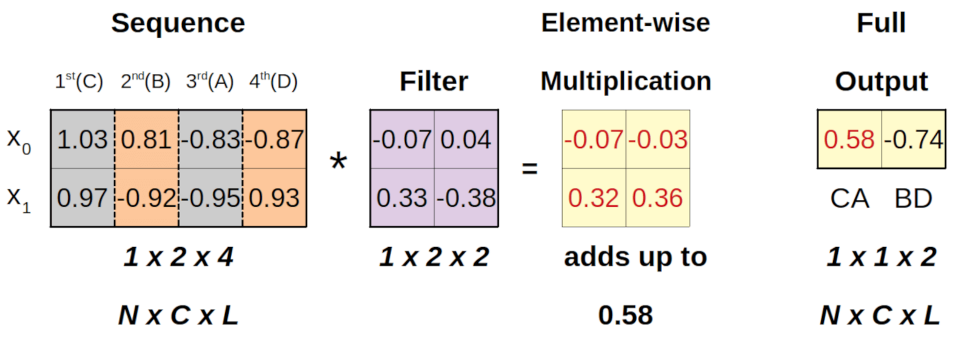


Figure 8.32 - Applying dilated filter over a sequence

The first **dilated region** (in gray) is given by the **first and third** values (corners C and A) which, convolved with the filter, will output a value of `0.58`. Then, the next **dilated region** will be given by the **second and fourth** values (corners B and D). Now, the convolutional layer has a `dilation` argument as well:

```
torch.manual_seed(17)
conv_dilated = nn.Conv1d(in_channels=2, out_channels=1,
                         kernel_size=2, dilation=2, bias=False)
conv_dilated.weight, conv_dilated.weight.shape
```

Output

```
(Parameter containing:
 tensor([[[-0.0658,  0.0351],
          [ 0.3302, -0.3761]]], requires_grad=True),
torch.Size([1, 2, 2]))
```

If we run our sequence through it, we get the same output as depicted in the figure above.

```
conv_dilated(seqs_length_last[0:1])
```

Output

```
tensor([[[ 0.5793, -0.7376]]], grad_fn=<SqueezeBackward1>)
```

This output is **smaller** than the previous one because the **dilation affects the shape of the output** according to the formula below (*d* stands for *dilation size*):

$$l_i * f = \frac{(l_i + 2p) - d(f-1) - 1}{s} + 1$$

Equation 8.16 - Resulting shape with dilation

If any of the resulting dimensions is not an integer, it must be **rounded down**.

Data Preparation

The data preparation step is very much like the others except for the fact that we need to **permute** the dimensions to comply with 1D convolution's "*sequence last*" (NFL) shape:

Data Preparation

```
train_data = TensorDataset(
    torch.as_tensor(points).float().permute(0, 2, 1),
    torch.as_tensor(directions).view(-1, 1).float()
)
test_data = TensorDataset(
    torch.as_tensor(test_points).float().permute(0, 2, 1),
    torch.as_tensor(test_directions).view(-1, 1).float()
)
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
test_loader = DataLoader(test_data, batch_size=16)
```

Model Configuration & Training

The model is quite simple: a single `Conv1d` layer followed by an activation function (ReLU), a flattening layer (which is only *squeezing* the channel dimension out), and a linear layer to combine the outputs (three values for each sequence, as shown in Figure 8.31) into logits for our binary classification.

Model Configuration

```
torch.manual_seed(21)
model = nn.Sequential()
model.add_module('conv1d', nn.Conv1d(in_channels=2,
                                     out_channels=1,
                                     kernel_size=2))
model.add_module('relu', nn.ReLU())
model.add_module('flatten', nn.Flatten())
model.add_module('output', nn.Linear(3, 1))

loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_conv1 = StepByStep(model, loss, optimizer)
sbs_conv1.set_loaders(train_loader, test_loader)
sbs_conv1.train(100)
```

```
fig = sbs_conv1.plot_losses()
```

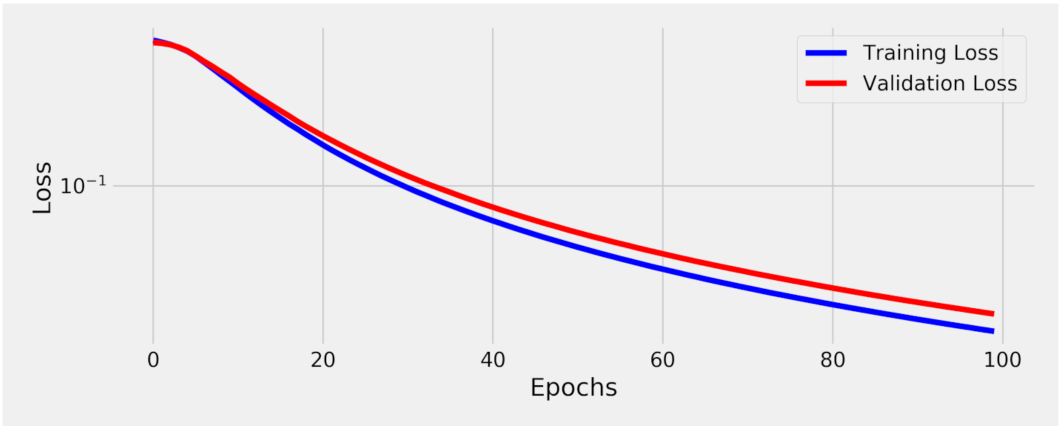


Figure 8.33 - Losses - The Edge Model

```
StepByStep.loader_apply(test_loader, sbs_conv1.correct)
```

Output

```
tensor([[53, 53],
        [75, 75]])
```

Once again, the model's accuracy is perfect. Maybe you've noticed it was much *faster* to train too. How come this simple model performed so well? Let's try to figure out what it did under the hood…

Visualizing the Model

The key component of our model is the `nn.Conv1d` layer, so let's take a look at its `state_dict`:

```
model.conv1d.state_dict()
```

Output

```
OrderedDict([('weight', tensor([[[-0.2186,  2.3289],
                                 [-2.3765, -0.1814]]], device='cuda:0')),
             ('bias', tensor([-0.5457], device='cuda:0'))])
```

Then, let's see what this filter is doing by feeding it a "*perfect*" square starting at corner **A** and going **counter-clockwise** (sequence ADCB):

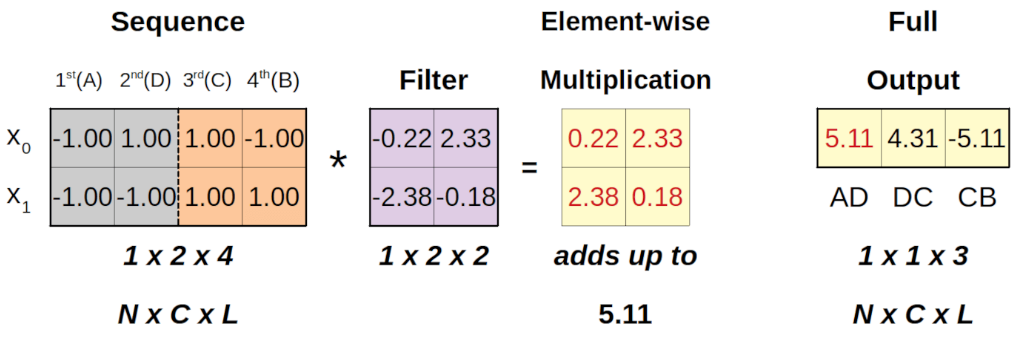


Figure 8.34 - Applying filter over the "perfect" square

The figure above shows us the element-wise multiplication and result for the **first region**, corresponding to the **AD edge** of our square (without including the *bias* value from the convolutional layer). The outputs on the right are the "*edge features*".

We can actually find the expression to compute them as a **weighted sum of the coordinates** for both **first** (x^{1st}) and **second** (x^{2nd}) corners included in the region being convolved:

$$edge\ \ feature = -0.22\ \ x_0^{1st} - 2.38\ \ x_1^{1st} + 2.33\ \ x_0^{2nd} - 0.18\ \ x_1^{2nd} - 0.5457$$

Equation 8.17 - Equation for "edge feature"

From the expression above, and given that the coordinates values are close to one (in absolute value), the only way for the **edge feature** to have a **positive value** is for x^{1st}_1 and x^{2nd}_0 to be approximately -1 and 1, respectively. This is the case for **two**

edges only, **AD** and **DC**:

$$\overline{AD}\ or\ \overline{DC} \implies x_1^{1st} \approx -1\ and\ x_0^{2nd} \approx 1 \implies edge\ \ feature > 0$$

Equation 8.18 - Detected edges

Every other edge will return a **negative value** and thus be clipped at **zero** by the ReLU activation function. Our model learned to **choose two edges with the same direction** to perform the classification.

*"Why **two** edges? Shouldn't a **single** edge suffice?"*

It *should* if our sequences actually **had four edges**… but they **don't**. We *do* have **four corners** but we can only build **three edges out of it** because we're missing the edge connecting the last and the first corners. So, any model that relies on a single edge will likely fail in those cases where that particular edge is the missing one. Thus, it needs to correctly classify at least two edges.

Putting It All Together

In this chapter we've used different **recurrent neural networks**, plain vanilla RNNs, GRUs, and LSTMs, to produce a **hidden state representing each sequence** that can be used for **sequence classification**. We used both **fixed** and **variable-length** sequences, **padding** or **packing** them with the help of a **collate function**, and built models that **ensured the right shape** of the data.

Fixed-Length Dataset

For fixed-length sequences, the data preparation was the usual:

Data Generation & Preparation

```
points, directions = generate_sequences(n=128, seed=13)
train_data = TensorDataset(
    torch.as_tensor(points).float(),
    torch.as_tensor(directions).view(-1, 1).float()
)
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
```

Variable-Length Dataset

For variable-length sequences, though, we built a **custom dataset** and a **collate function** to **pack the sequences**:

Data Generation

```
var_points, var_directions = generate_sequences(variable_len=True)
```

Data Preparation

```
class CustomDataset(Dataset):
    def __init__(self, x, y):
        self.x = [torch.as_tensor(s).float() for s in x]
        self.y = torch.as_tensor(y).float().view(-1, 1)

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

train_var_data = CustomDataset(var_points, var_directions)
```

Data Preparation

```
def pack_collate(batch):
    X = [item[0] for item in batch]
    y = [item[1] for item in batch]
    X_pack = rnn_utils.pack_sequence(X, enforce_sorted=False)

    return X_pack, torch.as_tensor(y).view(-1, 1)

train_var_loader = DataLoader(train_var_data,
                              batch_size=16,
                              shuffle=True,
                              collate_fn=pack_collate)
```

There Can Be Only ONE... Model

We've developed many models along this chapter, depending both on the **type of recurrent layer** that was used (RNN, GRU, or LSTM) and on the **type of sequence** (packed or not). The model below, though, is able to handle different configurations:

- its `rnn_layer` argument allows you to use whichever recurrent layer you prefer
- the `**kwargs` allows you to further configure the recurrent layer (using `num_layers` and `bidirectional` arguments, for example)
- the **output dimension** of the recurrent layer is automatically computed to build a **matching linear layer**
- if the input is a **packed sequence**, it handles the **unpacking and fancy indexing** to retrieve the **actual last hidden state**

Model Configuration

```
class SquareModelOne(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs,
                 rnn_layer=nn.LSTM, **kwargs):
```

```
        super(SquareModelOne, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        self.cell = None
        self.basic_rnn = rnn_layer(self.n_features,
                                   self.hidden_dim,
                                   batch_first=True, **kwargs)
        output_dim = (self.basic_rnn.bidirectional + 1) * \
                     self.hidden_dim
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(output_dim, self.n_outputs)

    def forward(self, X):
        is_packed = isinstance(X, nn.utils.rnn.PackedSequence)
        # X is a PACKED sequence, there is no need to permute

        rnn_out, self.hidden = self.basic_rnn(X)
        if isinstance(self.basic_rnn, nn.LSTM):
            self.hidden, self.cell = self.hidden

        if is_packed:
            # unpack the output
            batch_first_output, seq_sizes = \
                rnn_utils.pad_packed_sequence(rnn_out,
                                              batch_first=True)
            seq_slice = torch.arange(seq_sizes.size(0))
        else:
            batch_first_output = rnn_out
            seq_sizes = 0 # so it is -1 as the last output
            seq_slice = slice(None, None, None) # same as ':'

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[seq_slice, seq_sizes-1]

        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

Model Configuration & Training

The model below uses a **bidirectional LSTM** and already achieves a 100% accuracy on the training set. Feel free to experiment with different recurrent layers, the number of layers, single or bidirectional, as well as switching between fixed and variable-length sequences.

Model Configuration

```
torch.manual_seed(21)
model = SquareModelOne(n_features=2, hidden_dim=2, n_outputs=1,
                       rnn_layer=nn.LSTM, num_layers=1,
                       bidirectional=True)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_one = StepByStep(model, loss, optimizer)
#sbs_one.set_loaders(train_loader)
sbs_one.set_loaders(train_var_loader)
sbs_one.train(100)
```

```
#StepByStep.loader_apply(train_loader, sbs_one.correct)
StepByStep.loader_apply(train_var_loader, sbs_one.correct)
```

Recap

In this chapter, we've learned about **sequential data** and how to use **recurrent neural networks** to perform a classification task. We followed the **journey of a hidden state** through all its transformations happening inside of different recurrent layers: RNN, GRU, and LSTM. We understood the difference between **padding** and **packing** variable-length sequences, and how to **build a data loader for packed sequences**. We also brought back **convolutions**, using its **one-dimensional** version to process sequential data as well. This is what we've covered:

- understanding the importance of **order** in **sequential data**
- generating a synthetic two-dimensional dataset so we can visualize what's happening "*under the hood*" of our models
- learning what a **hidden state** is
- understanding how a hidden state is **modified by a data point** inside an **RNN cell**
- disassembling an RNN cell into its components: **two linear layers** and an **activation function**
- understanding the **reasoning** behind using **hyperbolic tangent** as activation function of choice in RNNs
- learning that **data points are independently transformed** while the **hidden state is sequentially transformed**
- using an **RNN layer** to automatically handle hidden state inputs and outputs without having to **loop over a sequence**
- discussing the issue with the **shape** of the data and the difference between **typical batch-first (N, L, F)** and **sequence-first (L, N, F)** shapes
- figuring that **stacked RNNs** and **bidirectional RNNs** are simply different ways of composing **two or more simple RNNs** and **concatenating their outputs**
- training a **square model** to classify our sequences into **clockwise** or **counterclockwise** direction

- visualizing the **transformed inputs** and the **decision boundary** separating the **final hidden states**
- visualizing the **journey of a hidden state** through each and every transformation happening inside an RNN layer
- adding **gates** to the RNN cell and turning it into a **Gated Recurrent Unit** cell
- learning that **gates** are simply **vectors of values between zero and one**, one value for **each dimension of the hidden state**
- disassembling a GRU cell into its components to better understand its internal mechanics
- visualizing the **effect of using a gate** on the **hidden state**
- adding **another state** and **more gates** to the RNN cell, making it a **Long Short-Term Memory** cell
- disassembling an LSTM cell into its many components to better understand its internal mechanics
- generating **variable-length sequences**
- understanding the **issues with having tensors of different sizes** and using **padding** to make all sequences equal in length
- **packing** sequences as an alternative to padding, and understanding the **way the data is organized in a packed sequence**
- using a **collate function** to make a data loader **yield a mini-batch of your own assembling**
- learning about **1D convolutions** and how they can be used with **sequential data**
- discussing **yet another issue** with the **shape (N, C, L)** expected by these convolutions
- understanding that **features** are considered **channels** in these convolutions
- learning about **dilated convolutions**
- visualizing how a **convolutional model** learned to classify sequences based on the **edges** of the square

Congratulations! You took your **first step** (quite a *long* one, I might add, so give yourself a pat on the back for it!) towards **building models using sequential data**. You're now familiar with the inner workings of **recurrent layers** and you learned the **importance of the hidden state** as the **representation of a sequence**. Moreover, you got the **tools** to **put your sequences in shape** (I *had* to make this pun!).

In the next chapter, we'll build on top of all this (especially the *hidden state* part), and develop models to **generate sequences**. These models use an **encoder-decoder** architecture, and we can add all sorts of bells and whistles to it, like **attention mechanisms**, **positional encoding**, and much more!

[131] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter08.ipynb

[132] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter08.ipynb

Chapter 9 - Part I

Sequence-To-Sequence

Spoilers

In the first half of this chapter, we will:

- learn about the **encoder-decoder** architecture
- build and train models to predict a **target sequence** from a **source sequence**
- understand the **attention** mechanism and its components ("keys", "values", and "queries")
- build a **multi-headed attention** mechanism

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 9[133] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[134].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 9's notebook. If not, just click on `Chapter09.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import copy
import numpy as np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, random_split, \
    TensorDataset

from data_generation.square_sequences import generate_sequences
from stepbystep.v4 import StepByStep
```

Sequence-To-Sequence

Sequence-to-sequence problems are more complex than those we handled in the last chapter. There are **two sequences** now: the **source** and the **target** sequences. We use the former to predict the latter, and they may even have *different lengths*.

A typical example of a sequence-to-sequence problem is **translation**: a sentence goes in (a sequence of words in English), and another sentence comes out (a sequence of words in French). This problem can be tackled using an **encoder-decoder architecture**, first described in the *"Sequence to Sequence Learning with Neural Networks"*[135] paper by Sutskever, I., et al.

Translating languages is an obviously difficult task, so we're falling back to a much simpler problem to illustrate how the encoder-decoder architecture works.

Data Generation

We'll **keep drawing the same squares** as before, but this time we'll **draw the first two corners** ourselves (the **source sequence**) and ask our model to **predict the next two corners** (the **target sequence**). As with every sequence-related problem, the **order is important**, so it is not enough to get the corner's coordinates right, but they should **follow the same direction** (clockwise or counter-clockwise).

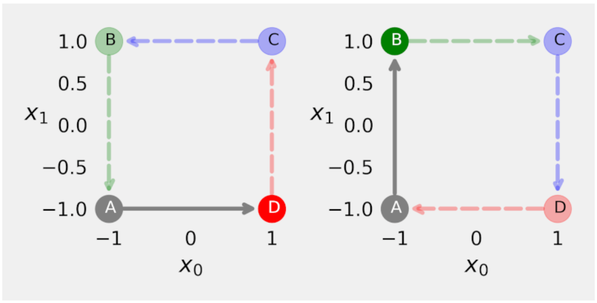


Figure 9.1 - Drawing first two corners, starting at A and moving towards either D or B

Since there are four corners to start from and two directions to follow, there are effectively eight possible sequences (solid colors indicate the corners in the source sequence, semi-transparent colors, the target sequence):

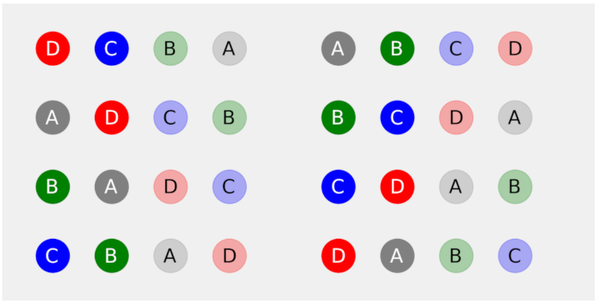


Figure 9.2 - Possible sequences of corners

Since the desired output of our model is a **sequence of coordinates (x_0, x_1)**, we're dealing with a **regression problem** now. Therefore, we'll be using a typical **mean squared error** loss to compare the **predicted and actual coordinates** for the **two points in the target sequence**.

Let's generate 128 random noisy squares:

Data Generation

```
points, directions = generate_sequences(n=128, seed=13)
```

And then let's visualize the first five squares (identical to those in Chapter 8):

```
fig = plot_data(points, directions, n_rows=1)
```

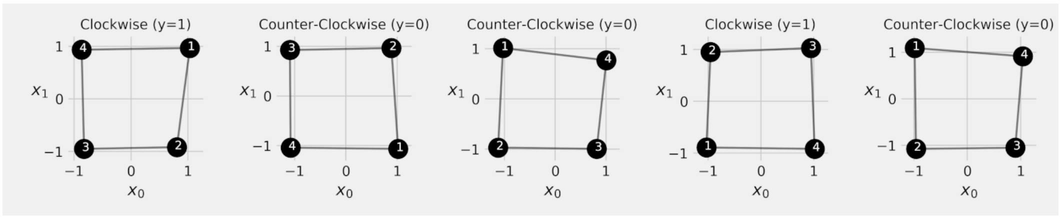


Figure 9.3 - Sequence dataset

The corners show the **order** in which they were drawn. In the first square, the drawing **started at the top-right corner**(corresponding to the *blue C* corner) and followed a **clockwise direction** (corresponding to the *CDAB* sequence). The **source sequence** for that square would include corners ***C* and *D* (1 and 2)**, while the **target sequence** would include corners ***A* and *B* (3 and 4)**, in that order.

In order to **output a sequence** we need a more complex architecture, we need an…

Encoder-Decoder Architecture

The encoder-decoder is a combination of **two models**: the **encoder** and the **decoder**.

Encoder

The **encoder's goal** is to **generate a representation** of the **source sequence**, that is, to **encode it**.

"*Wait, we've done that already, right?*"

Absolutely! That's what the **recurrent layers** did: they generated a **final hidden state** that was a **representation of the input sequence**. Now you know *why* I insisted *so much* on this idea and repeated it *over and over again* in Chapter 8 :-)

The figure below should look familiar: it is a **typical recurrent neural network** that we're using to **encode the source sequence**.

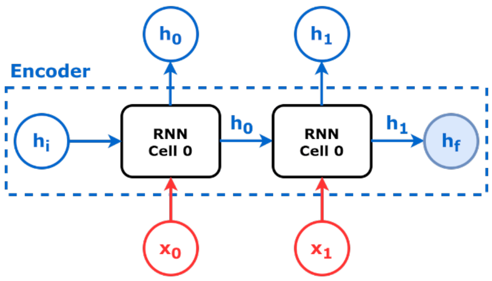


Figure 9.4 - Encoder

The **encoder** model is a slim version of our models from Chapter 8: it simply returns a **sequence of hidden states**.

Encoder

```python
class Encoder(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features,
                                self.hidden_dim,
                                batch_first=True)

    def forward(self, X):
        rnn_out, self.hidden = self.basic_rnn(X)

        return rnn_out # N, L, F
```

"*Don't we need only the **final hidden state**?*"

That's correct. We'll be using the **final hidden state only**… for now.

In the "*Attention*" section we'll be using **all hidden states**, and that's why we're implementing the encoder like that.

Let's go over a simple example of **encoding**: we start with a sequence of coordinates of a "*perfect*" square, and split it into **source and target sequences**:

```python
full_seq = (torch.tensor([[-1, -1], [-1, 1], [1, 1], [1, -1]])
            .float()
            .view(1, 4, 2))
source_seq = full_seq[:, :2] # first two corners
target_seq = full_seq[:, 2:] # last two corners
```

Now, let's **encode the source sequence** and take the **final hidden state**:

```
torch.manual_seed(21)
encoder = Encoder(n_features=2, hidden_dim=2)
hidden_seq = encoder(source_seq) # output is N, L, F
hidden_final = hidden_seq[:, -1:]   # takes last hidden state
hidden_final
```

Output

```
tensor([[[ 0.3105, -0.5263]]], grad_fn=<SliceBackward>)
```

Of course, the model is *untrained*, so the final hidden state above is totally *random*. In a **trained model**, however, the final hidden state will **encode information about the source sequence**. In Chapter 8, we used it to **classify the direction** in which the square was drawn, so it is safe to say that the **final hidden state encoded the drawing direction** (clockwise or counter-clockwise).

Pretty straightforward, right? Now, let's go over the…

Decoder

The **decoder's goal** is to **generate the target sequence** from **an initial representation**, that is, to **decode it**.

Sounds like a perfect match, doesn't it? **Encode the source sequence**, get its representation (**final hidden state**), and feed it to the **decoder** so it **generates the target sequence**.

"*How does the decoder transform a hidden state into a sequence?*"

We can use **recurrent layers** for that as well:

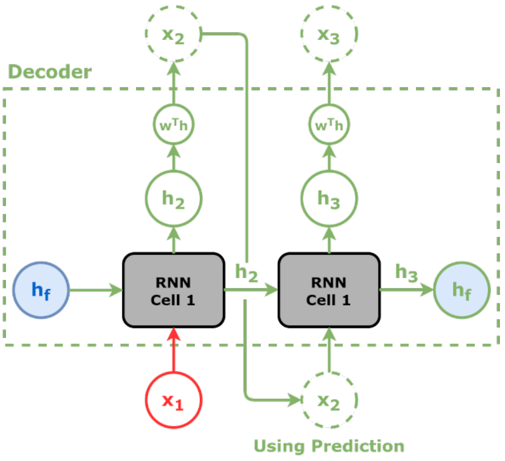


Figure 9.5 - Decoder

Let's analyze the figure above:

- in the **first step**, the **initial hidden state** is going to be the **encoder's final hidden state** (h_f, in blue)
- the first cell will **output** a **new hidden state** (h_2): that's both the **output of that cell** and **one of the inputs of the next cell**, as we've already seen in Chapter 8
- before, we'd only run the *final* hidden state through a *linear layer* to produce the logits, but now we'll **run the output of every cell through a linear layer** (w^Th) to **convert each hidden state into predicted coordinates** (x_2 and x_3)
- the **predicted coordinates** are then used as **one of the inputs of the second step** (x_2)

"*Great, but we're **missing one input in the first step**, right?*"

That's right! The **first cell** takes both an **initial hidden state** (h_f, in blue, the encoder's output) and a **first data point** (x_1, in red).

Let’s **pretend** for a moment that the encoder’s final hidden state (h_f) **encoded the direction of the drawing**. The decoder receives that information and, starting at its **first data point** (x_1), it follows the encoded direction to predict the coordinates of the next corner.

Of course, this is just a gross simplification for the sake of developing intuition. The encoded information is more complex than that.

In our case, the decoder’s first data point is actually the **last data point in the source sequence** because the **target sequence** is *not* a new sequence, but the **continuation of the source sequence**.

This is not always the case: in some Natural Language Processing tasks, like translation, where the target sequence *is* a new sequence, the first data point is some "*special*" token that indicates the start of that new sequence.

There is another small, yet **fundamental**, difference between the encoder and the decoder: since the decoder **uses the prediction of one step as input to the next**, we’ll have to **manually loop over the generation of the target sequence**.

This also means we need to **keep track of the hidden state** from one step to the next, using the **hidden state of one step as input to the next** as well.

But, instead of making the hidden state both an input and an output of the `forward` method, we can easily (and quite elegantly, I might add) handle this by **making the hidden state an attribute** of our decoder model.

The **decoder** model is actually quite similar to the models we’ve developed in Chapter 8:

Decoder

```python
class Decoder(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features,
                                self.hidden_dim,
                                batch_first=True)
        self.regression = nn.Linear(self.hidden_dim,
                                    self.n_features)

    def init_hidden(self, hidden_seq):
        # We only need the final hidden state
        hidden_final = hidden_seq[:, -1:] # N, 1, H
        # But we need to make it sequence-first
        self.hidden = hidden_final.permute(1, 0, 2) # 1, N, H  ①

    def forward(self, X):
        # X is N, 1, F
        batch_first_output, self.hidden = \
                                self.basic_rnn(X, self.hidden) ②

        last_output = batch_first_output[:, -1:]
        out = self.regression(last_output)

        # N, 1, F
        return out.view(-1, 1, self.n_features)                ③
```

① Initializing decoder's hidden state using encoder's **final hidden state**

② The recurrent layer both **uses** and **updates** the hidden state

③ The output has the same shape as the input (N, 1, F)

Let's go over the differences:

- since the **initial hidden state** has to come from the encoder, we need a method to initialize the hidden state and set the corresponding attribute - the encoder's output is *batch-first* though, and the **hidden state** must **always be sequence-first**, so we permute its first two dimensions
- the **hidden state attribute** is used both as an **input** and as an **output** of the recurrent layer
- the **shape of the output** must match the **shape of the input**, namely, a **sequence of length one**
- the `forward` method will be called **multiple times** as we loop over the generation of the target sequence

The whole thing is better understood with a hands-on example in code so, it's time to try some **decoding** to **generate a target sequence**:

```
torch.manual_seed(21)
decoder = Decoder(n_features=2, hidden_dim=2)

# Initial hidden state will be encoder's final hidden state
decoder.init_hidden(hidden_seq)
# Initial data point is the last element of source sequence
inputs = source_seq[:, -1:]

target_len = 2
for i in range(target_len):
    print(f'Hidden: {decoder.hidden}')
    out = decoder(inputs)   # Predicts coordinates
    print(f'Output: {out}\n')
    # Predicted coordinates are next step's inputs
    inputs = out
```

Output

```
Hidden: tensor([[[ 0.3105, -0.5263]]], grad_fn=<SliceBackward>)
Output: tensor([[[-0.2339,  0.4702]]], grad_fn=<ViewBackward>)

Hidden: tensor([[[ 0.3913, -0.6853]]], grad_fn=<StackBackward>)
Output: tensor([[[-0.0226,  0.4628]]], grad_fn=<ViewBackward>)
```

We created a **loop** to generate **a target sequence of length two**, using the predictions of one step as inputs to the next. The hidden state, on the other hand, was entirely handled by the model itself using its `hidden_state` attribute.

There is **one problem** with the approach above, though… an **untrained model** will make **really bad predictions**, and these predictions will still be used as inputs for subsequent steps. This makes model training **unnecessarily hard** because the **prediction error in one step** is caused by **both the (untrained) model and the prediction error in the previous step**.

"*Can't we use the* ***actual target sequence*** *instead?*"

Sure we can! This technique is called **teacher forcing**.

Teacher Forcing

The reasoning is simple: **ignore the predictions** and use **the real data from the target sequence** instead. In code, we only need to change the **last line**:

```
# Initial hidden state will be encoder's final hidden state
decoder.init_hidden(hidden_seq)
# Initial data point is the last element of source sequence
inputs = source_seq[:, -1:]

target_len = 2
for i in range(target_len):
    print(f'Hidden: {decoder.hidden}')
    out = decoder(inputs) # Predicts coordinates
    print(f'Output: {out}\n')
    # Completely ignores the predictions and uses real data instead
    inputs = target_seq[:, i:i+1]      ①
```

① Inputs to the next step are not predictions anymore

Output

```
Hidden: tensor([[[ 0.3105, -0.5263]]], grad_fn=<SliceBackward>)
Output: tensor([[[-0.2339,  0.4702]]], grad_fn=<ViewBackward>)

Hidden: tensor([[[ 0.3913, -0.6853]]], grad_fn=<StackBackward>)
Output: tensor([[[0.2265, 0.4529]]], grad_fn=<ViewBackward>)
```

Now, a **bad prediction** can only be traced to the **model** itself, and any bad predictions in previous steps have no effect whatsoever.

*"This is **great** for training time, sure… but what about **testing** time, when the target sequence is unknown?"*

In **testing** time, there is no escape from using only the model's **own predictions from previous steps**.

The problem is, a model trained using **teacher forcing** will minimize the loss given the **correct inputs at every step of the target sequence**. But, since this will **never be the case in testing time**, the model is likely to **perform poorly** when using its

own predictions as inputs.

"What can we do about it then?"

When in doubt, flip a coin. Literally. During training, **sometimes** the model will use **teacher forcing**, and **sometimes** it will use its **own predictions**. So we occasionally help the model by providing an actual input but we still force it to be robust enough to generate and use its own inputs. In code, we just have to add an **if statement** and **draw a random number**:

```python
# Initial hidden state will be encoder's final hidden state
decoder.init_hidden(hidden_seq)
# Initial data point is the last element of source sequence
inputs = source_seq[:, -1:]

teacher_forcing_prob = 0.5
target_len = 2
for i in range(target_len):
    print(f'Hidden: {decoder.hidden}')
    out = decoder(inputs)
    print(f'Output: {out}\n')
    # If it is teacher forcing
    if torch.rand(1) <= teacher_forcing_prob:
        # Takes the actual element
        inputs = target_seq[:, i:i+1]
    else:
        # Otherwise uses the last predicted output
        inputs = out
```

Output

```
Hidden: tensor([[[ 0.3105, -0.5263]]], grad_fn=<SliceBackward>)
Output: tensor([[[-0.2339,  0.4702]]], grad_fn=<ViewBackward>)

Hidden: tensor([[[ 0.3913, -0.6853]]], grad_fn=<StackBackward>)
Output: tensor([[[0.2265, 0.4529]]], grad_fn=<ViewBackward>)
```

You may set `teacher_forcing_prob` to `1.0` or `0.0` to replicate any of the two outputs we generated before.

Now it is time to put the two of them together…

Encoder + Decoder

The figure below illustrates the flow of information from encoder to decoder:

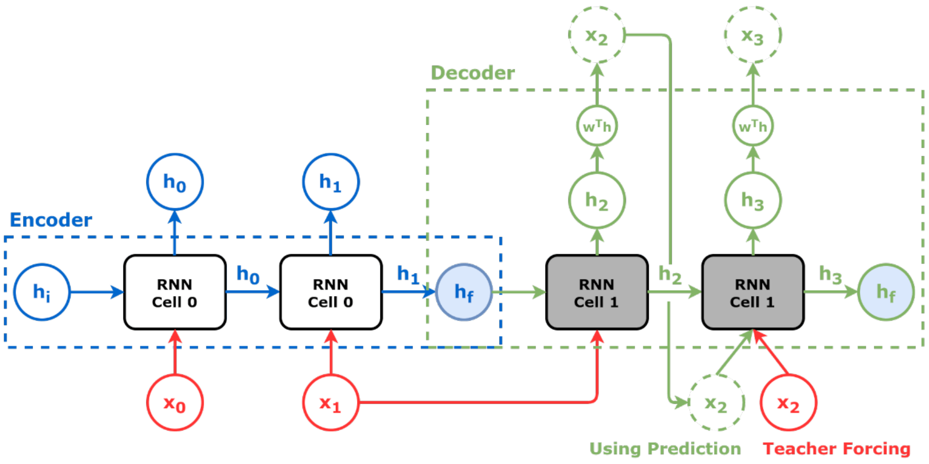


Figure 9.6 - Encoder + Decoder

Let’s go over it once again:

- the **encoder** receives the **source sequence** (x_0 and x_1, in red) and generates the **representation of the source sequence**, its **final hidden state** (h_f, in blue)

- the **decoder** receives the **hidden state from the encoder** (h_f, in blue), together with the **last known element of the sequence** (x_1, in red), to output a **hidden state** (h_2, in green) that is **converted into the first set of predicted coordinates** (x_2, in green) using a **linear layer** ($w^T h$, in green)
- in the next iteration of the loop, the model **randomly uses the predicted (x_2, in green) or the actual (x_2, in red) set of coordinates** as one of its inputs to output the **second set of predicted coordinates** (x_3) thus achieving the **target length**
- the **final output** of the **encoder + decoder** model is the **full sequence of predicted coordinates**: [x_2, x_3]

We can assemble the bits and pieces we've developed so far into a model that, given the encoder and decoder models, implements a `forward` method that **splits the input** into the source and target sequences, loops over the **generation of the target sequence**, and implements **teacher forcing** in training mode.

The model below is mostly about handling the **boilerplate** necessary to integrate both encoder and decoder:

Encoder + Decoder

```python
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder,
                 input_len, target_len,
                 teacher_forcing_prob=0.5):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.teacher_forcing_prob = teacher_forcing_prob
        self.outputs = None

    def init_outputs(self, batch_size):
        device = next(self.parameters()).device
        # N, L (target), F
        self.outputs = torch.zeros(batch_size,
                                   self.target_len,
                                   self.encoder.n_features).to(device)

    def store_output(self, i, out):
        # Stores the output
        self.outputs[:, i:i+1, :] = out

    def forward(self, X):
        # splits the data in source and target sequences
        # the target seq will be empty in testing mode
        # N, L, F
        source_seq = X[:, :self.input_len, :]
        target_seq = X[:, self.input_len:, :]
        self.init_outputs(X.shape[0])

        # Encoder expected N, L, F
        hidden_seq = self.encoder(source_seq)
        # Output is N, L, H
        self.decoder.init_hidden(hidden_seq)

        # The last input of the encoder is also
        # the first input of the decoder
        dec_inputs = source_seq[:, -1:, :]

        # Generates as many outputs as the target length
        for i in range(self.target_len):
            # Output of decoder is N, 1, F
            out = self.decoder(dec_inputs)
            self.store_output(i, out)

            prob = self.teacher_forcing_prob
            # In evaluation/test the target sequence is
            # unknown, so we cannot use teacher forcing
            if not self.training:
                prob = 0

            # If it is teacher forcing
            if torch.rand(1) <= prob:
                # Takes the actual element
                dec_inputs = target_seq[:, i:i+1, :]
            else:
                # Otherwise uses the last predicted output
                dec_inputs = out

        return self.outputs
```

The only *real* additions are the `init_outputs` method, which creates a tensor for storing the generated target sequence, and the `store_output` method, which actually stores the output produced by the decoder.

Let's create an instance of the model above using the other two we already had created:

```
encdec = EncoderDecoder(encoder, decoder,
                        input_len=2, target_len=2,
                        teacher_forcing_prob=0.5)
```

In **training mode**, the model expects the **full sequence** so it can randomly use **teacher forcing**:

```
encdec.train()
encdec(full_seq)
```

Output

```
tensor([[[-0.2339,  0.4702],
         [ 0.2265,  0.4529]]], grad_fn=<CopySlices>)
```

In **evaluation/test mode**, though, it **only needs the source sequence** as input:

```
encdec.eval()
encdec(source_seq)
```

Output

```
tensor([[[-0.2339,  0.4702],
         [-0.0226,  0.4628]]], grad_fn=<CopySlices>)
```

Let’s use this knowledge to build our **training and test sets**.

Data Preparation

For the **training set**, we need the **full sequences** as **features** (*X*) to use **teacher forcing**, and the **target sequences** as **labels** (*y*) so we can compute the **mean squared error** loss:

Data Generation - Train

```
points, directions = generate_sequences()
full_train = torch.as_tensor(points).float()
target_train = full_train[:, 2:]
```

For the **test set**, though, we only need **the source sequences** as **features** (*X*) and the **target sequences** as **labels** (*y*):

Data Generation - Test

```
test_points, test_directions = generate_sequences(seed=19)
full_test = torch.as_tensor(points).float()
source_test = full_test[:, :2]
target_test = full_test[:, 2:]
```

These are all simple tensors, so we can use `TensorDatasets` and simple data loaders:

Data Preparation

```
train_data = TensorDataset(full_train, target_train)
test_data = TensorDataset(source_test, target_test)
generator = torch.Generator()
train_loader = DataLoader(train_data, batch_size=16,
                          shuffle=True, generator=generator)
test_loader = DataLoader(test_data, batch_size=16)
```

In version 1.7, PyTorch introduced a **modification** to the random sampler in the `DataLoader` that's responsible for **shuffling** the data. In order to **ensure reproducibility**, we need to **assign a `Generator`** to the `DataLoader` (we did something similar in Chapter 4 when we used *other* samplers). Luckily, our `StepByStep` class **already sets a seed to the generator**, if any, in its `set_seed` method, so you don't need to worry about that.

By the way, we **didn't use the directions** to build the datasets this time.

We have everything we need to **train our first sequence-to-sequence model** now!

Model Configuration & Training

The model configuration is very straightforward: we create both encoder and decoder models, use them as arguments to the big `EncoderDecoder` model that handles the boilerplate, and create a loss and an optimizer as usual.

Model Configuration

```
torch.manual_seed(23)
encoder = Encoder(n_features=2, hidden_dim=2)
decoder = Decoder(n_features=2, hidden_dim=2)
model = EncoderDecoder(encoder, decoder,
                       input_len=2, target_len=2,
                       teacher_forcing_prob=0.5)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Next, we use the `StepByStep` class to train the model:

Model Training

```
sbs_seq = StepByStep(model, loss, optimizer)
sbs_seq.set_loaders(train_loader, test_loader)
sbs_seq.train(100)
```

```
fig = sbs_seq.plot_losses()
```

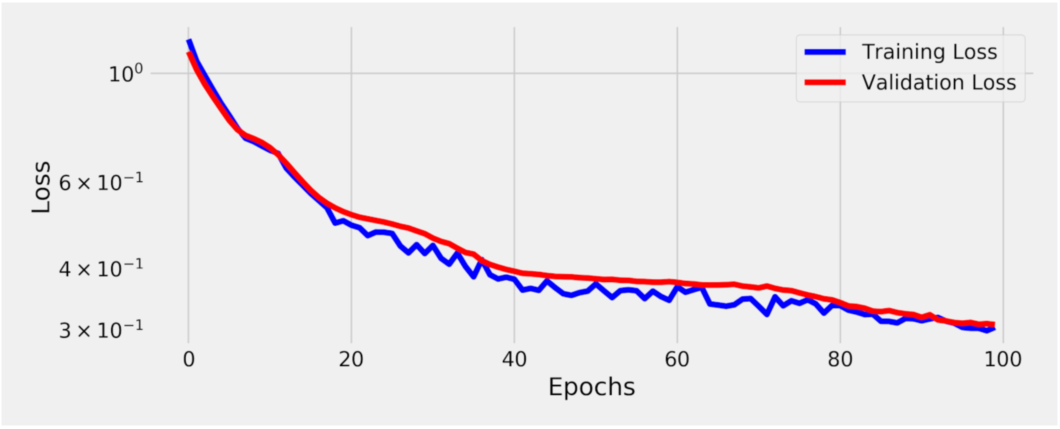


Figure 9.7 - Losses - Encoder + Decoder

It is hard to tell how badly our model is performing by looking at the loss only if we're dealing with a "*regression*" problem like this. It is much better to **visualize the predictions**.

Visualizing Predictions

Let's plot the **predicted coordinates** and connect them using **dashed lines** while using **solid lines** to connect the **actual coordinates**. The first **ten sequences** of the **test set** look like this:

```
fig = sequence_pred(sbs_seq, full_test, test_directions)
```

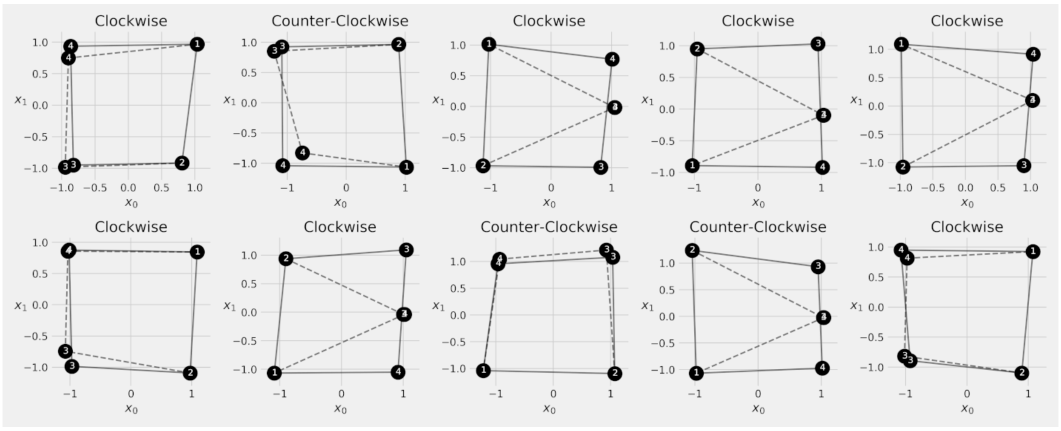


Figure 9.8 - Predictions

The results are, at the same time, very **good** and very **bad**. In **half** of the sequences, the **predicted coordinates** are **quite close** to the actual ones. But, in the other half, the predicted coordinates are **overlapping with each other** and close to the midpoint between the actual coordinates. For whatever reason, the model learned to make **good predictions** whenever the **first corner** is on the **right edge** of the square, but really bad ones otherwise.

Can We Do Better?

The encoder-decoder architecture is really interesting, but it has a **bottleneck**: the whole **source sequence** gets to be **represented by a single hidden state**, the final hidden state of the encoder part. Even for a very short source sequence like ours, that's quite a **big ask** to have the **decoder generating the target sequence with so little information**.

"*Can we make the decoder use* ***more information****?*"

Sure, we can!

Attention

Here is a (not so) crazy idea: what if the **decoder** could **choose** one (or more) of the **encoder's hidden states** to use instead of being forced to stick with only the final one? That would surely give it more **flexibility** to use the hidden state that's **more useful** at a given step of the **target sequence generation**.

Let's illustrate it with a simple, non-numerical, example: translating from English to French using Google Translate. The original sentence is "*the European economic zone*" and its French translation is "*la zone économique européenne*".

Now, let's compare their first words: "*la*", in French, obviously corresponds to "*the*" in English. My question to you is:

"*Could Google (or any translator) have translated "****the****" to "****la****" without any other information?*"

The answer is: **no**. The English language has only *one* definite article - "*the*" - while French (and many other languages) have *many* definite articles. It means that "*the*" **may be translated in many different ways**, and it is only possible to determine the correct (translated) article after finding the *noun* it refers to. The noun, in this case, is *zone* and it is the **last word** in the English sentence. Coincidentally, its translation is also *zone*, and it is a **singular feminine noun** in French, thus making "*la*" the correct translation of "*the*" in this case.

"So what? What does this have to do with hidden states?"

Well, if we consider the English sentence a **(source) sequence of words**, the French sentence is a **(target) sequence of words**. Assuming we can map each word to a numeric vector, we can use an encoder to encode the words in English, **each word corresponding to a hidden state**.

We know that the decoder's role is to **generate the translated words**, right? Then, if the decoder is allowed to **choose which hidden states from the encoder** it will use to generate each output, it means it can **choose which English words** it will use to generate each translated word.

We have already seen that, in order to translate "*the*" to "*la*", the translator (that will be the decoder) needs to know the corresponding noun too, so it is reasonable to assume that the decoder would choose the *first* and *last* encoded hidden states (corresponding to "*the*" and "*zone*" in the English sentence) to generate the first translated word.

In other words, we can say that the decoder is **paying attention** to different elements in the **source sequence**. That's the famous **attention mechanism** in a nutshell.

Now, try to answer this question:

"Which English word(s) is the decoder paying attention to in order to generate the second French word ("zone")?"

It is reasonable to assume it is paying attention only to the *last* English word, that is, "*zone*". The remaining English words shouldn't play a role in this particular piece of the translation.

Let's take it one step further and build a matrix with English words (the source sequence) as columns and French words (the target sequence) as rows. The entries in this matrix represent our guesses of **how much attention** the decoder is paying to **each English word** in order to generate **a given French word**:

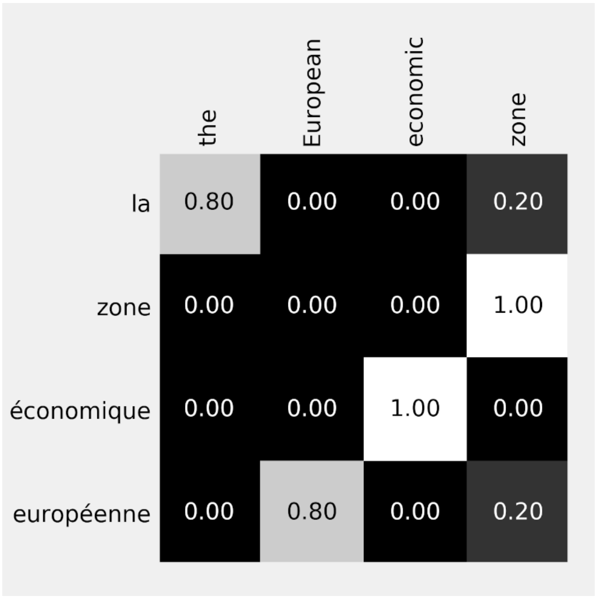


Figure 9.9 - Attention for translation

The numbers above are *completely made-up*, by the way. Since the translated "*la*" is based on "*the*" and "*zone*", I've just guessed that a translator (the decoder) could have assigned 80% of its attention to the definite article and the remaining 20% of its attention to the corresponding noun to determine its gender. These weights, or *alphas*, are the **attention scores**.

The **more relevant** to the decoder a **hidden state from the encoder** is, the **higher** the **score**.

"OK, I get what the attention scores represent, but ***how*** *does the decoder actually use them?"*

If you haven't noticed yet, the **attention scores** actually **add up to one**, so they will be used to compute a **weighted average of the encoder's hidden states**.

Let's work this out in more detail but, first, it helps to keep it short and simple: so we're translating two words only, from "*the zone*" in English to "*la zone*" in French.

Then, we can use the **encoder-decoder** architecture from the previous section:

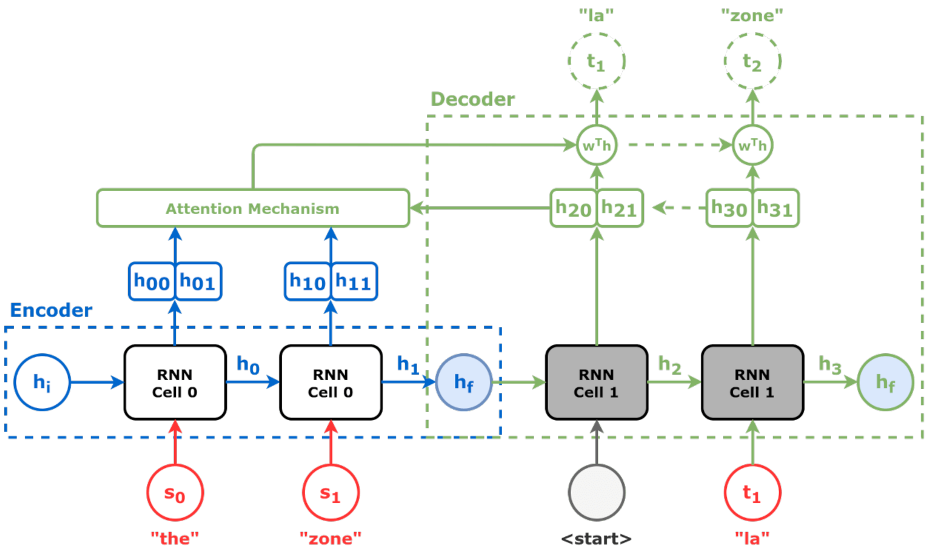


Figure 9.10 - Encoder-Decoder for translation

In the translation example above, the source and target sequences are independent, so the **first input** of the decoder *isn't* the last element of the source sequence anymore, but the **special token** that indicates the start of a **new sequence**.

The main difference is, instead of **generating predictions** solely based on its **own hidden states**, the **decoder** will recruit the **attention mechanism** to help it decide which parts of the source sequence it must pay attention to.

In our made-up example, the attention mechanism informed the decoder it should pay 80% of its attention to the encoder's hidden state corresponding to the word "*the*", and the remaining 20%, to the word "*zone*". The diagram below illustrates this:

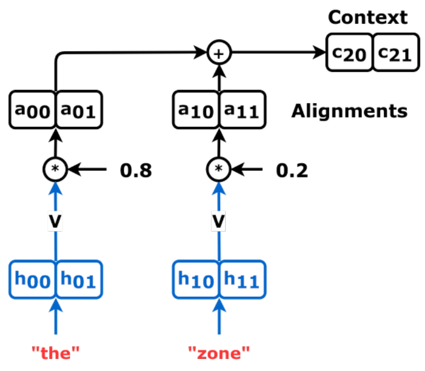


Figure 9.11 - Paying attention to words

"Values"

From now on, we'll be referring to the **encoder's hidden states** (or their affine transformations) as **"values" (V)**. The resulting multiplication of a "value" by its corresponding **attention score** is called an **alignment vector**. And, as you can see in the diagram, the **sum of all alignment vectors** (that is, the weighted average of the hidden states) is called a **context vector**.

$$context\ \ vector = \underbrace{\alpha_0 * h_0}_{alignment\ \ vector_0} + \underbrace{\alpha_1 * h_1}_{alignment\ \ vector_1} = 0.8 * value_{the} + 0.2 * value_{zone}$$

Equation 9.1 - Context vector

*"OK, but **where** do the attention scores come from?"*

"Keys" and "Queries"

The attention scores are based on **matching each hidden state of the decoder** (h_2) to **every hidden state of the encoder** (h_0 and h_1). Some of them will be **good matches** (higher attention scores) while some others will be **poor matches** (low attention scores):

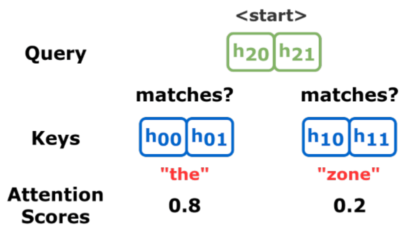


Figure 9.12 - Matching a query to the keys

The **encoder's hidden states** are called **"keys" (K)**, while the **decoder's hidden state** is called a **"query" (Q)**.

*"Wait a minute... I thought the **encoder's hidden states** were called **"values" (V)**..."*

You're absolutely right. The **encoder's hidden states** are used **both as "keys" (K) and "values" (V)**. Later on, we'll apply **affine transformations** to the hidden states, one for the "keys", another for the "values", so they will actually have different values.

"Where do these names come from anyway?"

Well, the general idea is that the **encoder** works like a **key-value store** as if it were some sort of database, and then the **decoder** would be able to **query** it. The attention mechanism would **look the query up in its keys** (the matching part) and **return its values**. Honestly, I don't think this idea helps much because the mechanism doesn't return a single original value, but a weighted average of all of them. But this naming convention is used everywhere, so you need to know it.

"Why is "the" a better match than "zone" in this case?"

Fair enough. These are made-up values and their sole purpose is to illustrate the attention mechanism. If it helps, consider that sentences are more likely to start with "*the*" than "*zone*", so the former is likely a better match to the special `<start>` token.

"*OK, I will play along...*"

Thanks! Even though we haven't actually discussed **how to match** a given "query" (Q) to the "keys" (K), we *can* update our diagram to include them:

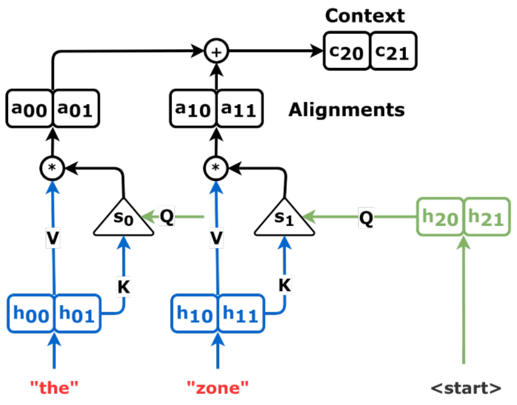


Figure 9.13 - Computing the context vector

The **"query" (Q)** is matched to both **"keys" (K)** to compute the **attention scores (s)** used to compute the **context vector**, which is simply the **weighted average of the "values" (V)**.

Computing the Context Vector

Let's go over a simple example in code using our own sequence-to-sequence problem, and the "*perfect*" square as input:

```
full_seq = (torch.tensor([[-1, -1], [-1, 1], [1, 1], [1, -1]])
            .float()
            .view(1, 4, 2))
source_seq = full_seq[:, :2]
target_seq = full_seq[:, 2:]
```

The **source sequence** is the input of the **encoder**, and the **hidden states** it outputs are going to be both **"values" (V)** and **"keys" (K)**:

```
torch.manual_seed(21)
encoder = Encoder(n_features=2, hidden_dim=2)
hidden_seq = encoder(source_seq)

values = hidden_seq # N, L, H
values
```

Output

```
tensor([[[ 0.0832, -0.0356],
         [ 0.3105, -0.5263]]], grad_fn=<PermuteBackward>)
```

```
keys = hidden_seq # N, L, H
keys
```

Output

```
tensor([[[ 0.0832, -0.0356],
         [ 0.3105, -0.5263]]], grad_fn=<PermuteBackward>)
```

The encoder-decoder dynamics stay exactly the same: we still use the **encoder's final hidden state** as the **decoder's initial hidden state** (although we're sending the whole sequence to the decoder, it *still* uses the last hidden state only), and we still use the **last element of the source sequence** as **input** to the first step of the **decoder**:

```
torch.manual_seed(21)
decoder = Decoder(n_features=2, hidden_dim=2)
decoder.init_hidden(hidden_seq)

inputs = source_seq[:, -1:]
out = decoder(inputs)
```

The **first "query" (Q)** is the **decoder's hidden state** (remember, hidden states are *always* sequence-first, so we're permuting it to batch-first):

```
query = decoder.hidden.permute(1, 0, 2)  # N, 1, H
query
```

Output

```
tensor([[[ 0.3913, -0.6853]]], grad_fn=<PermuteBackward>)
```

OK, we got the "keys" and a "query", so let's *pretend* we can compute **attention scores (*alphas*)** using them:

```
def calc_alphas(ks, q):
    N, L, H = ks.size()
    alphas = torch.ones(N, 1, L).float() * 1/L
    return alphas

alphas = calc_alphas(keys, query)
alphas
```

Output

```
tensor([[[0.5000, 0.5000]]])
```

We had to make sure *alphas* had the **right shape (N, 1, L)** so that, when multiplied by the **"values"** with **shape (N, L, H)**, it will result in a **weighted sum of the alignment vectors** with **shape (N, 1, H)**. We can use **batch matrix multiplication** (`torch.bmm`) for that:

$$(N, 1, L) \times (N, L, H) = (N, 1, H)$$

Equation 9.2 - Shapes for batch matrix multiplication

In other words, we can simply **ignore the first dimension** and PyTorch will go over all the elements in the mini-batch for us:

```
# N, 1, L x N, L, H -> 1, L x L, H -> 1, H
context_vector = torch.bmm(alphas, values)
context_vector
```

Output

```
tensor([[[ 0.1968, -0.2809]]], grad_fn=<BmmBackward0>)
```

"Why are you spending so much time on ***matrix multiplication*** *of all things?"*

Although it seems a fairly basic topic, **getting the shapes and dimensions right** is of utmost importance for the correct implementation of an algorithm or technique. Sometimes, using the *wrong* dimensions in an operation *may not raise an explicit error*, but it will **damage the model's ability to learn** nonetheless. For that reason, I believe it is worth it to spend some time going over it in full detail.

Once the **context vector** is ready, we can **concatenate** it to the **"query" (the decoder's hidden state)** and use it as the input for the linear layer that actually generates the predicted coordinates:

```
concatenated = torch.cat([context_vector, query], axis=-1)
concatenated
```

Output

```
tensor([[[ 0.1968, -0.2809,  0.3913, -0.6853]]], grad_fn
=<CatBackward>)
```

The diagram below illustrates the whole thing, encoder, decoder, and attention mechanism:

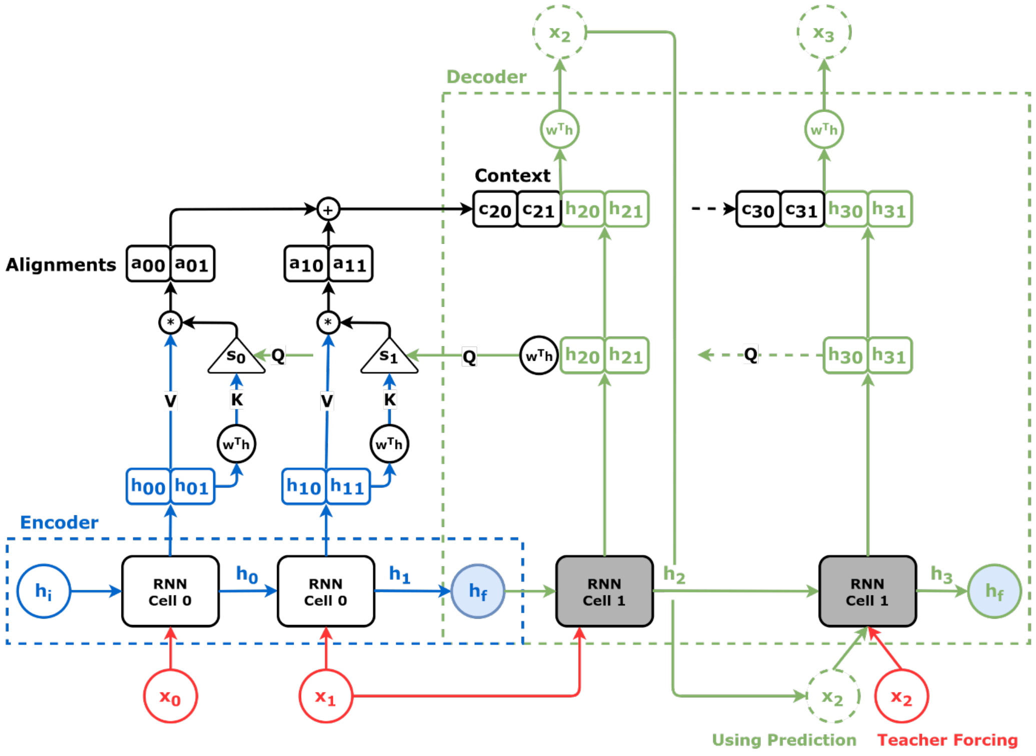


Figure 9.14 - Encoder + Decoder + Attention

*"The attention mechanism looks different… there are **affine transformations** ($w^T h$) for both "keys" and "queries" now…"*

Yes, the **scoring method** will use the **transformed "keys" and "queries"** to compute the **attention scores**, so I've already included them in the diagram above. By the way, this is a good moment to *summarize* the information in a table:

	Keys (K)	**Queries (Q)**	**Values (V)**
Source	Encoder	Decoder	Encoder
Affine Transformation	Yes	Yes	Not yet
Purpose	Scoring	Scoring	Alignment Vector

*"I guess now it is time to see **how** the **scoring method** works, right?"*

Absolutely! Let's understand how the **scoring method** transforms a **good match** between a "query" and a "key" into an **attention score**.

Scoring Method

A **"key" (K)** is a hidden state from the encoder. A **"query" (Q)** is a hidden state from the decoder. Both of them are **vectors** with the same number of dimensions, that is, the number of **hidden dimensions** of the underlying recurrent neural networks.

The **scoring method** needs to determine if **two vectors** are a **good match** or not, or, phrased differently, it needs to determine if **two vectors are similar** or not.

*"How do we compute the **similarity** between two vectors?"*

Well, that's actually easy: **cosine similarity** to the rescue! If two vectors are **pointing in the same direction**, their cosine similarity is a **perfect one**. If they are **orthogonal** (that is, there is a right-angle between them), their cosine similarity is **zero**. If they are pointing in **opposite directions**, their cosine similarity is **minus one**.

Its formula is:

$$cos \ \ \theta = \frac{\sum_i q_i k_i}{\sqrt{\sum_j q_j^2}\sqrt{\sum_j k_j^2}}$$

Equation 9.3 - Cosine similarity

Unfortunately, cosine similarity **does not** consider the **norm (size)** of the vectors, only its direction (the sizes of the vectors are in the denominator in the formula above).

*"What if we **scale** the similarity by the **norms** of both vectors?"*

Perfect! Let’s do that:

$$cos \ \ \theta \sqrt{\sum_j q_j^2}\sqrt{\sum_j k_j^2} = \sum_i q_i k_i$$

Equation 9.4 - Scaled cosine similarity

The two terms next to the cosine are the **norms** of the **"query" (Q)** and **"key" (K)** vectors, and the term on the right is actually the **dot product** between the two vectors:

$$cos\theta \ \ ||Q|| \ \ ||K|| = Q \cdot K$$

Equation 9.5 - Cosine similarity vs dot product

The **dot product** is equal to the **cosine similarity scaled by the norms of the vectors**. In other words, the dot product is:

- **higher** if both **"key" (K)** and **"query" (Q)** vectors are **aligned** (small angle and **high cosine value**)
- **proportional** to the **norm (size)** of the **"query" vector (Q)**
- **proportional** to the **norm (size)** of the **"key" vector (K)**

The **dot product** is one of the most common ways to compute **alignment (and attention) scores**, but it is *not* the only one. For more information on different mechanisms, and *attention* in general, please refer to Lilian Weng's amazing blog post[136] on the subject.

To compute the **dot products** all at once, we can use PyTorch's *batch matrix multiplication* (`torch.bmm`) once again. We have to **transpose the "keys" matrix**, though, by **permuting the last two dimensions**:

```
# N, 1, H x N, H, L -> N, 1, L
products = torch.bmm(query, keys.permute(0, 2, 1))
products
```

Output

```
tensor([[[0.0569, 0.4821]]], grad_fn=<BmmBackward0>)
```

*"But these values **do not add up to one**… they cannot be **attention scores**, right?"*

You're absolutely right, these are **alignment scores**.

Attention Scores

To transform alignment scores into **attention scores** we can use the **softmax** function:

```
alphas = F.softmax(products, dim=-1)
alphas
```

Output

```
tensor([[[0.3953, 0.6047]]], grad_fn=<SoftmaxBackward>)
```

That's more like it, they're adding up to one! The attention scores above mean that the **first hidden state** contributes to **roughly 40%** of the **context vector** while the **second hidden state** contributes to the remaining **60%** of the context vector.

We can also **update** our `calc_alphas` function to **actually compute them**:

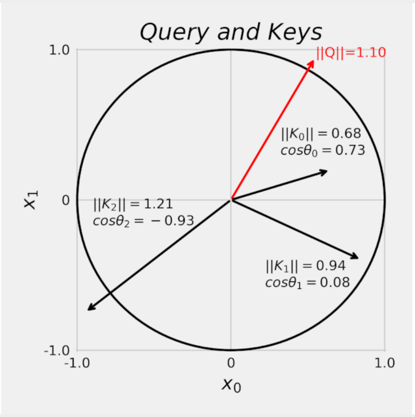
```
def calc_alphas(ks, q):
    # N, 1, H x N, H, L -> N, 1, L
    products = torch.bmm(q, ks.permute(0, 2, 1))
    alphas = F.softmax(products, dim=-1)
    return alphas
```

Visualizing the Context Vector

Let's start by creating one dummy "query" and three dummy "keys":

```
q = torch.tensor([.55, .95]).view(1, 1, 2) # N, 1, H
k = torch.tensor([[.65, .2],
                  [.85, -.4],
                  [-.95, -.75]]).view(1, 3, 2) # N, L, H
```

Then, let's **visualize** them as vectors, together with their **norms** and the **cosines** of the angles between each "key" and the "query":



We can use the values in the figure above to compute the **dot product** between each "key" and the "query":

$$
\begin{aligned}
Q \cdot K_0 &= cos\theta_0 \ ||Q|| \ ||K_0|| = && 0.73* && 1.10* && 0.68 = && 0.54 \\
Q \cdot K_1 &= cos\theta_1 \ ||Q|| \ ||K_1|| = && 0.08* && 1.10* && 0.94 = && 0.08 \\
Q \cdot K_2 &= cos\theta_2 \ ||Q|| \ ||K_2|| =- && 0.93* && 1.10* && 1.21 =- && 1.23
\end{aligned}
$$

Equation 9.6 - Dot products

```
# N, 1, H x N, H, L -> N, 1, L
prod = torch.bmm(q, k.permute(0, 2, 1))
prod
```

Output

```
tensor([[[ 0.5475,  0.0875, -1.2350]]])
```

Even though the **first** "key" (K_0) is the **smallest in size**, it is the **most well-aligned** to the "query", and, overall, the "key" with the **largest dot product**. This means that the **decoder** would **pay the most attention** to this particular key. Makes sense, right?

Applying the **softmax** to these values gives us the following **attention scores**:

```
scores = F.softmax(prod, dim=-1)
scores
```

Output

```
tensor([[[0.5557, 0.3508, 0.0935]]])
```

Unsurprisingly, the first key got the largest weight. Let's use these weights to compute the **context vector**:

$$\text{context vector} = 0.5557 * \begin{bmatrix} 0.65 \\ 0.20 \end{bmatrix} + 0.3508 * \begin{bmatrix} 0.85 \\ -0.40 \end{bmatrix} + 0.0935 * \begin{bmatrix} -0.95 \\ -0.75 \end{bmatrix}$$

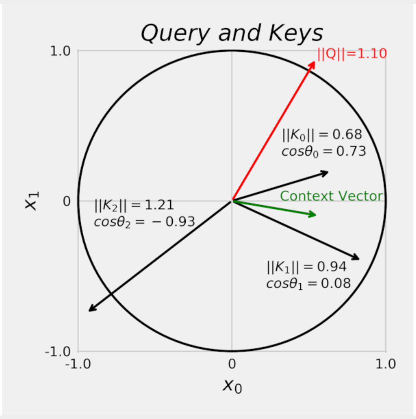
Equation 9.7 - Computing the context vector

```
v = k
context = torch.bmm(scores, v)
context
```

Output

```
tensor([[[ 0.5706, -0.0993]]])
```

Better yet, let's **visualize** the context vector:



Since the context vector is a **weighted sum of the values** (or keys, since we're not applying any affine transformations yet), it is only logical that its location is somewhere between the other vectors.

Scaled Dot Product

So far, we've used simple dot products between a "query" and each one of the "keys". But, given that the **dot product** between two vectors is the **sum of the elements** after an **element-wise multiplication** of both vectors, guess what happens as the vectors grow to a **larger number of dimensions**? The **variance** gets **larger** as well.

So we need to (somewhat) **standardize** it by **scaling the dot product** by the **inverse**

of its standard deviation:

$$\text{scaled dot product} = \frac{Q \cdot K}{\sqrt{d_k}}$$

Equation 9.8 - Scaled dot product

```
dims = query.size(-1)
scaled_products = products / np.sqrt(dims)
scaled_products
```

Output

```
tensor([[[0.0403, 0.3409]]], grad_fn=<DivBackward0>)
```

"*Why do we need to scale the dot product?*"

If we don't, the distribution of attention scores will get *too skewed* because the **softmax** function is actually affected by the **scale** of its inputs:

```
dummy_product = torch.tensor([4.0, 1.0])
(F.softmax(dummy_product, dim=-1),
 F.softmax(100*dummy_product, dim=-1))
```

Output

```
(tensor([0.9526, 0.0474]), tensor([1., 0.]))
```

See? As the *scale* of the dot products grows larger, the resulting distribution of the *softmax* gets more and more skewed.

In our case, there isn't much difference because our vectors have **only two dimensions**:

```
alphas = F.softmax(scaled_products, dim=-1)
alphas
```

Output

```
tensor([[[0.4254, 0.5746]]], grad_fn=<SoftmaxBackward>)
```

The computation of the **context vectors** using **scaled dot product** is usually depicted like this:

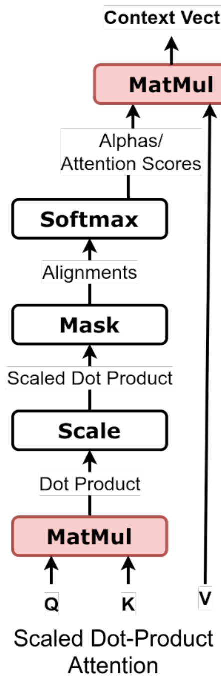


Figure 9.15 - Scaled dot product attention

Dot Product's Standard Deviation

You probably noticed that the **square root of the number of dimensions** as the **standard deviation** simply appeared out of thin air. We're not proving it or anything, but we can try and **simulate** a ton of dot products to see what happens, right?

```
n_dims = 10
vector1 = torch.randn(10000, 1, n_dims)
vector2 = torch.randn(10000, 1, n_dims).permute(0, 2, 1)
torch.bmm(vector1, vector2).squeeze().var()
```

Output

```
tensor(9.8681)
```

Even though the values in hidden states coming out of both encoder and decoder are **bounded to (-1, 1) by the hyperbolic tangent**, remember that we're likely performing an **affine transformation** on them to produce both "keys" and "query". This means that the simulation above, where values are drawn from a **normal distribution**, is not as far-fetched as it may seem at first sight.

If you try different values for the number of dimensions, you'll see that, on average, the **variance equals the number of dimensions**. So, the **standard deviation** is given by the **square root of the number of dimensions**:

$$
Var(vector_1 \cdot vector_2) = d_{vector_1} = d_{vector_2}
$$
$$
\sigma(vector_1 \cdot vector_2) = \sqrt{d_{vector_1}} = \sqrt{d_{vector_2}}
$$

Equation 9.9 - Standard deviation of the dot product

If you prefer to see it in code, it looks like this:

```
def calc_alphas(ks, q):
    dims = q.size(-1)
    # N, 1, H x N, H, L -> N, 1, L
    products = torch.bmm(q, ks.permute(0, 2, 1))
    scaled_products = products / np.sqrt(dims)
    alphas = F.softmax(scaled_products, dim=-1)
    return alphas
```

```
alphas = calc_alphas(keys, query)
# N, 1, L x N, L, H -> 1, L x L, H -> 1, H
context_vector = torch.bmm(alphas, values)
context_vector
```

Output

```
tensor([[[ 0.2138, -0.3175]]], grad_fn=<BmmBackward0>)
```

*"The **mask** is still missing in the code… what is that about?"*

We'll get back to it soon enough in the section after the next one. Hold that thought!

Now we need to organize it all, building a *class* to handle the attention mechanism.

Attention Mechanism

The **complete attention mechanism** is depicted in the figure below:

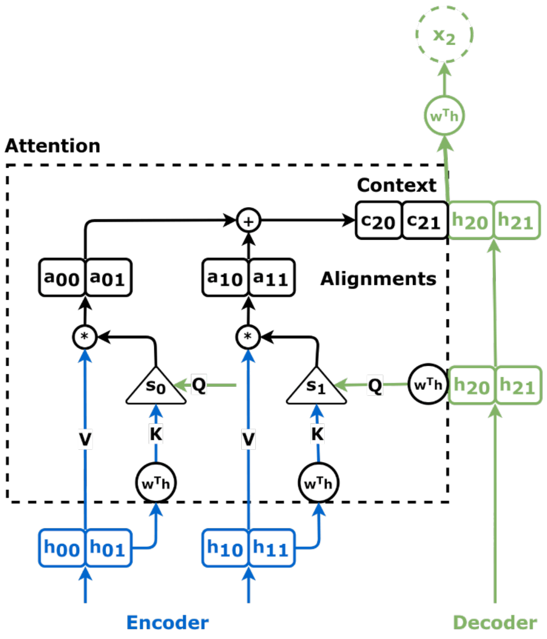


Figure 9.16 - Attention mechanism

For some cool **animations** of the attention mechanism, make sure to check these great posts: Jay Alammar's *"Visualizing A Neural Machine Translation Model"*[137] and Raimi Karim's *"Attn: Illustrated Attention"*[138].

In our sequence-to-sequence problem, both *features* and *hidden states* are *two-dimensional*. We could have chosen **any number of hidden dimensions**, but using *two hidden dimensions* allowed us to more easily visualize diagrams and plots.

More often than not, this **won't** be the case, and the number of **hidden dimensions** will be **different** than the number of **features** (the **input dimensions**). Currently, this **change in dimensionality** is performed by the **recurrent layers**.

The **self-attention** mechanism (our next topic) **does not** use recurrent layers anymore, so we'll have to tackle this change in dimensionality differently. Luckily, the **affine transformations** we're applying to "keys", "queries", and "values" (soon) can also be used to **change the dimensionality** from the number of *input* to the number of *hidden* dimensions. Therefore, we're including an `input_dim` argument in our `Attention` class to handle this.

The corresponding code, neatly organized into a *model* looks like this:

Attention Mechanism

```
class Attention(nn.Module):
    def __init__(self, hidden_dim, input_dim=None,
                 proj_values=False):
        super().__init__()
        self.d_k = hidden_dim
        self.input_dim = hidden_dim if input_dim is None \
                         else input_dim
        self.proj_values = proj_values
        # Affine transformations for Q, K, and V
        self.linear_query = nn.Linear(self.input_dim, hidden_dim)
        self.linear_key = nn.Linear(self.input_dim, hidden_dim)
        self.linear_value = nn.Linear(self.input_dim, hidden_dim)
        self.alphas = None

    def init_keys(self, keys):
        self.keys = keys
        self.proj_keys = self.linear_key(self.keys)
        self.values = self.linear_value(self.keys) \
                      if self.proj_values else self.keys

    def score_function(self, query):
        proj_query = self.linear_query(query)
        # scaled dot product
        # N, 1, H x N, H, L -> N, 1, L
        dot_products = torch.bmm(proj_query,
                                 self.proj_keys.permute(0, 2, 1))
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def forward(self, query, mask=None):
        # Query is batch-first N, 1, H
        scores = self.score_function(query) # N, 1, L   ①
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, 1, L    ②
        self.alphas = alphas.detach()

        # N, 1, L x N, L, H -> N, 1, H
        context = torch.bmm(alphas, self.values)        ③
        return context
```

① First step: alignment scores (scaled dot product)

② Second step: attention scores (alphas)

③ Third step: context vector

Let's go over each one of the methods:

- in the *constructor* method, there are:
    - **three linear layers** corresponding to the **affine transformations** for "keys" and "query" (and for the future transformation of "values" too)
    - one attribute for the number of hidden dimensions (to scale the dot product)
    - a placeholder for the **attention scores** (`alphas`)
- there is an `init_keys` method to receive a **batch-first sequence of hidden states** from the **encoder**
    - these are **computed once** at the beginning and will be used **over and over again** with every **new "query"** that is presented to the attention mechanism

    - therefore, it is better to **initialize "keys" and "values" once** than passing them as arguments to the `forward` method every time.

- the `score_function` is simply the **scaled dot product**, but using an affine transformation on the "query" this time.
- the `forward` method takes a **batch-first hidden state** as **"query"** and performs the **three steps** of the **attention mechanism**:
    - using "keys" and "query" to compute **alignment scores**
    - using alignment scores to compute **attention scores** (alphas)
    - using "values" and attention scores to generate the **context vector**

> *"There is that unexplained* `mask` *again…"*

I'm on it!

Source Mask

The `mask` can be used to, well, **mask some of the "values"** to **force the attention mechanism to ignore them**.

> *"Why would I want to force it to do that?"*

Padding comes to mind - you likely don't want to pay attention to **stuffed data points** in a sequence, right? Let's try out an example: pretend we have a source sequence with **one real and one padded data point**, and that it went through an encoder to generate the corresponding "keys":

```
source_seq = torch.tensor([[[-1., 1.], [0., 0.]]])
# pretend there's an encoder here...
keys = torch.tensor([[[-.38, .44], [.85, -.05]]])
query = torch.tensor([[[-1., 1.]]])
```

The **source mask** should be **`False`** for every **padded data point**, and its shape should be **(N, 1, L)** where **L** is the **length of the source sequence**.

```
source_mask = (source_seq != 0).all(axis=2).unsqueeze(1)
source_mask # N, 1, L
```

Output

```
tensor([[[ True, False]]])
```

The mask will **make the attention score equals to zero** for the **padded data points**. If we use the "keys" we've just made up to initialize an instance of the *attention mechanism* and call it using the **source mask** above, we'll see the result:

```
torch.manual_seed(11)
attnh = Attention(2)
attnh.init_keys(keys)

context = attnh(query, mask=source_mask)
attnh.alphas
```

Output

```
tensor([[[1., 0.]]])
```

The attention score of the **second data point**, as expected, was **set to zero**, leaving the whole attention to the first data point.

Decoder

We also need to make some **small adjustments** to the **decoder**:

Decoder + Attention

```
class DecoderAttn(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features,
                                self.hidden_dim,
                                batch_first=True)
        self.attn = Attention(self.hidden_dim)              ①
        self.regression = nn.Linear(2 * self.hidden_dim,
                                    self.n_features)        ①

    def init_hidden(self, hidden_seq):
        # the output of the encoder is N, L, H
        # and init_keys expects batch-first as well
        self.attn.init_keys(hidden_seq)                     ②
        hidden_final = hidden_seq[:, -1:]
        self.hidden = hidden_final.permute(1, 0, 2) # L, N, H

    def forward(self, X, mask=None):
        # X is N, 1, F
        batch_first_output, self.hidden = \
                                self.basic_rnn(X, self.hidden)
        query = batch_first_output[:, -1:]
        # Attention
        context = self.attn(query, mask=mask)              ③
        concatenated = torch.cat([context, query],
                                 axis=-1)                  ③
        out = self.regression(concatenated)

        # N, 1, F
        return out.view(-1, 1, self.n_features)
```

① Sets attention module and adjusts input dimensions of the regression layer

② Sets the "keys" (and "values") for the attention module

③ Feeds the "query" to the attention mechanism and concatenates it to the context vector

Let's go over a simple example in code, using the updated **decoder** and **attention** classes:

```
full_seq = (torch.tensor([[-1, -1], [-1, 1], [1, 1], [1, -1]])
            .float()
            .view(1, 4, 2))
source_seq = full_seq[:, :2]
target_seq = full_seq[:, 2:]
```

```
torch.manual_seed(21)
encoder = Encoder(n_features=2, hidden_dim=2)
decoder_attn = DecoderAttn(n_features=2, hidden_dim=2)

# Generates hidden states (keys and values)
hidden_seq = encoder(source_seq)
decoder_attn.init_hidden(hidden_seq)

# Target sequence generation
inputs = source_seq[:, -1:]
target_len = 2
for i in range(target_len):
    out = decoder_attn(inputs)
    print(f'Output: {out}')
    inputs = out
```

Output

```
Output: tensor([[[-0.3555, -0.1220]]], grad_fn=<ViewBackward>)
Output: tensor([[[-0.2641, -0.2521]]], grad_fn=<ViewBackward>)
```

The code above does the bare minimum to generate a target sequence using the attention mechanism. To actually **train a model** using **teacher forcing**, we need to put the **two (or three) classes together**…

Encoder + Decoder + Attention

The integration of encoder, decoder, and the attention mechanism, when applied to our sequence-to-sequence problem, is depicted in the figure below (that's the *same figure* from the "*Computing the Context Vector*" section):

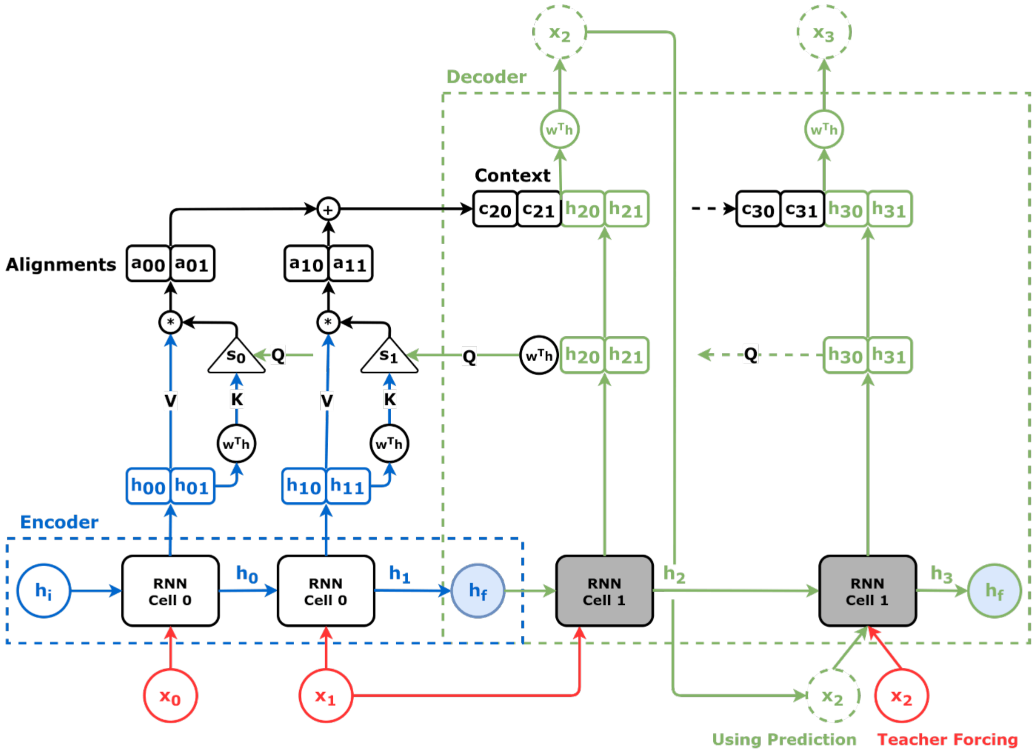


Figure 9.17 - Encoder + Decoder + Attention

Take your time to visualize the flow of information:

- first, the data points in the **source sequence** (in red) feed the **encoder** (in blue) and generate **"keys" (K)** and **"values" (V)** to the **attention mechanism** (in black)
- next, **each input** of the **decoder** (in green) generates **one "query" (Q) at a time** to produce a **context vector** (in black)
- finally, the context vector gets **concatenated** to the decoder's current **hidden state** (in green) and transformed to **predicted coordinates** (in green) by the **output layer** (in green)

Our former `EncoderDecoder` class works seamlessly with an instance of `DecoderAttn`:

```
encdec = EncoderDecoder(encoder, decoder_attn,
                        input_len=2, target_len=2,
                        teacher_forcing_prob=0.0)
encdec(full_seq)
```

Output

```
tensor([[[-0.3555, -0.1220],
         [-0.2641, -0.2521]]], grad_fn=<CopySlices>)
```

We *could* use it to train a model already… but we would *miss* something interesting: **visualizing the attention scores**. To visualize them, we need to **store** them first. The easiest way is to create a new class that **inherits** from `EncoderDecoder` and to **override** the `init_outputs` and `store_outputs` methods:

Encoder + Decoder + Attention

```python
class EncoderDecoderAttn(EncoderDecoder):
    def __init__(self, encoder, decoder, input_len, target_len,
                 teacher_forcing_prob=0.5):
        super().__init__(encoder, decoder, input_len, target_len,
                         teacher_forcing_prob)
        self.alphas = None

    def init_outputs(self, batch_size):
        device = next(self.parameters()).device
        # N, L (target), F
        self.outputs = torch.zeros(batch_size,
                              self.target_len,
                              self.encoder.n_features).to(device)
        # N, L (target), L (source)
        self.alphas = torch.zeros(batch_size,
                                  self.target_len,
                                  self.input_len).to(device)

    def store_output(self, i, out):
        # Stores the output
        self.outputs[:, i:i+1, :] = out
        self.alphas[:, i:i+1, :] = self.decoder.attn.alphas
```

The **attention scores** are stored in the `alphas` attribute of the **attention model** which, in turn, is the **decoder's `attn` attribute**. For each step in the target sequence generation, the corresponding scores are copied to the `alphas` attribute of the `EncoderDecoderAttn` model.

IMPORTANT: pay **attention** (pun very much intended!) to the **shape** of the `alphas` attribute: **(N, L_{target}, L_{source})**. For each one out of *N* sequences in a mini-batch, there is a **matrix**, where each **"query" (Q)** coming from the **target sequence** (a *row* in this matrix) has as many **attention scores** as there are **"keys" (K)** in the **source sequence** (the *columns* in this matrix).

We'll **visualize** these matrices shortly. Moreover, a proper understanding of *how* attention scores are organized in the `alphas` attribute will make it much easier to understand the next section: "*Self-Attention*".

Model Configuration & Training

We just have to replace the original classes for both **decoder** and **model** with their **attention** counterparts and we're good to go:

Model Configuration

```
torch.manual_seed(17)
encoder = Encoder(n_features=2, hidden_dim=2)
decoder_attn = DecoderAttn(n_features=2, hidden_dim=2)
model = EncoderDecoderAttn(encoder, decoder_attn,
                           input_len=2, target_len=2,
                           teacher_forcing_prob=0.5)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_seq_attn = StepByStep(model, loss, optimizer)
sbs_seq_attn.set_loaders(train_loader, test_loader)
sbs_seq_attn.train(100)
```

```
fig = sbs_seq_attn.plot_losses()
```

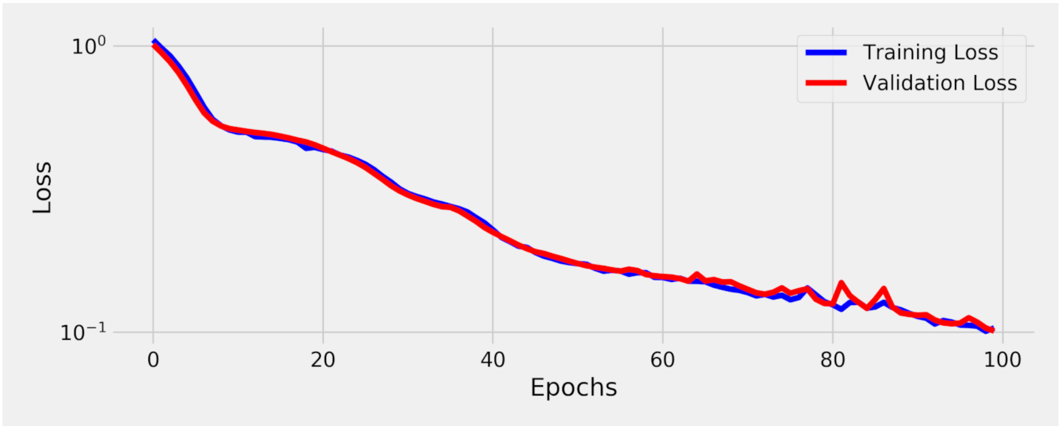


Figure 9.18 - Losses - Encoder + Decoder + Attention

The loss is **one order of magnitude smaller** than the previous, *distracted* (without attention?!), model. That looks promising!

Visualizing Predictions

Let's plot the **predicted coordinates** and connect them using **dashed lines**, while using **solid lines** to connect the **actual coordinates**, just like before:

```
fig = sequence_pred(sbs_seq_attn, full_test, test_directions)
```

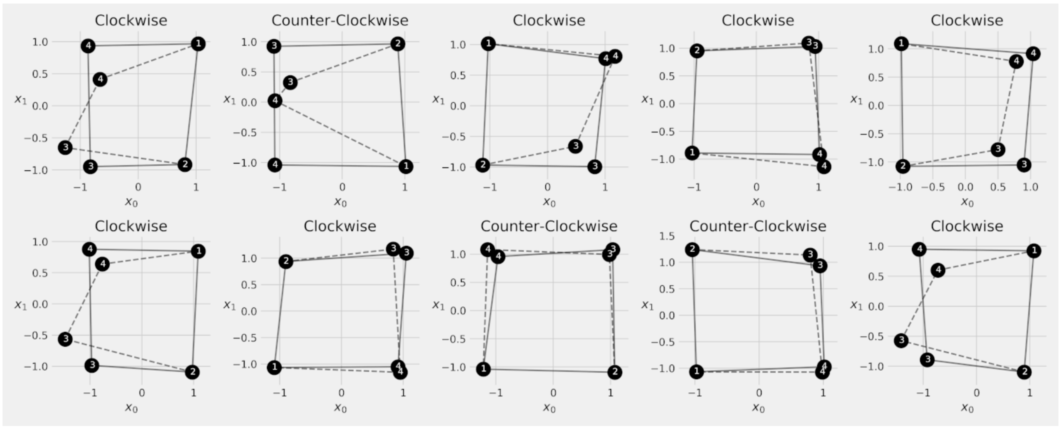


Figure 9.19 - Predicting the last two corners

Much, much better! No overlapping corners, no, Sir! The new model is definitely paying attention :-)

Visualizing Attention

Let's look at what the model is paying attention to by checking what's stored in the `alphas` attribute. The scores will be different for each source sequence, so let's try making predictions for the very first one:

```
inputs = full_train[:1, :2]
out = sbs_seq_attn.predict(inputs)
sbs_seq_attn.model.alphas
```

Output

```
tensor([[[0.0011, 0.9989],
         [0.0146, 0.9854]]], device='cuda:0')
```

"How do I interpret these attention scores?"

The **columns** represent the elements in the **source sequence**, and the **rows**, those in the **target sequence**:

$$\begin{array}{c|cc} & \text{source} & \\ \text{target} & x_0 & x_1 \\ \hline x_2 & \alpha_{20} & \alpha_{21} \\ x_3 & \alpha_{30} & \alpha_{31} \end{array} \implies \begin{array}{c|cc} & \text{source} & \\ \text{target} & x_0 & x_1 \\ \hline x_2 & 0.0011 & 0.9989 \\ x_3 & 0.0146 & 0.9854 \end{array}$$

Equation 9.10 - Attention score matrix

The attention scores we got tell us that the model **mostly paid attention to the second data point** of the **source sequence.** This is *not* going to be the case for every sequence, though. Let's check what the model is paying attention to for the first ten sequences in the training set:

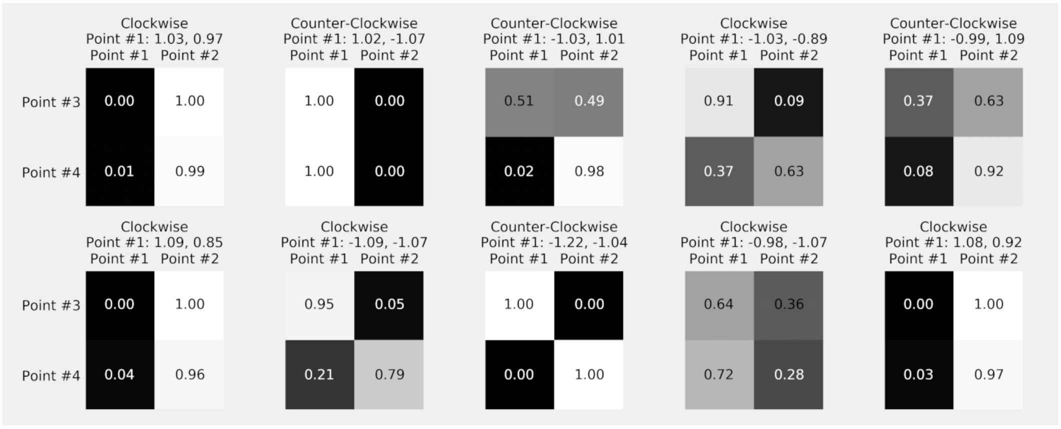


Figure 9.20 - Attention scores

See? For the *second sequence*, it **only** paid attention to the **first data point**. The model picks and chooses what it's going to look at depending on the inputs. How *amazing* is that?

Do you know what's even **better than one attention mechanism**?

Multi-Headed Attention

There is no reason to stick with **only one attention mechanism**: we can use **several** attention mechanisms at once, each one being referred to as an **attention head.**

Each attention head will output **its own context vector**, and they will all get **concatenated together** and **combined** using a **linear layer**. In the end, the **multi-headed attention mechanism** will *still* output a **single context vector**.

Wide vs Narrow Attention

This mechanism is known as **wide attention**: each **attention head** gets the **full "hidden state"** and produces a **context vector** of the **same size**. This is totally fine if the **number of "hidden dimensions"** is **small**.

For a **larger number of dimensions**, though, each **attention head** will get a **chunk** of the **affine transformation of the "hidden state"** to work with. This is a detail of **utmost importance**: it is **not a chunk of the original "hidden state"**, but of its transformation.

For example, say there are **512 dimensions** in the hidden state and we'd like to use **eight attention heads**: each **attention head** will work with a chunk of **64 dimensions** only. This mechanism is known as **narrow attention**, and we'll get back to it in the next chapter.

"Which one should I use?"

On the one hand, **wide attention** will likely yield better models compared to using **narrow attention** on the same number of dimensions. On the other hand, **narrow attention** makes it possible to use **more dimensions**, which may improve the quality of the model as well. It's hard to tell you which one is best overall but I *do* can tell you that **state-of-the-art large Transformer models** use **narrow attention**. In our much simpler and smaller model, though, we're sticking with **wide attention**.

The figure below illustrates the flow of information for **two attention heads**:

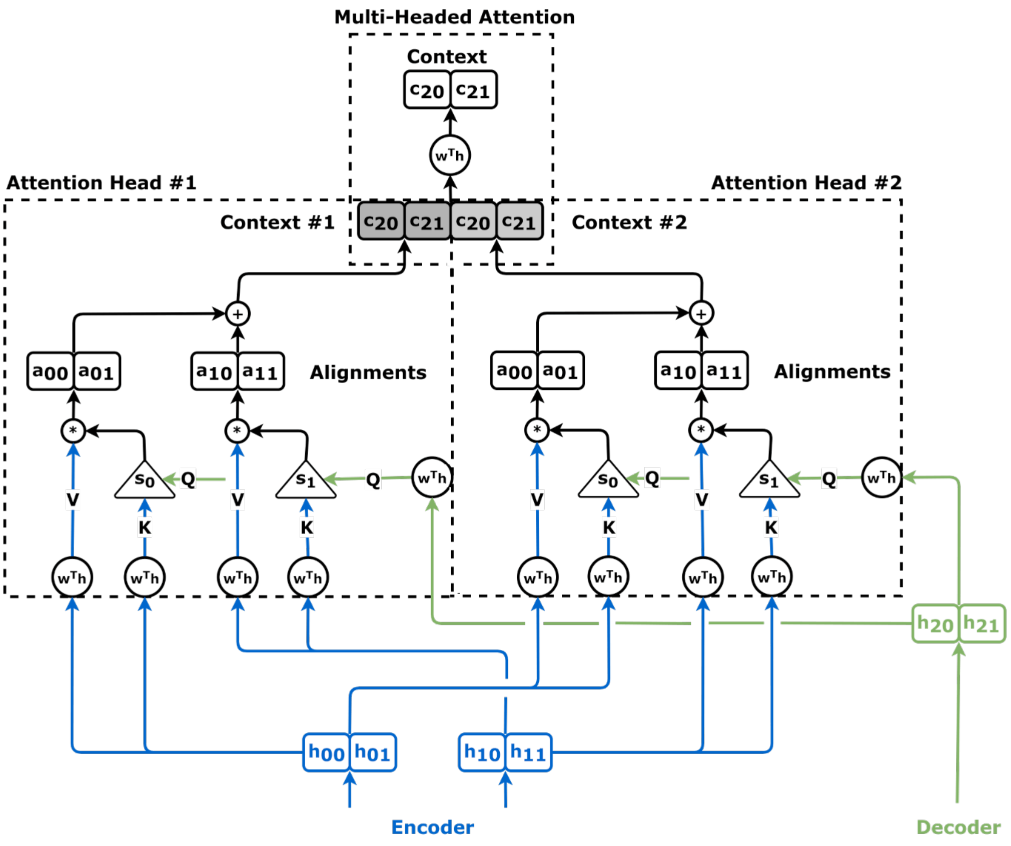


Figure 9.21 - Multi-headed attention mechanism

The **very same** hidden states from both **encoder** ("keys" and "values") and **decoder** ("query") will feed **all attention heads**. At first, you may think that the attention heads will end up being *redundant* (and it may indeed be the case some times) but, thanks to the **affine transformations** of both **"keys" (K)** and **"queries" (Q)**, and **"values" (V)** too, it is more likely that **each attention head** will learn a **distinct pattern**. Cool, right?

"Why are we transforming the "values" now?"

The *multi-headed attention* mechanism is commonly used together with **self-attention** (the next topic), and, as you'll shortly see, the hidden states will be replaced by the **raw data points**. For that reason, we throw *yet another* *transformation* into the mix to give the model a chance to **transform the data points** (which was the role played by the recurrent layer so far).

The **multi-headed attention mechanism** is usually depicted like this:

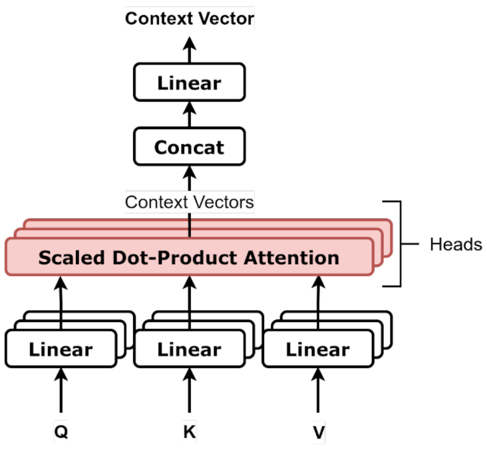


Figure 9.22 - Multi-headed attention mechanism

The code for the multi-headed attention looks like this:

Multi-Headed Attention

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, d_model,
                 input_dim=None, proj_values=True):
        super().__init__()
        self.linear_out = nn.Linear(n_heads * d_model, d_model)
        self.attn_heads = nn.ModuleList(
            [Attention(d_model,
                       input_dim=input_dim,
                       proj_values=proj_values)
             for _ in range(n_heads)]
        )

    def init_keys(self, key):
        for attn in self.attn_heads:
            attn.init_keys(key)

    @property
    def alphas(self):
        # Shape: n_heads, N, 1, L (source)
        return torch.stack(
            [attn.alphas for attn in self.attn_heads], dim=0
        )

    def output_function(self, contexts):
        # N, 1, n_heads * D
        concatenated = torch.cat(contexts, axis=-1)
        # Linear transf. to go back to original dimension
        out = self.linear_out(concatenated) # N, 1, D
        return out

    def forward(self, query, mask=None):
        contexts = [attn(query, mask=mask)
                    for attn in self.attn_heads]
        out = self.output_function(contexts)
        return out
```

It is pretty much a **list** of **attention mechanisms** with an extra linear layer on top. But it is not *any list*, it is a *special* list, it is a `ModuleList`.

"*What's so **special** about it?*"

Even though PyTorch recursively looks for models (and layers) listed as **attributes** to get a comprehensive list of **all parameters**, it **does not** look for models **inside Python lists**. Therefore, the only way to have a **list of models** (or layers) is to use the appropriate `ModuleList`, which you still can index and loop over as any other regular list.

This chapter is *so* long that I've split it into *two parts*, so you can take a break and let the **attention mechanism** sink in before moving on to its sibling, the **self-attention mechanism**.

TO BE CONTINUED...

[133] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter09.ipynb

[134] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter09.ipynb

[135] https://papers.nips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html

[136] https://bit.ly/3aEf81k

[137] http://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

[138] https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3

Chapter 9 - Part II
Sequence-To-Sequence

Spoilers

In the second half of this chapter, we will:

- use **self-attention** mechanisms to replace recurrent layers both in the encoder and the decoder
- understand the importance of the **target mask** to avoid data leakage
- learn how to use **positional encoding**

Self-Attention

Here is some **radical notion**: what about **replacing the recurrent layer** with an **attention mechanism**?

That's the main proposition of the famous "*Attention Is All You Need*"[139] paper by Vaswani, A., et al. It introduced the **Transformer** architecture, based on a self-attention mechanism, that was going to completely dominate the NLP landscape.

> "*I pity the fool using recurrent layers.*"
>
> Mr. T

The recurrent layer in the **encoder** took the **source sequence** in and, *one by one*, generated **hidden states**. But we don't *have to* generate hidden states like that. We can use **another, separate, attention mechanism** to **replace the encoder** (and, wait for it, the decoder too!).

These separate attention mechanisms are called **self-attention** mechanisms since all of its inputs, "keys", "values", and "query" are **internal** to either an **encoder** or a **decoder**.

The attention mechanism we discussed in the previous section, where "keys" and "values" come from the **encoder**, but the "query" comes from the **decoder**, is going to be referred to as **cross-attention** from now on.

Once again, the **affine transformations** we're applying to "keys", "queries", and "values" may also be used to **change the dimensionality** from the number of *input* to the number of *hidden* dimensions (this transformation was formerly performed by the recurrent layer).

Let's start with the encoder.

Encoder

The figure below depicts an **encoder** using **self-attention**: the **source sequence** (in red) works as **"keys" (K)**, **"values" (V)**, and "**queries (Q)"** as well. Did you notice I mentioned, "queries", *plural*, instead of "query"?

In the encoder, **each data point** is a **"query"** (the red arrow entering the self-attention mechanism from the side), and thus produces **its own context vector** using **alignment vectors** for **every data point in the source sequence, including itself**. It means it is possible for a data point to generate a context vector that is only paying attention to itself.

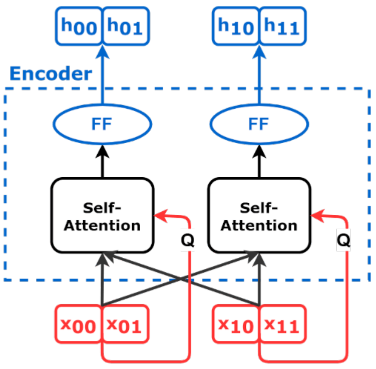


Figure 9.23 - Encoder with self-attention (simplified)

As the diagram above illustrates, each context vector produced by the self-attention mechanism goes through a **feed-forward network** to generate a **"hidden state"** as output. We can also dive deeper into the inner workings of the self-attention mechanisms:

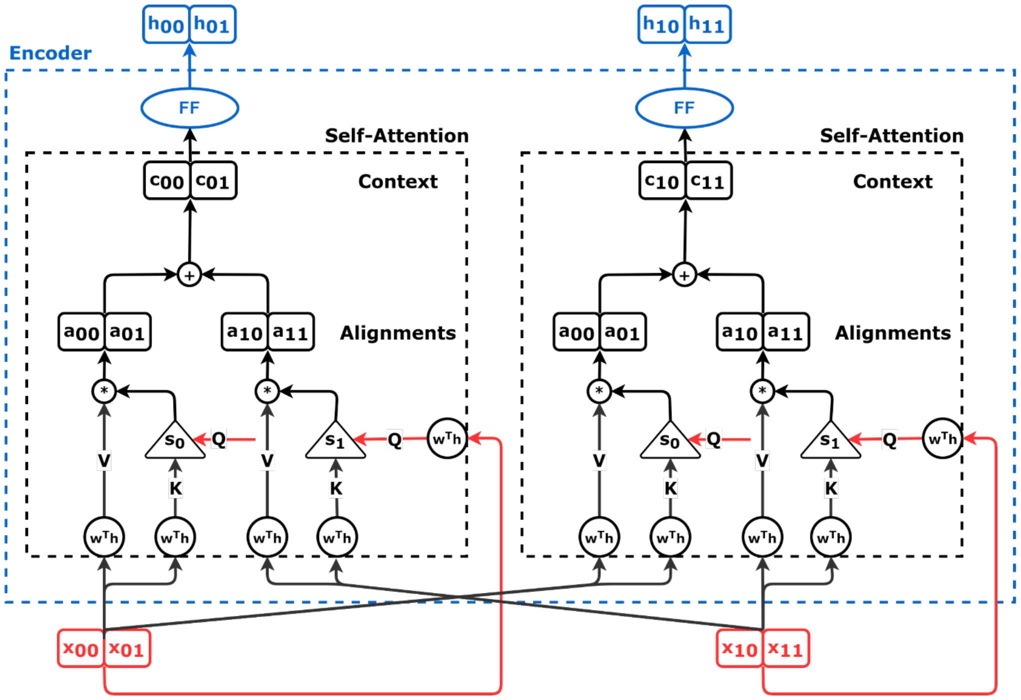


Figure 9.24 - Encoder with self-attention

Let’s focus on the self-attention mechanism on the *left*, used to generate the "hidden state" (h_{00}, h_{01}) corresponding to the **first data point** in the source sequence (x_{00}, x_{01}), and see how it works in great detail:

- the transformed coordinates of the first data point are used as **"query" (Q)**
- this "query" (Q) will be paired, *independently*, with each one of the two "keys" (K), one of them being a *different transformation* of the same coordinates (x_{00}, x_{01}), the other being a transformation of the second data point (x_{10}, x_{11})
- the pairing above will result in **two attention scores** (*alphas*) that, multiplied by their corresponding "values" (V), are added up to become the **context vector**:

$$\alpha_{00}, \alpha_{01} = softmax(\frac{Q_0 \cdot K_0}{\sqrt{2}}, \frac{Q_0 \cdot K_1}{\sqrt{2}})$$
$$context\ vector_0 = \alpha_{00} V_0 + \alpha_{01} V_1$$

Equation 9.11 - Context vector for first input (x_0)

- next, the context vector goes through the **feed-forward network**, and the first hidden state is born

Next, we shift our focus to the self-attention mechanism on the **right**:

- it is the **second data point**'s turn to be the **"query" (Q)**, being paired with both "keys" (K), generating attention scores and a context vector, resulting in the second hidden state:

$$\alpha_{10}, \alpha_{11} = softmax(\frac{Q_1 \cdot K_0}{\sqrt{2}}, \frac{Q_1 \cdot K_1}{\sqrt{2}})$$
$$context\ vector_1 = \alpha_{10} V_0 + \alpha_{11} V_1$$

Equation 9.12 - Context vector for second input (x_1)

As you probably already noticed, the **context vector** (and thus the **"hidden state"**) associated with a data point is basically a **function of the corresponding "query" (Q)**, and everything else ("keys" (K), "values" (V), and the parameters of the self-attention mechanism) is held constant for all queries.

Therefore we can simplify *a bit* our previous diagram and depict **only one self-attention mechanism**, assuming it will be fed a different "query" (Q) every time:

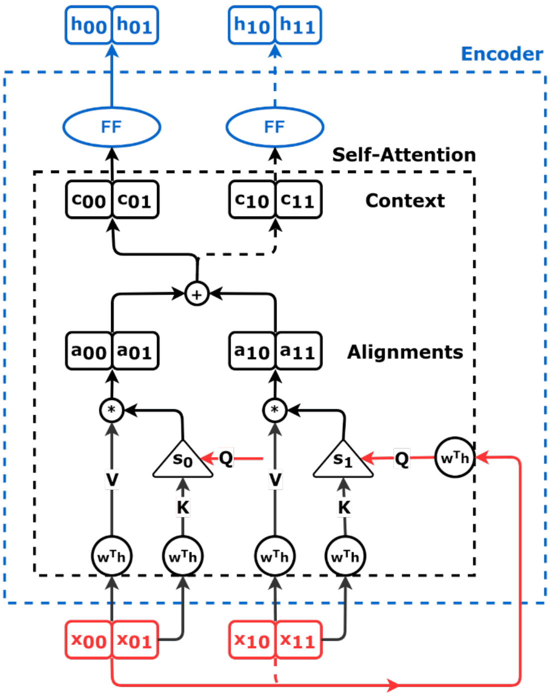


Figure 9.25 - Encoder with self-attention

The **alphas** are the **attention scores**, and they are organized like this in the `alphas` attribute (as we've already seen in the "*Visualizing Attention*" section):

$$\begin{array}{c|cc} & \multicolumn{2}{l}{source} \\ target & x_0 & x_1 \\ \hline h_0 & \alpha_{00} & \alpha_{01} \\ h_1 & \alpha_{10} & \alpha_{11} \end{array}$$

Equation 9.13 - Attention scores

For the **encoder**, the shape of the `alphas` attribute is given by **(N, L$_{source}$, L$_{source}$)** since it is **looking at itself**.

Even though I've described the process *as if* it was sequential, these operations can be **parallelized** to generate all "hidden states" at once, which is much more efficient than using a recurrent layer that is *sequential* in nature.

We can also use an even more **simplified diagram** of the encoder that abstracts the nitty-gritty details of the self-attention mechanism:

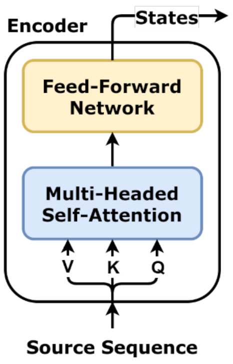


Figure 9.26 - Encoder with self-attention (diagram)

The code for our **encoder** with **self-attention** is actually quite simple since most of the moving parts are inside the attention heads:

Encoder + Self-Attention

```python
class EncoderSelfAttn(nn.Module):
    def __init__(self, n_heads, d_model,
                 ff_units, n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = n_features
        self.self_attn_heads = \
            MultiHeadAttention(n_heads,
                               d_model,
                               input_dim=n_features)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, d_model),
        )

    def forward(self, query, mask=None):
        self.self_attn_heads.init_keys(query)
        att = self.self_attn_heads(query, mask)
        out = self.ffn(att)
        return out
```

Remember that the **"query"** in the `forward` method actually gets the **data points** in the **source sequence**. These data points will be **transformed** into different **"keys"**, **"values"**, and **"queries"** inside each one of the **attention heads**. The output of the attention heads is a **context vector** (`att`) that goes through a **feed-forward network** to produce a **"hidden state"**.

By the way, now that we're got rid of the recurrent layer, we'll be talking about **model dimensions** (`d_model`) instead of **hidden dimensions** (`hidden_dim`). You still get to **choose it**, though.

The **mask** argument should receive the **source mask**, that is, the mask we use to **ignore padded data points** in our **source sequence**.

Let's create an encoder and feed it a source sequence:

```
torch.manual_seed(11)
encself = EncoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
query = source_seq
encoder_states = encself(query)
encoder_states
```

Output

```
tensor([[[-0.0498,  0.2193],
         [-0.0642,  0.2258]]], grad_fn=<AddBackward0>)
```

It produced a **sequence of states** that will be the input of the **(cross-)attention** mechanism used by the **decoder**. Business as usual.

Cross-Attention

The **cross-attention** was the first mechanism we discussed: the **decoder** provided a **"query" (Q)** which served not only as *input* but also got **concatenated to the resulting context vector**. That **won't** be the case anymore… instead of concatenation, the **context vector** will go through a **feed-forward** network in the **decoder** to generate the **predicted coordinates**.

The figure below illustrates the current state of the architecture: self-attention as encoder, cross-attention on top of it, and the modifications to the decoder part:

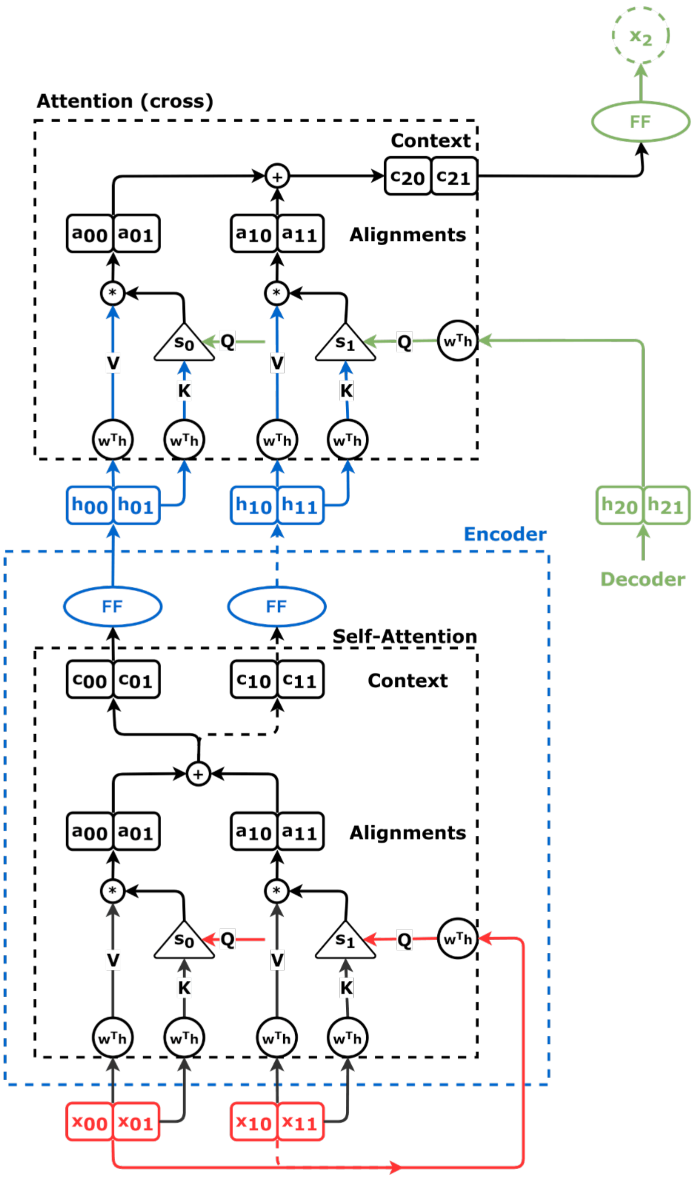


Figure 9.27 - Encoder with self- and cross-attentions

If you're wondering *why* we removed the concatenation part, here comes the answer: we're using **self-attention** as a **decoder** too.

Decoder

There is **one main difference** (in the code) between the **encoder** and the **decoder**: the latter includes a **cross-attention** mechanism, as you can see below:

Decoder + Self-Attention

```python
class DecoderSelfAttn(nn.Module):
    def __init__(self, n_heads, d_model,
                 ff_units, n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = d_model if n_features is None \
                          else n_features
        self.self_attn_heads = \
            MultiHeadAttention(n_heads, d_model,
                               input_dim=self.n_features)
        self.cross_attn_heads = \
            MultiHeadAttention(n_heads, d_model)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, self.n_features))

    def init_keys(self, states):                      ①
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        self.self_attn_heads.init_keys(query)
        att1 = self.self_attn_heads(query, target_mask)
        att2 = self.cross_attn_heads(att1, source_mask)   ①
        out = self.ffn(att2)
        return out
```

① Including **cross-attention**

The figure below depicts the **self-attention** part of a **decoder**:

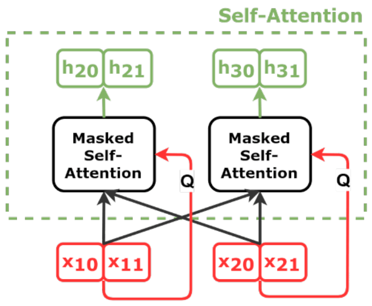


Figure 9.28 - Decoder with self-attention (simplified)

Once again, we can also dive deeper into the inner workings of the self-attention mechanism:

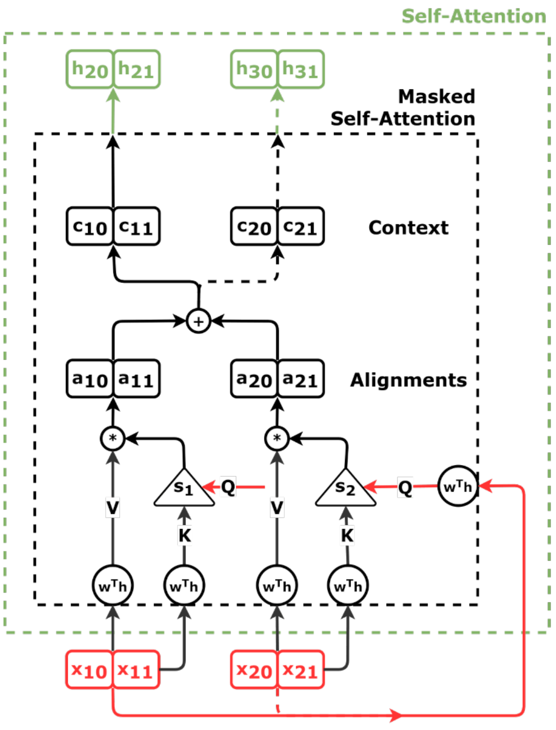


Figure 9.29 - Decoder with self-attention

There is **one small difference in the self-attention** architecture between encoder and decoder: the **feed-forward network** sits atop the *cross-attention* mechanism (not depicted in the figure above) instead of the *self-attention* mechanism. The *feed-forward network* also maps the **decoder's output** from the dimensionality of the model (`d_model`) back to the **number of features**, thus yielding **predictions**.

We can also use a **simplified diagram** for the decoder (Figure 9.29, although depicting a *single attention head*, corresponds to the "*Masked Multi-Headed Self-Attention*" box):

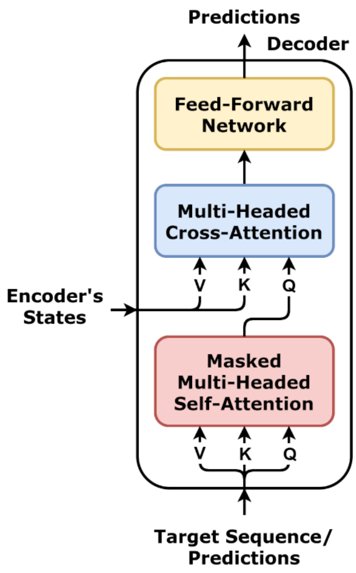

Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
V
K
Q
Encoder's States
Masked Multi-Headed Self-Attention
V
K
Q
Target Sequence/ Predictions

Figure 9.30 - Decoder with self- and cross-attentions (diagram)

The decoder's **first input** (x_{10}, x_{11}) is the **last known element of the source sequence**, as usual. The **source mask** is the same **mask used to ignore padded data points** in the **encoder**.

"*What about the **target mask**?*"

We'll get to that shortly. First, we need to discuss the **subsequent inputs**.

Subsequent Inputs and Teacher Forcing

In our problem, the first two data points are the *source sequence*, while the last two, the *target sequence*. Now, let's define the **shifted target sequence**, which includes the **last known element of the source sequence** and **all elements in the target sequence *but* the last one**:

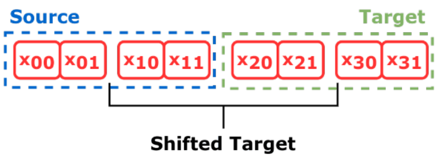


Figure 9.31 - Shifted target sequence

```
shifted_seq = torch.cat([source_seq[:, -1:],
                         target_seq[:, :-1]], dim=1)
```

The *shifted target sequence* was already used (even though we didn't have a name for it) when we discussed *teacher forcing*. There, at every step (after the first one), it randomly chose as the input to the subsequent step either an actual element from that sequence or a prediction. It worked very well with recurrent layers that were *sequential* in nature. But this **isn't** the case anymore.

One of the **advantages** of **self-attention over recurrent layers** is that operations can be **parallelized**. No need to do anything *sequentially* anymore, **teacher forcing included**. This means we're using **the whole shifted target sequence at once** as the **"query"** argument of the **decoder**.

That's very nice and cool, sure, but it raises **one big problem** involving the...

Attention Scores

To understand what the problem is, let's look at the **context vector** that will result in the **first "hidden state"** produced by the **decoder**, which, in turn, will lead to the

first prediction:

$$\alpha_{21}, \alpha_{22} = softmax(\frac{Q_1 \cdot K_1}{\sqrt{2}}, \frac{Q_1 \cdot K_2}{\sqrt{2}})$$

$$context\ vector_2 = \alpha_{21} V_1 + \alpha_{22} V_2$$

Equation 9.14 - Context vector for the first target

"*What's the problem with it?*"

The problem is that it is **using a "key" (K_2) and a "value" (V_2)** that are transformations of the **data point it is trying to predict**.

In other words, the model is being allowed to **cheat** by **peeking into the future** because we're giving it **all data points in the target sequence but the very last one**.

If we look at the context vector corresponding to the last prediction, it should be clear that the model simply *cannot* cheat:

$$\alpha_{31}, \alpha_{32} = softmax(\frac{Q_2 \cdot K_1}{\sqrt{2}}, \frac{Q_2 \cdot K_2}{\sqrt{2}})$$

$$context\ vector_3 = \alpha_{31} V_1 + \alpha_{32} V_2$$

Equation 9.15 - Context vector for the second target

We can also check it quickly by looking at the **subscript indices**: as long as the **indices of the "values"** are **smaller** than the **index of the context vector**, there is **no cheating**. By the way, it is even *easier* to check what's happening if we use the *alphas* matrix:

$$\begin{array}{c|cc} & source & \\ target & x_1 & x_2 \\ \hline h_2 & \alpha_{21} & \alpha_{22} \\ h_3 & \alpha_{31} & \alpha_{32} \end{array}$$

Equation 9.16 - Decoder's attention scores

For the **decoder**, the shape of the `alphas` attribute is given by **(N, L_{target}, L_{target})** since it is **looking at itself**. Any *alphas* **above the diagonal** are, literally, **cheating codes**. We need to **force** the self-attention mechanism to **ignore them**. If only there was a way to do it…

"*What about those **masks** we discussed earlier?*"

You're absolutely right! They are *perfect* for this case.

Target Mask (Training)

The purpose of the **target mask** is to **zero attention scores for "future" data points**. In our example, that's the *alphas* matrix we're aiming for:

$$\begin{array}{c|cc} & source & \\ target & x_1 & x_2 \\ \hline h_2 & \alpha_{21} & 0 \\ h_3 & \alpha_{31} & \alpha_{32} \end{array}$$

Equation 9.17 - Decoder's (masked) attention scores

Therefore we need a **mask** that **flags every element above the diagonal** as **invalid** as we did with the *padded data points* in the *source mask*. The **shape** of the **target mask**, though, must **match** the shape of the `alphas` attribute: **(1, L_{target}, L_{target})**.

We can create a function to **generate the mask** for us:

Subsequent Mask

```
def subsequent_mask(size):
    attn_shape = (1, size, size)
    subsequent_mask = (
        1 - torch.triu(torch.ones(attn_shape), diagonal=1)
    ).bool()
    return subsequent_mask
```

```
subsequent_mask(2) # 1, L, L
```

Output

```
tensor([[[ True, False],
         [ True,  True]]])
```

Perfect! The element above the diagonal is indeed set to `False`.

We **must** use this mask while querying the decoder to **prevent it from cheating**. You can choose to use an **additional mask** to "hide" more data from the decoder if you wish, but the **subsequent mask is a strong requirement** of the self-attention decoder.

Let's see it in practice:

```
torch.manual_seed(13)
decself = DecoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
decself.init_keys(encoder_states)

query = shifted_seq
out = decself(query, target_mask=subsequent_mask(2))

decself.self_attn_heads.alphas
```

Output

```
tensor([[[[1.0000, 0.0000],
          [0.4011, 0.5989]]],


        [[[1.0000, 0.0000],
          [0.4264, 0.5736]]],


        [[[1.0000, 0.0000],
          [0.6304, 0.3696]]]])
```

There we go, no cheating :)

Target Mask (Evaluation/Prediction)

The only difference between training and evaluation, concerning the **target mask**, is that we'll be using **larger masks** as we go. The very first mask is actually trivial since there are *no* elements above the diagonal:

$$1^{st}\ Step \left\{ \begin{array}{c|c} & source \\ target & x_1 \\ \hline h_2 & \alpha_{21} \end{array} \right.$$

Equation 9.18 - Decoder's (masked) attention scores for the first target

In evaluation/prediction time we *only* have the source sequence and, in our example, we use its last element as input for the decoder:

```
inputs = source_seq[:, -1:]
trg_masks = subsequent_mask(1)
out = decself(inputs, trg_masks)
out
```

Output

```
tensor([[[0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

The mask is not actually masking anything in this case, and we get a prediction for the coordinates of $\mathbf{x_2}$ as expected. Previously, this prediction would be used directly as the next input, but things are a *bit* different now…

The **self-attention decoder** expects the **full sequence** as **"query"**, so we **concatenate the prediction** to the **previous "query"**.

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

Now there are **two data points** for querying the decoder, so we adjust the mask accordingly:

$$2^{nd}\ Step \left\{ \begin{array}{c|cc} & source & \\ target & x_1 & x_2 \\ \hline h_2 & \alpha_{21} & 0 \\ h_3 & \alpha_{31} & \alpha_{32} \end{array} \right.$$

Equation 9.19 - Decoder's (masked) attention scores for the second target

The **mask** guarantees that the **predicted** ***x₂*** (in the first step) **won't change** the **predicted** ***x₂*** (in the second step) because predictions are made based on **past data points only**.

```
trg_masks = subsequent_mask(2)
out = decself(inputs, trg_masks)
out
```

Output

```
tensor([[[0.4137, 0.3727],
         [0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

These are the **predicted coordinates** of both x_2 and x_3. They are very close to each other, but that's just because we're using an *untrained model* to illustrate the mechanics of using target masks for prediction. The **last prediction** is, once again, **concatenated** to the previous "query".

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

But, since we're *actually* done with the predictions (the desired target sequence has a length of two), we simply *exclude* the first data point in the query (the one coming from the source sequence) and that's the **predicted target sequence**:

```
inputs[:, 1:]
```

Output

```
tensor([[[0.4132, 0.3728],
         [0.4132, 0.3728]]], grad_fn=<SliceBackward>)
```

Greedy Decoding vs Beam Search

This is called **greedy decoding** because **each prediction** is deemed **final**. "*No backsies*": once it's done, it's *really* done, you just move along to the next prediction and never look back. In the context of our sequence-to-sequence problem, a regression, it wouldn't make much sense to do otherwise anyway.

But that **may not** be the case for other types of sequence-to-sequence problems. In machine translation, for example, the decoder outputs **probabilities** for the **next word** in the sentence at each step. The **greedy** approach would simply take the **word with the highest probability** and move on to the next.

However, since **each prediction** is an input to the **next step**, taking the **top word at every step** is **not necessarily the winning approach** (translating from one language to another is not exactly "linear"). It is probably wiser to keep a **handful of candidates** at every step and **try their combinations** to choose the best one: that's called **beam search**. We're not delving into its details here, but you can find more information in Jason Brownlee's "*How to Implement a Beam Search Decoder for Natural Language Processing*"[140].

Encoder + Decoder + Self-Attention

Let's join the encoder and the decoder together again, each one using **self-attention** to compute their corresponding "hidden states", and the decoder using **cross-attention** to make predictions. The full picture looks like this (including the need for *masking* one of the inputs to avoid *cheating*):

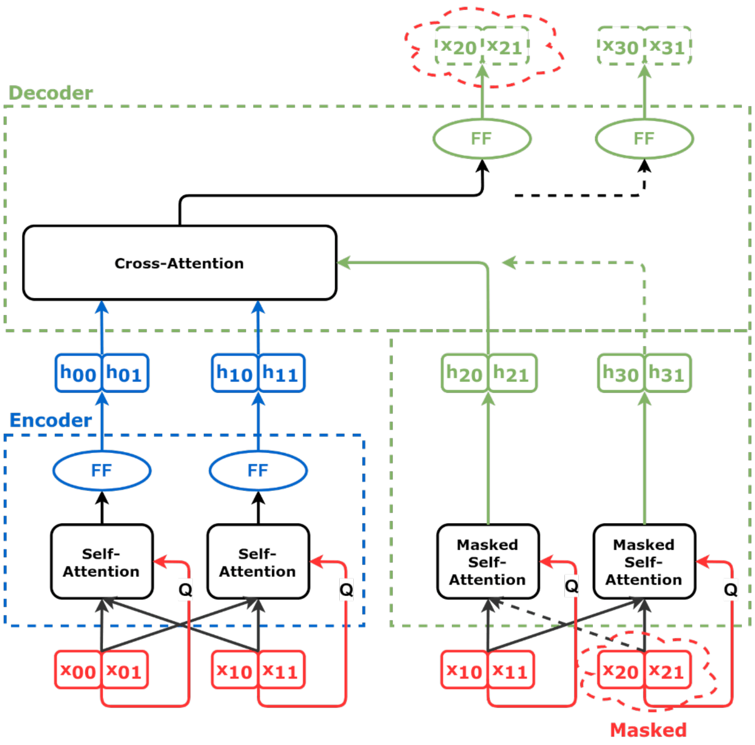


Figure 9.32 - Encoder + Decoder + Attention (simplified)

For some cool **animations** of the self-attention mechanism, make sure to check Raimi Karim's *"Illustrated: Self-Attention"*[141].

But, if you prefer an even more **simplified diagram**, here it is:

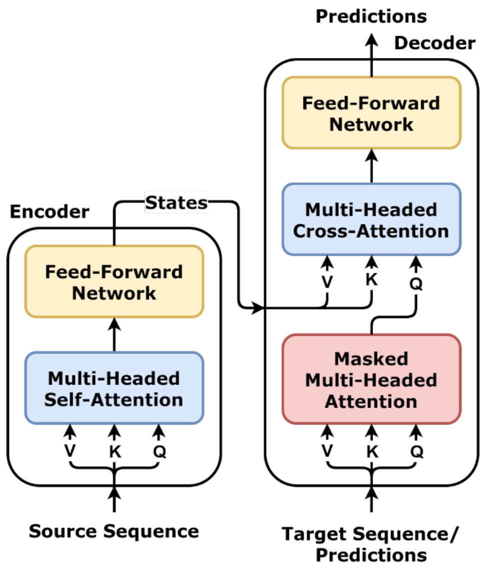

Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
V
K
Q
Masked Multi-Headed Attention
V
K
Q
Target Sequence/ Predictions
Encoder
States
Feed-Forward Network
Multi-Headed Self-Attention
V
K
Q
Source Sequence

Figure 9.33 - Encoder + Decoder + Attention (diagram)

The corresponding code for the architecture above looks like this:

Encoder + Decoder + Self-Attention

```
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)

    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequent_mask = (
            1 - torch.triu(torch.ones(attn_shape), diagonal=1)
        ).bool()
        return subsequent_mask
```

```python

    def encode(self, source_seq, source_mask):
        # Encodes the source sequence and uses the result
        # to initialize the decoder
        encoder_states = self.encoder(source_seq, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Decodes/generates a sequence using the shifted (masked)
        # target sequence - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs

    def predict(self, source_seq, source_mask):
        # Decodes/generates a sequence using one input
        # at a time - used in EVAL mode
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.decode(inputs,
                              source_mask,
                              self.trg_masks[:, :i+1, :i+1])
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = inputs[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        # Sends the mask to the same device as the inputs
        self.trg_masks = self.trg_masks.type_as(X).bool()
        # Slices the input to get source sequence
        source_seq = X[:, :self.input_len, :]
        # Encodes source sequence AND initializes decoder
        self.encode(source_seq, source_mask)
        if self.training:
            # Slices the input to get the shifted target seq
            shifted_target_seq = X[:, self.input_len-1:-1, :]
            # Decodes using the mask to prevent cheating
            outputs = self.decode(shifted_target_seq,
                                  source_mask,
                                  self.trg_masks)
        else:
            # Decodes using its own predictions
            outputs = self.predict(source_seq, source_mask)

        return outputs
```

The encoder-decoder class has *more methods* now to better organize the several steps formerly performed inside the `forward` method. Let's take a look at them:

- `encode`: takes the **source sequence and mask** and **encodes** it into a **sequence of states** that is immediately used to **initialize the "keys" (and "values")** in the **decoder**
- `decode`: takes the **shifted target sequence** and both **source and target masks** to generate a **target sequence - it is used for training** only
- `predict`: takes the **source sequence**, the **source mask**, and uses a subset of the **target mask** to **actually predict an unknown target sequence** - it is used for **evaluation/prediction** only
- `forward`: it splits the input into the **source** and **shifted target sequences** (if available), **encodes** the source sequence, and calls either `decode` or `predict` according to the model's *mode* (`train` or `eval`)

Moreover, the `subsequent_mask` became a *static method*, the mask is being generated in the constructor, and it is **sent to the same device as the inputs** using `tensor.type_as`. The last part is critical: we need to make sure that the *mask is in the same device* as the inputs (and the model, of course).

Model Configuration & Training

Once again, we create both encoder and decoder models, use them as arguments to the big `EncoderDecoderSelfAttn` model that handles the boilerplate, and we're good to go:

Model Configuration

```
torch.manual_seed(23)
encself = EncoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
decself = DecoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
model = EncoderDecoderSelfAttn(encself, decself,
                               input_len=2, target_len=2)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_seq_selfattn = StepByStep(model, loss, optimizer)
sbs_seq_selfattn.set_loaders(train_loader, test_loader)
sbs_seq_selfattn.train(100)
```

```
fig = sbs_seq_selfattn.plot_losses()
```

Even though we did our best to ensure the **reproducibility** of the results, you may **still** find some difference in the loss curves (and, consequently, in the attention scores as well). PyTorch's documentation about reproducibility states:

> *"Completely reproducible results are not guaranteed across PyTorch releases, individual commits, or different platforms. Furthermore, results may not be reproducible between CPU and GPU executions, even when using identical seeds."*

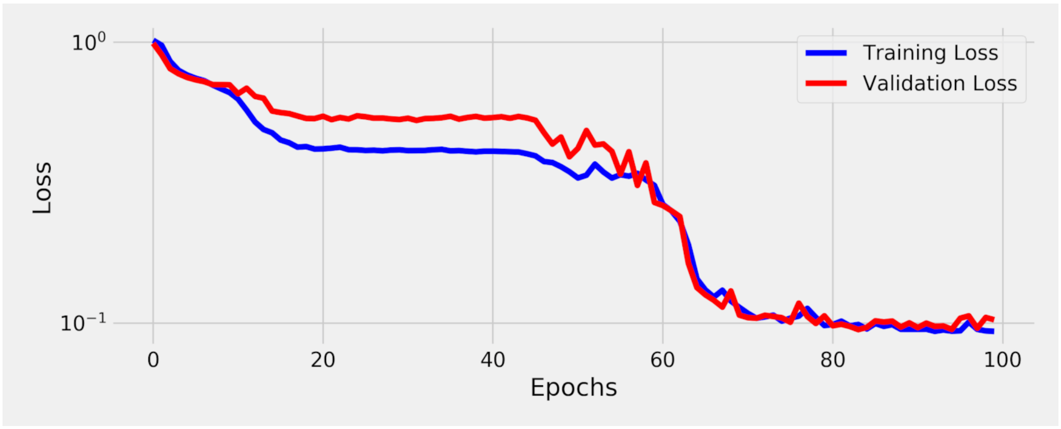


Figure 9.34 - Losses - Encoder + Decoder + Self-Attention

The losses are *worse* now… the model using *cross-attention only* was performing better than that. What about the predictions?

Visualizing Predictions

Let's plot the **predicted coordinates** and connect them using **dashed lines**, while using **solid lines** to connect the **actual coordinates**, just like before:

```
fig = sequence_pred(sbs_seq_selfattn, full_test, test_directions)
```

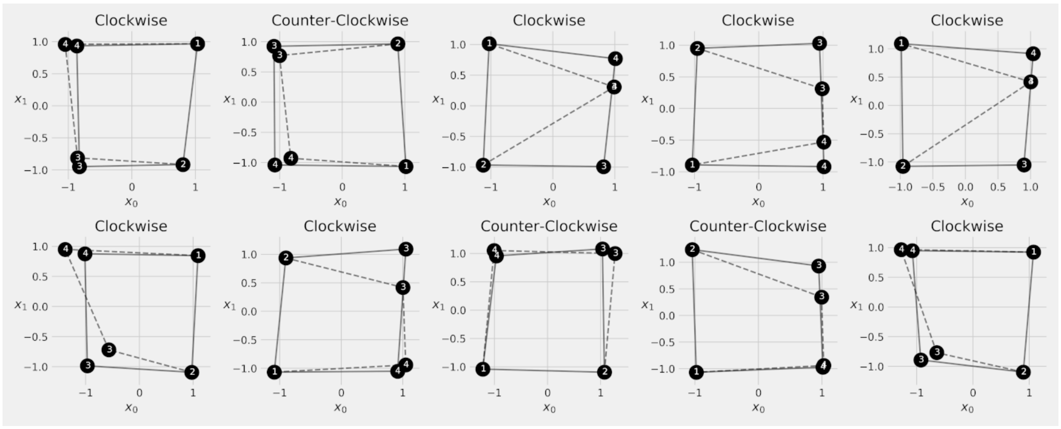


Figure 9.35 -

Well, that's a bit *disappointing*… the *triangles* made a comeback!

To be completely honest with you, it is perfectly feasible to achieve a **much better loss** (and no *triangles*) using the model above with a small tweak, namely, trying a *different seed*. But I decided to **keep the model above** for the sake of **highlighting the importance** of our next topic, **positional information**.

"*What happened here? Wasn't **self-attention** the best invention since the sliced bread?*"

Self-attention is great indeed, but it **misses one fundamental piece of information** that the recurrent layers had: the **order of the data points**. As we know, the **order** is of **utmost importance** in sequence problems but, for the self-attention mechanism, there is **no order** to data points in the **source sequence**.

"*I don't get it, **why** did it lose the order?*"

Sequential No More

Let's compare two **encoders**, one using **recurrent neural networks** (left), the other, **self-attention** (right):

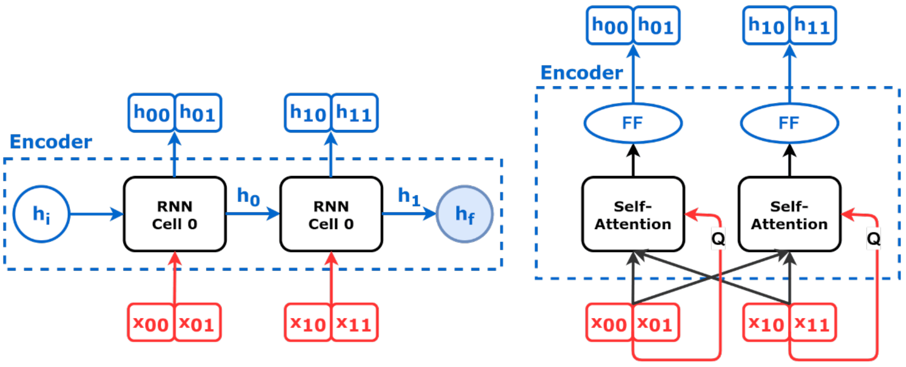


Figure 9.36 - RNN vs Self-Attention

The recurrent neural network ensured that the **output at each step is fed to the next**, depicted by the **h**$_0$ going from one cell to the next. Now, compare it to the encoder using **self-attention**: **every step is *independent* of the others**. If you *flip* the order of the data points in the source sequence, the encoder will output the "hidden states" in a different order, but it **won't change their values**.

That's exactly what makes it highly *parallelizable*, and it is both a **blessing** and a **curse**: on the one hand, it makes computation very efficient; on the other hand, it is **throwing away valuable information**.

"Can we fix it?"

Definitely! Instead of using a model designed to encode the order of the inputs (like the recurrent neural networks), let's **encode the positional information** ourselves and **add them to the inputs**.

Positional Encoding (PE)

We need to find a way to inform the model about the **position** of every **data point** such that it knows the **order of the data points**. In other words, we need to generate a **unique value** for each **position** in the input.

Let's put our simple sequence-to-sequence problem aside for a while, and imagine we have a **sequence of four data points** instead. The first idea that comes to mind is to use the **index** of the position itself, right? Let’s just use zero, one, two, and three, done! Maybe it wouldn’t be *so* bad for a short sequence like this but what about a sequence of, say, **one thousand** data points? The positional encoding of the last data point would be 999. We *shouldn’t* be using *unbounded values* like that as inputs for a neural network.

"What about "normalizing" the indices?"

Sure, we can try that and divide the indices by the *length* of our sequence (four):

Position	0	1	2	3
Pos/4	0.00	0.25	0.50	0.75

Figure 9.37 - "Normalizing" over the length

Unfortunately, that **didn’t** solve the problem… a **longer** sequence will **still generate values larger than one**:

Position	0	1	2	3	4	5	6	7
Pos/4	0.00	0.25	0.50	0.75	1.00	1.25	1.50	1.75

Figure 9.38 - "Normalizing" over a (shorter) length

"What about "normalizing" each sequence by its own length?"

It solves *that* problem but it raises **another** one, namely, the **same position** gets **different encoding** values depending on the length of the sequence:

Position	0	1	2	3	4	5	6	7
Pos/4	0.00	0.25	0.50	0.75				
Pos/8	0.00	0.13	0.25	0.38	0.50	0.63	0.75	0.88

Figure 9.39 - "Normalizing" over different lengths

Ideally, the **positional encoding** should remain **constant for a given position**, regardless of the length of the sequence.

"What if we take the ***module*** *first, and* ***then*** *"normalize" it?"*

Well, it indeed solves the two problems above, but the values **aren't unique anymore**:

Position	0	1	2	3	4	5	6	7
(Pos mod 4)/4	0.00	0.25	0.50	0.75	0.00	0.25	0.50	0.75

Figure 9.40 - "Normalizing" over a module of the length

"OK, I give up! How do we handle this?"

Let's think outside the box for a moment… no one said we must use *only one vector*, right? Let's build **three vectors** instead, using three hypothetical sequence lengths (four, five, and seven):

Position	0	1	2	3	4	5	6	7
(Pos mod 4)/4	0.00	0.25	0.50	0.75	0.00	0.25	0.50	0.75
(Pos mod 5)/5	0.00	0.20	0.40	0.60	0.80	0.00	0.20	0.40
(Pos mod 7)/7	0.00	0.14	0.29	0.43	0.57	0.71	0.86	0.00

Figure 9.41 - Combining results for different modules

The positional encoding above is **unique** up to the 140th position and we can easily extend that by adding more vectors.

"Are we done now? Is this good enough?"

Sorry, but no, not yet. Our solution still has one problem and it boils down to computing **distances** between two encoded positions. Let's take **position number three** and its two neighbors, positions number two and four. Obviously, the distance between position three and each of its closest neighbors is **one**. Now, let's see what happens if we compute distance using the positional encoding:

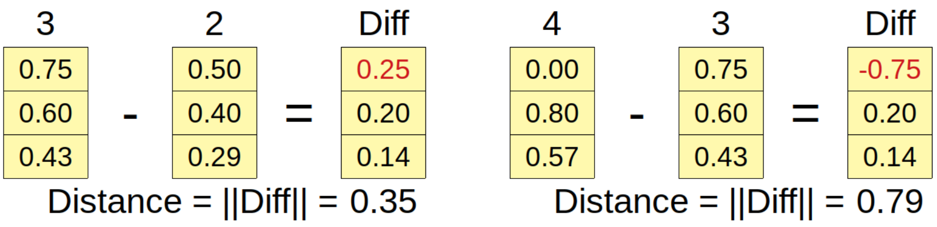


Figure 9.42 - Inconsistent distances

The distance between positions three and two (given by the norm of the difference vector) **is not equal** to the distance between positions three and four. That may seem a bit *too abstract*, but using an encoding with **inconsistent distances** would make it much harder for the model to make sense out of the encoded positions.

This inconsistency arises from the fact that our encoding is **resetting to zero** every time the **module** kicks in. The distance between positions three and four got much larger because, at position four, the first vector goes back to zero. We need some other function that has a **smoother cycle**…

*"What if we **actually** use a **cycle**, I mean, a **circle**?"*

Perfect! First, we take our encodings and **multiply them by 360**:

Position	0	1	2	3	4	5	6	7
Base 4	0	90	180	270	0	90	180	270
Base 5	0	72	144	216	288	0	72	144
Base 7	0	51	103	154	206	257	309	0

Figure 9.43 - From "normalized" module to degrees

Now, each value corresponds to a **number of degrees** that we can use to **move along a circle**. The figure below shows a **red arrow** rotated by the corresponding number of degrees for each position and base in the table above:

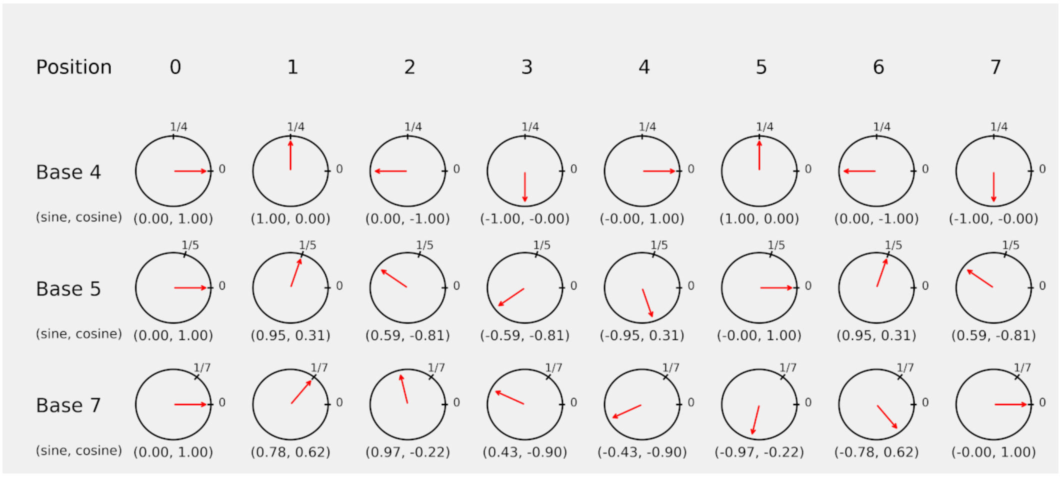
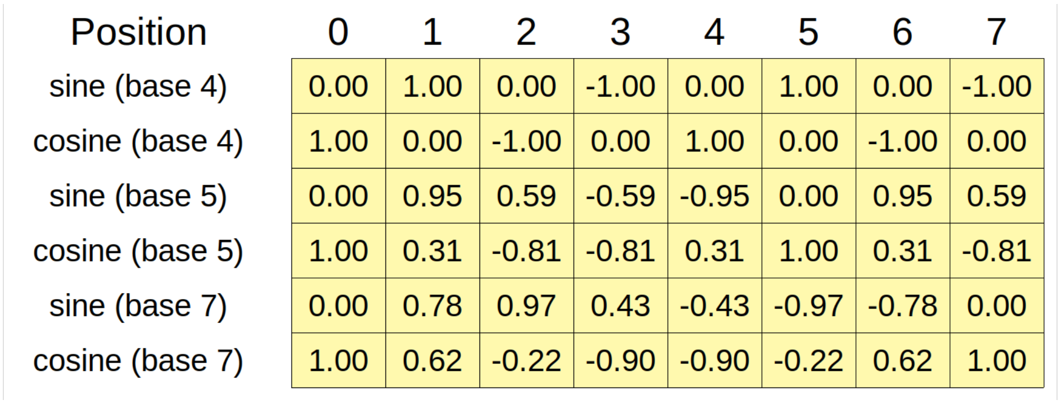

Position 0 1 2 3 4 5 6 7
Base 4
(sine, cosine) (0.00, 1.00) (1.00, 0.00) (0.00, -1.00) (-1.00, -0.00) (-0.00, 1.00) (1.00, 0.00) (0.00, -1.00) (-1.00, -0.00)
Base 5
(sine, cosine) (0.00, 1.00) (0.95, 0.31) (0.59, -0.81) (-0.59, -0.81) (-0.95, 0.31) (-0.00, 1.00) (0.95, 0.31) (0.59, -0.81)
Base 7
(sine, cosine) (0.00, 1.00) (0.78, 0.62) (0.97, -0.22) (0.43, -0.90) (-0.43, -0.90) (-0.97, -0.22) (-0.78, 0.62) (-0.00, 1.00)
1/4
1/5
1/7
0

Figure 9.44 - Representing degrees on a circle

Moreover, the circles above show the **sine** and **cosine** values corresponding to the **coordinates** of the tip of each **red arrow** (assuming a circle with a radius of one).

The **sine** and **cosine** values, that is, the **coordinates** of the **red arrow**, are the **actual positional encoding** of a given position.

We can simply read the sine and cosine values, from top to bottom, to build the encodings for each position:

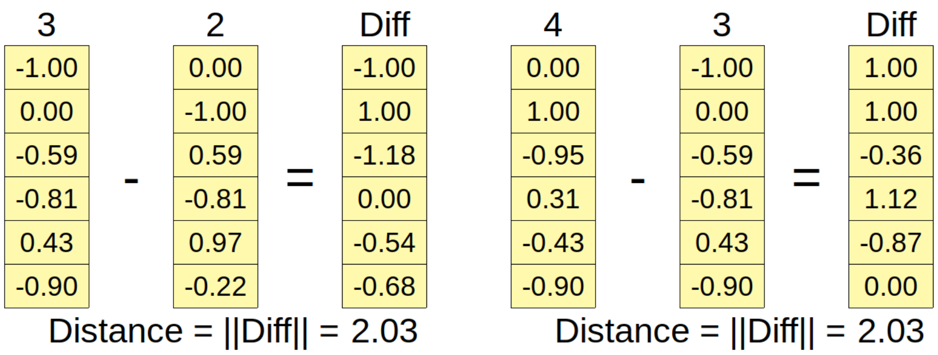


Position	0	1	2	3	4	5	6	7
sine (base 4)	0.00	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00
cosine (base 4)	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00	0.00
sine (base 5)	0.00	0.95	0.59	-0.59	-0.95	0.00	0.95	0.59
cosine (base 5)	1.00	0.31	-0.81	-0.81	0.31	1.00	0.31	-0.81
sine (base 7)	0.00	0.78	0.97	0.43	-0.43	-0.97	-0.78	0.00
cosine (base 7)	1.00	0.62	-0.22	-0.90	-0.90	-0.22	0.62	1.00

Figure 9.45 - Representing degrees using sine and cosine

There were **three vectors**, thus generating **six coordinates or dimensions** (three sines and three cosines).

Next, let’s use *these encodings* to calculate the distances, which should be **consistent** now:



3		2		Diff	4		3		Diff
-1.00		0.00		-1.00	0.00		-1.00		1.00
0.00		-1.00		1.00	1.00		0.00		1.00
-0.59	-	0.59	=	-1.18	-0.95	-	-0.59	=	-0.36
-0.81		-0.81		0.00	0.31		-0.81		1.12
0.43		0.97		-0.54	-0.43		0.43		-0.87
-0.90		-0.22		-0.68	-0.90		-0.90		0.00

Distance = ||Diff|| = 2.03

Distance = ||Diff|| = 2.03

Figure 9.46 - Consistent distances

Awesome, isn’t it? The **encoded distance between any two positions T steps apart is constant** now. In our encoding, the distance between any two positions **one step** apart will always be `2.03`.

"Great! But how do I choose the "bases" for the encoding?"

It turns out, you don’t have to. As the **first vector**, let’s simply move along the circle **as many radians as the index of the position** (one radian is approximately `57.3` degrees). Then, for each **new vector** we add to the encoding, we move along the circle with **exponentially slower** angular speeds. For example, in the **second vector**, we would move only **one-tenth of a radian** (approximately `5.73` degrees) for each new position. In the **third vector** we would move only **one-hundredth of a radian**, and so on and so forth. Figure 9.47 depicts the red arrow moving at increasingly slower angular speeds.

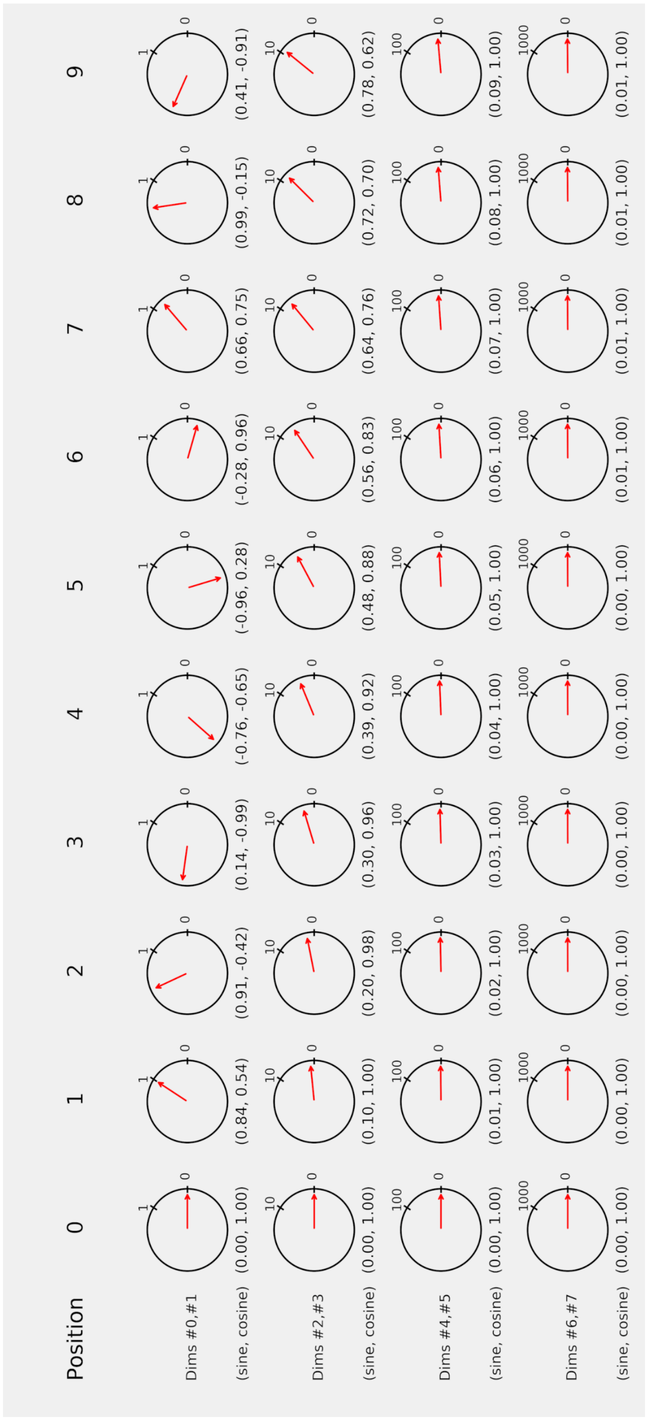


Figure 9.47 - Position encodings represented as circles

A Note on Encoded Distances

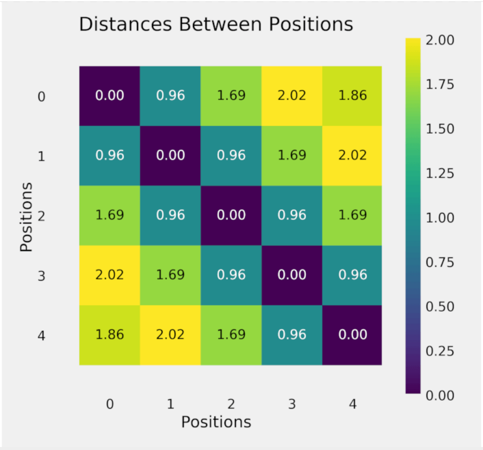
Let's recap what we've already seen:

- the **encoded distance** is defined by the **Euclidean distance** between **two vectors** or, in other words, it is the **norm (size) of the difference between two encoding vectors**
- the encoded distance between **positions zero and two (T=2)** should be exactly the **same** as the encoded distance between **positions one and three, two and four**, and so on

In other words, the **encoded distance** between any two positions **T steps apart remains constant**. Let's illustrate this by **computing the encoded distances** among the first **five positions** (by the way, we're using the encoding with eight dimensions now):

```
distances = np.zeros((5, 5))
for i, v1 in enumerate(encoding[:5]):
    for j, v2 in enumerate(encoding[:5]):
        distances[i, j] = np.linalg.norm(v1 - v2)
```

The resulting matrix looks is full of **pretty diagonals**, each **diagonal** containing a **constant value** for the **encoded distance** corresponding to **positions T steps apart**:



	0	1	2	3	4
0	0.00	0.96	1.69	2.02	1.86
1	0.96	0.00	0.96	1.69	2.02
2	1.69	0.96	0.00	0.96	1.69
3	2.02	1.69	0.96	0.00	0.96
4	1.86	2.02	1.69	0.96	0.00

For example, for **positions next to each other (T=1)**, our **encoded distance** is **always `0.96`**. That's an amazing property of this encoding scheme.

*"Great, but there is something weird... position **four** should have a **larger distance** than position **three** to position zero, right?"*

Not necessarily, no. The distance **does not necessarily need to always increase**. It is OK for the distance between positions zero and four (`1.86`) to be **less than** the distance between positions zero and three (`2.02`), as long as the **diagonals hold**.

For a more detailed discussion about using *sines* and *cosines* for positional encoding, check Amirhossein Kazemnejad's great post on the subject: *"Transformer Architecture: The Positional Encoding"*[142].

Since we're using **four different angular speeds**, the **positional encoding** depicted in Figure 9.47 has **eight dimensions**. Besides, notice that the red arrow barely moves in the last two rows.

In practice, we'll **choose the number of dimensions** first, and then compute the corresponding speeds. For example, for encoding with **eight dimensions** like the one from Figure 9.47, there are **four angular speeds**:

$$\left(\frac{1}{10000^{\frac{0}{8}}}, \frac{1}{10000^{\frac{2}{8}}}, \frac{1}{10000^{\frac{4}{8}}}, \frac{1}{10000^{\frac{6}{8}}}\right) = (1, 0.1, 0.01, 0.001)$$

Equation 9.20 - Angular speeds

The **positional encoding** is given by the two formulas below:

$$PE_{pos,\ 2d} = sin\left(\frac{1}{10000^{\frac{2d}{d_{model}}}}pos\right)$$
$$PE_{pos,\ 2d+1} = cos\left(\frac{1}{10000^{\frac{2d}{d_{model}}}}pos\right)$$

Equation 9.21 - Position encodings

Let's see it in code:

```
max_len = 10
d_model = 8
position = torch.arange(0, max_len).float().unsqueeze(1)
angular_speed = torch.exp(
  torch.arange(0, d_model, 2).float() * (-np.log(10000.0) / d_model)
)
encoding = torch.zeros(max_len, d_model)
encoding[:, 0::2] = torch.sin(angular_speed * position)
encoding[:, 1::2] = torch.cos(angular_speed * position)
```

As you can see, each **position** is multiplied by several different **angular speeds**, and the resulting **coordinates** (given by the sine and cosine) compose the **actual encoding**. Now, instead of plotting the circles, we can directly plot **all sine values** (the even dimensions of the encoding), and **all cosine values** (the odd dimensions of the encoding) instead:

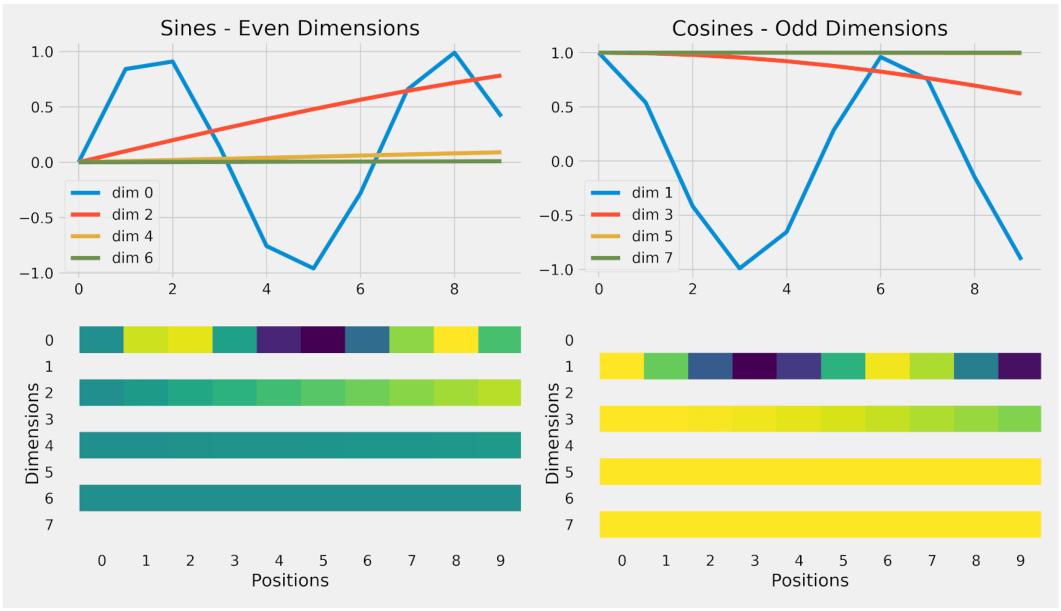


Figure 9.48 - Position encodings as a heatmap

The plots on the bottom show the **color-coded encoding**, ranging from **minus one (dark blue)** to **zero (green)**, all the way to **one (yellow)**. I chose to plot them with the *positions* on the *horizontal axis* so you can more easily associate them with the corresponding curves on the top. In most blog posts, however, you'll find the *transposed version*, that is, with *dimensions* on the *horizontal axis*.

Let's put both sine and cosine values together and look at the **first four positions**:

```
np.round(encoding[0:4], 4)  # first four positions
```

Output

```
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,
0.0000,  1.0000],
        [ 0.8415,  0.5403,  0.0998,  0.9950,  0.0100,  1.0000,
0.0010,  1.0000],
        [ 0.9093, -0.4161,  0.1987,  0.9801,  0.0200,  0.9998,
0.0020,  1.0000],
        [ 0.1411, -0.9900,  0.2955,  0.9553,  0.0300,  0.9996,
0.0030,  1.0000]])
```

Each line above represents the **encoding values** for each one of its **eight dimensions**. The **first position** will **always** have **alternated zeros and ones** (the sine and cosine of zero, respectively).

Let's put it all together into a class:

Positional Encoding (PE)

```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).float().unsqueeze(1)
        angular_speed = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-np.log(10000.0) / d_model)
        )
        # even dimensions
        pe[:, 0::2] = torch.sin(position * angular_speed)
        # odd dimensions
        pe[:, 1::2] = torch.cos(position * angular_speed)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x is N, L, D
        # pe is 1, maxlen, D
        scaled_x = x * np.sqrt(self.d_model)
        encoded = scaled_x + self.pe[:, :x.size(1), :]
        return encoded
```

There is a couple of things about this class I'd like to highlight:

- in the *constructor*, it uses `register_buffer` to define an **attribute** of the module
- in the `forward` method, it is **scaling the input** before adding the positional encoding

The `register_buffer` is used to define an attribute that is **part of the module's state**, yet **not a parameter**. The positional encoding is a good example: its values are computed according to the **dimension** and **length** used by the model, and even though these values *are going to be used* during training, they **shouldn't be updated** by gradient descent.

Another example of a *registered buffer* is the `running_mean` attribute of the **batch normalization** layer. It is used during training, and it is even *modified* during training (unlike positional encoding), but it isn't updated by gradient descent.

Let's create an instance of the positional encoding class and check its `parameters` and `state_dict`:

```
posenc = PositionalEncoding(2, 2)
list(posenc.parameters()), posenc.state_dict()
```

Output

```
([], OrderedDict([('pe', tensor([[[0.0000, 1.0000],
                                  [0.8415, 0.5403]]]))]))
```

The registered buffer can be accessed just like any other attribute:

```
posenc.pe
```

Output

```
tensor([[[0.0000, 1.0000],
         [0.8415, 0.5403]]])
```

Now, let's see what happens if we **add the positional encoding** to a **source sequence**:

```
source_seq # 1, L, D
```

Output

```
tensor([[[-1., -1.],
         [-1.,  1.]]])
```

```
source_seq + posenc.pe
```

Output

```
tensor([[[-1.0000,  0.0000],
         [-0.1585,  1.5403]]])
```

"*What am I looking at?*"

It turns out, the *original coordinates* were somewhat **crowded-out** by the addition of the positional encoding (especially the first row). This may happen if the **data points** have values roughly in the **same range** as the **positional encoding**. Unfortunately, this is **fairly common**: both **standardized inputs** and **word embeddings** (we'll get back to them in Chapter 11) are likely to have **most of their values** inside the `[-1, 1]` range of the positional encoding.

"*How can we handle it then?*"

That's what the **scaling** in the `forward` method is for: it's as if we were "***reversing the standardization***" of the inputs (using a **standard deviation** equals the **square root of their dimensionality**) to retrieve the hypothetical "*raw*" inputs.

$$\text{standardized } x = \frac{\text{"raw" } x}{\sqrt{d_x}} \implies \text{"raw"} x = \sqrt{d_x} \text{ standardized } x$$

Equation 9.22 - "Reversing" the standardization

By the way, we **scaled** the **dot product** using the **inverse** of the **square root of its dimensionality**, which was its **standard deviation**.

Even though this **is not** the same thing, the analogy might help you **remember** that the **inputs** are also **scaled** by the **square root of its number of dimensions** before the positional encoding gets added to them.

In our example, the **dimensionality is two (coordinates)**, so the inputs are going to be **scaled by the square root of two**:

```
posenc(source_seq)
```

Output

```
tensor([[[-1.4142, -0.4142],
         [-0.5727,  1.9545]]])
```

The results above (after the encoding) illustrate the effect of scaling the inputs: it seems to have lessened the *crowding-out* effect of the positional encoding. For inputs with **many dimensions**, the **effect** will be **much more pronounced**: a 300-dimensions embedding will have a scaling factor around 17, for example.

"*Wait, isn't this* ***bad*** *for the model?*"

Left unchecked, yes, it could be bad for the model. That's why we'll pull off **yet another normalization** trick: **layer normalization**. We'll discuss it in detail in the next chapter.

For now, *scaling* the coordinates by the *square root of two* isn't going to be an issue, so we can move on and **integrate positional encoding** into our model.

Encoder + Decoder + PE

The new encoder and decoder classes are just **wrapping** their **self-attention** counterparts by assigning the latter as the `layer` attribute of the former, and **encoding the inputs** prior to calling the corresponding `layer`:

Encoder with Positional Encoding

```python
class EncoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = EncoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

    def forward(self, query, mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, mask)
        return out
```

Decoder with Positional Encoding

```python
class DecoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = DecoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

    def init_keys(self, states):
        self.layer.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, source_mask, target_mask)
        return out
```

*"Why are we calling the **self-attention encoder (and decoder)** a **layer** now? It's a bit confusing..."*

You're right, it may be a bit confusing indeed. Unfortunately, *naming conventions* aren't so great in our field. A **layer** is (loosely) used here as a **building block** of a **larger model**. It may look a bit silly, after all, there is **only one layer** (apart from the encoding). Why even bother making it a "*layer*", right?

In the next chapter, we'll use **multiple layers** (of attention mechanisms) to build the famous **Transformer**.

Model Configuration & Training

Since we haven't changed the big Encoder-Decoder model, we only need to update its arguments (encoder and decoder) to use the new positional encoding-powered classes:

Model Configuration

```python
torch.manual_seed(43)
encpe = EncoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)
decpe = DecoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)

model = EncoderDecoderSelfAttn(encpe, decpe,
                               input_len=2, target_len=2)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```python
sbs_seq_selfattnpe = StepByStep(model, loss, optimizer)
sbs_seq_selfattnpe.set_loaders(train_loader, test_loader)
sbs_seq_selfattnpe.train(100)
```

```python
fig = sbs_seq_selfattnpe.plot_losses()
```

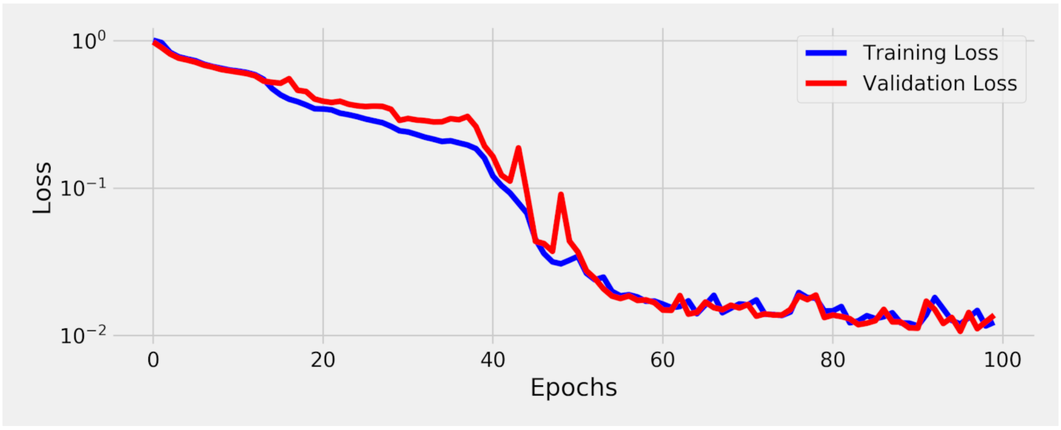


Figure 9.49 - Losses - using positional encoding

Good, the **loss** broke **below 10^{-1}** once again.

Visualizing Predictions

Let’s plot the **predicted coordinates** and connect them using **dashed lines**, while using **solid lines** to connect the **actual coordinates**, just like before:

```
fig = sequence_pred(sbs_seq_selfattnpe, full_test, test_directions)
```

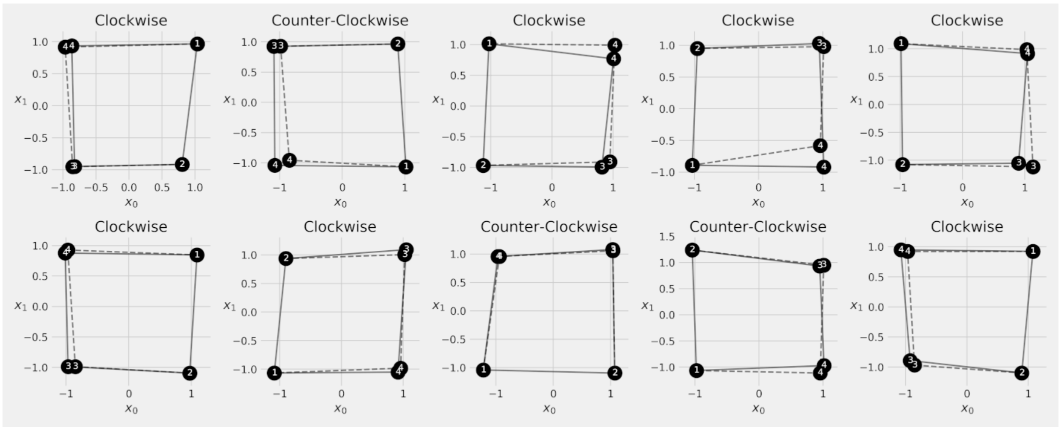


Figure 9.50 - Predicting the last two corners

Awesome, it looks like **positional encoding** is working well indeed - the predicted coordinates are quite close to the actual ones.

Visualizing Attention

Now, let’s check what the model is paying attention to for the first **two sequences** in the training set. Unlike last time, though, there are **three heads** and **three attention mechanisms** to visualize now.

We’re starting with the **three heads** of the **self-attention** mechanism of the **encoder**. There are **two data points** in our **source sequence**, so each attention head has a **two-by-two** matrix of attention scores:

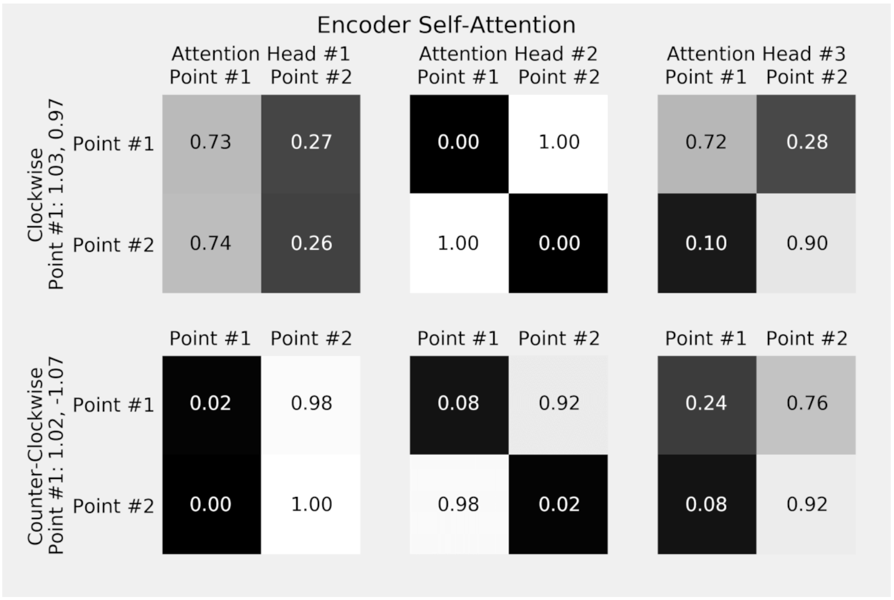


Figure 9.51 - Encoder's self-attention scores for its three heads

It seems like that, in **Attention Heads #1 and #3**, each **data point** is dividing its attention between itself and the other data point. In the **second** attention head, though, the data points barely pay any attention to themselves. Of course, these are just two data points used for visualization: the attention scores are *different* for each source sequence.

Next, we're moving to the **three heads** of the **self-attention** mechanism of the **decoder**. There are **two data points** in our **target sequence** as well, but do not forget that there's a **target mask** to **prevent cheating**:

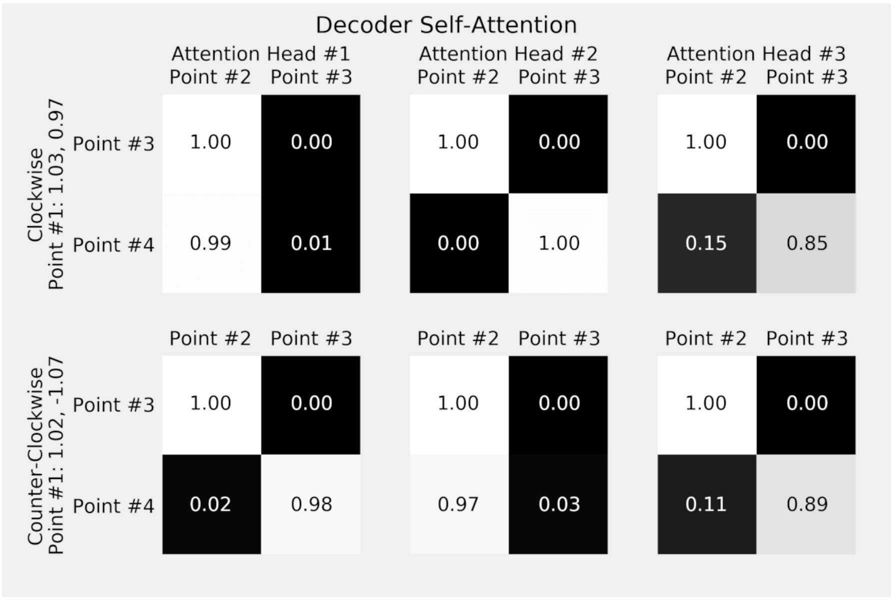


Figure 9.52 - Decoder's self-attention scores for its three heads

The *top-right* value of every matrix is **zero** thanks to the **target mask**: **point #3** (first row) is not allowed to pay attention to its (supposedly unknown, at training time) **own value** (second column), it can pay attention to **point #2 only** (first column).

On the other hand, **point #4** may pay attention to either one of its predecessors. From the matrices above, it seems to pay attention almost exclusively to one of the two points depending on which head and sequence are being considered.

Then, there is the **cross-attention** mechanism, the first one we discussed:

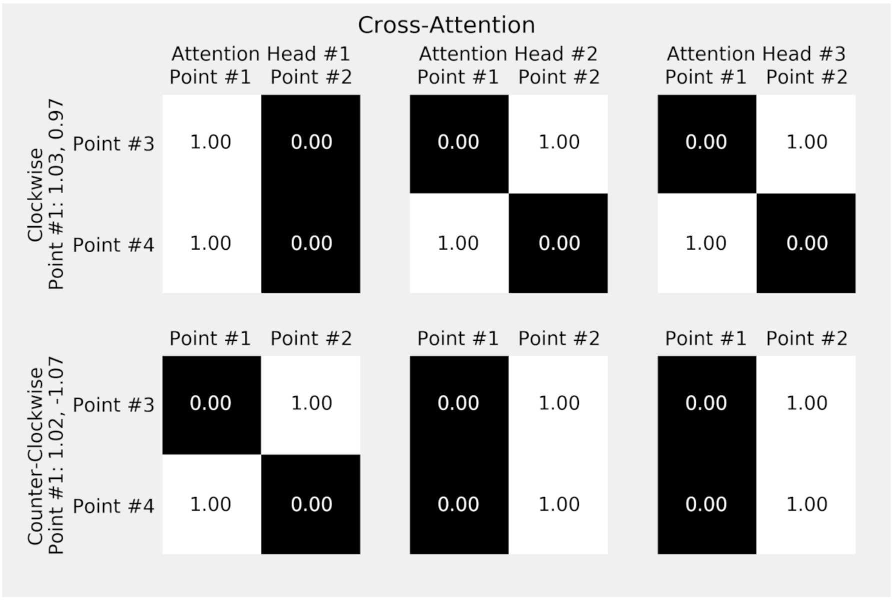


Figure 9.53 - Cross-attention scores for its three heads

There is a lot of variation in the matrices above: in the first sequence and head, for example, **points #3 and #4** pay attention to **point #1 only** while the other two heads pay attention to the preceding point only; in the second sequence, though, it's the other way around.

Putting It All Together

In this chapter, we used the same dataset of **colored squares** but this time we focused on **predicting the coordinates** of the last two corners (**target sequence**) given the coordinates of the first two corners (**source sequence**). In the beginning, we used familiar recurrent neural networks to build an **encoder-decoder architecture**. Then, we progressively built on top of it by using **(cross-)attention**, **self-attention**, and **positional encoding**.

Data Preparation

The training set has the **full sequences** as **features**, while the test set has only the **source sequences** as **features**:

Data Generation & Preparation

```
# Training set
points, directions = generate_sequences()
full_train = torch.as_tensor(points).float()
target_train = full_train[:, 2:]
train_data = TensorDataset(full_train, target_train)
generator = torch.Generator()
train_loader = DataLoader(train_data, batch_size=16,
                          shuffle=True, generator=generator)
# Validation/Test Set
test_points, test_directions = generate_sequences(seed=19)
full_test = torch.as_tensor(points).float()
source_test = full_test[:, :2]
target_test = full_test[:, 2:]
test_data = TensorDataset(source_test, target_test)
test_loader = DataLoader(test_data, batch_size=16)
```

Model Assembly

During this chapter, we used the usual **bottom-up** approach for building ever more complex models. Now, we're revisiting the **current stage** of development in a **top-down** approach, starting from the **encoder-decoder architecture**:

Model Configuration

```
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)

    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequent_mask = (
            1 - torch.triu(torch.ones(attn_shape), diagonal=1)
        ).bool()
        return subsequent_mask

    def encode(self, source_seq, source_mask):
        # Encodes the source sequence and uses the result
        # to initialize the decoder
        encoder_states = self.encoder(source_seq, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Decodes/generates a sequence using the shifted (masked)
        # target sequence - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs

    def predict(self, source_seq, source_mask):
        # Decodes/generates a sequence using one input
        # at a time - used in EVAL mode
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.decode(inputs,
                              source_mask,
                              self.trg_masks[:, :i+1, :i+1])
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = inputs[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        # Sends the mask to the same device as the inputs
        self.trg_masks = self.trg_masks.type_as(X).bool()
        # Slices the input to get source sequence
        source_seq = X[:, :self.input_len, :]
        # Encodes source sequence AND initializes decoder
        self.encode(source_seq, source_mask)
        if self.training:
            # Slices the input to get the shifted target seq
            shifted_target_seq = X[:, self.input_len-1:-1, :]
            # Decodes using the mask to prevent cheating
            outputs = self.decode(shifted_target_seq,
                                  source_mask,
                                  self.trg_masks)
        else:
            # Decodes using its own predictions
            outputs = self.predict(source_seq, source_mask)

        return outputs
```

Encoder + Decoder + Positional Encoding

In the **second level**, we'll find both **encoder** and **decoder** using **positional encodings** to prepare the inputs before calling the **"layer"** that implements the corresponding **self-attention** mechanism.

In the next chapter, we'll *modify this code* to include **multiple "layers" of self-attention**.

Model Configuration

```
class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).float().unsqueeze(1)
        angular_speed = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-np.log(10000.0) / d_model)
        )
        # even dimensions
        pe[:, 0::2] = torch.sin(position * angular_speed)
        # odd dimensions
        pe[:, 1::2] = torch.cos(position * angular_speed)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x is N, L, D
        # pe is 1, maxlen, D
        scaled_x = x * np.sqrt(self.d_model)
        encoded = scaled_x + self.pe[:, :x.size(1), :]
        return encoded

class EncoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = EncoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

    def forward(self, query, mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, mask)
        return out

class DecoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = DecoderSelfAttn(n_heads, d_model,
                                    ff_units, n_features)

    def init_keys(self, states):
        self.layer.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, source_mask, target_mask)
        return out
```

Self-Attention "Layers"

At first, both classes below were full-fledged encoder and decoder. Now, they've been "*downgraded*" to mere **"layers"** of the soon-to-be-larger **encoder** and **decoder** above. The *encoder layer* has a single self-attention mechanism, and the *decoder* layer has both a self-attention and a cross-attention mechanism.

In the next chapter, we'll **add a lot of bells and whistles** to this part. Wait for it!

Model Configuration

```
class EncoderSelfAttn(nn.Module):
    def __init__(self, n_heads, d_model,
                 ff_units, n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = n_features
        self.self_attn_heads = \
            MultiHeadAttention(n_heads,
                               d_model,
                               input_dim=n_features)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, d_model),
        )

    def forward(self, query, mask=None):
        self.self_attn_heads.init_keys(query)
        att = self.self_attn_heads(query, mask)
        out = self.ffn(att)
        return out

class DecoderSelfAttn(nn.Module):
    def __init__(self, n_heads, d_model,
                 ff_units, n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = d_model if n_features is None \
                          else n_features
        self.self_attn_heads = \
            MultiHeadAttention(n_heads,
                               d_model,
                               input_dim=self.n_features)
        self.cross_attn_heads = \
            MultiHeadAttention(n_heads, d_model)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, self.n_features),
        )

    def init_keys(self, states):
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        self.self_attn_heads.init_keys(query)
        att1 = self.self_attn_heads(query, target_mask)
        att2 = self.cross_attn_heads(att1, source_mask)
        out = self.ffn(att2)
        return out
```

Attention Heads

Both self-attention and cross-attention mechanisms are implemented using **wide multi-headed attention**, that is, a straightforward **concatenation** of the results of **several basic attention mechanisms** followed by a **linear projection** to get the **original context vector dimensions** back.

In the next chapter, we'll develop a **narrow multi-headed attention** mechanism.

Model Configuration

```
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, d_model,
                 input_dim=None, proj_values=True):
        super().__init__()
        self.linear_out = nn.Linear(n_heads * d_model, d_model)
        self.attn_heads = nn.ModuleList(
            [Attention(d_model,
                       input_dim=input_dim,
                       proj_values=proj_values)
             for _ in range(n_heads)]
        )

    def init_keys(self, key):
        for attn in self.attn_heads:
            attn.init_keys(key)

    @property
    def alphas(self):
        # Shape: n_heads, N, 1, L (source)
        return torch.stack(
            [attn.alphas for attn in self.attn_heads], dim=0
        )

    def output_function(self, contexts):
        # N, 1, n_heads * D
        concatenated = torch.cat(contexts, axis=-1)
        out = self.linear_out(concatenated) # N, 1, D
        return out

    def forward(self, query, mask=None):
        contexts = [attn(query, mask=mask)
                    for attn in self.attn_heads]
        out = self.output_function(contexts)
        return out

class Attention(nn.Module):
    def __init__(self, hidden_dim,
                 input_dim=None, proj_values=False):
        super().__init__()
        self.d_k = hidden_dim
        self.input_dim = hidden_dim if input_dim is None \
                         else input_dim
        self.proj_values = proj_values
        self.linear_query = nn.Linear(self.input_dim, hidden_dim)
        self.linear_key = nn.Linear(self.input_dim, hidden_dim)
        self.linear_value = nn.Linear(self.input_dim, hidden_dim)
        self.alphas = None

    def init_keys(self, keys):
        self.keys = keys
        self.proj_keys = self.linear_key(self.keys)
        self.values = self.linear_value(self.keys) \
                      if self.proj_values else self.keys

    def score_function(self, query):
        proj_query = self.linear_query(query)
        # scaled dot product
        # N, 1, H x N, H, L -> N, 1, L
        dot_products = torch.bmm(proj_query,
                                 self.proj_keys.permute(0, 2, 1))
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def forward(self, query, mask=None):
        # Query is batch-first N, 1, H
        scores = self.score_function(query) # N, 1, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, 1, L
        self.alphas = alphas.detach()

        # N, 1, L x N, L, H -> N, 1, H
        context = torch.bmm(alphas, self.values)
        return context
```

Model Configuration & Training

Model Configuration

```
torch.manual_seed(43)
encpe = EncoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)
decpe = DecoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)
model = EncoderDecoderSelfAttn(encpe, decpe,
                               input_len=2, target_len=2)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model Training

```
sbs_seq_selfattnpe = StepByStep(model, loss, optimizer)
sbs_seq_selfattnpe.set_loaders(train_loader, test_loader)
sbs_seq_selfattnpe.train(100)
sbs_seq_selfattnpe.losses[-1], sbs_seq_selfattnpe.val_losses[-1]
```

Output

```
(0.01233051170129329, 0.013784115784801543)
```

Recap

In this chapter, we've introduced **sequence-to-sequence** problems and the **encoder-decoder architecture**. At first, we used **recurrent neural networks** to **encode a source sequence** so that its representation (hidden state) could be used to **generate the target sequence**. Then, we improved the architecture by using a **(cross-)attention mechanism** that allowed the **decoder** to use the **full sequence of hidden states** produced by the **encoder**. Next, we replaced the recurrent neural networks with **self-attention mechanisms** which, although more efficient, cause the **loss of information** regarding the **order of the inputs**. Finally, the addition of **positional encoding** allowed to account for the order of the inputs once again. This is what we've covered:

- generating a synthetic **dataset** of **source** and **target sequences**
- understanding the purpose of the **encoder-decoder architecture**
- using the **encoder** to generate a **representation of the source sequence**
- using **encoder's final hidden state** as **decoder's initial hidden state**
- using the **decoder** to generate the **target sequence**
- using **teacher forcing** to help the decoder during **training**
- combining both encoder and decoder into a single **encoder-decoder model**
- understanding the **limitations** of using a **single hidden state** to encode the source sequence
- defining the **sequence of (transformed) hidden states** from the **encoder** as **"values" (V)**

- defining the **sequence of (transformed) hidden states** from the **encoder** as **"keys" (K)**
- defining **(transformed) hidden states** produced by the **decoder** as **"queries" (Q)**
- computing **similarities (alignment scores)** between a given **"query"** and **all the "keys"** using **scaled dot-product**
- visualizing the **geometric** interpretation of the **dot-product**
- **scaling the dot-product** according to the **number of dimensions** to keep its **variance constant**
- using **softmax** to transform similarities into **attention scores (alphas)**
- computing a **context vector** as an **average of "values" (V)** weighted by the corresponding **attention scores**
- **concatenating** the **context vector** to the **decoder's hidden state** and running it through a linear layer to get predictions
- building a class for the **attention mechanism**
- ignoring **padded data points** in the source sequence using a **mask**
- visualizing the attention scores
- combining **multiple attention heads** into a **multi-headed (wide) attention mechanism**
- learning the difference between **wide and narrow attention**
- using `ModuleList` to add a **list of layers** as a model attribute
- replacing the recurrent layers with **(self-)attention mechanisms**, after all, **attention is all you need**
- understanding that, in **self-attention** mechanisms, **each data point** will be used to generate a **"value" (V)**, a **"key" (K)**, and a **"query" (Q)**, but they will still have distinct values because of the **different affine transformations**
- building an **encoder** using a **self-attention** mechanism and a simple **feed-forward network**

- realizing that **self-attention scores** are a **square matrix** since **every "hidden state" is a weighted average of all elements** in the input sequence
- reusing the attention mechanism as a **cross-attention mechanism**, such that the decoder still has access to the full sequence from the encoder
- understanding that **self-attention mechanisms leak future data**, thus allowing the **decoder to cheat**
- using a **target mask** to prevent the decoder from paying attention to "future" elements of the sequence
- building a **decoder** using a **(masked) self-attention** mechanism, a **cross-attention** mechanism, and a simple **feed-forward network**
- understanding that **self-attention** mechanisms **cannot** account for the **sequential order** of the data
- figuring that attention is not enough and that we also need **positional encoding** to incorporate sequential order back into the model
- using alternated **sines and cosines** of different frequencies as **positional encoding**
- learning that combining sines and cosines yield interesting properties, such as keeping **constant** the **encoded distance between any two positions** T steps apart
- using `register_buffer` to add an attribute that should be part of the **module's state without being a parameter**
- visualizing self- and cross-attention scores

Congratulations! That was definitely an *intense* chapter. The **attention mechanism** in its different forms - single head, multi-headed, **self-attention**, and cross-attention - is very flexible and built on top of fairly simple concepts, but the whole thing is definitely ***not* that easy** to grasp. Maybe you feel a bit *overwhelmed* by the huge amount of information and details involved in it, but don't worry. I guess everyone does feel like that at first, I know I did. It gets better with time!

The good thing is, you have already learned most of the techniques that make up for the famous **Transformer** architecture: attention mechanisms, masks, and positional encoding. There are still a few things left to learn about it, like **layer normalization**, and we'll cover them all in the next chapter.

Transform and roll out!

[139] https://arxiv.org/abs/1706.03762

[140] https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/

[141] https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a

[142] https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Chapter 10
Transform and Roll Out

Spoilers

In this chapter, we will:

- modify the **multi-headed attention** mechanism to use **narrow attention**
- use **layer normalization** to standardize individual data points
- **stack "layers"** together to build **Transformer encoders and decoders**
- add **layer normalization**, **dropout**, and **residual connections** to each "sub-layer" operation
- learn the difference between **norm-last** and **norm-first** "sub-layers"
- train a **Transformer** to predict a target sequence from a source sequence
- build and train a **Vision Transformer** to perform image classification

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 10[143] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[144].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 10's

notebook. If not, just click on `Chapter10.ipynb` in your Jupyter's Home Page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import copy
import numpy as np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, random_split, \
    TensorDataset
from torchvision.transforms import Compose, Normalize, Pad

from data_generation.square_sequences import generate_sequences
from data_generation.image_classification import generate_dataset
from helpers import index_splitter, make_balanced_sampler
from stepbystep.v4 import StepByStep
# These are the classes we built in Chapter 9
from seq2seq import PositionalEncoding, subsequent_mask, \
    EncoderDecoderSelfAttn
```

Transform and Roll Out

We're actually **quite close** to developing our own version of the famous **Transformer** model. The **encoder-decoder architecture with positional encoding** is missing only a few details to effectively "*transform and roll out*" :-)

"*What's missing?*"

First, we need to revisit the **multi-headed attention** mechanism to make it less computationally expensive by using **narrow attention**. Then, we'll learn about a new kind of normalization: **layer normalization**. Finally, we'll add some more bells and whistles: **dropout**, **residual connections**, and **more "layers"** (like the encoder and decoder "layers" from the last chapter).

Narrow Attention

In the last chapter, we've used *full* attention heads to build a **multi-headed attention** and we called it **wide attention**. Although this mechanism works well, it gets prohibitively expensive as the number of dimensions grows. That's when the **narrow attention** comes in: each **attention head** will get a **chunk** of the **transformed data points (projections)** to work with.

Chunking

This is a detail of **utmost importance**: the attention heads **do not** use **chunks of the original data points**, but of their projections.

"*Why?*"

To understand why, let's take an example of an **affine transformation**, one that generates **"values"** (v_0) from the first data point (x_0):

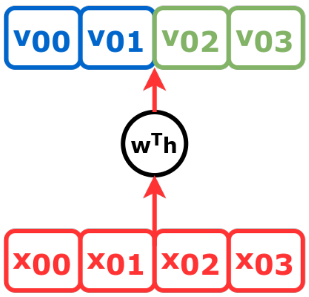


Figure 10.1 - Narrow attention

The transformation above takes a single **data point** of **four dimensions** (features) and turns it into a **"value"** (also with four dimensions) that's going to be used in the **attention mechanism**.

At first sight, it *may* look like we'll get the *same* result, either if we *split the inputs into chunks* or if we *split the projections into chunks*. But that's definitely **not** the case. So, let's **zoom in** and look at the **individual weights** inside that transformation:

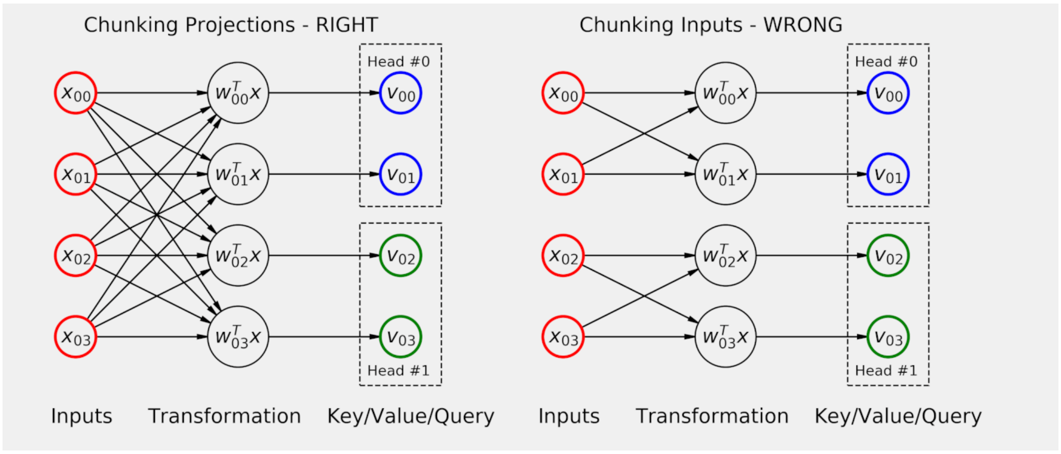


Figure 10.2 - Chunking: the wrong and the right way

On the left, the **correct approach**: it computes the **projections first** and **chunks them later**. It is clear that **each value in the projection** (from $\mathbf{v}_{00}$ to $\mathbf{v}_{03}$) is a **linear combination** of **all features** in the data point.

Since **each head** is working with a **subset of the projected dimensions**, these projected dimensions **may** end up representing a **different aspect of the underlying data**. For Natural Language Processing tasks, for example, some attention heads may correspond to linguistic notions of syntax and coherence. A particular head may attend to the **direct objects of verbs** while another head may attend to **objects of prepositions**, and so on[145].

Now, compare it to the **wrong approach**, on the right: by **chunking it first**, each value in the projection is a linear combination of **a subset of the features only**.

"*Why is it so bad?*"

First, it is a *simpler* model (the wrong approach has only *eight weights* while the correct one has *sixteen*), so its learning capacity is limited. Second, since each head can only look at a **subset of the features**, they simply **cannot learn** about **long-range dependencies** in the inputs.

Now, let's use a **source sequence** of **length two** as input, each data point having **four features** like the chunking example above, to illustrate our new **self-attention** mechanism:

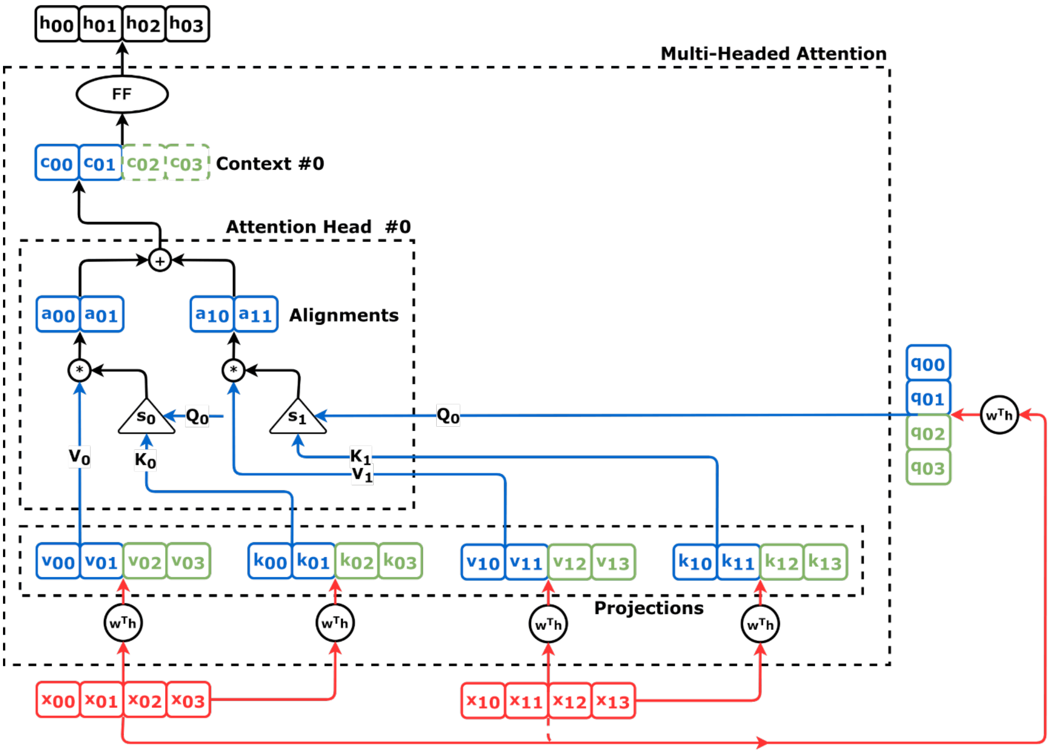


Figure 10.3 - Self-(narrow)attention mechanism

The flow of information goes like this:

- both **data points** (x_0 and x_1) go through **distinct affine transformations** to generate the corresponding **"values"** (v_0 and v_1) and **"keys"** (k_0 and k_1), which we'll be calling **projections**

- both data points also go through **another affine transformation** to generate the corresponding **"queries"** (q_0 and q_1), but we'll be focusing on the **first query** (q_0) only now
- each **projection** has the same number of **dimensions** as the **inputs** (four)
- instead of simply using the projections, as former attention heads did, this attention head uses **only a chunk of the projections** to compute the **context vector**
- since projections have four dimensions, let's split them into **two chunks** - blue (left) and green (right) - of **two dimensions each**
- the **first attention head** uses **only blue chunks** to compute its **context vector** which, like the projections, has **only two dimensions**
- the **second attention head** (not depicted in the figure above) uses the **green chunks** to compute the **other half** of the **context vector** which, in the end, has the desired dimension
- like the former multi-headed attention mechanism, the **context vector** goes through a **feed-forward network** to generate the **"hidden states"** (only the first one is depicted in the figure above)

It *looks* complicated, I know, but it really isn't *that* bad. Maybe it helps to see it in code.

Multi-Headed Attention

The new multi-headed attention class is more than a combination of both `Attention` and `MultiHeadAttention` classes from the previous chapter: it implements the *chunking* of the projections and it introduces **dropout for attention scores**.

Multi-Headed Attention

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = int(d_model / n_heads)                           ①
        self.linear_query = nn.Linear(d_model, d_model)
        self.linear_key = nn.Linear(d_model, d_model)
        self.linear_value = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(p=dropout)                        ④
        self.alphas = None

    def make_chunks(self, x):                                       ①
        batch_size, seq_len = x.size(0), x.size(1)
        # N, L, D -> N, L, n_heads * d_k
        x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
        # N, n_heads, L, d_k
        x = x.transpose(1, 2)
        return x

    def init_keys(self, key):
        # N, n_heads, L, d_k
        self.proj_key = self.make_chunks(self.linear_key(key)) ①
        self.proj_value = \
                self.make_chunks(self.linear_value(key))       ①

    def score_function(self, query):
        # Scaled dot product
        proj_query = self.make_chunks(self.linear_query(query))①
        # N, n_heads, L, d_k x N, n_heads, d_k, L ->
        # N, n_heads, L, L
        dot_products = torch.matmul(                                ②
            proj_query, self.proj_key.transpose(-2, -1)
        )
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def attn(self, query, mask=None):                                 ③
        # Query is batch-first: N, L, D
        # Score function will generate scores for each head
        scores = self.score_function(query) # N, n_heads, L, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, n_heads, L, L

        alphas = self.dropout(alphas)                                 ④
        self.alphas = alphas.detach()

        # N, n_heads, L, L x N, n_heads, L, d_k ->
        # N, n_heads, L, d_k
        context = torch.matmul(alphas, self.proj_value)               ②
        return context

    def output_function(self, contexts):
        # N, L, D
        out = self.linear_out(contexts) # N, L, D
        return out

    def forward(self, query, mask=None):
        if mask is not None:
            # N, 1, L, L - every head uses the same mask
            mask = mask.unsqueeze(1)

        # N, n_heads, L, d_k
        context = self.attn(query, mask=mask)
        # N, L, n_heads, d_k
        context = context.transpose(1, 2).contiguous()                ⑤
        # N, L, n_heads * d_k = N, L, d_model
        context = context.view(query.size(0), -1, self.d_model)⑤
        # N, L, d_model
        out = self.output_function(context)
        return out
```

① Chunking the projections

② Using `torch.matmul` instead of `torch.bmm`

③ Former `forward` method of `Attention` class

④ Dropout for the attention scores

⑤ "Concatenating" the context vectors

Let’s go over its methods:

- `make_chunks`: it takes a tensor of shape (N, L, D) and splits its last dimension in two, resulting in a **(N, L, n_heads, d_k)** shape where **d_k** is the size of the chunk (**d_k = D / n_heads**)
- `init_keys`: it makes projections for "keys" and "values", and *chunks* them
- `score_function`: it *chunks* the projected "queries" and computes the **scaled dot product** (it uses `torch.matmul` as a replacement for `torch.bmm` because there is **one extra dimension** due to *chunking*, see the aside below for more details)
- `attn`: corresponds to the `forward` method of the former `Attention` class, and it computes the **attention scores** (*alphas*) and the **chunks of the context vector**
  - it uses **dropout** on the attention scores for regularization: dropping an attention score (zeroing it) means that the corresponding **element** in the sequence will be **ignored**
- `output_function`: it simply runs the contexts through the feed-forward network since the concatenation of the contexts is going to happen in the `forward` method now
- `forward`: it calls the `attn` method and **reorganizes the dimensions** of the result to **"concatenate" the chunks** of the **context vector**
  - if a `mask` is provided - (N, 1, L) shape for the source mask (in the encoder) or (N, L, L) shape for the target mask (in the decoder) - it *unsqueezes* a new dimension after the first one to accommodate the **multiple heads** since every head should use the **same mask**

`torch.bmm` vs `torch.matmul`

In the last chapter, we used `torch.bmm` to perform **batch matrix multiplication**. It was the right tool for the task at hand since we had **two three-dimensional tensors** (for example, computing the context vector using *alphas* and "values"):

$$(N, 1, L) \times (N, L, H) = (N, 1, H)$$

Equation 10.1 - Batch matrix multiplication using `torch.bmm`

Unfortunately, `torch.bmm` **cannot** handle tensors with more dimensions than that. Since we have a **four-dimensional tensor** after **chunking**, we need something more powerful: `torch.matmul`. It is a **more generic** operation that, depending on its inputs, behaves like `torch.dot`, `torch.mm`, or `torch.bmm`.

If we're using `torch.matmul` to multiply *alphas* and "values" once again, while using multiple heads and chunking, it looks like this:

$$(N, n\_heads, L, L) \times (N, n\_heads, L, d_k) = (N, n\_heads, L, d_k)$$

Equation 10.2 - Batch matrix multiplication using `torch.matmul`

It is quite similar to batch matrix multiplication but you're free to have as many extra dimensions as you want: it still looks at the **last two dimensions** only.

We can generate some *dummy points* corresponding to a mini-batch of **16 sequences (N)**, each sequence having **two data points (L)**, each data point having **four features (F)**:

```
dummy_points = torch.randn(16, 2, 4) # N, L, F
mha = MultiHeadedAttention(n_heads=2, d_model=4, dropout=0.0)
mha.init_keys(dummy_points)
out = mha(dummy_points) # N, L, D
out.shape
```

Output

```
torch.Size([16, 2, 4])
```

Since we’re using the data points both as "keys", "values", and as "queries", this is a **self-attention** mechanism.

The figure below depicts a multi-headed attention mechanism with its **two heads**, **blue (left)** and **green (right)**, and the **first data point** being used as **"query"** to generate the first "hidden state" ($\mathbf{h}_0$):

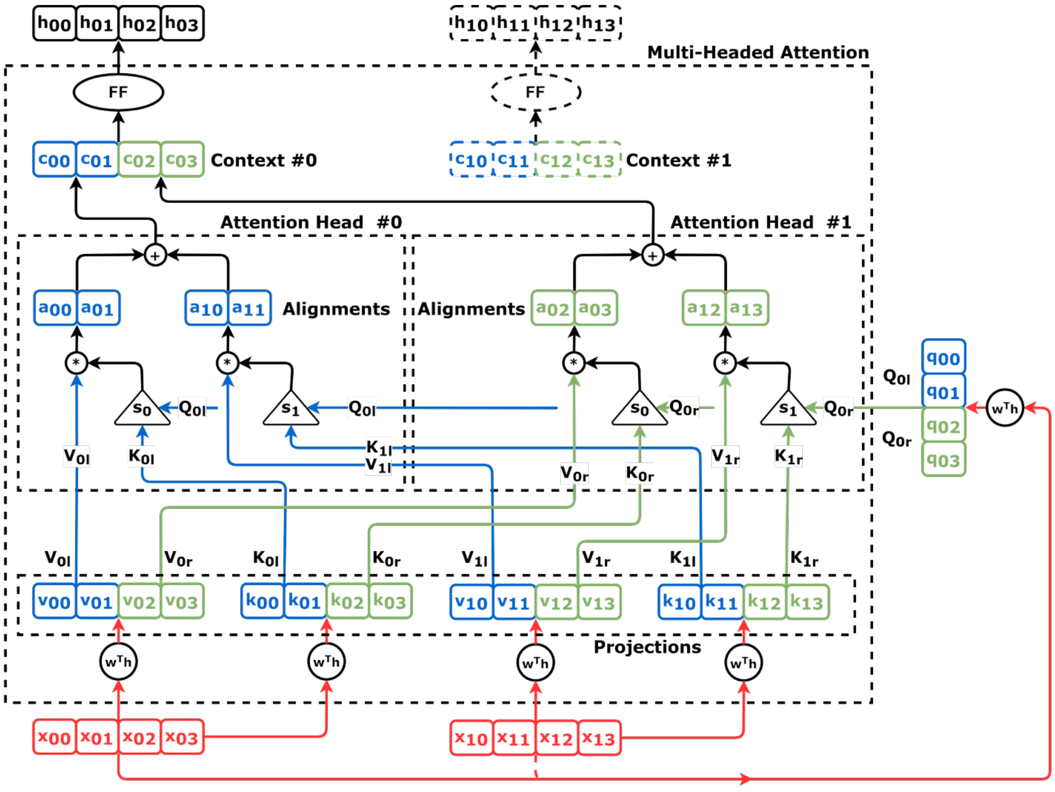


Figure 10.4 - Self-(narrow)attention mechanism (both heads)

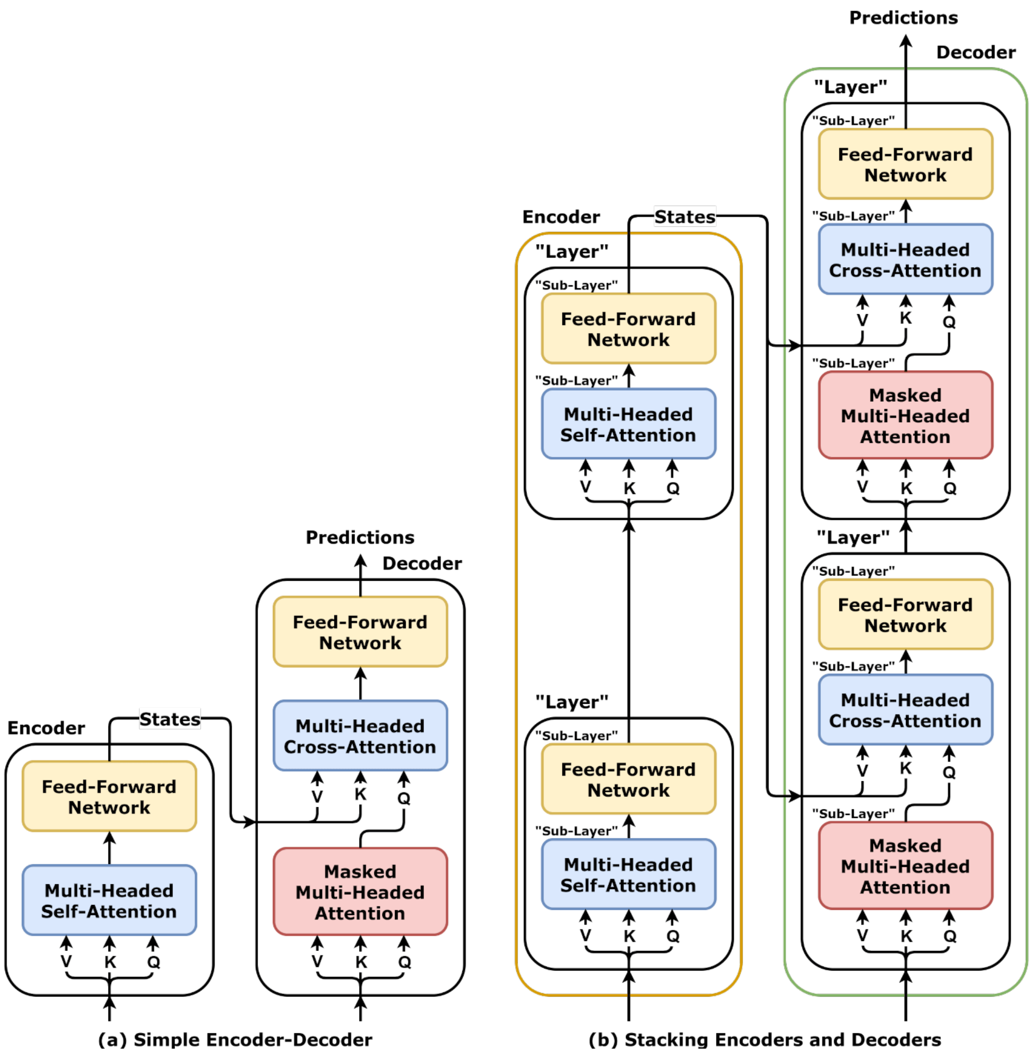
To help you out (especially if you're seeing it in black and white), I've **labeled the arrows** with their corresponding **role (V, K, or Q)** followed by a **subscript** indicating both the **index of the data point** being used **(zero or one)** and **which head** is using it **(left or right)**.

If you find the figure above too confusing, don't sweat about it, I've included it for the sake of *completion* since Figure 10.3 depicted only the first head. The important thing to remember here is: "**multi-headed attention chunks the projections, not the inputs**".

Let's move on to the next topic…

Stacking Encoders and Decoders

Let's make our encoder-decoder architecture **deeper** by **stacking two encoders** on top of one another, and then do the same with **two decoders**. It looks like this:



Encoder
States
Feed-Forward Network
Multi-Headed Self-Attention
V K Q
Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
Masked Multi-Headed Attention
(a) Simple Encoder-Decoder
"Layer"
"Sub-Layer"
(b) Stacking Encoders and Decoders

Figure 10.5 - Stacking Encoders and Decoders

The output of one encoder feeds the next, and the last encoder outputs **states** as usual. These states will feed the **cross-attention** mechanism of **all stacked**

decoders. The output of one decoder feeds the next, and the last decoder outputs **predictions** as usual.

The **former encoder** is now a so-called **"layer"**, and a **stack of "layers"** compose the **new, deeper, encoder**. The same holds true for the **decoder**. Moreover, **each operation** (multi-headed self- and cross-attention mechanisms and feed-forward networks) inside a "layer" is now a **"sub-layer"**.

The figure above represents an encoder-decoder architecture with **two "layers"** each. But we're *not* stopping there: we're stacking **six "layers"**! It would be *somewhat* hard to draw a diagram for it, so we're simplifying it a bit:

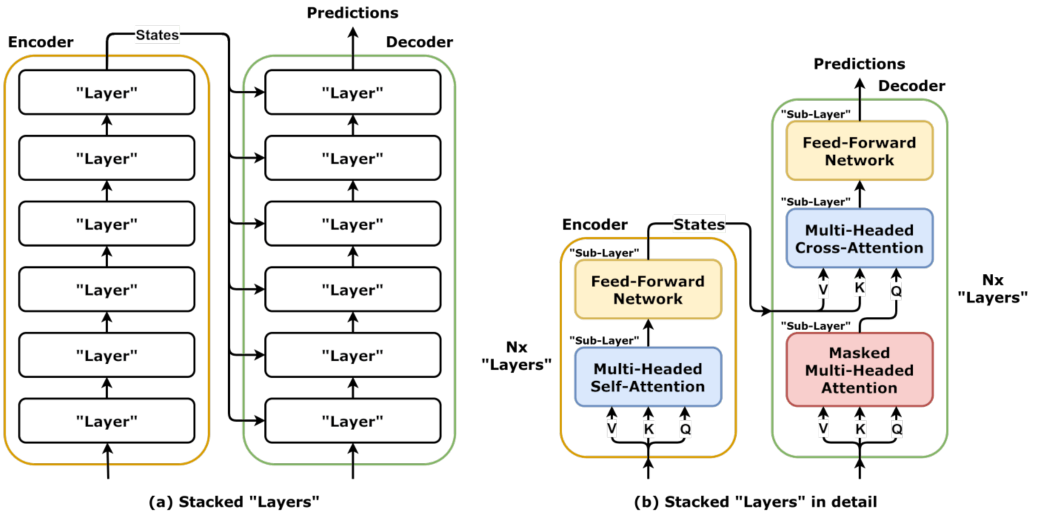


Figure 10.6 - Stacked "Layers"

On the one hand, we could simply draw both *stacks of "layers"* and abstract away their inner operations. That's the diagram **(a)** in the figure above. On the other hand, since **all "layers" are identical**, we can **keep representing the inner operations** and just *hint* at the **stack** by adding "*Nx "Layers"*" next to it. That's the diagram **(b)** in the figure above, and it will be our representation of choice from now on.

By the way, that's exactly how a **Transformer** is built!

"*Cool! Is this a Transformer already then?*"

Not yet, no. We need to work further on the **"sub-layers"** to **transform** (ahem!) the architecture above into a real **Transformer**.

Wrapping "Sub-Layers"

As our models grows **deeper** with many **stacked "layers"**, we're going to run into familiar issues, like the vanishing gradients problem. In computer vision models, this issue was successfully addressed by the addition of other components, like **batch normalization** and **residual connections**, for example. Moreover, we know that…

"*With great depth comes great complexity.*"

Peter Parker

…and along with that, overfitting.

But we *also* know that **dropout** works pretty well as a **regularizer**, so we can throw it in the mix as well.

"*How are we adding normalization, residual connections, and dropout to our model?*"

We'll **wrap each and every "sub-layer"** with them! Cool, right? But that brings up *another* question: **how** to wrap them? It turns out, we can choose to wrap a "sub-layer" in one out of two ways, **norm-last** or **norm-first**:

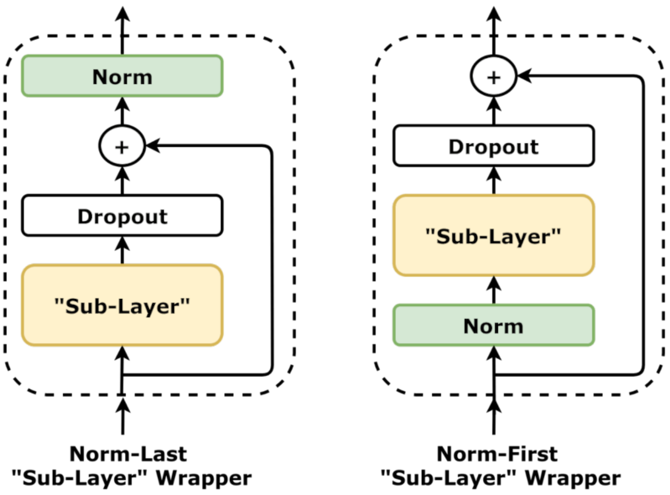


Figure 10.7 - "Sub-Layers" - Norm-last vs Norm-first

The **norm-last** wrapper follows the *"Attention Is All you Need"*[146] paper to the letter:

> "*We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is* `LayerNorm(x+Sublayer(x))`, *where* `Sublayer(x)` *is the function implemented by the sub-layer itself.*"

The **norm-first** wrapper follows the "sub-layer" implementation described in *"The Annotated Transformer"*[147], which **explicitly** places **norm first** as opposed to last for the sake of code simplicity.

Let's turn the diagrams above into equations:

$$\begin{aligned} outputs_{norm-last} &= norm(inputs + dropout(sublayer(inputs))) \\ outputs_{norm-first} &= inputs + dropout(sublayer(norm(inputs))) \end{aligned}$$

Equation 10.3 - Outputs - norm-first vs norm-last

The equations are **almost** the same, except for the fact that the **norm-last** wrapper (from "*Attention Is All You Need*") **normalizes the outputs** and the **norm-first** wrapper (from "*The Annotated Transformer*") **normalizes the inputs**. That's a **small, yet important, difference**.

"*Why?*"

If you're using **positional encoding**, you *want* to **normalize your inputs**, so **norm-first** is more convenient.

"*What about the outputs?*"

We'll **normalize the *final* outputs**, that is, the output of **the last "layer"** (which is the output of its last, not normalized, "sub-layer"). Any intermediate output is simply the input of the subsequent "sub-layer", and each "sub-layer" normalizes its own inputs.

There is *another* important difference that will be discussed in the next section.

From now on, we're **sticking with norm-first**, thus **normalizing the inputs**:

$$outputs_{norm-first} = inputs + dropout(sublayer(norm(inputs)))$$

Equation 10.4 - Outputs - norm-first

By **wrapping each and every "sub-layer"** inside both **encoder "layers"** and **decoder "layers"**, we'll arrive at the desired **Transformer architecture**.

Let's start with the...

Transformer Encoder

We'll be representing the encoder using "stacked" layers in detail (like Figure 10.6 (*b*)), that is, showing the internal **wrapped "sub-layers"** (the dashed rectangles):

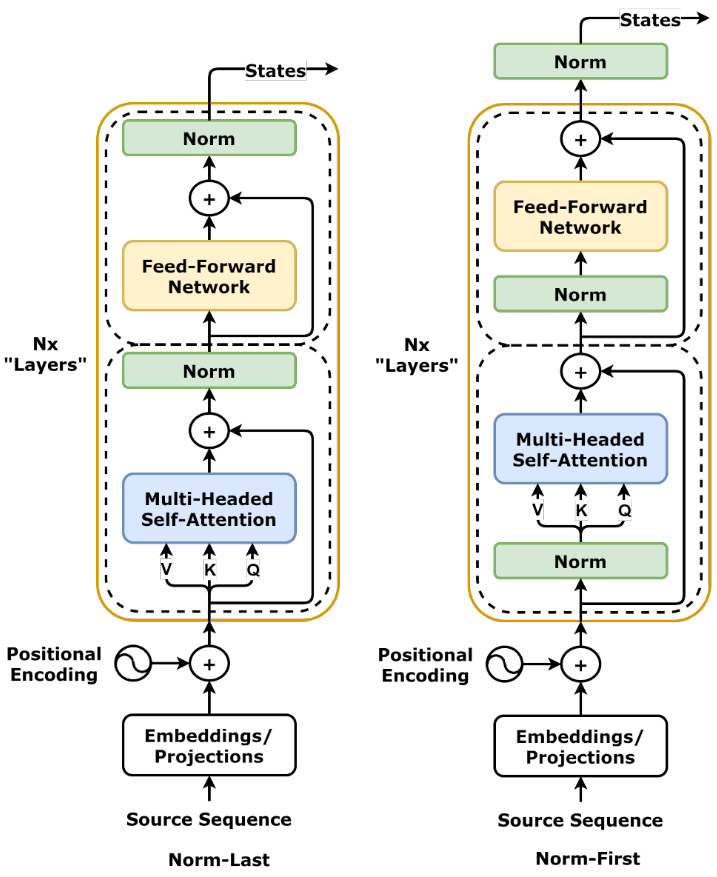


Figure 10.8 - Transformer Encoder - norm-last vs norm-first

On the left, the encoder uses a *norm-last wrapper*, and its output (the encoder's **states**) is given by:

$$outputs_{norm-last} = norm(\underbrace{norm(inputs + att(inputs))}_{Output\ \ of\ \ SubLayer_0} + ffn(\underbrace{norm(inputs + att(inputs))}_{Output\ \ of\ \ SubLayer_0}))$$

Equation 10.5 - Encoder's output: norm-last

On the right, the encoder uses a *norm-first wrapper*, and its output (the encoder's **states**) is given by:

$$outputs_{norm-first} = \underbrace{inputs + att(norm(inputs))}_{Output\ \ of\ \ SubLayer_0} + ffn(norm(\underbrace{inputs + att(norm(inputs))}_{Output\ \ of\ \ SubLayer_0}))$$

Equation 10.6 - Encoder's output: norm-first

The **norm-first wrapper** allows the **inputs to flow unimpeded** (the inputs aren't normalized) all the way to the top while adding the results of each "sub-layer" along the way (the *last* normalization of *norm-first* happens outside of the "sub-layers" so it's not included in the equation).

"Which one is best?"

There is no straight answer to this question. It actually reminds me of the discussions about placing the batch normalization layer *before* or *after* the activation function. Now, once again, there is no "right" and "wrong", and the order of the different components is not etched in stone.

Let's see it in code, starting with the "layer", and all its wrapped "sub-layers":

Encoder "Layer"

```python
class EncoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.self_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ff_units, d_model),
        )

        self.norm1 = nn.LayerNorm(d_model)  ①
        self.norm2 = nn.LayerNorm(d_model)  ①
        self.drop1 = nn.Dropout(dropout)
        self.drop2 = nn.Dropout(dropout)

    def forward(self, query, mask=None):
        # Sublayer #0
        # Norm
        norm_query = self.norm1(query)
        # Multi-headed Attention
        self.self_attn_heads.init_keys(norm_query)
        states = self.self_attn_heads(norm_query, mask)
        # Add
        att = query + self.drop1(states)

        # Sublayer #1
        # Norm
        norm_att = self.norm2(att)
        # Feed Forward
        out = self.ffn(norm_att)
        # Add
        out = att + self.drop2(out)
        return out
```

① What is that?

Its constructor takes **four arguments**:

- `n_heads`: the number of **attention heads** in the **self-attention mechanism**
- `d_model`: the **number of features** of the inputs (remember, this number will be *split* among the attention heads, so it must be a multiple of the number of heads)
- `ff_units`: the number of **units** in the **hidden layer** of the **feed-forward network**
- `dropout`: the **probability** of dropping out inputs

The `forward` method takes a "query" and a source mask (to ignore padded data points) as usual.

"*What is that* `LayerNorm`*?*"

It is one *teeny-tiny detail* I haven't mentioned before… Transformers do not use *batch normalization*, but **layer normalization**.

"*What's the difference?*"

Short answer: batch normalization *normalizes features*, while **layer normalization normalizes data points**. Long answer: there is a *whole section* on it, we'll get back to it soon enough.

In PyTorch, the encoder "layer" is implemented as `nn.TransformerEncoderLayer`, and its constructor method expects the same arguments (`d_model`, `nhead`, `dim_feedforward`, and `dropout`) and an optional `activation` function for the feed-forward network.

Its `forward` method, though, has **three arguments**:

- `src`: the **source sequence**, that's the `query` argument in our class

  IMPORTANT: PyTorch's **transformer layers** use **sequence-first** shapes for their inputs (L, N, F), and there is **no batch-first option**.

- `src_key_padding_mask`: the mask for **padded data points**, that's the `mask` argument in our class
- `src_mask`: this mask is used to **purposefully hide some of the inputs** in the source sequence - we're not doing that, so our class doesn't have a corresponding argument - which can be used for training **language models** (more on that in Chapter 11)

Now we can stack a bunch of "layers" like that to build an **actual encoder**:

Transformer Encoder

```
class EncoderTransf(nn.Module):
    def __init__(self, encoder_layer, n_layers=1, max_len=100):
        super().__init__()
        self.d_model = encoder_layer.d_model
        self.pe = PositionalEncoding(max_len, self.d_model)
        self.norm = nn.LayerNorm(self.d_model)
        self.layers = nn.ModuleList([copy.deepcopy(encoder_layer)
                                     for _ in range(n_layers)])

    def forward(self, query, mask=None):
        # Positional Encoding
        x = self.pe(query)
        for layer in self.layers:
            x = layer(x, mask)
        # Norm
        return self.norm(x)
```

Its constructor takes an **instance of an `EncoderLayer`**, the **number of "layers"** we'd like to stack on top of one another, and a **max length** of the source sequence that's going to be used for the positional encoding.

We're using `deepcopy` to make sure we create *real copies* of the encoder layer, and `nn.ModuleList` to make sure PyTorch can find the "layers" inside the list. Our *default* for the number of "layers" is only one, but the **original Transformer uses six "layers"**.

The `forward` method is quite straightforward (I was actually missing making puns...): it adds positional encoding to the "query", loops over the "layers", and normalizes the outputs in the end. The final outputs are, as usual, the **states of the encoder** that will feed the **cross-attention mechanism** of **every "layer"** of the **decoder**.

In PyTorch, the encoder is implemented as `nn.TransformerEncoder`, and its constructor method expects similar arguments: `encoder_layer`, `num_layers`, and an optional normalization layer to normalize (or not) the outputs.

```
enclayer = nn.TransformerEncoderLayer(
    d_model=6, nhead=3, dim_feedforward=20
)
enctransf = nn.TransformerEncoder(
    enclayer, num_layers=1, norm=nn.LayerNorm
)
```

Therefore, it behaves a bit *differently* than ours, since it **does not** (at the time of writing) implement **positional encoding** for the inputs, and it does not normalize the outputs by default.

Transformer Decoder

We'll be representing the decoder using "stacked" layers in detail (like Figure 10.6 (*b*)), that is, showing the internal **wrapped "sub-layers"** (the dashed rectangles):

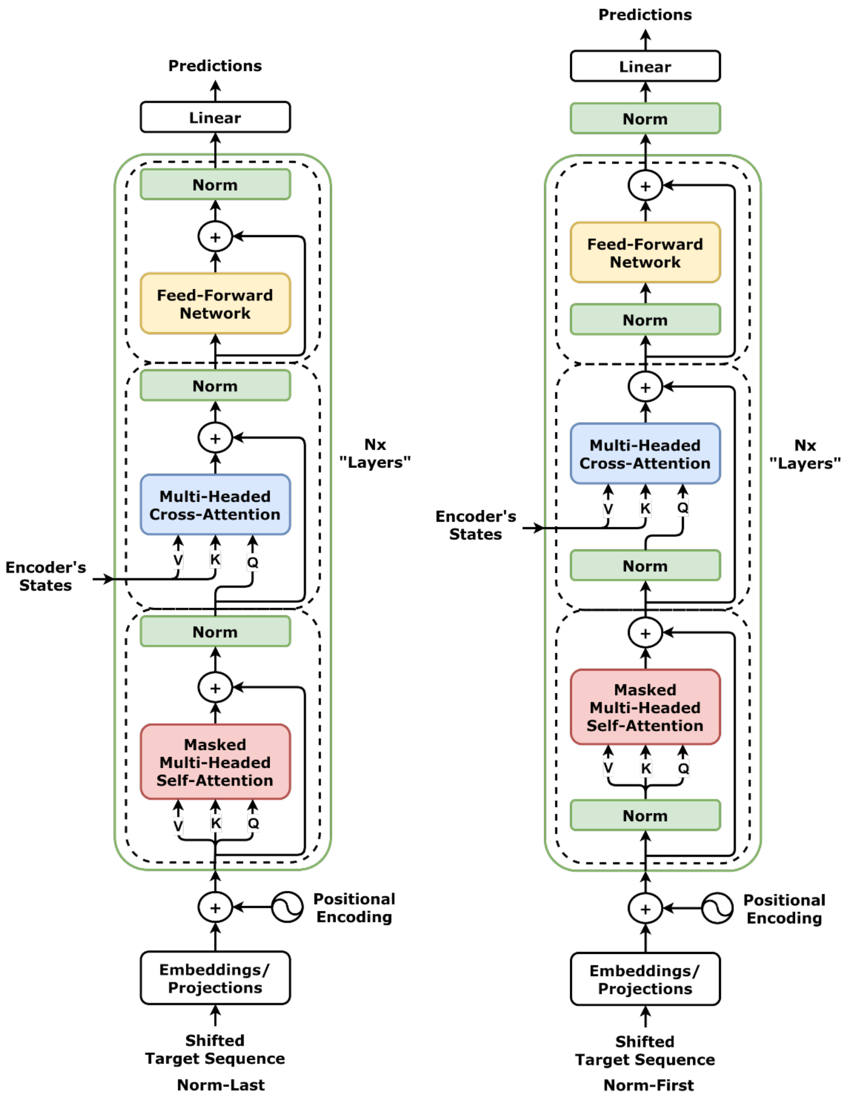


Figure 10.9 - Transformer Decoder - norm-last vs norm-first

The **small arrow** on the left represents the **states** produced by the **encoder**, which will be used as inputs for "keys" and "values" of the **(cross-)multi-headed attention** mechanism in **each "layer"**.

Moreover, there is **one final linear layer** responsible for projecting the decoder's output back to the original number of dimensions (corner's coordinates, in our case). This linear layer is *not* included in our decoder's class, though: it will be part of the encoder-decoder (or Transformer) class.

Let's see it in code, starting with the "layer", and all its wrapped "sub-layers". By the way, the code below is remarkably similar to that of the `EncoderLayer`, except for the fact that it has a third "sub-layer" (cross-attention) in between the other two:

Decoder "Layer"

```python
class DecoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.self_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.cross_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ff_units, d_model),
        )

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.drop1 = nn.Dropout(dropout)
        self.drop2 = nn.Dropout(dropout)
        self.drop3 = nn.Dropout(dropout)

    def init_keys(self, states):
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        # Sublayer #0
        # Norm
        norm_query = self.norm1(query)
        # Masked Multi-head Attention
        self.self_attn_heads.init_keys(norm_query)
        states = self.self_attn_heads(norm_query, target_mask)
        # Add
        att1 = query + self.drop1(states)

        # Sublayer #1
        # Norm
        norm_att1 = self.norm2(att1)
        # Multi-head Attention
        encoder_states = self.cross_attn_heads(norm_att1,
                                               source_mask)
        # Add
        att2 = att1 + self.drop2(encoder_states)

        # Sublayer #2
        # Norm
        norm_att2 = self.norm3(att2)
        # Feed Forward
        out = self.ffn(norm_att2)
        # Add
        out = att2 + self.drop3(out)
        return out
```

The constructor method of the **decoder "layer"** takes the **same arguments** as the **encoder "layer"'s**. The `forward` method takes three arguments: the "query", the **source mask** that's going to be used to ignore padded data points in the source sequence during **cross-attention**, and the **target mask** used to **avoid cheating** by peeking into the future.

In PyTorch, the decoder "layer" is implemented as `nn.TransformerDecoderLayer`, and its constructor method expects the same arguments (`d_model`, `nhead`, `dim_feedforward`, and `dropout`) and an optional `activation` function for the feed-forward network.

Its `forward` method, though, has **six arguments**. Three of them are equivalent to those arguments in our own `forward` method:

- `tgt`: the **target sequence**, that's the `query` argument in our class (required)

  IMPORTANT: PyTorch's **transformer layers** use **sequence-first** shapes for their inputs (L, N, F), and there is **no batch-first option**.

- `memory_key_padding_mask`: the mask for **padded data points** in the **source sequence**, that's the `source_mask` argument in our class (optional), and the same as `src_key_padding_mask` of `nn.TransformerEncoderLayer`
- `tgt_mask`: the mask used to **avoid cheating**, that's the `target_mask` argument in our class (although quite important, this argument is still considered *optional*)

Then, there is the **other required argument**, which corresponds to the `states` argument of the `init_keys` method in our own class:

- `memory`: the **encoded states** of the **source sequence** as returned by the **encoder**

There remaining two arguments *do not exist* in our own class:

- `memory_mask`: this mask is used to **purposefully hide some of the encoded states** used by the decoder

- `tgt_key_padding_mask`: this mask is used for **padded data points** in the **target sequence**

Now we can stack a bunch of "layers" like that to build an **actual decoder**:

Transformer Decoder

```python
class DecoderTransf(nn.Module):
    def __init__(self, decoder_layer, n_layers=1, max_len=100):
        super(DecoderTransf, self).__init__()
        self.d_model = decoder_layer.d_model
        self.pe = PositionalEncoding(max_len, self.d_model)
        self.norm = nn.LayerNorm(self.d_model)
        self.layers = nn.ModuleList([copy.deepcopy(decoder_layer)
                                     for _ in range(n_layers)])

    def init_keys(self, states):
        for layer in self.layers:
            layer.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        # Positional Encoding
        x = self.pe(query)
        for layer in self.layers:
            x = layer(x, source_mask, target_mask)
        # Norm
        return self.norm(x)
```

Its constructor takes an **instance of a `DecoderLayer`**, the **number of "layers"** we'd like to stack on top of one another, and a **max length** of the source sequence that's going to be used for the positional encoding. Once again, we're using `deepcopy` and `nn.ModuleList` to create multiple layers.

In PyTorch, the decoder is implemented as `nn.TransformerDecoder`, and its constructor method expects similar arguments: `decoder_layer`, `num_layers`, and an optional normalization layer to normalize (or not) the outputs.

```
declayer = nn.TransformerDecoderLayer(
    d_model=6, nhead=3, dim_feedforward=20
)
dectransf = nn.TransformerDecoder(
    declayer, num_layers=1, norm=nn.LayerNorm
)
```

PyTorch's decoder also behaves a bit *differently* than ours, since it **does not** (at the time of writing) implement **positional encoding** for the inputs, and it does not normalize the outputs by default.

Before putting the **encoder** and the **decoder** together, we still have to make a short pit-stop and address that *teeny-tiny detail*…

Layer Normalization

Layer normalization was introduced by Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffery E. Hinton in their 2016 paper "*Layer Normalization*"[148], but it only got *really* popular after being used in the hugely successful *Transformer* architecture. They say: "*…we transpose batch normalization into layer normalization by computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a* ***single training case***", the highlight is mine.

Simply put: layer normalization **standardizes individual data points**, not features.

This is completely different than the standardizations we've performed so far. Before, each **feature**, either in the whole training set (using Scikit-learn's

`StandardScaler` way back in Chapter 0), or in a mini-batch (using *batch norm* in Chapter 7), was standardized to have **zero mean** and **unit standard deviation**. In a tabular dataset, we were **standardizing the columns**.

Layer normalization, in a tabular dataset, **standardizes the rows**. Each data point will have **the average of its features equals zero**, and the **standard deviation of its features equals one**.

Let's assume we have a *mini-batch* of three sequences *(N=3)*, each sequence having a *length* of two *(L=2)*, each data point having four *features (D=4)*, and, to illustrate the importance of layer normalization, let's add **positional encoding** to it too:

```
d_model = 4
seq_len = 2
n_points = 3

torch.manual_seed(34)
data = torch.randn(n_points, seq_len, d_model)
pe = PositionalEncoding(seq_len, d_model)
inputs = pe(data)
inputs
```

Output

```
tensor([[[-3.8049,  1.9899, -1.7325,  2.1359],
         [ 1.7854,  0.8155,  0.1116, -1.7420]],

        [[-2.4273,  1.3559,  2.8615,  2.0084],
         [-1.0353, -1.2766, -2.2082, -0.6952]],

        [[-0.8044,  1.9707,  3.3704,  2.0587],
         [ 4.2256,  6.9575,  1.4770,  2.0762]]])
```

It should be straightforward to identify the different dimensions, *N* (three vertical groups), *L* (two rows in each group), and *D* (four columns) in the tensor above. There are **six data points** in total, and their value range is mostly due to the addition of positional encoding.

Well, **layer normalization standardizes individual data points**, the *rows* in the tensor above, so we need to **compute statistics over the corresponding dimension (*D*)**. Let's start with the **means**:

$$\overline{X}_{n,l} = \frac{1}{D}\sum_{d=1}^{D} x_{n,l,d}$$

Equation 10.7 - Data points' means over features (D)

```
inputs_mean = inputs.mean(axis=2).unsqueeze(2)
inputs_mean
```

Output

```
tensor([[[-0.3529],
         [ 0.2426]],

        [[ 0.9496],
         [-1.3038]],

        [[ 1.6489],
         [ 3.6841]]])
```

As expected, **six mean values**, one for each data point. The `unsqueeze` is there to preserve the original dimensionality, thus making the result a tensor of (N, L, 1) shape.

Next, we compute the *biased standard deviations* over the same dimension (*D*):

$$\sigma_{n,l}(X) = \sqrt{\frac{1}{D}\sum_{d=1}^{D}(x_{n,l,d} - \overline{X}_{n,l})^2}$$

Equation 10.8 - Data points' standard deviations over features (D)

```
inputs_var = inputs.var(axis=2, unbiased=False).unsqueeze(2)
inputs_var
```

Output

```
tensor([[[6.3756],
         [1.6661]],

        [[4.0862],
         [0.3153]],

        [[2.3135],
         [4.6163]]])
```

No surprises here.

The **actual standardization** is then computed using the *mean*, *biased standard deviation*, and a tiny *epsilon* to guarantee numerical stability:

$$standardized\ x_{n,l,d} = \frac{x_{n,l,d} - \overline{X}_{n,l}}{\sigma_{n,l}(X) + \epsilon}$$

Equation 10.9 - Layer Normalization

```
(inputs - inputs_mean)/torch.sqrt(inputs_var+1e-5)
```

Output

```
tensor([[[-1.3671,  0.9279, -0.5464,  0.9857],
         [ 1.1953,  0.4438, -0.1015, -1.5376]],

        [[-1.6706,  0.2010,  0.9458,  0.5238],
         [ 0.4782,  0.0485, -1.6106,  1.0839]],

        [[-1.6129,  0.2116,  1.1318,  0.2695],
         [ 0.2520,  1.5236, -1.0272, -0.7484]]])
```

The values above are **layer normalized**. It is possible to achieve the very same results by using PyTorch's own `nn.LayerNorm`, of course:

```
layer_norm = nn.LayerNorm(d_model)
normalized = layer_norm(inputs)

normalized[0][0].mean(), normalized[0][0].std(unbiased=False)
```

Output

```
(tensor(-1.4901e-08, grad_fn=<MeanBackward0>),
 tensor(1.0000, grad_fn=<StdBackward0>))
```

Zero mean and unit standard deviation, as expected.

"*Why do they have a* `grad_fn` *attribute?*"

Like *batch normalization*, **layer normalization** can also **learn affine transformations**. Yes, plural: each **feature has its own affine transformation**. Since we're using layer normalization on `d_model`, and its dimensionality is **four**, there will be **four weights** and **four biases** in the `state_dict`:

```
layer_norm.state_dict()
```

Output

```
OrderedDict([('weight', tensor([1., 1., 1., 1.])),
             ('bias', tensor([0., 0., 0., 0.]))])
```

The weights and biases are used to scale, and translate, respectively, the standardized values:

$$layer\ \ normed\ \ x_{n,l,d} = b_d + w_d\ \ standardized\ \ x_{n,l,d}$$

Equation 10.10 - Layer Normalization (with affine transformation)

In PyTorch's documentation, though, you'll find **gamma** and **beta** instead:

$$layer\ \ normed\ \ x_{n,l,d} = standardized\ \ x_{n,l,d}\ \ \gamma_d + \beta_d$$

Equation 10.11 - Layer Normalization (with affine transformation)

Batch and *layer* normalization look quite similar to one another, but there are some important differences between them that we need to point out.

Batch vs Layer

Although both normalizations **compute statistics**, namely, mean and biased standard deviation, to **standardize** the inputs, **only batch norm** needs to keep track of **running statistics**.

Moreover, since **layer normalization** considers **data points individually**, it **exhibits the same behavior** either if the model is in **training** or in **evaluation** mode.

To illustrate the difference between the two types of normalization, let's generate yet another dummy example (again adding positional encoding to it):

```
torch.manual_seed(23)
dummy_points = torch.randn(4, 1, 256)
dummy_pe = PositionalEncoding(1, 256)
dummy_enc = dummy_pe(dummy_points)
dummy_enc
```

Output

```
tensor([[[-14.4193,  10.0495,  -7.8116, ..., -18.0732,  -3.9566]],

        [[  2.6628,  -3.5462, -23.6461, ..., -18.4375, -37.4197]],

        [[-24.6397,  -1.9127, -16.4244, ..., -26.0550, -14.0706]],

        [[ 13.7988,  21.4612,  10.4125, ..., -17.0188,   3.9237]]])
```

There are *four sequences*, and let's pretend there are **two mini-batches of two sequences each (N=2)**. Each sequence has a *length of one* (**L=1** is not *quite* a sequence, I know), and their sole data points have **256 features (D=256)**. The figure below illustrates the difference between applying *batch norm* (over features / columns) and **layer norm** (over data points / rows):

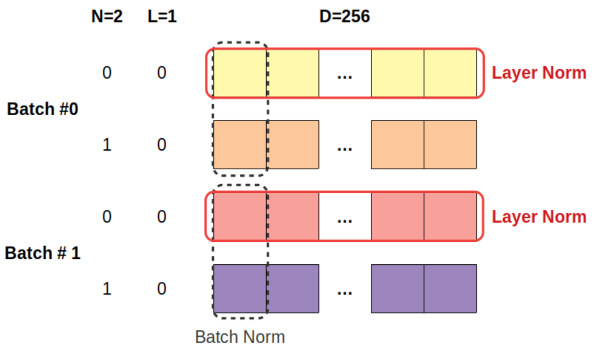


Figure 10.10 - Layer Norm vs Batch Norm

In Chapter 7 we learned that the **size of the mini-batch** strongly impacts the **running statistics** of the batch normalization. We also learned that **batch norm's oscillating statistics** may introduce a **regularizing effect**.

None of this happens with **layer normalization**: it steadily delivers data points with zero mean and unit standard deviation regardless of our choice of mini-batch size or anything else. Let's see it in action!

First, we're visualizing the **distribution of the positionally-encoded features** that we generated:

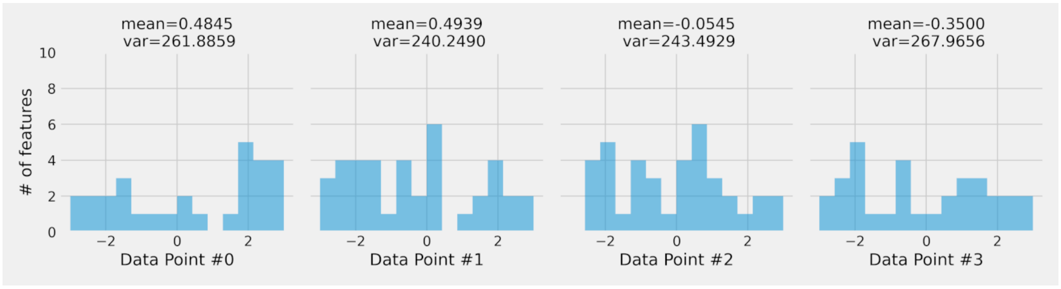


Figure 10.11 - Distribution of feature values

The *actual range* is much bigger than that (like -50 to 50), and the *variance* is approximately the same as the dimensionality (256) due to the addition of positional encoding. Let's apply **layer normalization** to it:

```
layer_normalizer = nn.LayerNorm(256)
dummy_normed = layer_normalizer(dummy_enc)
dummy_normed
```

Output

```
tensor([[[-0.9210,  0.5911, -0.5127,  ..., -1.1467, -0.2744]],

        [[ 0.1399, -0.2607, -1.5574,  ..., -1.2214, -2.4460]],

        [[-1.5755, -0.1191, -1.0491,  ..., -1.6662, -0.8982]],

        [[ 0.8643,  1.3324,  0.6575,  ..., -1.0183,  0.2611]]],
       grad_fn=<NativeLayerNormBackward>)
```

Then, we're visualizing **both distributions**, original and standardized:

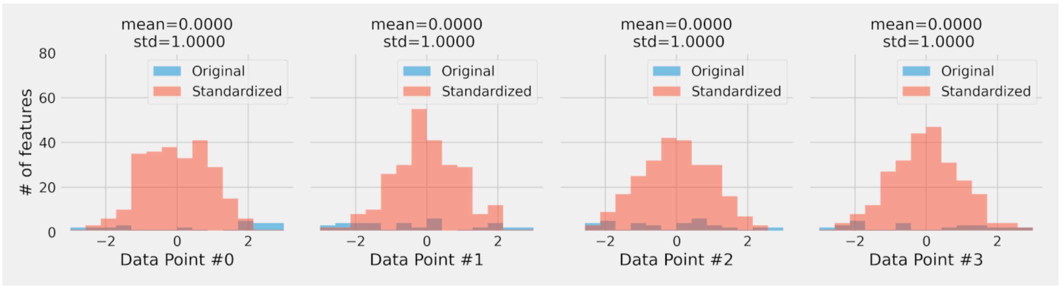


Figure 10.12 - Distribution of layer-normalized feature values

Each **data point** has its **feature values** distributed with **zero mean** and **unit standard deviation**. Beautiful!

Our Seq2Seq Problem

So far, I've been using dummy examples to illustrate how layer normalization works. Let's go back to our **sequence-to-sequence** problem, where the **source sequence** had **two data points**, each data point **representing the coordinates of**

two corners. As usual, we're adding **positional encoding** to it:

```
pe = PositionalEncoding(max_len=2, d_model=2)

source_seq = torch.tensor([[[ 1.0349,  0.9661],
                            [ 0.8055, -0.9169]]])
source_seq_enc = pe(source_seq)
source_seq_enc
```

Output

```
tensor([[[ 1.4636,  2.3663],
         [ 1.9806, -0.7564]]])
```

Next, we **normalize it**:

```
norm = nn.LayerNorm(2)
norm(source_seq_enc)
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 1.0000, -1.0000]]], grad_fn=<NativeLayerNormBackward>)
```

"*Wait, what happened here?*"

That's what happens when one tries to **normalize two features only**: they become either **minus one** or **one**. Even worse, it will be the *same* for *every data point*. These values won't get us anywhere, that's for sure.

We need to do better, we need…

Projections or Embeddings

Sometimes **projections** and **embeddings** are used interchangeably. Here, though, we're sticking with **embeddings** for **categorical** values and **projections** for **numerical values**.

In Chapter 11, we'll be using **embeddings** to get a **numerical representation** (a vector) for a given **word** or **token**. Since words or tokens are **categorical values**, the **embedding layer** works like a **big lookup table**: it will look up a given word or token in its keys and return the corresponding tensor.

But, since we're dealing with **coordinates**, that is, **numerical values**, we are using **projections** instead. A simple linear layer is all that it takes to project our pair of coordinates into a **higher-dimensional** feature space:

```
torch.manual_seed(11)
proj_dim = 6
linear_proj = nn.Linear(2, proj_dim)
pe = PositionalEncoding(2, proj_dim)

source_seq_proj = linear_proj(source_seq)
source_seq_proj_enc = pe(source_seq_proj)
source_seq_proj_enc
```

Output

```
tensor([[[-2.0934,  1.5040,  1.8742,  0.0628,  0.3034,  2.0190],
         [-0.8853,  2.8213,  0.5911,  2.4193, -2.5230,  0.3599]]],
       grad_fn=<AddBackward0>)
```

See? Now each data point in our source sequence has **six features** (the projected dimensions), and they are **positionally-encoded** too. Sure, *this* particular projection is totally random, but that won't be the case anymore once we add the corresponding linear layer to our model. It will learn a meaningful projection that, after being positionally-encoded, will be normalized:

```
norm = nn.LayerNorm(proj_dim)
norm(source_seq_proj_enc)
```

Output

```
tensor([[[-1.9061,  0.6287,  0.8896, -0.3868, -0.2172,  0.9917],
         [-0.7362,  1.2864,  0.0694,  1.0670, -1.6299, -0.0568]]],
       grad_fn=<NativeLayerNormBackward>)
```

Problem solved!

In Chapter 9, we used the *affine transformations* inside the **attention heads** to map from **input dimensions** to **hidden (or model) dimensions**.

Now, this **change in dimensionality** is performed using **projections** directly on the **input sequences** before they are passed to the encoder and the decoder.

Finally, we have **everything** to build a full-blown **Transformer**!

The Transformer

Let’s start with the diagram, which is nothing more than an encoder and a decoder side-by-side (we’re sticking with **norm-first "sub-layer" wrappers**):

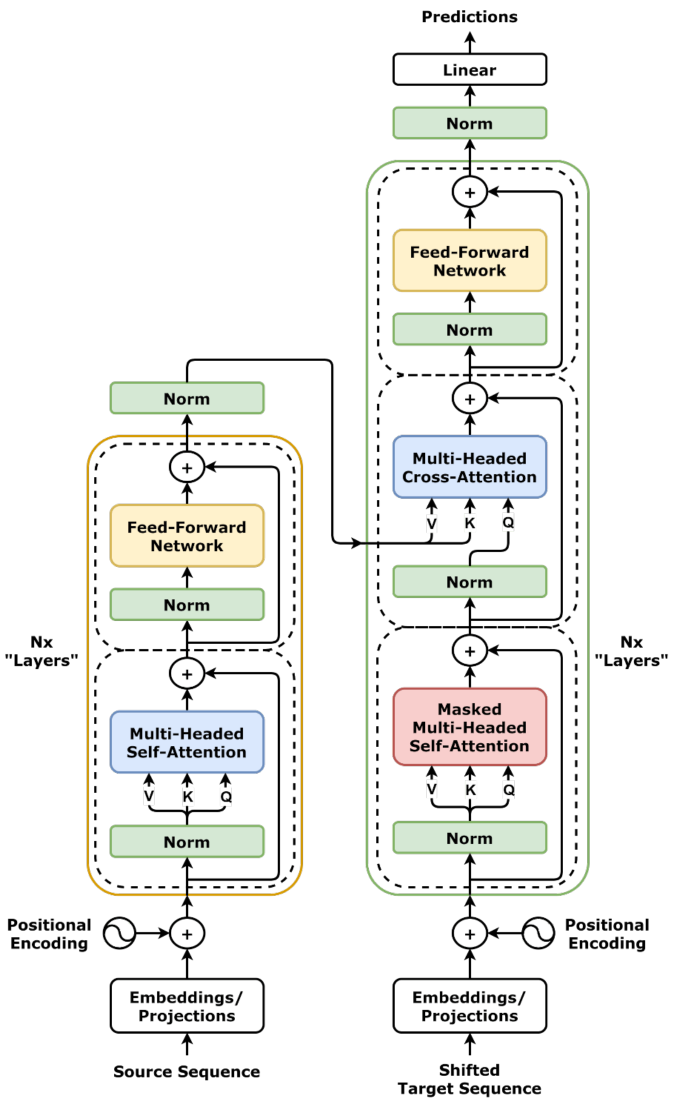


Figure 10.13 - The Transformer (norm-first)

The **Transformer** still is an **encoder-decoder architecture** like the one we developed in the previous chapter, so it should be no surprise that we can actually use our former `EncoderDecoderSelfAttn` class as a *parent class* and add **two extra components** to it:

- a **projection** layer to map our original **features** (`n_features`) to the

dimensionality of both encoder and decoder (`d_model`)

- a **final linear** layer to map the decoder's outputs back to the original **feature space** (the coordinates we're trying to predict)

We also need to make some **small modifications** to the `encode` and `decode` methods to account for the components above:

Transformer Encoder-Decoder

```
class EncoderDecoderTransf(EncoderDecoderSelfAttn):
    def __init__(self, encoder, decoder,
                 input_len, target_len, n_features):
        super(EncoderDecoderTransf, self).__init__(
            encoder, decoder, input_len, target_len
        )
        self.n_features = n_features
        self.proj = nn.Linear(n_features, encoder.d_model)      ①
        self.linear = nn.Linear(encoder.d_model, n_features)    ②

    def encode(self, source_seq, source_mask=None):
        # Projection
        source_proj = self.proj(source_seq)                     ①
        encoder_states = self.encoder(source_proj, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Projection
        target_proj = self.proj(shifted_target_seq)             ①
        outputs = self.decoder(target_proj,
                               source_mask=source_mask,
                               target_mask=target_mask)
        # Linear
        outputs = self.linear(outputs)                          ②
        return outputs
```

① Projecting features to model dimensionality

② Final linear transformation from model to feature space

Let's briefly review the model's methods:

- `encode`: takes the **source sequence and mask** and **encodes its projection** into a **sequence of states** that is immediately used to **initialize the "keys" (and "values")** in the **decoder**
- `decode`: takes the **shifted target sequence** and use **its projection** together with both **source and target masks** to generate a **target sequence** that goes through the **last linear layer** to be transformed back to **feature space** - it is used for **training** only

The parent class is reproduced below for your convenience:

Encoder + Decoder + Self-Attention

```python
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)

    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequent_mask = (
            1 - torch.triu(torch.ones(attn_shape), diagonal=1)
        ).bool()
        return subsequent_mask

    def encode(self, source_seq, source_mask):
        # Encodes the source sequence and uses the result
        # to initialize the decoder
        encoder_states = self.encoder(source_seq, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Decodes/generates a sequence using the shifted (masked)
        # target sequence - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs

    def predict(self, source_seq, source_mask):
        # Decodes/generates a sequence using one input
        # at a time - used in EVAL mode
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.decode(inputs,
                              source_mask,
                              self.trg_masks[:, :i+1, :i+1])
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = inputs[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        # Sends the mask to the same device as the inputs
        self.trg_masks = self.trg_masks.type_as(X).bool()
        # Slices the input to get source sequence
        source_seq = X[:, :self.input_len, :]
        # Encodes source sequence AND initializes decoder
        self.encode(source_seq, source_mask)
        if self.training:
            # Slices the input to get the shifted target seq
            shifted_target_seq = X[:, self.input_len-1:-1, :]
            # Decodes using the mask to prevent cheating
```

```
                outputs = self.decode(shifted_target_seq,
                                      source_mask,
                                      self.trg_masks)
            else:
                # Decodes using its own predictions
                outputs = self.predict(source_seq, source_mask)

            return outputs
```

Since the **Transformer** is an **encoder-decoder architecture**, we can use it in a **sequence-to-sequence** problem. Well, we already have one of those, right? Let's reuse Chapter 9's "*Data Preparation*" code.

Data Preparation

We'll **keep drawing the first two corners of the squares** ourselves, the **source sequence**, and ask our model to **predict the next two corners**, the **target sequence**, same as in Chapter 9.

Data Preparation

```
# Generating training data
points, directions = generate_sequences()
full_train = torch.as_tensor(points).float()
target_train = full_train[:, 2:]
# Generating test data
test_points, test_directions = generate_sequences(seed=19)
full_test = torch.as_tensor(points).float()
source_test = full_test[:, :2]
target_test = full_test[:, 2:]
# Datasets and data loaders
train_data = TensorDataset(full_train, target_train)
test_data = TensorDataset(source_test, target_test)

generator = torch.Generator()
train_loader = DataLoader(train_data, batch_size=16,
                          shuffle=True, generator=generator)
test_loader = DataLoader(test_data, batch_size=16)
```

```
fig = plot_data(points, directions, n_rows=1)
```

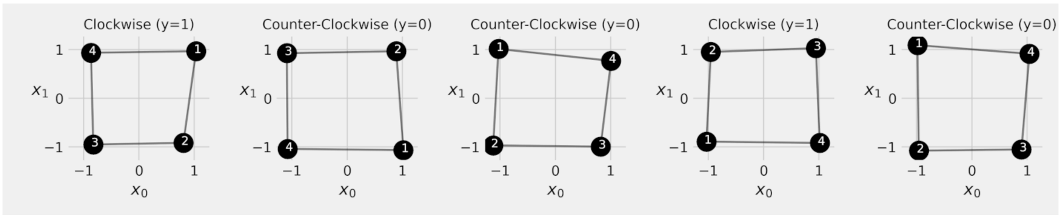


Figure 10.14 - Seq2Seq Dataset

The corners show the **order** in which they were drawn. In the first square, the drawing **started at the top-right corner** and followed a **clockwise direction**. The **source sequence** for that square would include corners **on the right edge (1 and 2)**, while the **target sequence** would include corners **on the left edge (3 and 4)**, in that order.

Model Configuration & Training

Let’s train our **Transformer**! We start by creating the corresponding "layers" for both encoder and decoder, and use them both as arguments of the `EncoderDecoderTransf` class:

Model Configuration

```python
torch.manual_seed(42)
# Layers
enclayer = EncoderLayer(n_heads=3, d_model=6,
                        ff_units=10, dropout=0.1)
declayer = DecoderLayer(n_heads=3, d_model=6,
                        ff_units=10, dropout=0.1)
# Encoder and Decoder
enctransf = EncoderTransf(enclayer, n_layers=2)
dectransf = DecoderTransf(declayer, n_layers=2)
# Transformer
model_transf = EncoderDecoderTransf(
    enctransf, dectransf, input_len=2, target_len=2, n_features=2
)
loss = nn.MSELoss()
optimizer = torch.optim.Adam(model_transf.parameters(), lr=0.01)
```

The original Transformer model was initialized using **Glorot/Xavier uniform** distribution, so we’re sticking with it:

Weight Initialization

```python
for p in model_transf.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

Next, we use the `StepByStep` class to train the model as usual:

Model Training

```
sbs_seq_transf = StepByStep(model_transf, loss, optimizer)
sbs_seq_transf.set_loaders(train_loader, test_loader)
sbs_seq_transf.train(50)
```

```
fig = sbs_seq_transf.plot_losses()
```

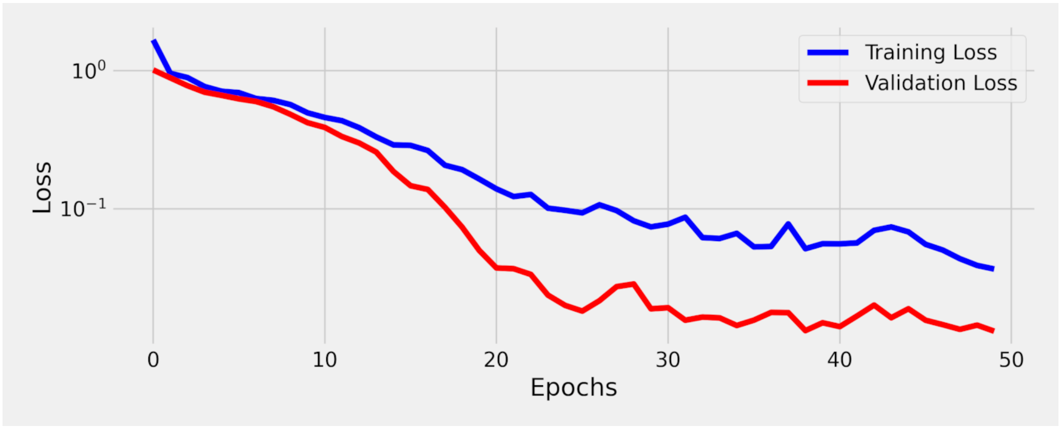


Figure 10.15 - Losses - Transformer model

*"Why is the validation loss **so much better** than the training loss?"*

This phenomenon may happen for a variety of reasons, from an *easier* validation set to a "*side effect*" of **regularization** (e.g. dropout) in our current model. The regularization makes it *harder* for the model to learn or, in other words, yields **higher losses**. In our Transformer model, there are **many dropout layers**, so it gets increasingly more difficult for the model to learn.

Let's observe this effect by using the **same mini-batch** to compute the **loss** using the **trained model** both in **`train`** and **`eval`** modes:

```
torch.manual_seed(11)
x, y = next(iter(train_loader))
device = sbs_seq_transf.device
# Training
model_transf.train()
loss(model_transf(x.to(device)), y.to(device))
```

Output

```
tensor(0.0480, device='cuda:0', grad_fn=<MseLossBackward>)
```

```
# Validation
model_transf.eval()
loss(model_transf(x.to(device)), y.to(device))
```

Output

```
tensor(0.0101, device='cuda:0')
```

See the difference? The loss is roughly **three times** larger in training mode. You can also **set dropout to zero** and retrain the model to verify that both loss curves get much closer to each other (by the way, the overall loss level gets *better* without dropout, but that's just because our sequence-to-sequence problem is actually quite simple).

Visualizing Predictions

Let's plot the **predicted coordinates** and connect them using **dashed lines**, while using **solid lines** to connect the **actual coordinates**, just like before:

```
fig = sequence_pred(sbs_seq_transf, full_test, test_directions)
```

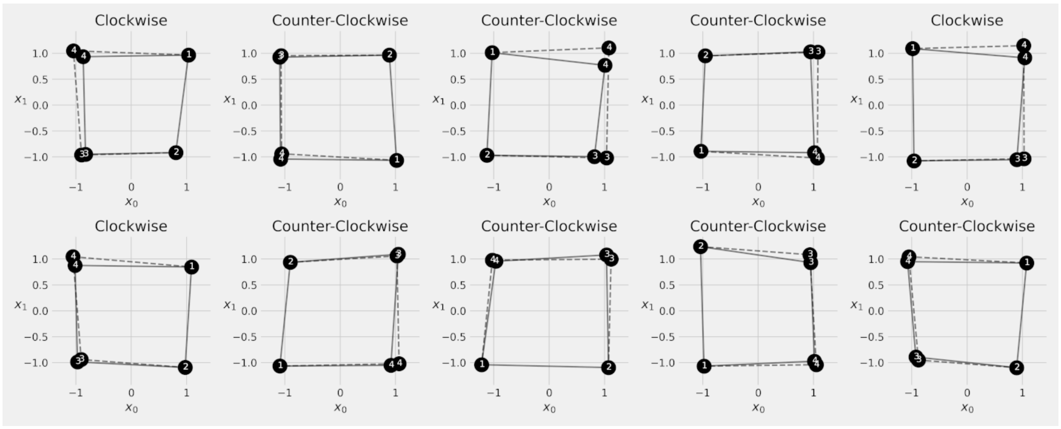


Figure 10.16 - Predictions

Looking good, right?

The PyTorch Transformer

So far we've been using our own classes to build encoder and decoder "layers" and assemble them all together into a Transformer. We **don't have to** do it like that, though. PyTorch implements a full-fledged **Transformer** class of its own: `nn.Transformer`.

There are some differences between PyTorch's implementation and our own:

- first, and most importantly, PyTorch implements **norm-last "sub-layer" wrappers**, normalizing the output of each "sub-layer":

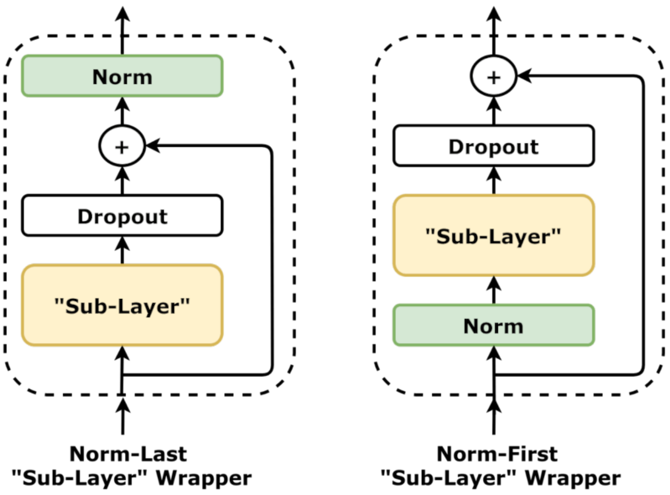


Figure 10.17 - "Sub-Layer" - norm-last vs norm-first

- it **does not** implement **positional encoding**, the **final linear layer**, and the **projection layer**, so we have to handle those ourselves

Let's take a look at its constructor and `forward` methods. The constructor expects *many arguments* because PyTorch's Transformer actually **builds both encoder and decoder by itself**:

- `d_model`: the **number of features** of the inputs (remember, this number will be *split* among the attention heads, so it must be a multiple of the number of heads, its default value is 512)
- `nhead`: the number of **attention heads** in each **attention mechanism** (default is eight, so each attention head gets 64 out of the 512 dimensions)
- `num_encoder_layers`: the number of "layers" in the encoder (the Transformer uses six layers by default)
- `num_decoder_layers`: the number of "layers" in the decoder (the Transformer uses six layers by default)
- `dim_feedforward`: the number of **units** in the **hidden layer** of the **feed-forward network** (default is 2048)

- `dropout`: the **probability** of dropping out inputs (default is 0.1)
- `activation`: the activation function to be used in the feed-forward network (ReLU by default)

It is also possible to use a **custom encoder or decoder** by setting the corresponding arguments: `custom_encoder` and `custom_decoder`. But don't forget that the PyTorch Transformer expects **sequence-first** inputs.

The `forward` method expects both sequences, **source** and **target**, and **all sorts of (optional) masks**.

IMPORTANT: PyTorch's **Transformer** uses **sequence-first** shapes for its inputs (L, N, F), and there is **no batch-first option**.

There are **masks** for **padded data points**:

- `src_key_padding_mask`: the mask for **padded data points** in the **source sequence**
- `memory_key_padding_mask`: also a mask for **padded data points** in the **source sequence** that should be, in most cases, the *same* as `src_key_padding_mask`
- `tgt_key_padding_mask`: this mask is used for **padded data points** in the **target sequence**

And there are **masks** to **purposefully hide some of the inputs**:

- `src_mask`: hides inputs in the **source sequence**, which can be used for training **language models** (more on that in Chapter 11)
- `tgt_mask`: the mask used to **avoid cheating** (although quite important, this argument is still considered *optional*)
    - the Transformer has a method named `generate_square_subsequent_mask` that generates the appropriate mask given the size (length) of the sequence

- `memory_mask`: hides **encoded states** used by the decoder

Also, notice that there is no **memory argument** anymore: the encoded states are handled internally by the Transformer and fed directly to the decoder part.

In the code, we'll be replacing the two former methods, `encode` and `decode`, by a single one, `encode_decode`, that calls the **Transformer** itself and runs its output through the **last linear layer** to transform them into coordinates. Since the Transformer expects and outputs **sequence-first** shapes, there is some back-and-forth **permuting** as well.

```python
def encode_decode(self, source, target,
                  source_mask=None, target_mask=None):
    # Projections
    # PyTorch Transformer expects L, N, F
    src = self.preprocess(source).permute(1, 0, 2)
    tgt = self.preprocess(target).permute(1, 0, 2)

    out = self.transf(src, tgt,
                      src_key_padding_mask=source_mask,
                      tgt_mask=target_mask)

    # Linear
    # Back to N, L, D
    out = out.permute(1, 0, 2)
    out = self.linear(out) # N, L, F
    return out
```

By the way, we're keeping the masks to a minimum for the sake of simplicity: only `src_key_padding_mask` and `tgt_mask` are used.

Moreover, we're implementing a `preprocess` method that takes an **input sequence** and:

- **projects** the original features into the model dimensionality

* adds **positional encoding**
* and **(layer) normalizes** the result (remember that PyTorch's implementation *does not normalize the inputs*, so we have to do it ourselves)

The full code looks like this:

Transformer

```
class TransformerModel(nn.Module):
    def __init__(self, transformer,
                 input_len, target_len, n_features):
        super().__init__()
        self.transf = transformer
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = \
            self.transf.generate_square_subsequent_mask(
                self.target_len
            )
        self.n_features = n_features
        self.proj = nn.Linear(n_features, self.transf.d_model) ①
        self.linear = nn.Linear(self.transf.d_model,           ②
                                n_features)

        max_len = max(self.input_len, self.target_len)
        self.pe = PositionalEncoding(max_len,
                                     self.transf.d_model)      ③
        self.norm = nn.LayerNorm(self.transf.d_model)          ③

    def preprocess(self, seq):
        seq_proj = self.proj(seq)                              ①
        seq_enc = self.pe(seq_proj)                            ③
        return self.norm(seq_enc)                              ③

    def encode_decode(self, source, target,
                      source_mask=None, target_mask=None):
```

```
        # Projections
        # PyTorch Transformer expects L, N, F
        src = self.preprocess(source).permute(1, 0, 2)          ③
        tgt = self.preprocess(target).permute(1, 0, 2)          ③

        out = self.transf(src, tgt,
                          src_key_padding_mask=source_mask,
                          tgt_mask=target_mask)

        # Linear
        # Back to N, L, D
        out = out.permute(1, 0, 2)
        out = self.linear(out) # N, L, F                        ②
        return out

    def predict(self, source_seq, source_mask=None):
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.encode_decode(
                source_seq, inputs,
                source_mask=source_mask,
                target_mask=self.trg_masks[:i+1, :i+1]
            )
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = out[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        self.trg_masks = self.trg_masks.type_as(X)
        source_seq = X[:, :self.input_len, :]

        if self.training:
            shifted_target_seq = X[:, self.input_len-1:-1, :]
            outputs = self.encode_decode(
                source_seq, shifted_target_seq,
                source_mask=source_mask,
                    target_mask=self.trg_masks
                )
        else:
            outputs = self.predict(source_seq, source_mask)

        return outputs
```

① Projecting features to model dimensionality

② Final linear transformation from model to feature space

③ Adding positional encoding and normalizing inputs

Its constructor takes an instance of the `nn.Transformer` class followed by the typical sequence lengths *and* the number of features (so it can map the predicted sequence back to our feature space, that is, the coordinates). Both `predict` and `forward` methods are roughly the same, but they call the `encode_decode` method now.

Model Configuration & Training

Let's train PyTorch's **Transformer**! We start by creating an instance of it to use as an argument of our `TransformerModel` class, followed by the same initialization scheme as before, and the typical training procedure:

Model Configuration

```python
torch.manual_seed(42)
transformer = nn.Transformer(d_model=6,
                             nhead=3,
                             num_encoder_layers=1,
                             num_decoder_layers=1,
                             dim_feedforward=20,
                             dropout=0.1)
model_transformer = TransformerModel(transformer, input_len=2,
                                     target_len=2, n_features=2)
loss = nn.MSELoss()
optimizer = torch.optim.Adam(model_transformer.parameters(),
                             lr=0.01)
```

Weight Initialization

```python
for p in model_transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

Model Training

```python
sbs_seq_transformer = StepByStep(
    model_transformer, loss, optimizer
)
sbs_seq_transformer.set_loaders(train_loader, test_loader)
sbs_seq_transformer.train(50)
```

```python
fig = sbs_seq_transformer.plot_losses()
```

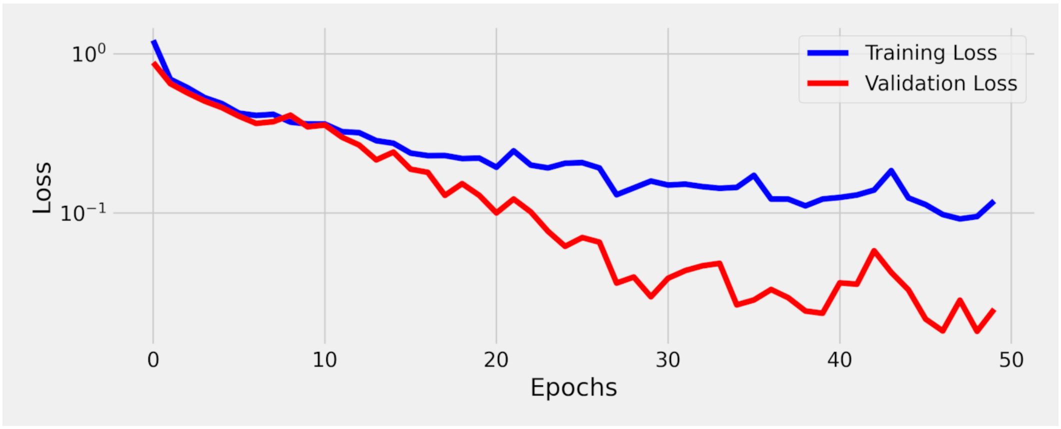


Figure 10.18 - Losses - PyTorch's Transformer

Once again, the validation loss is significantly lower than the training loss. No surprises here since it is roughly the same model.

Visualizing Predictions

Let's plot the **predicted coordinates** and connect them using **dashed lines**, while using **solid lines** to connect the **actual coordinates**, just like before:

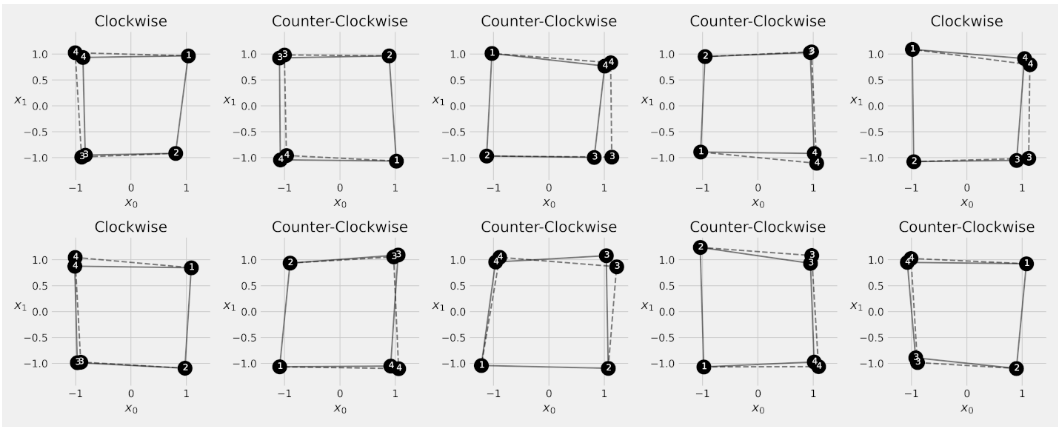


Figure 10.19 - Predictions

Once again, looking good, right?

Vision Transformer

The Transformer architecture is fairly flexible and, although it was devised to handle NLP tasks in the first place, it is already starting to spread to different areas, including Computer Vision. Let's take a look at one of the latest developments in the field: the Vision Transformer (ViT). It was introduced by Dosovitskiy, A., et al. in their paper *"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale"*[149].

*"Cool, but I thought the Transformer handled **sequences**, not images... "*

That's a fair point. The answer is deceptively simple: let's **break an image** into a **sequence of patches**.

Data Generation & Preparation

First, let's bring back our multiclass classification problem from Chapter 5. We're generating a synthetic dataset of images that are going to have either a diagonal or a parallel line, and labeling them according to the table below:

Line	Label/Class Index
Parallel (Horizontal OR Vertical)	0
Diagonal, Tilted to the Right	1
Diagonal, Tilted to the Left	2

Data Generation

```
images, labels = generate_dataset(img_size=12, n_images=1000,
                                  binary=False, seed=17)
```

Each image, like the example below, is 12x12 pixels in size and has a single channel:

```
img = torch.as_tensor(images[2]).unsqueeze(0).float()/255.
```

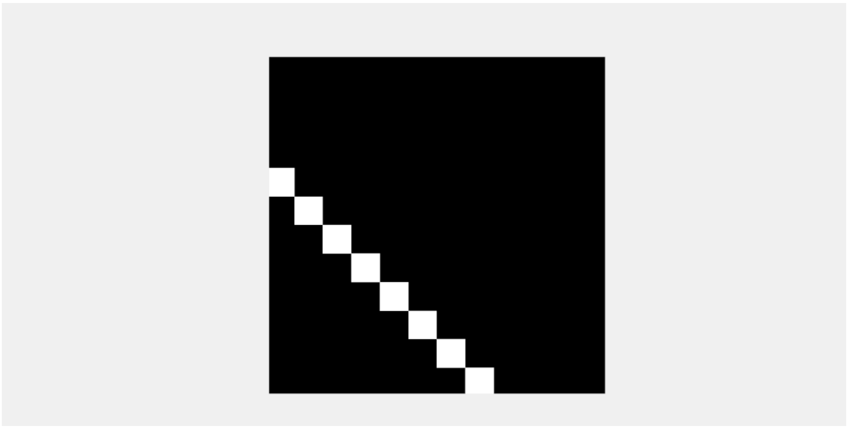


Figure 10.20 - Sample image - Label "2"

*"But this is a **classification problem**, not a sequence-to-sequence one… why are we using a Transformer then?"*

Well, we're *not* using the *full* Transformer architecture, but its **encoder only**. In Chapter 8, we used recurrent neural networks to generate a *final hidden state* that we used as the input for classification. Similarly, the **encoder** generates a **sequence of "hidden states"** (the *memory*, in Transformer lingo), and we're using **one "hidden state"** as the input for classification again.

"Which one? The last "hidden state"?"

No, not the last one, but a **special one**. We'll **prepend a special classifier token** **`[CLS]`** to our sequence and use its corresponding "hidden state" as input to a classifier. The figure below illustrates the idea:

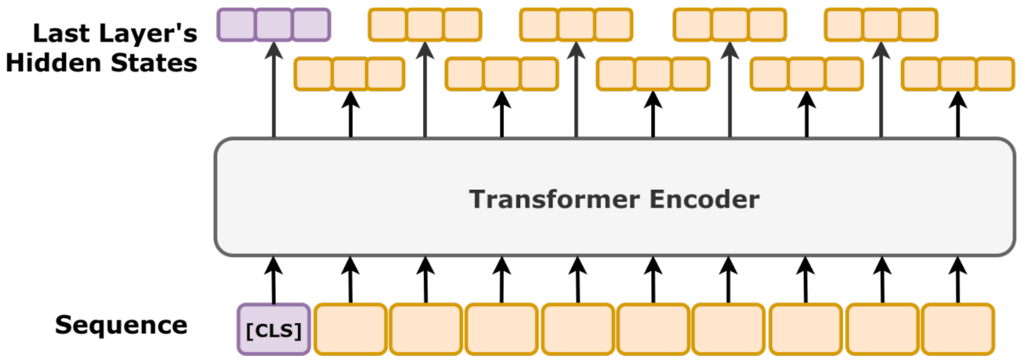


Figure 10.21 - Hidden States and the special classifier token `[CLS]`

But I'm *jumping the gun* here… we'll get back to that in the "*Special Classifier Token*" section.

The *data preparation* step is exactly the same one we used in Chapter 5:

Data Preparation

```
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]
        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)

# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels).long()

# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]

# We're not doing any data augmentation now
train_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])

# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(
    x_train_tensor, y_train_tensor, transform=train_composer)
val_dataset = TransformedTensorDataset(
    x_val_tensor, y_val_tensor, transform=val_composer)

# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)

# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(
    dataset=train_dataset, batch_size=16, sampler=sampler)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Patches

There are different ways of breaking up an image into patches. The most straightforward one is simply **rearranging** the pixels, so let's start with that one.

Rearranging

Tensorflow has a utility function called `tf.image.extract_patches` that does the job, and we're implementing a simplified version of this function in PyTorch with `tensor.unfold` (using only a *kernel size* and a *stride*, but no *padding* or anything else):

```
# Adapted from https://discuss.pytorch.org/t/tf-extract-image-
# patches-in-pytorch/43837
def extract_image_patches(x, kernel_size, stride=1):
    # Extract patches
    patches = x.unfold(2, kernel_size, stride)
    patches = patches.unfold(3, kernel_size, stride)
    patches = patches.permute(0, 2, 3, 1, 4, 5).contiguous()

    return patches.view(n, patches.shape[1], patches.shape[2], -1)
```

It works *as if* we were applying a convolution to the image. Each **patch** is actually a *receptive field* (the region the filter is moving over to convolve) but, instead of convolving the region, we're just taking it as it is. The **kernel size** is the **patch size** and the **number of patches** depends on the **stride** - the smaller the stride, the more patches. If the **stride matches the kernel size**, we're effectively **breaking up** the image into **non-overlapping patches**, so let's do that:

```
kernel_size = 4
patches = extract_image_patches(
    img, kernel_size, stride=kernel_size
)
patches.shape
```

Output

```
torch.Size([1, 3, 3, 16])
```

Since kernel size is four, **each patch has 16 pixels**, and there are **nine patches** in total. Even though each patch is a tensor of 16 elements, if we plot them as if they were four-by-four images instead, it would look like this:

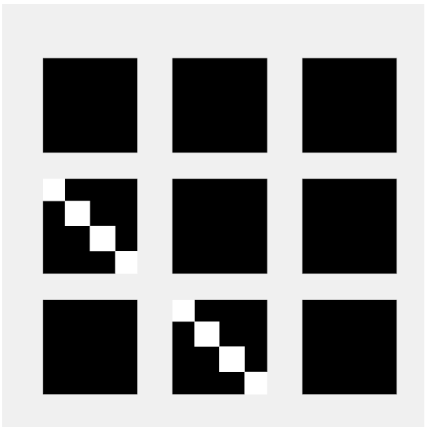


Figure 10.22 - Sample image - split into patches

It is very easy to see how the image was broken up in the figure above. In reality, though, the Transformer needs a **sequence of flattened patches**. Let's reshape them:

```
seq_patches = patches.view(-1, patches.size(-1))
```

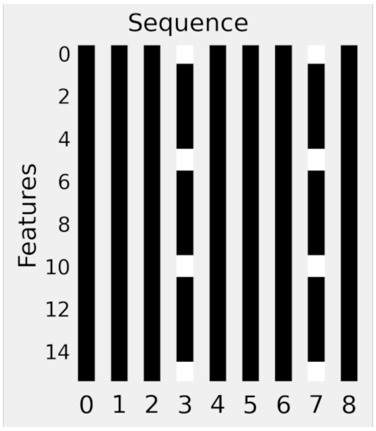


Figure 10.23 - Sample image - split into a sequence of flattened patches

That's more like it: each image is turned into a **sequence of length nine**, each element in the sequence having **16 features** (pixel values in this case).

Einops

"*There is more than one way to skin a cat*" as the saying goes, and so there is more than one way to rearrange the pixels into sequences. An alternative approach uses a package called einops[150]: it is **very** minimalistic (maybe even a bit too much) and it allows you to express complex rearrangements in a couple lines of code. It may take a while to get the hang of how it works, though.

We're not using it here but, if you're interested, this is the einops equivalent of the extract_image_patches function above:

```
# Adapted from https://github.com/lucidrains/vit-pytorch/blob/
# main/vit_pytorch/vit_pytorch.py
# !pip install einops
from einops import rearrange
patches = rearrange(padded_img,
                    'b c (h p1) (w p2) -> b (h w) (p1 p2 c)',
                    p1 = kernel_size, p2 = kernel_size)
```

Embeddings

If each **patch** is like a *receptive field*, and we even talked about **kernel size** and **stride**, why not go **full convolution** then? That's how the **Visual Transformer (ViT)** actually implemented **patch embeddings**:

Patch Embeddings

```
# Adapted from https://amaarora.github.io/2021/01/18/ViT.html
class PatchEmbed(nn.Module):
    def __init__(self, img_size=224, patch_size=16,
                 in_channels=3, embed_dim=768, dilation=1):
        super().__init__()
        num_patches = (img_size // patch_size) * \
                      (img_size // patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.proj = nn.Conv2d(in_channels,
                              embed_dim,
                              kernel_size=patch_size,
                              stride=patch_size)

    def forward(self, x):
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

The **patch embedding** is not the original receptive field anymore, but the **convolved receptive field**. After convolving the image, given a kernel size and a stride, the **patches get flattened**, so we end up with the same **sequence of nine patches**:

```
torch.manual_seed(13)
patch_embed = PatchEmbed(
    img.size(-1), kernel_size, 1, kernel_size**2
)
embedded = patch_embed(img)
embedded.shape
```

Output

```
torch.Size([1, 9, 16])
```

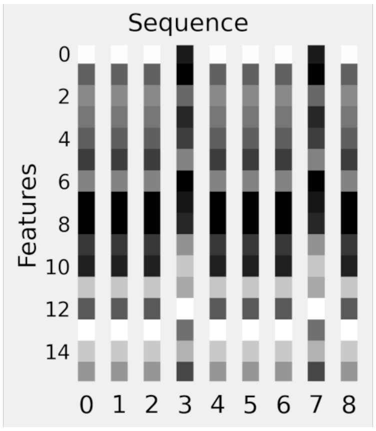


Figure 10.24 - Sample image - split into a sequence of **patch embeddings**

But the patches are **linear projections** of the original **pixel values** now. We're projecting each 16-pixel patch into a 16-dimensional feature space - we're not changing the number of dimensions, but using **different linear combinations of the original dimensions**.

The original image had **144 pixels** and it was split into **nine patches** of **16 pixels** each. Each patch **embedding *still* has 16 dimensions** so, overall, each image is *still* represented by **144 values**. This is by no means a coincidence: the idea behind this choice of values is to **preserve the dimensionality** of the inputs.

Special Classifier Token

In Chapter 8, the *final hidden state* represented the *full sequence*. This approach had its shortcomings (the attention mechanism was developed to compensate for them) but it leveraged the fact that **there was an underlying, sequential, structure** to the data.

This is *not quite the same* for images, though. The **sequence of patches** is a clever way of making the data suitable for the **encoder**, sure, but it does not necessarily reflect a sequential structure, after all, we end up with **two different sequences** depending on which direction we choose to go over the patches: **row-wise** or **column-wise**.

*"Can't we use the **full sequence** of "hidden states" then? Or maybe **average** them?"*

It is definitely *possible* to use the **average** of the "hidden states" produced by the encoder as input for the classifier.

But it is also common to use a **special classifier token `[CLS]`**, especially in NLP tasks (as we'll see in Chapter 11). The idea is quite simple and elegant: **add the same token to the beginning of every sequence**. This special token has **an embedding** as well, and it will be **learned** by the model like any other parameter.

The **first "hidden state"** produced by the encoder, the output corresponding to the **added special token**, plays the role of **overall representation** of the image - just like the final hidden state represented the overall sequence in recurrent neural networks.

Remember that the Transformer Encoder uses **self-attention** and that **every token** can pay attention **to every other token** in the sequence. Therefore, the **special classifier token** can actually learn **which tokens (patches, in our case)** it needs to pay attention to in order to correctly classify the sequence.

Let's illustrate the addition of the `[CLS]` token by taking two images from our dataset:

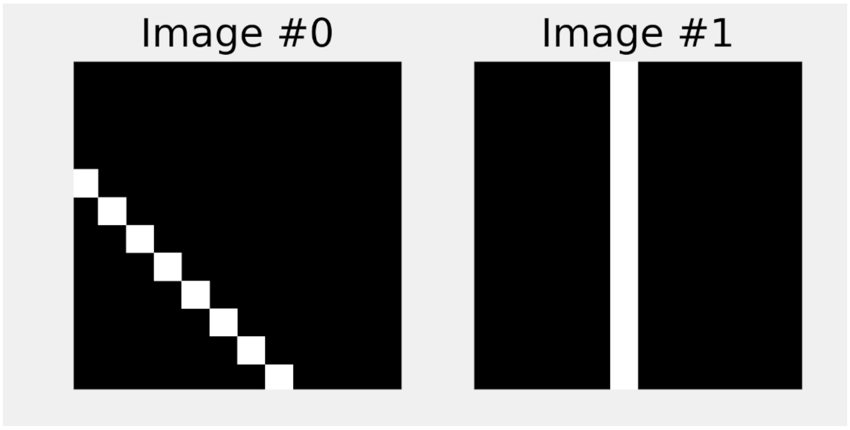


Figure 10.25 - Two images

Next, we get their corresponding **patch embeddings**:

```
embeddeds = patch_embed(imgs)
```

Our images were transformed into **sequences of nine patch embeddings of size 16** each, so they can be represented like this (the features are on the horizontal axis now):

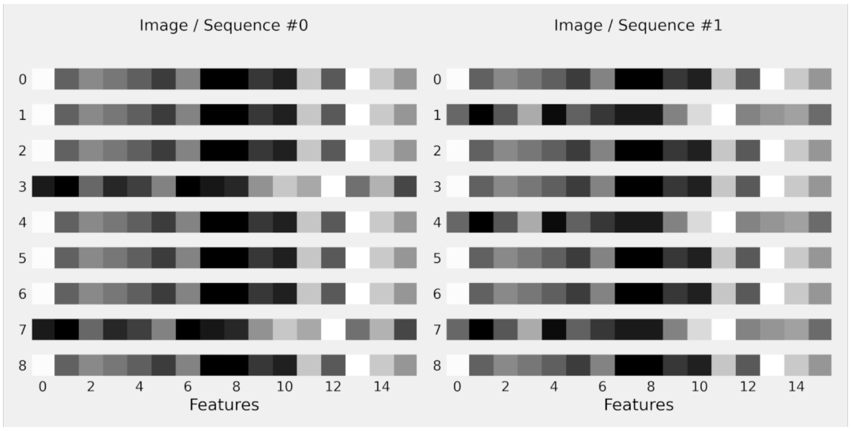


Figure 10.26 - Two patch embeddings

The **patch embeddings** are obviously different for each image but the **embedding** corresponding to the **special classifier token** that's prepended to the patch embeddings is always the **same**:

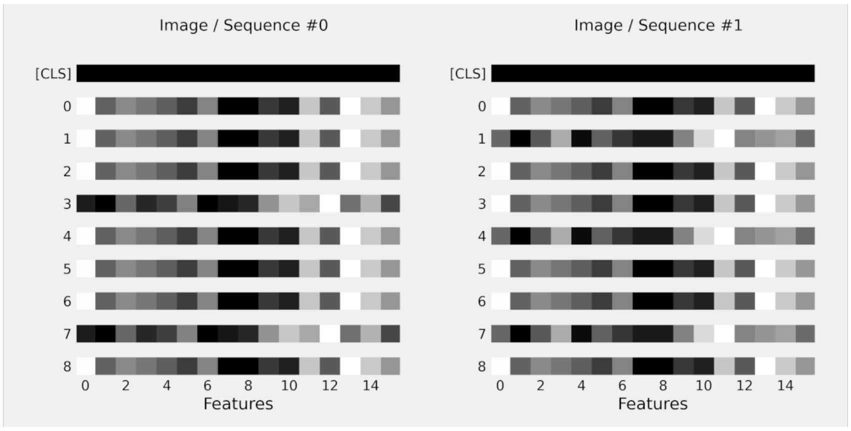


Figure 10.27 - Two patch embeddings + `[CLS]` *embedding*

"How do we do that?"

It's actually simple: we need to define a **parameter** in our model (using `nn.Parameter`) to represent this **special embedding** and **concatenate** it at the beginning of every sequence of embeddings. Let's start by creating the parameter itself (it will be an attribute of our model later):

```
cls_token = nn.Parameter(torch.zeros(1, 1, 16))
cls_token
```

Output

```
Parameter containing:
tensor([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.]]], requires_grad=True)
```

It is just a **vector full of zeros**. That's it. But, since it is a parameter, its values will be updated as the model gets trained.

Then, let's fetch a mini-batch of images and get their **patch embeddings**:

```
images, labels = next(iter(train_loader))
images.shape # N, C, H, W
```

Output

```
torch.Size([16, 1, 12, 12])
```

```
embed = patch_embed(images)
embed.shape # N, L, D
```

Output

```
torch.Size([16, 9, 16])
```

There are **16 images**, each image represented by a **sequence of nine patches with 16 dimensions each**. The **special embedding** should be the **same** for all 16 images, so we use `tensor.expand` to **replicate it** along the batch dimension before concatenation:

```
cls_tokens = cls_token.expand(embed.size(0), -1, -1)
embed_cls = torch.cat((cls_tokens, embed), dim=1)
embed_cls.shape # N, L+1, D
```

Output

```
torch.Size([16, 10, 16])
```

Now each **sequence has ten elements**, and we have everything we need to build our model.

The Model

The *main* part of the model is the **Transformer encoder** which, coincidentally, is implemented **normalizing the inputs** (norm-first) like our own `EncoderLayer` and `EncoderTransf` classes (and unlike PyTorch's default implementation).

The encoder outputs a sequence of "hidden states" (*memory*) and the **first** one is used as input to a **classifier** ("*MLP Head*"), as briefly discussed in the previous section. So, the model is all about **preprocessing the inputs**, our images, using a series of transformations:

- computing **a sequence of patch embeddings**
- prepending the same **special classifier token `[CLS]` embedding** to every sequence
- adding position embedding (or, in our case, **position encoding** implemented in our **encoder**)

The figure below illustrates the architecture:

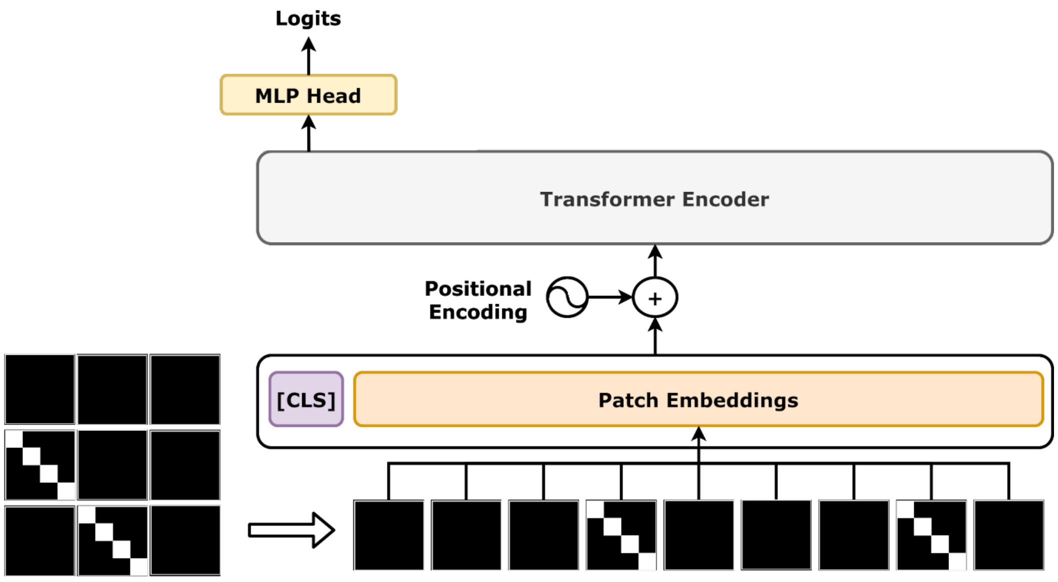


Figure 10.28 - The Vision Transformer (ViT)

In code, it looks like this:

Vision Transformer

```python
class ViT(nn.Module):
    def __init__(self, encoder, img_size,
                 in_channels, patch_size, n_outputs):
        super().__init__()
        self.d_model = encoder.d_model
        self.n_outputs = n_outputs
        self.encoder = encoder
        self.mlp = nn.Linear(encoder.d_model, n_outputs)

        self.embed = PatchEmbed(img_size, patch_size,
                                in_channels, encoder.d_model)
        self.cls_token = nn.Parameter(
            torch.zeros(1, 1, encoder.d_model)
        )

    def preprocess(self, X):
        # Patch embeddings
        # N, L, F -> N, L, D
        src = self.embed(X)
        # Special classifier token
        # 1, 1, D -> N, 1, D
        cls_tokens = self.cls_token.expand(X.size(0), -1, -1)
        # Concatenates CLS tokens -> N, 1 + L, D
        src = torch.cat((cls_tokens, src), dim=1)
        return src

    def encode(self, source):
        # Encoder generates "hidden states"
        states = self.encoder(source)
        # Gets state from first token: CLS
        cls_state = states[:, 0]  # N, 1, D
        return cls_state

    def forward(self, X):
        src = self.preprocess(X)
        # Featurizer
        cls_state = self.encode(src)
        # Classifier
        out = self.mlp(cls_state) # N, 1, outputs
        return out
```

It takes an instance of a **Transformer encoder**, and a series of image-related arguments (size, number of channels, and patch/kernel size), besides the desired **number of outputs** (logits) corresponding to the number of existing classes.

The `forward` method takes a mini-batch of images, pre-process them, encodes them (featurizer), and outputs logits (classifier). It is not *that* different from our typical image classifier from Chapter 5. It even uses **convolutions**!

For more details on the *original Vision Transformer*, make sure to check this amazing post[151] by Aman Arora and Dr. Habib Bukhari.

You can also check Phil Wang's implementation here[152].

Model Configuration & Training

Let's train our **Vision Transformer**! You know the drill:

Model Configuration

```python
torch.manual_seed(17)
layer = EncoderLayer(n_heads=2, d_model=16, ff_units=20)
encoder = EncoderTransf(layer, n_layers=1)
model_vit = ViT(encoder, img_size=12,
                in_channels=1, patch_size=4, n_outputs=3)
multi_loss_fn = nn.CrossEntropyLoss()
optimizer_vit = optim.Adam(model_vit.parameters(), lr=1e-3)
```

Model Training

```
sbs_vit = StepByStep(model_vit, multi_loss_fn, optimizer_vit)
sbs_vit.set_loaders(train_loader, val_loader)
sbs_vit.train(20)
```

```
fig = sbs_vit.plot_losses()
```

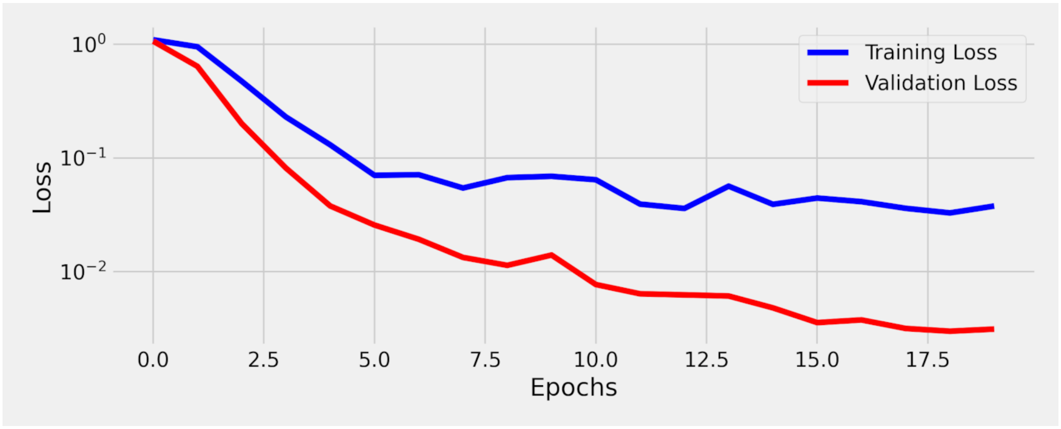


Figure 10.29 - Losses - Vision Transformer

Validation losses smaller than training losses - "*thank you*", dropout!

Once the model is trained, we can check the **embeddings** of our **special classifier token**:

```
model_vit.cls_token
```

Output

```
Parameter containing:
tensor([[[ 0.0557, -0.0345,  0.0126, -0.0300,  0.0335, -0.0422,
0.0479, -0.0248,  0.0128, -0.0605,  0.0061, -0.0178,  0.0921,
-0.0384, 0.0424, -0.0423]]], device='cuda:0', requires_grad=True)
```

Finally, let's see how **accurate** the Vision Transformer is:

```
StepByStep.loader_apply(sbs_vit.val_loader, sbs_vit.correct)
```

Output

```
tensor([[76, 76],
        [65, 65],
        [59, 59]])
```

Nailed it!

Putting It All Together

In this chapter, we used the same dataset of **colored squares** to, once again, **predict the coordinates** of the last two corners (**target sequence**) given the coordinates of the first two corners (**source sequence**). We built on top of our self-attention-based encoder-decoder architecture, turning the **former encoder and decoder** classes into **"layers"**, and wrapping up its internal operations with **"sub-layers"** to add **layer normalization**, **dropout**, and **residual connections** to each operation.

Data Preparation

The training set has the **full sequences** as **features**, while the test set has only the **source sequences** as **features**:

Data Preparation

```
# Training set
points, directions = generate_sequences()
full_train = torch.as_tensor(points).float()
target_train = full_train[:, 2:]
train_data = TensorDataset(full_train, target_train)
generator = torch.Generator()
train_loader = DataLoader(train_data, batch_size=16,
                          shuffle=True, generator=generator)
# Validation/Test Set
test_points, test_directions = generate_sequences(seed=19)
full_test = torch.as_tensor(points).float()
source_test = full_test[:, :2]
target_test = full_test[:, 2:]
test_data = TensorDataset(source_test, target_test)
test_loader = DataLoader(test_data, batch_size=16)
```

Model Assembly

Once again, we used the usual **bottom-up** approach to increasingly extend our encoder-decoder architecture. Now, we're revisiting the **Transformer** in a **top-down** approach, starting from the **encoder-decoder module (1)**. In its `encode` and `decode` methods, it makes a call to an instance of the **encoder (2)** followed by a call to an instance of the **decoder (3)**. We'll be representing the two called modules (2 and 3) as boxes inside the box corresponding to the caller module (1).

If we follow the complete sequence of calls, that's the resulting diagram for the **Transformer** architecture:

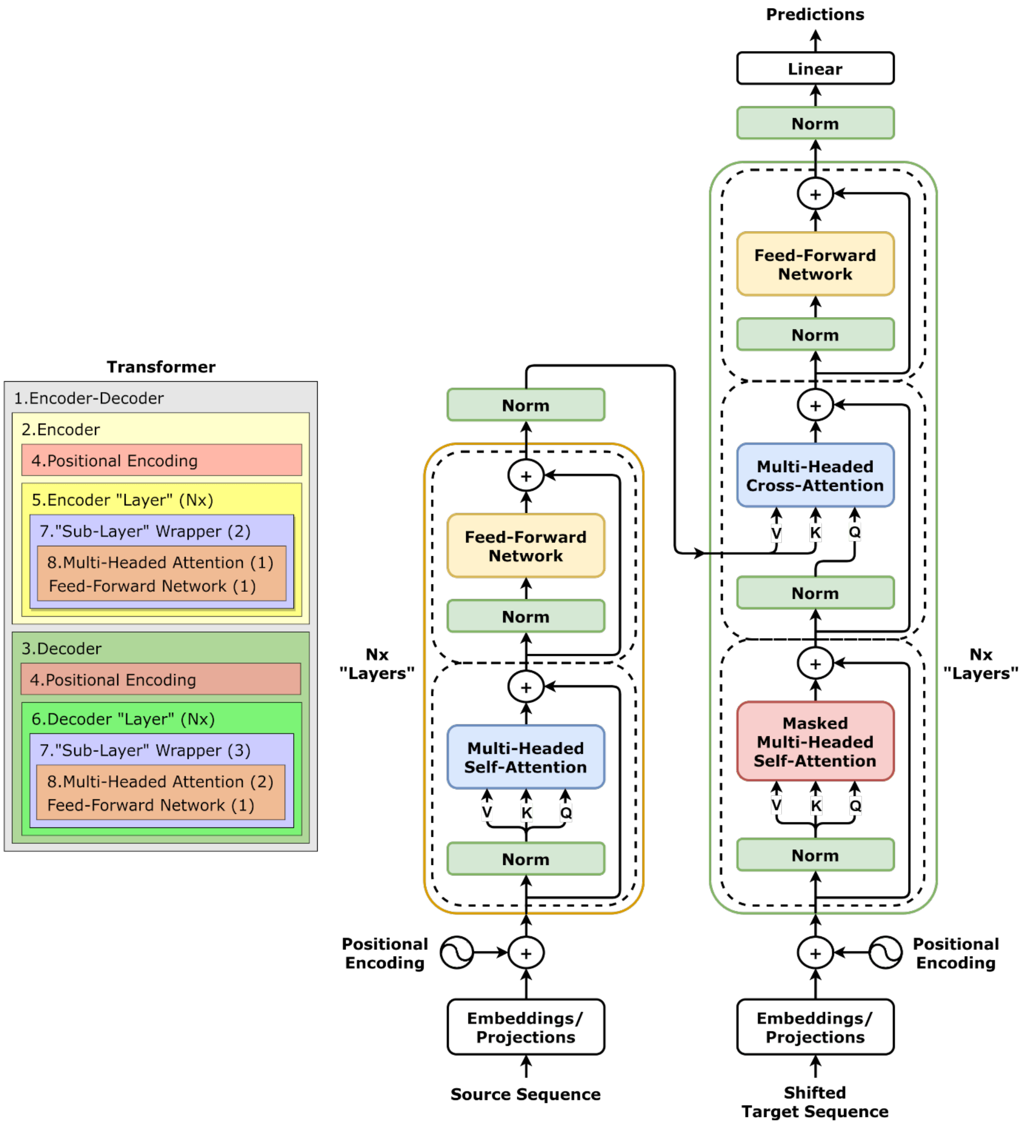


Figure 10.30 - The Transformer

Now, let's revisit the code of the represented modules. They're numbered accordingly, and so are the corresponding calls to them.

1. Encoder-Decoder

The encoder-decoder architecture was actually extended from the one developed in the previous chapter (`EncoderDecoderSelfAttn`), which handled **training** and **prediction** using **greedy decoding**. There are *no changes* here, except for the omission of both `encode` and `decode` methods, that are going to be overridden anyway:

Encoder + Decoder + Self-Attention

```python
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)

    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequent_mask = (
            1 - torch.triu(torch.ones(attn_shape), diagonal=1)
        ).bool()
        return subsequent_mask

    def encode(self, source_seq, source_mask):
        # Encodes the source sequence and uses the result
        # to initialize the decoder
        encoder_states = self.encoder(source_seq, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Decodes/generates a sequence using the shifted (masked)
        # target sequence - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs

    def predict(self, source_seq, source_mask):
        # Decodes/generates a sequence using one input
        # at a time - used in EVAL mode
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.decode(inputs,
                              source_mask,
                              self.trg_masks[:, :i+1, :i+1])
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = inputs[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        # Sends the mask to the same device as the inputs
        self.trg_masks = self.trg_masks.type_as(X).bool()
        # Slices the input to get source sequence
        source_seq = X[:, :self.input_len, :]
        # Encodes source sequence AND initializes decoder
        self.encode(source_seq, source_mask)
        if self.training:
            # Slices the input to get the shifted target seq
            shifted_target_seq = X[:, self.input_len-1:-1, :]
            # Decodes using the mask to prevent cheating
            outputs = self.decode(shifted_target_seq,
                                  source_mask,
                                  self.trg_masks)
        else:
            # Decodes using its own predictions
            outputs = self.predict(source_seq, source_mask)

        return outputs
```

This is the *actual* encoder-decoder Transformer, re-implementing both `encode` and `decode` methods to include the **input projections** and the **last linear layer** of the **decoder**. Notice the **numbered calls** to the **encoder (2)** and **decoder (3)**:

Transformer Encoder-Decoder

```python
class EncoderDecoderTransf(EncoderDecoderSelfAttn):
    def __init__(self, encoder, decoder,
                 input_len, target_len, n_features):
        super(EncoderDecoderTransf, self).__init__(
            encoder, decoder, input_len, target_len
        )
        self.n_features = n_features
        self.proj = nn.Linear(n_features, encoder.d_model)
        self.linear = nn.Linear(encoder.d_model, n_features)

    def encode(self, source_seq, source_mask=None):
        # Projection
        source_proj = self.proj(source_seq)
        encoder_states = self.encoder(source_proj, source_mask)①
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq,
               source_mask=None, target_mask=None):
        # Projection
        target_proj = self.proj(shifted_target_seq)
        outputs = self.decoder(target_proj,                 ②
                               source_mask=source_mask,
                               target_mask=target_mask)
        # Linear
        outputs = self.linear(outputs)
        return outputs
```

① Calls **Encoder** (2)

② Calls **Decoder** (3)

2. Encoder

The Transformer encoder has a **list** of stacked **encoder "layers" (3)** (remember that our "layers" are *norm-first*). It also adds **positional encoding (4)** to the inputs, and **normalizes** the outputs at the end:

Transformer Encoder

```
class EncoderTransf(nn.Module):
    def __init__(self, encoder_layer, n_layers=1, max_len=100):
        super().__init__()
        self.d_model = encoder_layer.d_model
        self.pe = PositionalEncoding(max_len, self.d_model)
        self.norm = nn.LayerNorm(self.d_model)
        self.layers = nn.ModuleList([copy.deepcopy(encoder_layer)
                                     for _ in range(n_layers)])

    def forward(self, query, mask=None):
        # Positional Encoding
        x = self.pe(query)        ①
        for layer in self.layers:
            x = layer(x, mask)    ②
        # Norm
        return self.norm(x)
```

① Calls **Positional Encoding** (4)

② Calls **Encoder "Layer"** multiple times (3)

3. Decoder

The Transformer decoder has a **list** of stacked **decoder "layers" (6)** (remember that our "layers" are *norm-first*). It also adds **positional encoding (4)** to the inputs, and **normalizes** the outputs at the end:

Transformer Decoder

```
class DecoderTransf(nn.Module):
    def __init__(self, decoder_layer, n_layers=1, max_len=100):
        super(DecoderTransf, self).__init__()
        self.d_model = decoder_layer.d_model
        self.pe = PositionalEncoding(max_len, self.d_model)
        self.norm = nn.LayerNorm(self.d_model)
        self.layers = nn.ModuleList([copy.deepcopy(decoder_layer)
                                     for _ in range(n_layers)])

    def init_keys(self, states):
        for layer in self.layers:
            layer.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        # Positional Encoding
        x = self.pe(query)                              ①
        for layer in self.layers:
            x = layer(x, source_mask, target_mask)  ②
        # Norm
        return self.norm(x)
```

① Calls **Positional Encoding** (4)

② Calls **Decoder "Layer"** multiple times (6)

4. Positional Encoding

We haven't changed the positional encoding module, it is here for the sake of completion only:

Positional Encoding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).float().unsqueeze(1)
        angular_speed = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-np.log(10000.0) / d_model)
        )
        # even dimensions
        pe[:, 0::2] = torch.sin(position * angular_speed)
        # odd dimensions
        pe[:, 1::2] = torch.cos(position * angular_speed)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x is N, L, D
        # pe is 1, maxlen, D
        scaled_x = x * np.sqrt(self.d_model)
        encoded = scaled_x + self.pe[:, :x.size(1), :]
        return encoded
```

5. Encoder "Layer"

The encoder "layer" implements a **list** of **two "sub-layers" (7)** which are going to be called with their corresponding operations (self-attention and feed-forward network):

Encoder "Layer"

```python
class EncoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.self_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ff_units, d_model),
        )
        self.sublayers = nn.ModuleList(
            [SubLayerWrapper(d_model, dropout) for _ in range(2)]
        )

    def forward(self, query, mask=None):
        # SubLayer 0 - Self-Attention
        att = self.sublayers[0](query,                           ①
                                sublayer=self.self_attn_heads,
                                is_self_attn=True,
                                mask=mask)
        # SubLayer 1 - FFN
        out = self.sublayers[1](att, sublayer=self.ffn)          ①
        return out
```

① Calls **"Sub-Layer" Wrapper** twice (self-attention and feed-forward network) (7)

"*Wait, I don't remember this* `SubLayerWrapper` *module…*"

Good catch! It is indeed **brand new**! We're defining it shortly (it's number seven, hang in there!). It was built to simplify the standard sequence of operations that was repeated several times: normalizing the inputs, calling the "sub-layer" itself, applying dropout, and adding the residual connection.

6. Decoder "Layer"

The decoder "layer" implements a **list** of **three"sub-layers" (7)** which are going to be called with their corresponding operations (masked self-attention, cross-attention, and feed-forward network):

Decoder "Layer"

```python
class DecoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.self_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.cross_attn_heads = \
            MultiHeadedAttention(n_heads, d_model, dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ff_units, d_model),
        )
        self.sublayers = nn.ModuleList(
            [SubLayerWrapper(d_model, dropout) for _ in range(3)]
        )

    def init_keys(self, states):
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        # SubLayer 0 - Masked Self-Attention
        att1 = self.sublayers[0](query,                          ①
                                   sublayer=self.self_attn_heads,
                                   is_self_attn=True,
                                   mask=target_mask)
        # SubLayer 1 - Cross-Attention
        att2 = self.sublayers[1](att1,                          ①
                                   sublayer=self.cross_attn_heads,
                                   mask=source_mask)
        # SubLayer 2 - FFN
        out = self.sublayers[2](att2, sublayer=self.ffn)        ①
        return out
```

① Calls **"Sub-Layer" Wrapper** three times (self-attention, cross-attention, and feed-forward network) (7)

7. "Sub-Layer" Wrapper

The **"sub-layer" wrapper** implements the **norm-first** approach to **wrapping "sub-layers"**:

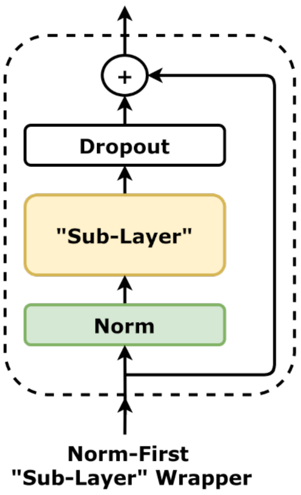


Figure 10.31 - "Sub-Layer" wrapper - norm-first

As stated above, it normalizes the inputs, calls the "sub-layer" itself (passed as argument), applies dropout, and adds the residual connection at the end:

Sub-Layer Wrapper

```python
class SubLayerWrapper(nn.Module):
    def __init__(self, d_model, dropout):
        super().__init__()
        self.norm = nn.LayerNorm(d_model)
        self.drop = nn.Dropout(dropout)

    def forward(self, x, sublayer, is_self_attn=False, **kwargs):
        norm_x = self.norm(x)
        if is_self_attn:
            sublayer.init_keys(norm_x)
        out = x + self.drop(sublayer(norm_x, **kwargs))        ①
        return out
```

① Calls **Multi-Headed Attention** (and the feed-forward network as well, depending on the `sublayer` argument) (8)

To make it more clear how this module was used to replace most of the code in the `forward` method of encoder and decoder "layers", here it is a *before-after* comparison of the encoder "layer"'s first "sub-layer" (self-attention):

```
# Before
def forward(self, query, mask=None):
    # query and mask go in
    norm_query = self.norm1(query)
    self.self_attn_heads.init_keys(norm_query)
    # the sublayer is the self-attention
    states = self.self_attn_heads(norm_query, mask)
    att = query + self.drop1(states)
    # att comes out
    ...

# After
def forward(self, query, mask=None):
    # query and mask go in
    # the sublayer is the self-attention
    # norm, drop, and residual are inside the wrapper
    att = self.sublayers[0](query,
                            sublayer=self.self_attn_heads,
                            is_self_attn=True,
                            mask=mask)
    # att comes out
    ...
```

8. Multi-Headed Attention

The **multi-headed attention** mechanism below replicates the implemented **narrow attention** described at the start of this chapter, **chunking the projections** of "keys" (K), "values" (V), and "queries" (Q) to make the size of the model more manageable:

Multi-Headed Attention

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = int(d_model / n_heads)
        self.linear_query = nn.Linear(d_model, d_model)
        self.linear_key = nn.Linear(d_model, d_model)
        self.linear_value = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(p=dropout)
        self.alphas = None

    def make_chunks(self, x):
        batch_size, seq_len = x.size(0), x.size(1)
        # N, L, D -> N, L, n_heads * d_k
        x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
        # N, n_heads, L, d_k
        x = x.transpose(1, 2)
        return x

    def init_keys(self, key):
        # N, n_heads, L, d_k
        self.proj_key = self.make_chunks(self.linear_key(key))
        self.proj_value = \
            self.make_chunks(self.linear_value(key))

    def score_function(self, query):
        # scaled dot product
        # N, n_heads, L, d_k x # N, n_heads, d_k, L
        # -> N, n_heads, L, L
        proj_query = self.make_chunks(self.linear_query(query))
        dot_products = torch.matmul(
            proj_query, self.proj_key.transpose(-2, -1)
        )
        scores =  dot_products / np.sqrt(self.d_k)
        return scores

    def attn(self, query, mask=None):
        # Query is batch-first: N, L, D
        # Score function will generate scores for each head
        scores = self.score_function(query) # N, n_heads, L, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, n_heads, L, L
        alphas = self.dropout(alphas)
        self.alphas = alphas.detach()

        # N, n_heads, L, L x N, n_heads, L, d_k
        # -> N, n_heads, L, d_k
        context = torch.matmul(alphas, self.proj_value)
        return context

    def output_function(self, contexts):
        # N, L, D
        out = self.linear_out(contexts) # N, L, D
        return out

    def forward(self, query, mask=None):
        if mask is not None:
            # N, 1, L, L - every head uses the same mask
            mask = mask.unsqueeze(1)

        # N, n_heads, L, d_k
        context = self.attn(query, mask=mask)
        # N, L, n_heads, d_k
        context = context.transpose(1, 2).contiguous()
        # N, L, n_heads * d_k = N, L, d_model
        context = context.view(query.size(0), -1, self.d_model)
        # N, L, d_model
        out = self.output_function(context)
        return out
```

Model Configuration & Training

Model Configuration

```
torch.manual_seed(42)
# Layers
enclayer = EncoderLayer(n_heads=3, d_model=6,
                        ff_units=10, dropout=0.1)
declayer = DecoderLayer(n_heads=3, d_model=6,
                        ff_units=10, dropout=0.1)
# Encoder and Decoder
enctransf = EncoderTransf(enclayer, n_layers=2)
dectransf = DecoderTransf(declayer, n_layers=2)
# Transformer
model_transf = EncoderDecoderTransf(enctransf,
                                    dectransf,
                                    input_len=2,
                                    target_len=2,
                                    n_features=2)
loss = nn.MSELoss()
optimizer = torch.optim.Adam(model_transf.parameters(), lr=0.01)
```

Weight Initialization

```
for p in model_transf.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

Model Configuration

```
sbs_seq_transf = StepByStep(model_transf, loss, optimizer)
sbs_seq_transf.set_loaders(train_loader, test_loader)
sbs_seq_transf.train(50)
```

```
sbs_seq_transf.losses[-1], sbs_seq_transf.val_losses[-1]
```

Output

```
(0.036566647700965405, 0.012972458032891154)
```

Recap

In this chapter, we've extended the **encoder-decoder architecture** and *transformed* it into a **Transformer** (the last pun of the chapter, I couldn't resist it!). First, we modified the **multi-headed attention** mechanism to use **narrow** attention. Then, we introduced **layer normalization** and the need to change the dimensionality of the inputs using either **projections** or **embeddings**. Next, we used our *former encoder and decoder* as **"layers"** that could be **stacked** together to form the *new Transformer encoder and decoder*. That made our model much deeper, thus raising the need for **wrapping** each "layer"'s internal operations (self-, cross-attention, and feed-forward network, now called **"sub-layers"**) with a combination of **layer normalization**, **dropout**, and **residual connection**. This is what we've covered:

- using **narrow attention** in the multi-headed attention mechanism
- **chunking the projections** of the inputs to implement narrow attention
- learning that chunking projections allow **different heads** to focus on, literally, **different dimensions** of the inputs
- **standardizing** individual **data points** using **layer normalization**
- using layer normalization to **standardize positionally-encoded inputs**
- changing the **dimensionality** of the inputs using **projections (embeddings)**
- defining an **encoder "layer"** that uses **two "sub-layers"**, a **self-attention** mechanism, and a **feed-forward** network
- **stacking** encoder "layers" to build a **Transformer encoder**
- **wrapping "sub-layer" operations** with a combination of **layer normalization**, **dropout**, and **residual connection**
- learning the difference between **norm-last** and **norm-first "sub-layers"**

- understanding that **norm-first "sub-layers"** allow the inputs to **flow unimpeded** all the way to the top through the **residual connections**
- defining a **decoder "layer"** that uses **three "sub-layers"**, a **masked self-attention** mechanism, a **cross-attention** mechanism, and a **feed-forward** network
- **stacking** decoder "layers" to build a **Transformer decoder**
- combining both **encoder and decoder** into a full-blown, norm-first, **Transformer** architecture
- **training the Transformer** to tackle our sequence-to-sequence problem
- understanding that the **validation loss** may be much **smaller** than the **training loss** due to regularizing effect of **dropout**
- training another model using **PyTorch's (norm-last) Transformer** class
- using the **Vision Transformer** architecture to tackle an **image classification problem**
- splitting an **image** into flattened **patches** either by **rearranging or embedding** them
- adding a **special classifier token** to the embeddings
- using the **encoder's output** corresponding to the special classifier token as **features** for the classifier

Congratulations! You've just assembled and trained your first **Transformers** (and even a cutting-edge **Vision Transformer**!): this is *no small feat*. Now you know what "layers" and "sub-layers" stand for and how they're brought up together to build a Transformer. Keep in mind, though, that you may find *slightly different* implementations around. It may be either **norm-first** or **norm-last** or maybe yet another customization. The details may be different, but the overall concept remains: it is all about **stacking attention-based "layers"**.

"Hey, what about BERT? Shouldn't we use Transformers to tackle NLP problems?"

I was actually *waiting* for this question: **yes**, we should, and we will, in the next chapter. As you have seen, it is already *hard enough* to understand the Transformer even when it's used to tackle such a simple sequence-to-sequence problem as ours. Trying to train a model to handle a more complex Natural Language Processing problem would only make it *even harder*.

In the next chapter, we'll start with *some* **NLP concepts and techniques** like tokens, tokenization, word embeddings, and language models, and work our way up to **contextual word embeddings**, **GPT-2**, and **BERT**. We'll be using several Python packages, including the famous **HuggingFace** :-)

[143] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter10.ipynb

[144] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter10.ipynb

[145] https://arxiv.org/abs/1906.04341

[146] https://arxiv.org/abs/1706.03762

[147] http://nlp.seas.harvard.edu/2018/04/03/attention

[148] https://arxiv.org/abs/1607.06450

[149] https://arxiv.org/abs/2010.11929

[150] https://github.com/arogozhnikov/einops

[151] https://amaarora.github.io/2021/01/18/ViT.html

[152] https://github.com/lucidrains/vit-pytorch

Part IV
Natural Language Processing

Chapter 11

Down the Yellow Brick Rabbit Hole

Spoilers

In this chapter, we will:

- learn about many useful packages for NLP: `nltk`, `gensim`, `flair`, and HuggingFace
- build our own dataset from scratch using **HuggingFace's `Dataset`**
- use different **tokenizers** on our dataset
- learn and load **word embeddings** using **Word2Vec** and **GloVe**
- train many models using **embeddings** in different ways
- use **ELMo** and **BERT** to retrieve **contextual word embeddings**
- use **HuggingFace's `Trainer`** to fine-tune BERT
- fine-tune **GPT-2** and use it in a **pipeline** to **generate text**

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 11[153] is part of the official **Deep Learning with PyTorch Step-by-Step** repository on GitHub. You can also run it directly in **Google Colab**[154].

If you're using a *local Installation*, open your terminal or Anaconda Prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 11's notebook. If not, just click on `Chapter11.ipynb` in your Jupyter's Home Page.

Additional Setup

This is a *special chapter* when it comes to its setup: we won't be using *only* PyTorch anymore but a handful of other packages as well, including the *de facto* standard for NLP tasks - HuggingFace.

Before proceeding, make sure you have all of them installed by running the commands below:

```
!pip install gensim==3.8.3
!pip install allennlp==0.9.0
!pip install flair==0.8.0.post1 # uses PyTorch 1.7.1
# HuggingFace
!pip install transformers==4.5.1
!pip install datasets==1.6.0
```

Some packages, like `flair`, may have *strict dependencies* and eventually require the **downgrading** of some other packages in your environment, even PyTorch itself.

The versions above were used to generate the outputs presented in this chapter, but you can use newer versions if you want (except for the `allennlp` package since this specific version is required by `flair` for retrieving ELMo embeddings).

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import os
import json
import errno
import requests
import numpy as np
from copy import deepcopy
from operator import itemgetter

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, random_split, \
    TensorDataset

from data_generation.nlp import ALICE_URL, WIZARD_URL, download_text
from stepbystep.v4 import StepByStep
# These are the classes we built in Chapters 9 and 10
from seq2seq import *

import nltk
from nltk.tokenize import sent_tokenize

import gensim
from gensim import corpora, downloader
from gensim.parsing.preprocessing import *
from gensim.utils import simple_preprocess
from gensim.models import Word2Vec

from flair.data import Sentence
from flair.embeddings import ELMoEmbeddings, WordEmbeddings, \
    TransformerWordEmbeddings, TransformerDocumentEmbeddings

from datasets import load_dataset, Split
from transformers import (
    DataCollatorForLanguageModeling,
    BertModel, BertTokenizer, BertForSequenceClassification,
    DistilBertModel, DistilBertTokenizer,
    DistilBertForSequenceClassification,
    AutoModelForSequenceClassification,
    AutoModel, AutoTokenizer, AutoModelForCausalLM,
    Trainer, TrainingArguments, pipeline, TextClassificationPipeline
)
from transformers.pipelines import SUPPORTED_TASKS
```

"Down the Yellow Brick Rabbit Hole"

Where does the sentence in the title come from? On the one hand, if it were "*down the rabbit hole*", one could guess "*Alice's Adventures in Wonderland*". On the other hand, if it were "*the yellow brick road*", one could guess "*The Wonderful Wizard of Oz*". But it is neither (or maybe it is **both**?). What if, instead of trying to guess it ourselves, we **train a model** to **classify sentences**? That's a book about deep learning, after all :-)

Training models on text data *is* what **Natural Language Processing (NLP)** is all about. The whole field is enormous, and we'll be only *scratching the surface* of it in this chapter. We'll start with the most obvious question, "*how to convert text data into numerical data*", and end up using a **pretrained model** - our famous muppet friend, **BERT** - to **classify sentences**.

Building a Dataset

There are *many* freely available datasets for NLP. The texts are usually already nicely organized into **sentences** that you can easily feed to a pretrained model like BERT. Isn't it awesome? Well, yeah, but...

"But what?"

But the texts you'll find in the real world are *not* nicely organized into sentences. **You** have to organize them yourself.

So, we'll start our NLP journey by following the steps of Alice and Dorothy, from *"Alice's Adventures in Wonderland"*[155] by Lewis Carroll and *"The Wonderful Wizard of Oz"*[156] by L. Frank Baum.

Both texts are freely available at the Oxford Text Archive (OTA)[157] under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) license.

Figure 11.1 - Left: "Alice and the Baby Pig" illustration by John Tenniel, from "Alice's Adventures in Wonderland" (1865). Right: "Dorothy meets the Cowardly Lion" illustration by W. W. Denslow, from "The Wonderful Wizard of Oz" (1900).

The direct links to both texts are alice28-1476.txt[158] (we're naming it `ALICE_URL`) and wizoz10-1740.txt[159] (we're naming it `WIZARD_URL`). You can download both of them to a local folder using the helper method `download_text` (included in `data_generation.nlp`):

Data Loading

```
localfolder = 'texts'
download_text(ALICE_URL, localfolder)
download_text(WIZARD_URL, localfolder)
```

If you open these files in a text editor, you'll see that there is a *lot* of information at the beginning (and some at the end) that has been added to the original text of the book for legal reasons. We need to remove these additions to the original texts:

Downloading books

```
fname1 = os.path.join(localfolder, 'alice28-1476.txt')
with open(fname1, 'r') as f:
    alice = ''.join(f.readlines()[104:3704])
fname2 = os.path.join(localfolder, 'wizoz10-1740.txt')
with open(fname2, 'r') as f:
    wizard = ''.join(f.readlines()[310:5100])
```

The actual texts of the books are contained between lines 105 and 3703 (remember Python's zero-based indexing) and 309 and 5099, respectively. Moreover, we're **joining all the lines** together into a single, large, string of text for each book because we're going to organize the resulting texts into **sentences** and, in a regular book, there are line breaks mid-sentence all over.

We definitely *do not* want to do that manually every time, right? Although it would be more difficult to automatically remove any additions to the original text, we *can* at least automate the removal of the extra lines by setting the **real start and end** lines of each text in a configuration file (`lines.cfg`):

Configuration file

```
text_cfg = """fname,start,end
alice28-1476.txt,104,3704
wizoz10-1740.txt,310,5100"""
bytes_written = open(
    os.path.join(localfolder, 'lines.cfg'), 'w'
).write(text_cfg)
```

Your local folder (`texts`) should have *three files* now: `alice28-1476.txt`, `lines.cfg`, and `wizoz10-1740.txt`. Now, it is time to perform…

Sentence Tokenization

A **token** is a **piece of a text**, and to **tokenize** a text means to **split it into pieces**, that is, into a **list of tokens**.

"*What kind of pieces are we talking about here?*"

The most common kind of piece is a **word**. So, **tokenizing a text** usually means to **split it into words** using the **white space** as a separator:

```
sentence = "I'm following the white rabbit"
tokens = sentence.split(' ')
tokens
```

Output

```
["I'm", 'following', 'the', 'white', 'rabbit']
```

"*What about "I'm"? Isn't it **two words**?*"

Yes, and no. Not helpful, right? As usual, it depends… word contractions like that are fairly common, and maybe you *want* to keep them as single tokens. But it is also possible to have the token itself split into its two basic components, "*I*" and "*am*", such that the sentence above has *six tokens* instead of five. For now, we're only interested in **sentence tokenization** which, as you probably already guessed, means to **split a text into its sentences**.

We'll get back to the topic of **tokenization** at **word** (and **subword**) levels later.

For a *brief* introduction to the topic, check the Tokenization[160] section of the "*Introduction to Information Retrieval*"[161] book by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, Cambridge University Press (2008).

We're using NLTK's `sent_tokenize` to accomplish that instead of trying to devise the splitting rules ourselves (NLTK is the Natural Language Toolkit library and it is one of the most traditional tools for handling NLP tasks):

```
import nltk
from nltk.tokenize import sent_tokenize
nltk.download('punkt')
corpus_alice = sent_tokenize(alice)
corpus_wizard = sent_tokenize(wizard)
len(corpus_alice), len(corpus_wizard)
```

Output

```
(1612, 2240)
```

There are 1,612 sentences in "*Alice's Adventures in Wonderland*" and 2,240 sentences in "*The Wonderful Wizard of Oz*".

"*What is this* `punkt`?"

That's the **Punkt Sentence Tokenizer**, and its pretrained model (for the English language) is included in the NLTK package.

"*And what is a* ***corpus***?"

A **corpus** is a structured **set of documents**. But there is quite a lot of wiggle room in this definition: one *can* define a *document* as a **sentence**, a **paragraph**, or even a **whole book**. In our case, the document is a **sentence**, so each **book is actually a set of sentences**, and thus each **book** may be considered a **corpus**. The *plural of corpus* is actually **corpora** (yay Latin!), so we do have a corpora.

Let's check one sentence from the first corpus of text:

```
corpus_alice[2]
```

Output

```
'There was nothing so VERY remarkable in that; nor did Alice\nthink
it so VERY much out of the way to hear the Rabbit say to\nitself,
`Oh dear!'
```

Notice that it **still** includes the **line breaks (\n)** from the original text. The sentence tokenizer *only* handles the sentence splitting, it does *not* clean up the line breaks.

Let's check one sentence from the second corpus of text:

```
corpus_wizard[30]
```

Output

```
'"There\'s a cyclone coming, Em," he called to his wife.'
```

No line breaks here, but notice the **quotation marks (")** in the text.

"Why do we care about line breaks and quotation marks anyway?"

Our dataset is going to be a **collection of CSV files**, one file for each book, each CSV file containing **one sentence per line**.

Therefore, we need to:

- **clean the line breaks** to make sure each sentence is in one line only
- **define** an appropriate **quote char** to "wrap" the sentence such that the original *commas* and *semicolons* in the original text do not get misinterpreted as *separation chars* of the CSV file
- add a **second column** to the CSV file (the first one is the sentence itself) to identify the original *source of the sentence* since we'll be **concatenating** and **shuffling** the sentences before training a model on our corpora

The sentence above should end up looking like this:

```
\"There's a cyclone coming, Em," he called to his wife.\,wizoz10
-1740.txt
```

The escape character "\" is a good choice for *quote char* because it is *not present* in any of the books (we would probably have to choose something else if our books of choice were about coding).

The function below does the grunt work of cleaning, splitting, and saving the sentences to a CSV file for us:

Method to generate CSV of sentences

```python
def sentence_tokenize(source, quote_char='\\', sep_char=',',
                      include_header=True, include_source=True,
                      extensions=('txt'), **kwargs):
    nltk.download('punkt')
    # If source is a folder, goes through all files inside it
    # that match the desired extensions ('txt' by default)
    if os.path.isdir(source):
        filenames = [f for f in os.listdir(source)
                     if os.path.isfile(os.path.join(source, f))
                     and os.path.splitext(f)[1][1:] in
extensions]
    elif isinstance(source, str):
        filenames = [source]

    # If there is a configuration file, builds a dictionary with
    # the corresponding start and end lines of each text file
    config_file = os.path.join(source, 'lines.cfg')
    config = {}
    if os.path.exists(config_file):
        with open(config_file, 'r') as f:
            rows = f.readlines()

        for r in rows[1:]:
            fname, start, end = r.strip().split(',')
            config.update({fname: (int(start), int(end))})

    new_fnames = []
    # For each file of text
    for fname in filenames:
        # If there's a start and end line for that file, use it
        try:
            start, end = config[fname]
        except KeyError:
            start = None
            end = None

        # Opens the file, slices the configures lines (if any)
        # cleans line breaks and uses the sentence tokenizer
        with open(os.path.join(source, fname), 'r') as f:
            contents = (
                ''.join(f.readlines()[slice(start, end, None)])
                .replace('\n', ' ').replace('\r', '')
            )
        corpus = sent_tokenize(contents, **kwargs)

        # Builds a CSV file containing tokenized sentences
        base = os.path.splitext(fname)[0]
        new_fname = f'{base}.sent.csv'
        new_fname = os.path.join(source, new_fname)
        with open(new_fname, 'w') as f:
            # Header of the file
            if include_header:
                if include_source:
                    f.write('sentence,source\n')
                else:
                    f.write('sentence\n')
            # Writes one line for each sentence
            for sentence in corpus:
                if include_source:
                    f.write(f'{quote_char}{sentence}{quote_char}\
                              {sep_char}{fname}\n')
                else:
                    f.write(f'{quote_char}{sentence}\
                              {quote_char}\n')
        new_fnames.append(new_fname)

    # Returns list of the newly generated CSV files
    return sorted(new_fnames)
```

It takes a `source` **folder** (or a single file) and goes through the files with the right `extensions` (only `.txt` by default), removing lines based on the `lines.cfg` file (if any), applying the **sentence tokenizer** to each file, and generating the corresponding CSV file of sentences using the configured `quote_char` and `sep_char`. It may also `include_header` and `include_source` in the CSV file.

The CSV files are named after the corresponding text files by dropping the original extension and appending `.sent.csv` to it. Let's see it in action:

Generating dataset of sentences

```
new_fnames = sentence_tokenize(localfolder)
new_fnames
```

Output

```
['texts/alice28-1476.sent.csv', 'texts/wizoz10-1740.sent.csv']
```

Each **CSV file** contains the **sentences of a book**, and we'll use both of them to build our own **dataset**.

Sentence Tokenization in spaCy

By the way, NLTK is *not* the only option for sentence tokenization: it is also possible to use spaCy's sentencizer for this task. The snippet below shows an example of a spaCy pipeline:

```
# conda install -c conda-forge spacy
# python -m spacy download en_core_web_sm
import spacy
nlp = spacy.blank("en")
nlp.add_pipe(nlp.create_pipe("sentencizer"))
sentences = []
for doc in nlp.pipe(corpus_alice):
    sentences.extend(sent.text for sent in doc.sents)
len(sentences), sentences[2]
```

Output

```
(1615, 'There was nothing so VERY remarkable in that; nor did
Alice\nthink it so VERY much out of the way to hear the Rabbit
say to\nitself, `Oh dear!')
```

Since spaCy uses a different model for tokenizing sentences, it is no surprise that it found a *slightly different* number of sentences in the text.

HuggingFace's Dataset

We'll be using HuggingFace's datasets instead of regular PyTorch ones.

"Why?"

First, we'll be using HuggingFace's **pretrained models** (like BERT) later on, so it is only logical to go for their implementation of datasets as well. Second, there are many **datasets already available** in their library, so it makes sense to get used to handling text data using their implementation of datasets.

Even though we're using HuggingFace's `Dataset` class to build our own dataset, we're only using a fraction of its capabilities. For a more detailed view of what it has to offer, make sure to check its extensive documentation:

- Quick Tour[162]
- What's in the Dataset object[163]
- Loading a Dataset[164]

And, for a complete list of every dataset available, check the HuggingFace Hub[165].

Loading a Dataset

We can use HF's (I will abbreviate HuggingFace as HF from now on) `load_dataset` to load from local files:

Data Preparation

```
from datasets import load_dataset, Split
dataset = load_dataset(path='csv',
                       data_files=new_fnames,
                       quotechar='\\',
                       split=Split.TRAIN)
```

The name of the first argument (`path`) may be a bit misleading… it is actually the **path** to the **dataset processing script**, not the actual files. To load CSV files, we can simply use HF's `csv` as in the example above. The list of *actual files* containing the text (sentences, in our case) must be provided in the `data_files` argument. The `split` argument is used to designate which *split* the dataset represents (`Split.TRAIN`, `Split.VALIDATION`, or `Split.TEST`).

Moreover, the CSV script offers more options to control parsing and reading of the CSV files, like `quotechar`, `delimiter`, `column_names`, `skip_rows`, and `quoting`. For more details, please check the documentation on loading CSV files.

It is also possible to load data from JSON files, text files, Python dictionaries, and Pandas dataframes.

Attributes

The `Dataset` has many attributes, like `features`, `num_columns`, and `shape`:

```
dataset.features, dataset.num_columns, dataset.shape
```

Output

```
({'sentence': Value(dtype='string', id=None),
  'source': Value(dtype='string', id=None)},
 2,
 (3852, 2))
```

Our dataset has two columns, `sentence` and `source`, and there are 3,852 sentences in it.

It can be indexed like a **list**:

```
dataset[2]
```

Output

```
{'sentence': 'There was nothing so VERY remarkable in that; nor did
Alice think it so VERY much out of the way to hear the Rabbit say to
itself, `Oh dear!',
 'source': 'alice28-1476.txt'}
```

That's the third sentence in our dataset, and it is from "*Alice's Adventures in Wonderland*".

And its columns can be accessed as a **dictionary** too:

```
dataset['source'][:3]
```

Output

```
['alice28-1476.txt', 'alice28-1476.txt', 'alice28-1476.txt']
```

The first few sentences all come from "*Alice's Adventures in Wonderland*" because we haven't *shuffled* the dataset yet.

Methods

The `Dataset` also has many methods, like `unique`, `map`, `filter`, `shuffle`, and `train_test_split` (for a comprehensive list of operations, check HF's *"Processing data in a Dataset"*[166]).

We can easily check the unique sources:

```
dataset.unique('source')
```

Output

```
['alice28-1476.txt', 'wizoz10-1740.txt']
```

We can use `map` to **create new columns** by using a **function** that **returns a dictionary** with the **new column as key**:

Data Preparation

```
def is_alice_label(row):
    is_alice = int(row['source'] == 'alice28-1476.txt')
    return {'labels': is_alice}

dataset = dataset.map(is_alice_label)
```

Each element in the dataset is a `row` corresponding to a dictionary (`{'sentence': ..., 'source': ...}`, in our case), so the function has access to all columns in a given row. Our `is_alice_label` function tests the `source` column and **creates a `labels` columns**. There is **no need** to return the original columns since it is automatically handled by the dataset.

If we retrieve the third sentence from our dataset once again, the new column will already be there:

```
dataset[2]
```

Output

```
{'labels': 1,
 'sentence': 'There was nothing so VERY remarkable in that; nor did
Alice think it so VERY much out of the way to hear the Rabbit say to
itself, `Oh dear!',
 'source': 'alice28-1476.txt'}
```

Now that the labels are in place, we can finally **shuffle the dataset** and **split** it into training and test sets:

Data Preparation

```
shuffled_dataset = dataset.shuffle(seed=42)
split_dataset = shuffled_dataset.train_test_split(test_size=0.2)
split_dataset
```

Output

```
DatasetDict({
    train: Dataset({
        features: ['sentence', 'source'],
        num_rows: 3081
    })
    test: Dataset({
        features: ['sentence', 'source'],
        num_rows: 771
    })
})
```

The *splits* are actually a **dataset dictionary**, so you may want to retrieve the actual datasets from it:

Data Preparation

```
train_dataset = split_dataset['train']
test_dataset = split_dataset['test']
```

Done! We have two - training and test- randomly shuffled datasets.

Word Tokenization

The naïve word tokenization, as we've already seen, simply **splits a sentence into words** using the **white space** as a separator:

```
sentence = "I'm following the white rabbit"
tokens = sentence.split(' ')
tokens
```

Output

```
["I'm", 'following', 'the', 'white', 'rabbit']
```

But, as we've *also* seen, there are issues with the naïve approach (how to handle contractions, for example). Let's try using Gensim[167], a popular library for topic modeling, which offers some out-of-the-box tools for performing word tokenization:

```
from gensim.parsing.preprocessing import *
preprocess_string(sentence)
```

Outpu

```
['follow', 'white', 'rabbit']
```

"*That doesn't look right… some words are simply gone!*"

Welcome to the world of tokenization :-) It turns out, Gensim's `preprocess_sring` applies **many filters** by default, namely:

- `strip_tags` (for removing HTML-like tags between brackets)
- `strip_punctuation`
- `strip_multiple_whitespaces`
- `strip_numeric`

The filters above are pretty straightforward, and they are used to remove typical elements from the text. But `preprocess_string` *also* includes the following filters:

- `strip_short`: it **discards** any word **less than three characters** long
- `remove_stopwords`: it **discards** any word that is considered a **stopword** (like "*the*", "*but*", "*then*", and so on)
- `stem_text`: it **modifies** words by **stemming** them, that is, reducing them to a **common base form** (from "*following*" to its base "*follow*", for example)

For a brief introduction to **stemming** (and the related **lemmatization**) procedures, please check the Stemming and lemmatization[168] section of the "*Introduction to Information Retrieval*"[169] book by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, Cambridge University Press (2008).

We **won't** be removing stopwords or performing stemming here. Since our goal is to use HF's pretrained BERT model, we'll also use its corresponding **pretrained tokenizer**.

So, let's use the *first four filters only* (and make everything lower case too):

```
filters = [lambda x: x.lower(),
           strip_tags,
           strip_punctuation,
           strip_multiple_whitespaces, strip_numeric]
preprocess_string(sentence, filters=filters)
```

Output

```
['i', 'm', 'following', 'the', 'white', 'rabbit']
```

Another option is to use Gensim's `simple_preprocess`, which converts the text into a list of lowercase tokens, discarding tokens that are either too short (less than three characters) or too long (more than fifteen characters):

```
from gensim.utils import simple_preprocess
tokens = simple_preprocess(sentence)
tokens
```

Output

```
['following', 'the', 'white', 'rabbit']
```

"*Why are we using Gensim? Can't we use NLTK to perform word tokenization?*"

Fair enough. NLTK *can* be used to tokenize words as well, but Gensim *cannot* be used to tokenize sentences. Besides, since Gensim has many *other* interesting tools for building vocabularies, bag-of-words (BoW) models, and *Word2Vec* models (we'll get to that soon), it makes sense to introduce it as soon as possible.

Data Augmentation

Let's briefly address the topic of **augmentation** for text data. Although we're not actually including it in our pipeline here, it's worth knowing about some possibilities and techniques regarding data augmentation.

The most basic technique is called **word dropout** and, as you probably already guessed, it simply **randomly replaces words** by some other random word or a special `[UNK]` token (word) that indicates a non-existing word.

It is also possible to replace words with their **synonyms**, so the meaning of the text is preserved. One can use WordNet[170], a lexical database for the English language, to look up synonyms. Finding synonyms is not so easy, and this approach is limited to the English language, though.

To circumvent the limitations of the synonyms approach, it is also possible to replace words with **similar words**, numerically speaking. We haven't yet talked about **word embeddings** - numerical representations of words - but they can be used to identify words that **may** have a similar meaning. For now, it suffices to say that there are packages that perform data augmentation on text data using embeddings, like TextAttack[171].

Let's try augmenting Richard P. Feynman as an example:

```
# !pip install textattack
from textattack.augmentation import EmbeddingAugmenter
augmenter = EmbeddingAugmenter()
feynman = 'What I cannot create, I do not understand.'
```

```
for i in range(5):
    print(augmenter.augment(feynman))
```

Output

```
['What I cannot create, I do not fathom.']
['What I cannot create, I do not understood.']
['What I notable create, I do not understand.']
['What I cannot creating, I do not understand.']
['What I significant create, I do not understand.']
```

Some are OK, some are changing the tense, some are simply weird. No one said data augmentation was easy, right?

Vocabulary

The **vocabulary** is a **list of unique words** that appear in text corpora.

To build our own vocabulary, we need to tokenize our training set first:

```
sentences = train_dataset['sentence']
tokens = [simple_preprocess(sent) for sent in sentences]
tokens[0]
```

Output

```
['and', 'so', 'far', 'as', 'they', 'knew', 'they', 'were', 'quite',
'right']
```

The `tokens` variable is a **list of lists of words**, each (inner) list containing all the words (tokens) in a sentence. These tokens can then be used to build a **vocabulary** using Gensim's `corpora.Dictionary`:

```
from gensim import corpora
dictionary = corpora.Dictionary(tokens)
print(dictionary)
```

Output

```
Dictionary(3704 unique tokens: ['and', 'as', 'far', 'knew', 'quite'
]...)
```

The corpora's dictionary **is not** a typical Python dictionary. It has some specific (and useful) attributes:

```
dictionary.num_docs
```

Output

```
3081
```

The `num_docs` attribute tells us how many *documents* were processed (sentences, in our case), and it corresponds to the length of the (outer) list of tokens.

```
dictionary.num_pos
```

Output

```
50802
```

The `num_pos` attribute tells us how many *tokens* (words) were processed over all documents (sentences).

```
dictionary.token2id
```

Output

```
{'and': 0,
 'as': 1,
 'far': 2,
 'knew': 3,
 'quite': 4,
 ...
```

The `token2id` attribute is a (Python) dictionary containing **the unique words** found in the text corpora, and a **unique id sequentially assigned** to the words.

The **keys** of the `token2id` dictionary are the actual **vocabulary** of our corpora:

```
vocab = list(dictionary.token2id.keys())
vocab[:5]
```

Output

```
['and', 'as', 'far', 'knew', 'quite']
```

The `cfs` attribute stands for **collection frequencies** and it tells us **how many times** a given token appears on the text corpora:

```
dictionary.cfs
```

Output

```
{0: 2024,
 6: 362,
 2: 29,
 1: 473,
 7: 443,
 ...
```

The token corresponding to the **id zero** ("*and*") appeared 2,024 times across all sentences. But, in **how many distinct documents** (sentences) did it appear? That's what the `dfs` attribute, which stands for **document frequencies**, tells us:

```
dictionary.dfs
```

Output

```
{0: 1306,
 6: 351,
 2: 27,
 1: 338,
 7: 342,
 ...
```

The token corresponding to the **id zero** ("*and*") appeared in 1,306 sentences.

Finally, if we want to **convert a list of tokens** into a **list of their corresponding indices in the vocabulary**, we can use the `doc2idx` method:

```
sentence = 'follow the white rabbit'
new_tokens = simple_preprocess(sentence)
ids = dictionary.doc2idx(new_tokens)
print(new_tokens)
print(ids)
```

Output

```
['follow', 'the', 'white', 'rabbit']
[1482, 20, 497, 333]
```

The problem is, however *large* we make the vocabulary, **there will always be a new word that's not in there**.

"*What do we do with words that **aren't** in the vocabulary?*"

If the word **isn't** in the vocabulary, it is an **unknown** word, and it is going to be replaced by the corresponding **special token**: `[UNK]`. This means we need to **add `[UNK]` to the vocabulary**. Luckily, Gensim's Dictionary has a `patch_with_special_tokens` method that makes it very easy to *patch* our vocabulary:

```
special_tokens = {'[PAD]': 0, '[UNK]': 1}
dictionary.patch_with_special_tokens(special_tokens)
```

Besides, since we're at it, let's add **yet another special token: `[PAD]`**. At some point, we'll have to *pad* our sequences (like we did in Chapter 8), so it will be useful to have a token ready for it.

What if, instead of adding more tokens to the vocabulary, we try **removing words** from it? Maybe we'd like to remove *rare words* (**aardvark** always comes to my mind…) to get a smaller vocabulary, or maybe we'd like to remove *bad words* (profanity) from it.

Gensim's dictionary has a couple of methods that we can use for it:

- `filter_extremes`: keeps the first **`keep_n`** most frequent words only (it is also possible to **keep** words that appear in **at least `no_below` documents** or to **remove** words that appear in **more than `no_above` fraction of documents**)
- `filter_tokens`: **removes tokens** from a list of **`bad_ids`** (`doc2idx` can be used to get a list of the corresponding ids of the bad words) or **keeps** only the tokens from a list of **`good_ids`**

"What if I want to remove words that appear less than X times in all documents?"

That's not directly supported by Gensim's Dictionary, but we can use its `cfs` attribute to find those tokens with low frequency, and then filter them out using `filter_tokens`:

Method to find rare tokens

```python
def get_rare_ids(dictionary, min_freq):
    rare_ids = [t[0] for t in dictionary.cfs.items()
                if t[1] < min_freq]
    return rare_ids
```

Once we're happy with the size and scope of a vocabulary, we can **save it to disk** as a plain text file, one token (word) per line. The helper function below takes a **list of sentences**, generates the corresponding **vocabulary**, and saves it to a file named `vocab.txt`:

Method to build a vocabulary from a dataset of sentences

```python
def make_vocab(sentences, folder=None, special_tokens=None,
               vocab_size=None, min_freq=None):
    if folder is not None:
        if not os.path.exists(folder):
            os.mkdir(folder)

    # tokenizes the sentences and create a Dictionary
    tokens = [simple_preprocess(sent) for sent in sentences]
    dictionary = corpora.Dictionary(tokens)
    # keeps only the most frequent words (vocab size)
    if vocab_size is not None:
        dictionary.filter_extremes(keep_n=vocab_size)
    # removes rare words (in case the vocab size still
    # includes words with low frequency)
    if min_freq is not None:
        rare_tokens = get_rare_ids(dictionary, min_freq)
        dictionary.filter_tokens(bad_ids=rare_tokens)
    # gets the whole list of tokens and frequencies
    items = dictionary.cfs.items()
    # sorts the tokens in descending order
    words = [dictionary[t[0]]
             for t in sorted(dictionary.cfs.items(),
                             key=lambda t: -t[1])]
    # prepends special tokens, if any
    if special_tokens is not None:
        to_add = []
        for special_token in special_tokens:
            if special_token not in words:
                to_add.append(special_token)
        words = to_add + words

    with open(os.path.join(folder, 'vocab.txt'), 'w') as f:
        for word in words:
            f.write(f'{word}\n')
```

We can take the sentences from our training set, add special tokens to the vocabulary, filter out any words appearing only once, and save the vocabulary file to the `our_vocab` folder:

```
make_vocab(train_dataset['sentence'],
           'our_vocab/',
           special_tokens=['[PAD]', '[UNK]'],
           min_freq=2)
```

And now we can use this vocabulary file with a **tokenizer**.

*"But I thought we were **already** using tokenizers… aren't we?"*

Yes, we are. First, we used a sentence tokenizer to split the texts into sentences. Then, we used a word tokenizer to split each sentence into words. But there is **yet another tokenizer**…

HuggingFace's Tokenizer

Since we're using HF's datasets, it is only logical that we use HF's tokenizers as well, right? Besides, in order to use a **pretrained BERT model**, we need to use the model's **corresponding pretrained tokenizer**.

"Why?"

Just like pretrained computer vision models require that the input images are standardized using ImageNet statistics, pretrained language models like BERT require that the inputs are properly tokenized. The tokenization used in BERT is *different* than the simple word tokenization we've just discussed. We'll get back to that in due time, but let's stick with the simple tokenization for now.

So, before loading a pretrained tokenizer, let's create our **own tokenizer** using our **own vocabulary**. HuggingFace's tokenizers also expect a **sentence** as input, and they also proceed to perform *some sort* of word tokenization. But, instead of simply returning the tokens themselves, these tokenizers return the **indices in the vocabulary** corresponding to the tokens, and **lots of additional information**. It's like Gensim's `doc2idx`, but on steroids! Let's see it in action!

We'll be using the `BertTokenizer` class to create a tokenizer based on our own vocabulary:

```
from transformers import BertTokenizer
tokenizer = BertTokenizer('our_vocab/vocab.txt')
```

The purpose of this is to **illustrate how the tokenizer works** using simple word tokenization only! The (pretrained) tokenizer you'll use for real with a (pretrained) BERT model **does not need a vocabulary**.

The tokenizer class is very rich and it offers a plethora of methods and arguments. We're just using some basic methods that barely scratch the surface. For more details, please refer to HuggingFace's documentation on the `tokenizer` and `BertTokenizer` classes.

Then, let's tokenize a new sentence using its `tokenize` method:

```
new_sentence = 'follow the white rabbit neo'
new_tokens = tokenizer.tokenize(new_sentence)
new_tokens
```

Output

```
['follow', 'the', 'white', 'rabbit', '[UNK]']
```

Since Neo (from "*The Matrix*") **isn't** part of the original "*Alice's Adventures in*

Wonderland", it couldn't possibly be in our vocabulary and thus it is treated as an **unknown** word with its corresponding special token.

"There is nothing new here… wasn't it supposed to return ***indices*** *and more?"*

Wait for it… First, we actually *can* get the **indices** (the token ids) using the `convert_tokens_to_ids` method:

```
new_ids = tokenizer.convert_tokens_to_ids(new_tokens)
new_ids
```

Output

```
[1219, 5, 229, 200, 1]
```

"OK, fine, but that doesn't seem very practical…"

You're absolutely right. We can use the `encode` method to perform two steps at once:

```
new_ids = tokenizer.encode(new_sentence)
new_ids
```

Output

```
[3, 1219, 5, 229, 200, 1, 2]
```

There we go, from sentence to token ids in one call!

"Nice try… there are ***more ids*** *than tokens in this output! Something must be wrong…"*

Yes, there are more ids than tokens. No, there's nothing wrong, it's actually meant to be like that. These extra tokens are **special tokens** too. We *could* look them up in the vocabulary using their indices (three and two), but it's nicer to use the tokenizer's `convert_ids_to_tokens` method:

```
tokenizer.convert_ids_to_tokens(new_ids)
```

Output

```
['[CLS]', 'follow', 'the', 'white', 'rabbit', '[UNK]', '[SEP]']
```

The tokenizer not only **appended a special separation token (`[SEP]`)** to the output, but it also **prepended a special classification token (`[CLS]`)** to it. We've already added a classification token to the inputs of a *Vision Transformer* to use its corresponding output in a classification task. We can do the same here to classify text using BERT.

"*What about the* ***separation token****?*"

This special token is used to, well, **separate** inputs into **two distinct sentences**. Yes, it is possible to feed BERT with two sentences at once, and this kind of input is used for the **next sentence prediction** task. We won't be using that in our example, but we'll get back to it while discussing how BERT is trained.

We can actually *get rid of the special tokens* if we're not using them:

```
tokenizer.encode(new_sentence, add_special_tokens=False)
```

Output

```
[1219, 5, 229, 200, 1]
```

"*OK, but where is the promised **additional information**?*"

That’s easy enough - we can simply **call the tokenizer itself** instead of a particular method and it will produce an enriched output:

```
tokenizer(new_sentence,
          add_special_tokens=False,
          return_tensors='pt')
```

Output

```
{'input_ids': tensor([[1219,    5,  229,  200,    1]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1]])}
```

By default, the outputs are *lists*, but we used the `return_tensors` argument to get PyTorch tensors instead (`pt` stands for PyTorch). There are **three outputs** in the dictionary, `input_ids`, `token_type_ids`, and `attention_mask`.

The first one, the `input_ids`, is the familiar list of token ids. They are the most fundamental input, and sometimes the only one, required by the model.

The second output, the `token_type_ids`, works as a **sentence index**, and it only makes sense if the input has **more than one sentence** (and the special separation tokens between them). For example:

```
sentence1 = 'follow the white rabbit neo'
sentence2 = 'no one can be told what the matrix is'
tokenizer(sentence1, sentence2)
```

Output

```
{'input_ids': [3, 1219, 5, 229, 200, 1, 2, 51, 42, 78, 32, 307, 41,
5, 1, 30, 2], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1]}
```

Although the tokenizer received **two sentences** as arguments, it considered them a **single input**, thus producing a **single sequence of ids**. Let's convert the ids back to tokens and inspect the result:

```
print(
    tokenizer.convert_ids_to_tokens(joined_sentences['input_ids'])
)
```

Output

```
['[CLS]', 'follow', 'the', 'white', 'rabbit', '[UNK]', '[SEP]',
'no', 'one', 'can', 'be', 'told', 'what', 'the', '[UNK]', 'is',
'[SEP]']
```

The two sentences were concatenated together with a special separation token (`[SEP]`) at the end of each one.

The last output, the `attention_mask`, works as the **source mask** we used in the **Transformer encoder** and it indicates the **padded positions**. In a **batch of sentences**, for example, we may *pad* the sequences to get them all with the same length:

```
separate_sentences = tokenizer([sentence1, sentence2], padding=True)
separate_sentences
```

Output

```
{'input_ids': [[3, 1219, 5, 229, 200, 1, 2, 0, 0, 0, 0], [3, 51, 42,
78, 32, 307, 41, 5, 1, 30, 2]], 'token_type_ids': [[0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0], [1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1]]}
```

The tokenizer received a **list of two sentences**, and it took them as **two independent inputs**, thus producing **two sequences of ids**. Moreover, since the `padding` argument was `True`, it padded the shortest sequence (five tokens) to match the longest one (nine tokens). Let's convert the ids back to tokens once again:

```
print(
    tokenizer.convert_ids_to_tokens(
        separate_sentences['input_ids'][0]
    )
)
print(separate_sentences['attention_mask'][0])
```

Output

```
['[CLS]', 'follow', 'the', 'white', 'rabbit', '[UNK]', '[SEP]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]']
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
```

Each **padded element** in the sequence has a **corresponding zero** in the **attention mask**.

*"Then how can I have a **batch** where **each input** has **two, separate, sentences**?"*

Excellent question! It's actually easy: simply use **two batches**, one containing the

first sentence of each pair, the other containing the **second sentence** of each pair:

```
first_sentences = [sentence1, 'another first sentence']
second_sentences = [sentence2, 'a second sentence here']
batch_of_pairs = tokenizer(first_sentences, second_sentences)
first_input = tokenizer.convert_ids_to_tokens(
                    batch_of_pairs['input_ids'][0]
              )
second_input = tokenizer.convert_ids_to_tokens(
                    batch_of_pairs['input_ids'][1]
               )
print(first_input)
print(second_input)
```

Output

```
['[CLS]', 'follow', 'the', 'white', 'rabbit', '[UNK]', '[SEP]',
'no', 'one', 'can', 'be', 'told', 'what', 'the', '[UNK]', 'is',
'[SEP]']
['[CLS]', 'another', 'first', 'sentence', '[SEP]', '[UNK]',
'second', 'sentence', 'here', '[SEP]']
```

The batch above has only two inputs, and each input has two sentences.

Finally, let's apply our tokenizer to our dataset of sentences, padding them, and returning PyTorch tensors:

```
tokenized_dataset = tokenizer(dataset['sentence'],
                              padding=True,
                              return_tensors='pt',
                              max_length=50,
                              truncation=True)
tokenized_dataset['input_ids']
```

Output

```
tensor([[  3,  27,   1,  ...,   0,   0,   0],
        [  3,  24,  10,  ...,   0,   0,   0],
        [  3,  49,  12,  ...,   0,   0,   0],
        ...,
        [  3,   1,   6,  ...,   0,   0,   0],
        [  3,   6, 132,  ...,   0,   0,   0],
        [  3,   1,   1,  ...,   0,   0,   0]])
```

Since our books may have some *really long sentences*, we can use both `max_length` and `truncation` arguments to ensure that sentences longer than 50 tokens get truncated, and those shorter than that, padded.

For more details on **padding** and **truncation**, please check the awesomely named *"Everything you always wanted to know about padding and truncation"*[172] section of HuggingFace's documentation.

"Are we done? Can we feed the `input_ids` to BERT and watch the magic happen?"

Well, yeah, we could… but don't you prefer to **peek behind the curtain** instead? I thought so :-)

Behind the curtain, BERT is actually using **vectors** to represent the **words**. The **token ids** we'll be sending it are simply the **indices** of an **enormous lookup table**. That lookup table has a very nice name: **word embeddings**.

Each **row** of the lookup table corresponds to a different **token**, and each row is represented by a **vector**. The **size of the vectors** is the **dimensionality of the embedding**.

"How do we build these vectors?"

That’s the million-dollar question! We can either *build them* or **learn them**.

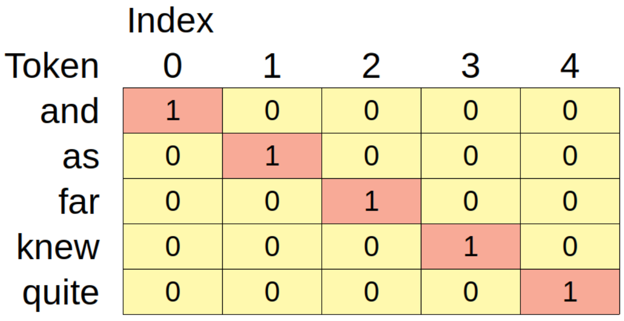
Before Word Embeddings

Before getting to the *actual* word embeddings, let’s start with the **basics** and *build* some simple vectors…

One-Hot Encoding (OHE)

The idea behind OHE is quite simple: each **unique token** (word) is represented by a **vector full of zeros except for one position**, the position corresponding to the **token’s index**. As vectors go, it doesn’t get any simpler than that.

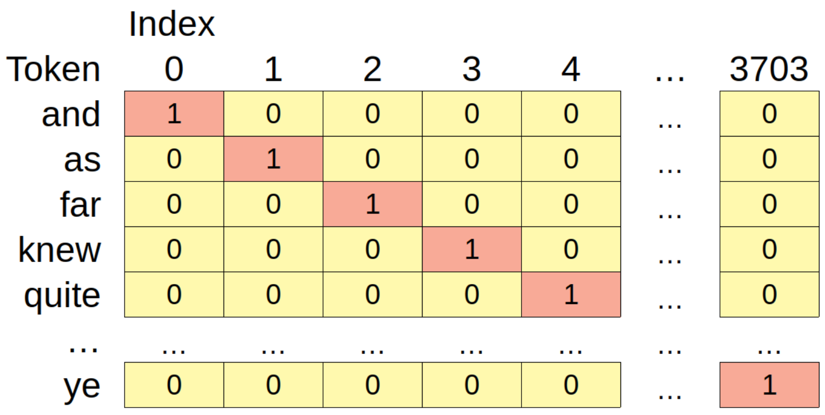
Let’s see it in action using only five tokens - "*and*", "*as*", "*far*", "*knew*", and "*quite*" - and generate **one-hot encoding** representations for them:



	Index				
Token	0	1	2	3	4
and	1	0	0	0	0
as	0	1	0	0	0
far	0	0	1	0	0
knew	0	0	0	1	0
quite	0	0	0	0	1

Figure 11.2 - One-Hot Encoding - vocabulary of five words

The figure above would be the OHE representations of these five tokens **if there were only five tokens in total**. But there are 3,704 unique tokens in our text corpora (not counting the added special tokens), so the OHE actually looks like this:



	Index						
Token	0	1	2	3	4	…	3703
and	1	0	0	0	0	…	0
as	0	1	0	0	0	…	0
far	0	0	1	0	0	…	0
knew	0	0	0	1	0	…	0
quite	0	0	0	0	1	…	0
…	…	…	…	…	…	…	…
ye	0	0	0	0	0	…	1

Figure 11.3 - One-Hot Encoding - our full vocabulary

That's quite a **large** and **sparse** (that's *fancy* for *way more zeros than non-zeros*) vector, right? And our vocabulary is not even *that* large! If we were to use a typical English vocabulary, we would need vectors of 100,000 dimensions. Clearly, this isn't very practical. Nonetheless, the sparse vectors produced by the one-hot encoding are the basis of a **fairly basic NLP model**: the **bag-of-words (BoW)**.

Bag-of-Words (BoW)

The bag-of-words model is **literally** a bag of words: it simply **sums up the corresponding OHE vectors**, completely disregarding any underlying structure or relationships between the words. The resulting vector has only the **counts** of the words appearing in the text.

We don't have to do it manually, though, since Gensim's Dictionary has a `doc2bow` method that does the job for us:

```
sentence = 'the white rabbit is a rabbit'
bow_tokens = simple_preprocess(sentence)
bow_tokens
```

Output

```
['the', 'white', 'rabbit', 'is', 'rabbit']
```

```
bow = dictionary.doc2bow(bow_tokens)
bow
```

Output

```
[(20, 1), (69, 1), (333, 2), (497, 1)]
```

The word "*rabbit*" appears *twice* in the sentence, so its *index* (333) shows the corresponding *count* (2). Also, notice that the fifth word in the original sentence ("*a*") did not qualify as a valid token because it was filtered out by the `simple_preprocess` function for being too short.

The BoW model is obviously very **limited** since it represents the **frequencies** of each word in a piece of text and nothing else. Moreover, representing **words** using **one-hot encoded vectors** also presents severe limitations: not only the vectors become more and more **sparse** (that is, having more zeros in them) as the vocabulary grows, but also **every word is orthogonal to all the other words**.

"*What do you mean by one word being* ***orthogonal*** *to the others?*"

Remember the **cosine similarity** from Chapter 9? Two vectors are said to be **orthogonal** to each other if there is a **right angle** between them, corresponding to a **similarity of zero**. So, if we use **one-hot encoded** vectors to represent words, we're basically saying that **no two words are similar to each other**. This is obviously **wrong** (take synonyms, for example).

"*How can we get better vectors to represent words then?*"

Well, we can try to explore the **structure** and the **relationship** between words in a given sentence. That’s the role of…

Language Models

A **Language Model (LM)** is a model that estimates the **probability of a token** or sequence of tokens. We’ve been using **token** and **word** somewhat interchangeably, but a token can be a single character, or a sub-word too. In other words, a language model will predict the tokens more likely to **fill in a blank**.

Now, pretend you’re a language model and fill in the blank in the following sentence:

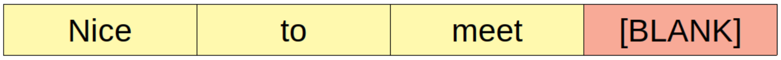


Figure 11.4 - What’s the next word?

You probably filled the blank in with the word "**you**":

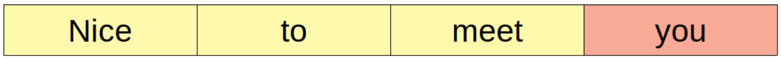


Figure 11.5 - Filling in the [BLANK]

What about this sentence?

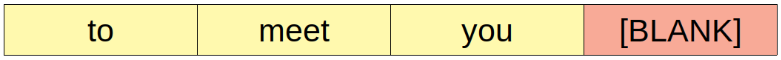


Figure 11.6 - What’s the next word?

Maybe you filled this blank in with "**too**", or maybe you chose a different word like "**there**" or "**now**", depending on what you assumed to be preceding the first word:

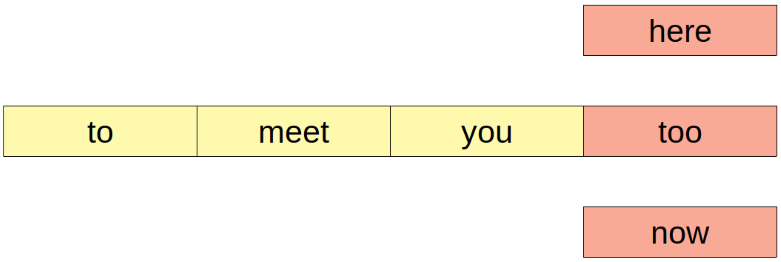


Figure 11.7 - Many options for filling in the [BLANK]

That's easy, right? How did you do it, though? How do you know that "***you***" should follow "*nice to meet*"? You've probably read and said "*nice to meet you*" thousands of times. But have you ever read or said: "*nice to meet aardvark*"? Me neither!

What about the second sentence? It's not that obvious anymore, but I bet you can still rule out "*to meet you aardvark*" (or at least admit that's *very* unlikely to be the case).

It turns out, we have a language model in our heads too, and it's straightforward to guess which words are good choices to fill in the blanks using **sequences** that are familiar to us.

N-grams

The structure, in the examples above, is composed of **three words** and a **blank**: a **four-gram**. If we were using *two words* and blank, that would be a *trigram*, and, for a given number of words (***n-1***) followed by a blank, an ***n-gram***.

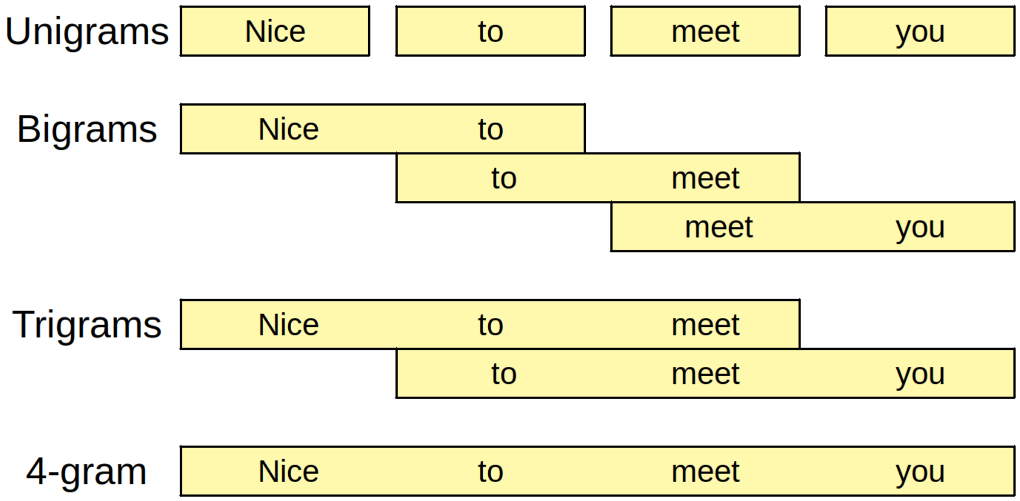


Figure 11.8 - N-grams

N-gram models are based on pure statistics: they fill in the blanks using the **most common sequence** that matches the **words preceding the blank** (that's called the **context**). On the one hand, larger values of ***n*** (longer sequences of words) may yield better predictions; on the other hand, they may yield **no predictions** since a particular sequence of words may have never been observed. In the latter case, one can always fall back to a *shorter n-gram* and try again (that's called a *stupid back-off* by the way).

For a more detailed explanation of n-gram models, please check the *"N-gram Language Models"*[173] section of Lena Voita's amazing *"NLP Course | For You"*[174].

These models are simple, but they are somewhat limited because they can only **look back**.

*"Can we **look ahead** too?"*

Sure, we can!

Continuous Bag-of-Words (CBoW)

In these models, the **context** is given by the **surrounding words**, both **before and after the blank**. That way, it becomes **much easier** to predict the word that best fills in the blank. Let's say we're trying to fill in the following blank:

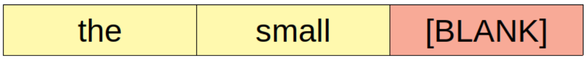


Figure 11.9 - Filling the [BLANK] at the end

That's what a *trigram model* would have to work with. It doesn't look good... the possibilities are endless. Now, consider the same sentence once again, this time containing the words that **follow the blank**:

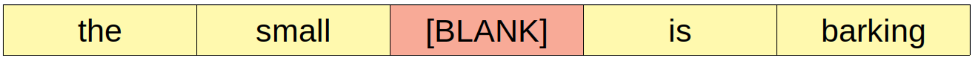


Figure 11.10 - Filling the [BLANK] in the center

Well, that's easy: the blank is "***dog***".

"*Cool, but what does the bag-of-words has to do with it?*"

It is a bag-of-words because it **sums up (or averages) the vectors** of the **context words** ("*the*", "*small*", "*is*", and "*barking*") and uses it to **predict the central word**.

"*Why is it continuous? What does it even mean?*"

It means the vectors are *not one-hot encoded* anymore and have **continuous values** instead. The vector of continuous values that represents a given word is a called **word embedding**.

Word Embeddings

"*How do we find the values that best represent each word?*"

We need to train a model to learn them. This model is called…

Word2Vec

Word2Vec was proposed by Mikolov, T. et al. in their 2013 paper "*Efficient Estimation of Word Representations in Vector Space*"[175], and it included two model architectures: continuous bag-of-words (CBoW) and skip-gram (SG). We're focusing on the former.

In the CBoW architecture, the **target** is the **central word**. In other words, we're dealing with a **multi-class classification problem** where the **number of classes** is given by the **size of the vocabulary** (any word in the vocabulary can be the central word). And we'll be using the **context words**, better yet, their **corresponding embeddings (vectors)**, as **inputs**.

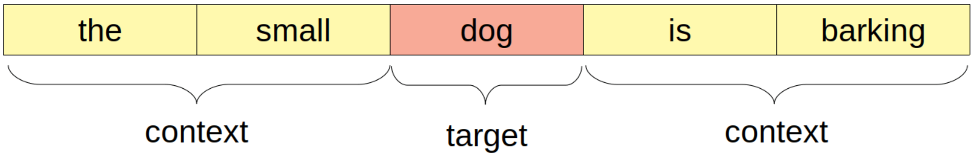


Figure 11.11 - Target and context words

"Wait, how come we're using the embeddings as inputs? That's what we're trying to learn in the first place!"

Exactly! The **embeddings** are also **parameters of the model** and, as such, they are **randomly initialized** as well. As training progresses, their weights are being updated by gradient descent like any other parameter and, in the end, we'll have **embeddings** for each word in the vocabulary.

For each pair of context words and corresponding target, the model will **average the embeddings of the context words**, and feed the result to a **linear layer** that will compute **one logit for each word in the vocabulary**. That's it! Let's check the corresponding code:

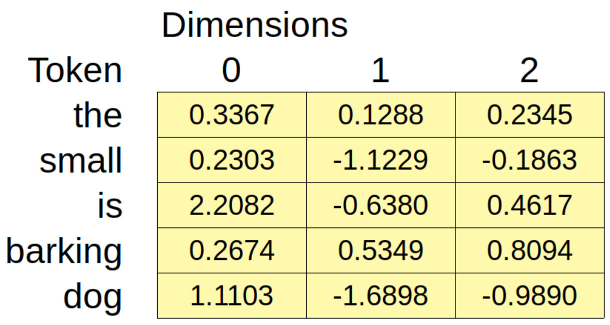
```
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size)
        self.linear = nn.Linear(embedding_size, vocab_size)

    def forward(self, X):
        embeddings = self.embedding(X)
        bow = embeddings.mean(dim=1)
        logits = self.linear(bow)
        return logits
```

That's a fairly simple model, right? If our vocabulary had only five words ("*the*", "*small*", "*is*", "*barking*", and "*dog*"), we could try to represent each word with an **embedding of three dimensions**. Let's create a dummy model to inspect its (randomly initialized) embeddings:

```
torch.manual_seed(42)
dummy_cbow = CBOW(vocab_size=5, embedding_size=3)
dummy_cbow.embedding.state_dict()
```

Output

```
OrderedDict([('weight', tensor([[ 0.3367,  0.1288,  0.2345],
                      [ 0.2303, -1.1229, -0.1863],
                      [ 2.2082, -0.6380,  0.4617],
                      [ 0.2674,  0.5349,  0.8094],
                      [ 1.1103, -1.6898, -0.9890]]))])
```



	Dimensions		
Token	0	1	2
the	0.3367	0.1288	0.2345
small	0.2303	-1.1229	-0.1863
is	2.2082	-0.6380	0.4617
barking	0.2674	0.5349	0.8094
dog	1.1103	-1.6898	-0.9890

Figure 11.12 - Word embeddings

As depicted in the figure above, PyTorch's `nn.Embedding` layer is a **big lookup table**. It may be *randomly initialized* given the **size of the vocabulary** (`num_embeddings`) and the **number of dimensions** (`embedding_dim`). To actually **retrieve the values** we need to call the embedding layer with a **list of tokens indices**, and it will return the corresponding **rows** of the table.

For example, we can retrieve the embeddings for the tokens "*is*" and "*barking*" using their corresponding indices (two and three):

```
# tokens: ['is', 'barking']
dummy_cbow.embedding(torch.as_tensor([2, 3]))
```

Output

```
tensor([[ 2.2082, -0.6380,  0.4617],
        [ 0.2674,  0.5349,  0.8094]], grad_fn=<EmbeddingBackward>)
```

That's why the **main job** of the **tokenizer** is to **transform a sentence into a list of token ids**. That list is used as an **input to the embedding layer**, and from then on, the tokens are represented by a dense vector.

"How do you choose the number of dimensions?"

It is commonplace to use between 50 and 300 dimensions for word embeddings, but some embeddings may be as large as 3,000 dimensions. That may look like a lot but, compared to one-hot encoded vectors, it is still a bargain! The vocabulary of our tiny dataset would already require more than 3,000 dimensions if it were one-hot encoded.

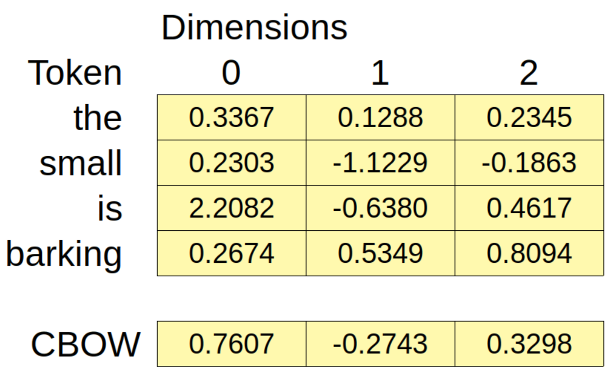
In our former example, "***dog***" was the **central word** and the other four words were the **context words**:

```
tiny_vocab = ['the', 'small', 'is', 'barking', 'dog']
context_words = ['the', 'small', 'is', 'barking']
target_words = ['dog']
```

Now, let's pretend that we tokenized the words and got their corresponding indices:

```
batch_context = torch.as_tensor([[0, 1, 2, 3]]).long()
batch_target = torch.as_tensor([4]).long()
```

In its very first training step, the model would then **compute the continuous bag-of-words** for the inputs by **averaging the corresponding embeddings**:



	Dimensions		
Token	0	1	2
the	0.3367	0.1288	0.2345
small	0.2303	-1.1229	-0.1863
is	2.2082	-0.6380	0.4617
barking	0.2674	0.5349	0.8094
CBOW	0.7607	-0.2743	0.3298

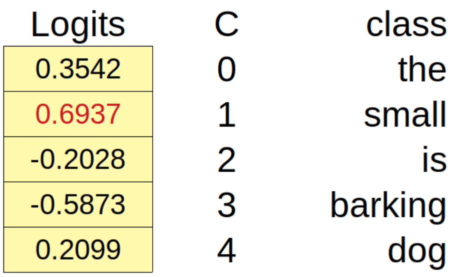
Figure 11.13 - Continuous Bag-of-Words

```
cbow_features = dummy_cbow.embedding(batch_context).mean(dim=1)
cbow_features
```

Output

```
tensor([[ 0.7606, -0.2743,  0.3298]], grad_fn=<MeanBackward1>)
```

The bag-of-words have **three dimensions** and these dimensions are the **features** used to **compute the logits** for our multi-class classification problem:



Logits	C	class
0.3542	0	the
0.6937	1	small
-0.2028	2	is
-0.5873	3	barking
0.2099	4	dog

Figure 11.14 - Logits

```
logits = dummy_cbow.linear(cbow_features)
logits
```

Output

```
tensor([[ 0.3542,  0.6937, -0.2028, -0.5873,  0.2099]],
       grad_fn=<AddmmBackward>)
```

The *largest logit* corresponds to the word "***small***" (class index one), so that would be the **predicted central word**: "*the small **small** is barking*". The prediction is obviously **wrong** but, then again, that’s still a randomly initialized model. Given a large enough dataset of context and target words, we could train the `CBOW` model above using a `CrossEntropyLoss` to **learn actual word embeddings**.

The Word2Vec model may *also* be trained using the **skip-gram** approach instead of continuous bag-of-words. The skip-gram uses the **central word** to **predict the surrounding words**, thus being a **multi-label multi-class classification problem**. In our simple example, the input would be the central word "*dog*", and the model would try to predict the four context words ("*the*", "*small*", "*is*", and "*barking*") at once.

We're not diving any deeper into the inner workings of the Word2Vec model, but you can check Jay Alammar's *"The Illustrated Word2Vec"*[176] and Lilian Weng's *"Learning Word Embedding"*[177] amazing posts on the subject.

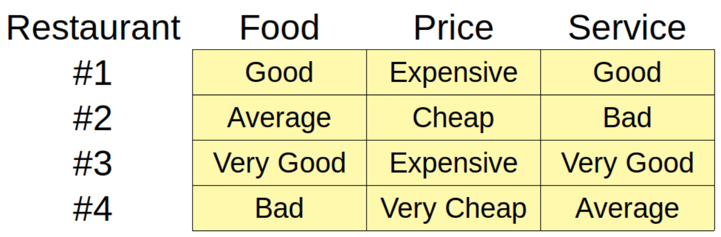
If you're interested in training a Word2Vec model yourself, follow Jason Brownlee's great tutorial: *"How to Develop Word Embeddings in Python with Gensim"*[178].

So far, it looks like we're learning **word embeddings** just for the sake of getting **more compact (dense)** representations than one-hot encoding can offer for each word. But word embeddings are *more than that*.

What Is an Embedding Anyway?

An embedding is a **representation** of an entity (a word, in our case), and each one of its **dimensions** can be seen as an **attribute** or **feature**.

Let's forget about words for a moment and talk about **restaurants** instead. We can rate restaurants over many different **dimensions**, like **food**, **price**, and **service**, for example:



Restaurant	Food	Price	Service
#1	Good	Expensive	Good
#2	Average	Cheap	Bad
#3	Very Good	Expensive	Very Good
#4	Bad	Very Cheap	Average

Figure 11.15 - Reviewing restaurants

Clearly, restaurants #1 and #3 have good food and service but are expensive, and restaurants #2 and #4 are cheap but either the food or the service is bad. It's fair to say that restaurants #1 and #3 are similar to each other, and they are both very different from restaurants #2 and #4 which, in turn, are somewhat similar to each other as well.

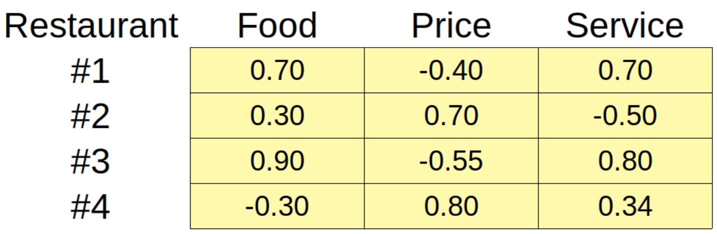
"*What about the **cuisine**? We can't properly compare restaurants without that information!*"

I agree with you, so let's just pretend that all of them are **pizza places** :-)

Although it's fairly obvious to spot the similarities and differences among the restaurants in the table above, it wouldn't be so easy to spot them if there were *dozens of dimensions* to compare. Besides, it would be very hard to **objectively measure the similarity** between any two restaurants using categorical scales like that.

"*What if we use **continuous scales** instead?*"

Perfect! Let's do that and assign values in the range `[-1, 1]`, from very bad (-1) to very good (1), or from very expensive (-1) to very cheap (1):



Restaurant	Food	Price	Service
#1	0.70	-0.40	0.70
#2	0.30	0.70	-0.50
#3	0.90	-0.55	0.80
#4	-0.30	0.80	0.34

Figure 11.16 - Restaurant "embeddings"

These values are like "**restaurant embeddings**" :-)

Well, they're not *quite* embeddings, but at least we can use **cosine similarity** to find out how similar to each other two restaurants are:

```
ratings = torch.as_tensor([[.7, -.4, .7],
                           [.3, .7, -.5],
                           [.9, -.55, .8],
                           [-.3, .8, .34]]).float()
sims = torch.zeros(4, 4)
for i in range(4):
    for j in range(4):
        sims[i, j] = F.cosine_similarity(ratings[i],
                                         ratings[j],
                                         dim=0)
sims
```

Output

```
tensor([[ 1.0000, -0.4318,  0.9976, -0.2974],
        [-0.4318,  1.0000, -0.4270,  0.3581],
        [ 0.9976, -0.4270,  1.0000, -0.3598],
        [-0.2974,  0.3581, -0.3598,  1.0000]])
```

As expected, restaurants #1 and #3 are remarkably similar (`0.9976`), and restaurants #2 and #4 are somewhat similar (`0.3581`). Restaurant #1 is quite different from restaurants #2 and #4 (`-0.4318` and `-0.2974`, respectively), and so is restaurant #3 (`-0.4270` and `-0.3598`, respectively).

Although we *can* compute the cosine similarity between two restaurants now, the values in the table above **are not real embeddings**. It was only an example that illustrates well the **concept of embedding dimensions as attributes**.

Unfortunately, the **dimensions** of the **word embeddings** learned by the Word2Vec model **do not have clear-cut meanings** like that.

On the bright side, though, it is possible to do **arithmetic with word embeddings**!

"*Say what?*"

You got that right - **arithmetic** - really! Maybe you've seen this "*equation*" somewhere else already:

> KING - MAN + WOMAN = QUEEN

Awesome, right? We'll try this "*equation*" out shortly, hang in there!

Pretrained Word2Vec

Word2Vec is a simple model but it still requires a sizable amount of text data to learn meaningful embeddings. Luckily for us, someone else had already done the hard work of training these models, and we can use Gensim's `downloader` to choose from a variety of pretrained word embeddings.

For a detailed list of the available models (embeddings), please check Gensim-data's repository[179] on GitHub.

"*Why so many embeddings? How are they different from each other?*"

Good question! It turns out, using **different text corpora** to train a Word2Vec model produces **different embeddings**. On the one hand, this shouldn't be a surprise, after all, these are **different datasets** and it's expected that they will produce **different results**. On the other hand, if these datasets all contain **sentences in the same language** (English, for example), how come the embeddings are different?

The embeddings will be influenced by the **kind of language** used in the text: the phrasing and wording used in novels are different from those used in news articles and radically different from those used in Twitter, for example.

☺ "*Choose your word embeddings wisely.*"

Grail Knight

Moreover, not every word embedding is learned using a Word2Vec model architecture. There are *many* different ways of learning word embeddings, one of them being…

Global Vectors (GloVe)

The Global Vectors model was proposed by Pennington, J. et al. in their 2014 paper "*GloVe: Global Vectors for Word Representation*"[180]. It combines the **skip-gram** model with **co-occurrence statistics** at the **global level** (hence the name). We're not diving into its inner workings here but, if you're interested in knowing more about it, make sure to check its official website: https://nlp.stanford.edu/projects/glove/.

The pretrained GloVe embeddings come in many sizes and shapes: dimensions vary between 25 and 300, vocabularies vary between 400,000 and 2,200,000 words. Let's use Gensim's `downloader` to retrieve the smallest one: `glove-wiki-gigaword-50`. It was trained on Wikipedia 2014 and Gigawords 5, it contains 400,000 words in its vocabulary, and its embeddings have 50 dimensions.

Downloading pretrained word embeddings

```
from gensim import downloader
glove = downloader.load('glove-wiki-gigaword-50')
len(glove.vocab)
```

Output

```
400000
```

Let's check the embeddings for "***alice***" (the vocabulary is uncased):

```
glove['alice']
```

Output

```
array([ 0.16386,  0.57795, -0.59197, -0.32446,  0.29762,  0.85151,
       -0.76695, -0.20733,  0.21491, -0.51587, -0.17517,  0.94459,
        0.12705, -0.33031,  0.75951,  0.44449,  0.16553, -0.19235,
        0.06553, -0.12394,  0.61446,  0.89784,  0.17413,  0.41149,
        1.191  , -0.39461, -0.459  ,  0.02216, -0.50843, -0.44464,
        0.68721, -0.7167 ,  0.20835, -0.23437,  0.02604, -0.47993,
        0.31873, -0.29135,  0.50273, -0.55144, -0.06669,  0.43873,
       -0.24293, -1.0247 ,  0.02937,  0.06849,  0.25451, -1.9663 ,
        0.26673,  0.88486], dtype=float32)
```

There we go, 50 dimensions! It's time to try the famous "*equation*": KING - MAN + WOMAN = QUEEN. We're calling the result a "*synthetic queen*":

```
synthetic_queen = glove['king'] - glove['man'] + glove['woman']
```

These are the corresponding embeddings:

```
fig = plot_word_vectors(
    glove, ['king', 'man', 'woman', 'synthetic', 'queen'],
    other={'synthetic': synthetic_queen}
)
```

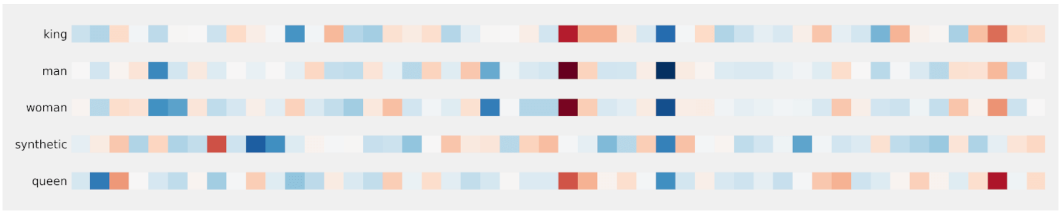


Figure 11.17 - Synthetic Queen

How similar is the "***synthetic queen***" to the **actual "*queen*"**, you ask. It’s hard to tell by looking at the vectors above alone, but Gensim’s word vectors have a `similar_by_vector` method that computes **cosine similarity** between a **given vector** and *the whole vocabulary* and returns the **top N most similar words**:

```
glove.similar_by_vector(synthetic_queen, topn=5)
```

Output

```
[('king', 0.8859835863113403),
 ('queen', 0.8609581589698792),
 ('daughter', 0.7684512138366699),
 ('prince', 0.7640699148178101),
 ('throne', 0.7634971141815186)]
```

*"The most similar word to the "synthetic queen" is… **king**?"*

Yes. It’s not *always* the case, but it’s fairly common to find out that, after performing **word embedding arithmetic**, the word most similar to the result is the **original word** itself. For this reason, it’s usual to **exclude the original word** from the similarity results. In this case, the most similar word to the "*synthetic queen*" is, indeed, the actual "*queen*".

"OK, cool, but ***how*** *does this arithmetic work?"*

The general idea is that the embeddings learned to **encode abstract dimensions**, like "*gender*", "*royalty*", "*genealogy*", or "*profession*". None of these abstract dimensions corresponds to a single numerical dimension, though.

In its large 50-dimensional feature space, the model learned to place "*man*" as far apart from "*woman*" as "*king*" is from "*queen*" (roughly approximating the gender difference between the two). Similarly, the model learned to place "*king*" as far apart from "*man*" as "*queen*" is from "*woman*" (roughly approximating the difference of being a royal).

The figure below depicts a hypothetical projection in two dimensions for easier visualization:

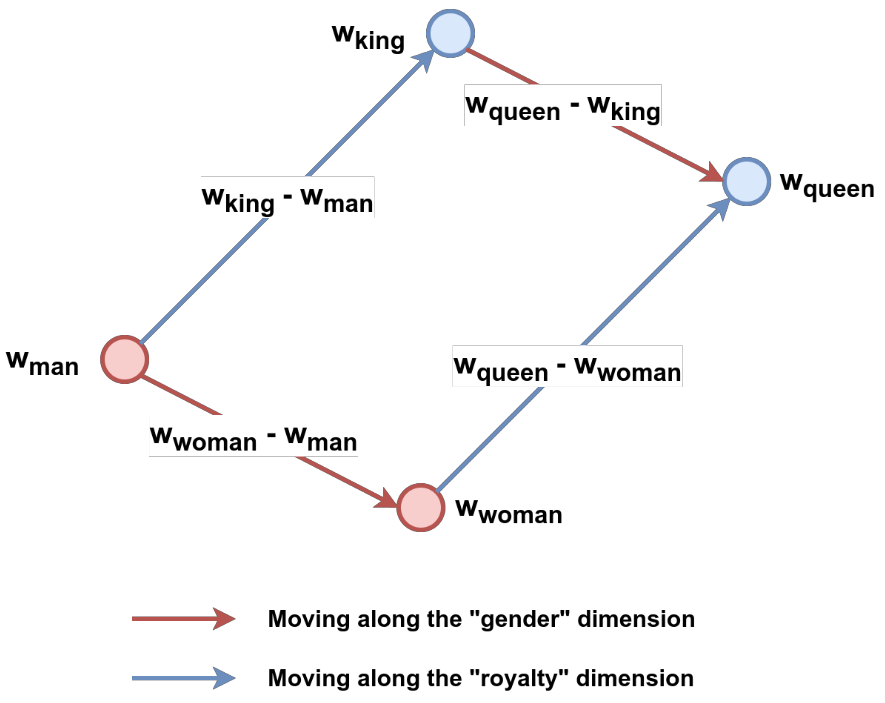


Figure 11.18 - Projection of embeddings

From the figure above, it should be relatively clear that both arrows pointing up (blue) are approximately the same size thus resulting in the equation below:

$$w_{king} - w_{man} \approx w_{queen} - w_{woman}$$
$$\implies w_{king} - w_{man} + w_{woman} \approx w_{queen}$$

Equation 11.1 - Embedding arithmetic

This arithmetic is cool and all, but you won't be actually **using it** much, the whole point was to show you that the **word embeddings** indeed **capture the relationship** between different words. We *can* use them to train *other models*, though…

Using Word Embeddings

It seems easy enough: get the text corpora tokenized, look the tokens up in the table of pretrained word embeddings, and then use the embeddings as inputs of another model. But, *what if* the **vocabulary** of your corpora is **not quite properly represented** in the embeddings? Even worse, *what if* the **preprocessing steps** you used resulted in a lot of tokens that **do not exist in the embeddings**?

"*Choose your word embeddings wisely.*"

Grail Knight

Vocabulary Coverage

Once again, the Grail Knight has a point… the chosen word embeddings must provide a good **vocabulary coverage**. First and foremost, **most of the usual preprocessing steps do not apply** whenever you're using pretrained word embeddings like GloVe: no lemmatization, no stemming, no stop word removal. These steps would likely end up producing a lot of `[UNK]` tokens.

Second, even without those preprocessing steps, maybe the words used in the given text corpora are simply *not a good match* for a particular pretrained set of word embeddings.

Let's see how good a match the `glove-wiki-gigaword-50` embeddings are to our own vocabulary. Our vocabulary has 3,706 words (3,704 from our text corpora plus the padding and unknown special tokens):

```
vocab = list(dictionary.token2id.keys())
len(vocab)
```

Output

```
3706
```

Let's see **how many words of our own vocabulary** are **unknown to the embeddings**:

```
unknown_words = sorted(
    list(set(vocab).difference(set(glove.vocab)))
)
print(len(unknown_words))
print(unknown_words[:5])
```

Output

```
44
['[PAD]', '[UNK]', 'arrum', 'barrowful', 'beauti']
```

There are only 44 unknown words: the two special tokens, and some other weird words like "*arrum*" and "*barrowful*". It looks good, right? It means that there are 3,662 matches out of 3,706 words, hinting at 98.81% coverage. But it is actually *better* than that.

If we look at **how often the unknown words** show up in our text corpora, we'll have a precise measure of **how many tokens** will be unknown to the embeddings. To actually get the total count we need to get the ids of the unknown words first, and then look at its frequencies in the corpora:

```
unknown_ids = [dictionary.token2id[w]
               for w in unknown_words
               if w not in ['[PAD]', '[UNK]']]
unknown_count = np.sum([dictionary.cfs[idx]
                        for idx in unknown_ids])
unknown_count, dictionary.num_pos
```

Output

```
(82, 50802)
```

Only 82 out of 50,802 words in the text corpora cannot be matched to the vocabulary of the word embeddings. That's an impressive 99.84% coverage!

The helper function below can be used to compute the **vocabulary coverage** given a **Gensim's Dictionary** and **pretrained embeddings**:

Method for vocabulary coverage

```python
def vocab_coverage(gensim_dict, pretrained_wv,
                   special_tokens=('[PAD]', '[UNK]')):
    vocab = list(gensim_dict.token2id.keys())
    unknown_words = sorted(
        list(set(vocab).difference(set(pretrained_wv.vocab)))
    )
    unknown_ids = [gensim_dict.token2id[w]
                   for w in unknown_words
                   if w not in special_tokens]
    unknown_count = np.sum([gensim_dict.cfs[idx]
                            for idx in unknown_ids])
    cov = 1 - unknown_count / gensim_dict.num_pos
    return cov
```

```python
vocab_coverage(dictionary, glove)
```

Output

```
0.9983858903192788
```

Tokenizer

Once we're happy with the **vocabulary coverage** of our pretrained embeddings, we can **save the vocabulary of the embeddings to disk** as a plain text file once again, so we can use it with the HF's tokenizer:

Method to save a vocabulary from pretrained embeddings

```python
def make_vocab_from_wv(wv, folder=None, special_tokens=None):
    if folder is not None:
        if not os.path.exists(folder):
            os.mkdir(folder)

    words = wv.index2word
    if special_tokens is not None:
        to_add = []
        for special_token in special_tokens:
            if special_token not in words:
                to_add.append(special_token)
        words = to_add + words

    with open(os.path.join(folder, 'vocab.txt'), 'w') as f:
        for word in words:
            f.write(f'{word}\n')
```

Saving GloVe's vocabulary to a file

```python
make_vocab_from_wv(glove,
                   'glove_vocab/',
                   special_tokens=['[PAD]', '[UNK]'])
```

We'll be using the `BertTokenizer` class once again to create a tokenizer based on GloVe's vocabulary:

```python
glove_tokenizer = BertTokenizer('glove_vocab/vocab.txt')
```

One more time: the (pretrained) tokenizer you'll use for real with a (pretrained) BERT model **does not need a vocabulary**.

Now we can use its `encode` method to get the indices for the tokens in a sentence:

```
glove_tokenizer.encode('alice followed the white rabbit',
                        add_special_tokens=False)
```

Output

```
[7101, 930, 2, 300, 12427]
```

These are the **indices** we'll use to **retrieve the corresponding word embeddings**. There is **one small detail** we need to take care of first, though…

Special Tokens' Embeddings

Our vocabulary has 400,002 tokens now, but the original pretrained word embeddings have only 400,000 entries:

```
len(glove_tokenizer.vocab), len(glove.vectors)
```

Output

```
(400002, 400000)
```

The difference is due to the **two special tokens**, `[PAD]` and `[UNK]` that were **prepended to the vocabulary** when we saved it to disk. Therefore, we need to **prepend their corresponding embeddings** too.

"*How would I know the embeddings for these tokens?*"

That's actually easy, these embeddings are just 50-dimensional vectors of **zeros**, and we concatenate them to the GloVe's pretrained embeddings, making sure that the special embeddings come first:

Adding embeddings for the special tokens

```
special_embeddings = np.zeros((2, glove.vector_size))
extended_embeddings = np.concatenate(
    [special_embeddings, glove.vectors], axis=0
)
extended_embeddings.shape
```

Output

```
(400002, 50)
```

Now, if we encode "*alice*" to get its corresponding index, and use that index to retrieve the corresponding values from our *extended embeddings*, they should match the original GloVe embeddings:

```
alice_idx = glove_tokenizer.encode(
    'alice', add_special_tokens=False
)
np.all(full_embeddings[alice_idx] == glove['alice'])
```

Output

```
True
```

OK, it looks like we're set! Let's put these embeddings to good use and, *finally*, train a model in PyTorch!

Model I - GloVE + Classifier

Data Preparation

It all starts with the *data preparation* step. As we already know, we need to **tokenize** the sentences to get their corresponding **sequences of token ids**. The sentences (and the labels) can be easily retrieved from HF's dataset like a dictionary:

Data Preparation

```
train_sentences = train_dataset['sentence']
train_labels = train_dataset['labels']

test_sentences = test_dataset['sentence']
test_labels = test_dataset['labels']
```

Next, we use our `glove_tokenizer` to tokenize the sentences, making sure that we **pad** and **truncate** them so they all end up with 60 tokens (like we did in the "*HuggingFace's Tokenizer*" section). We only need the `inputs_ids` to fetch their corresponding embeddings later on:

Data Preparation - Tokenizing

```
train_ids = glove_tokenizer(train_sentences,
                            truncation=True,
                            padding=True,
                            max_length=60,
                            add_special_tokens=False,
                            return_tensors='pt')['input_ids']
train_labels = torch.as_tensor(train_labels).float().view(-1, 1)

test_ids = glove_tokenizer(test_sentences,
                           truncation=True,
                           padding=True,
                           max_length=60,
                           add_special_tokens=False,
                           return_tensors='pt')['input_ids']
test_labels = torch.as_tensor(test_labels).float().view(-1, 1)
```

Both sequences of token ids and labels are regular PyTorch tensors now, so we can use the familiar `TensorDataset`:

Data Preparation

```
train_tensor_dataset = TensorDataset(train_ids, train_labels)
generator = torch.Generator()
train_loader = DataLoader(
    train_tensor_dataset, batch_size=32,
    shuffle=True, generator=generator
)
test_tensor_dataset = TensorDataset(test_ids, test_labels)
test_loader = DataLoader(test_tensor_dataset, batch_size=32)
```

"*Hold on! Why are we going back to* `TensorDataset` *instead of using HF's* `Dataset`*?*"

Well, even though HF's `Dataset` was **extremely useful** to load and manipulate all the files from our text corpora, and it will surely work seamlessly with HF's pretrained models, it's *not ideal* to work with our regular, pure PyTorch, training routine.

"*Why not?*"

It boils down to the fact that, while a `TensorDataset` returns a **typical (features, label) tuple**, the HF's `Dataset` always returns a **dictionary**. So, instead of jumping through hoops to accommodate this difference in their outputs, it's easier to fall back to the familiar `TensorDataset` for now.

We already have **token ids** and **labels**. But we also have to **load the pretrained embeddings** that **match the ids** produced by the tokenizer.

Pretrained PyTorch Embeddings

The **embedding layer** in PyTorch, `nn.Embedding`, can be either **trained** like any other layer or **loaded** using its `from_pretrained` method. Let's load the *extended* version of the pretrained GloVe embeddings:

```
extended_embeddings = torch.as_tensor(extended_embeddings).float()
torch_embeddings = nn.Embedding.from_pretrained(extended_embeddings)
```

By default, the embeddings are **frozen**, that is, they **won't be updated** during model training. You can change this behavior by setting the `freeze` argument to `False`, though.

Then, let's take the first mini-batch of tokenized sentences and their labels:

```
token_ids, labels = next(iter(train_loader))
token_ids
```

Output

```
tensor([[  36,   63,    1,  ...,    0,    0,    0],
        [ 934,   16,   14,  ...,    0,    0,    0],
        [  57,  311,    8,  ...,  140,    3,   83],
        ...,
        [7101,   59, 1536,  ...,    0,    0,    0],
        [  43,   59, 1995,  ...,    0,    0,    0],
        [ 102,   41,  210,  ...,  685,    3,    7]])
```

There are 32 sentences of 60 tokens each. We can use this batch of token ids to retrieve their corresponding **embeddings**:

```
token_embeddings = torch_embeddings(token_ids)
token_embeddings.shape
```

Output

```
torch.Size([32, 60, 50])
```

Since **each embedding has 50 dimensions**, the resulting tensor has the shape above: 32 sentences, 60 tokens each, 50 dimensions for each token.

Let's make it a bit *simpler* and **average the embeddings** corresponding to **all tokens in a sentence**:

```
token_embeddings.mean(dim=1)
```

Output

```
tensor([[ 0.0665, -0.0071, -0.0534,  ..., -0.0202, -0.1432],
        [ 0.0514,  0.0495,  0.0083,  ...,  0.0162,  0.0687],
        ...,
        [ 0.0516,  0.1091,  0.0917,  ...,  0.0037,  0.0553],
        [ 0.1972,  0.1069, -0.2049,  ..., -0.1026, -0.3731]])
```

Now **each sentence** is represented by an **average embedding of its tokens**. That's a **bag-of-words** or, better yet, a **bag-of-embeddings**. Each tensor is a numerical representation of a sentence and we can use it as **features** for a classification algorithm.

By the way, for training simple models using bag-of-embeddings as inputs, it is better to use PyTorch's nn.EmbeddingBag instead. The outcome is exactly the same as the one above, but it is faster:

```
boe_mean = nn.EmbeddingBag.from_pretrained(
    extended_embeddings, mode='mean'
)
boe_mean(token_ids)
```

Besides, we don't have to take the mean manually anymore and therefore we can use it in a simple `Sequential` model.

Model Configuration & Training

Let’s build a `Sequential` model to classify our sentences according to their source ("*Alice’s Adventures in Wonderland*" or "*The Wonderful Wizard of Oz*") using PyTorch’s `nn.EmbeddingBag`:

Model Configuration

```python
extended_embeddings = torch.as_tensor(
    extended_embeddings
).float()
boe_mean = nn.EmbeddingBag.from_pretrained(
    extended_embeddings, mode='mean'
)
torch.manual_seed(41)
model = nn.Sequential(
    # Embeddings
    boe_mean,
    # Classifier
    nn.Linear(boe_mean.embedding_dim, 128),
    nn.ReLU(),
    nn.Linear(128, 1)
)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

The model is quite simple and straightforward: the bag-of-embeddings generates a **batch of average embeddings** (each sentence is represented by a tensor of `embedding_dim` dimensions), and those embeddings work as **features** for the classifier part of the model.

We can train the model in the usual way:

Model Training

```
sbs_emb = StepByStep(model, loss_fn, optimizer)
sbs_emb.set_loaders(train_loader, test_loader)
sbs_emb.train(20)
```

```
fig = sbs_emb.plot_losses()
```

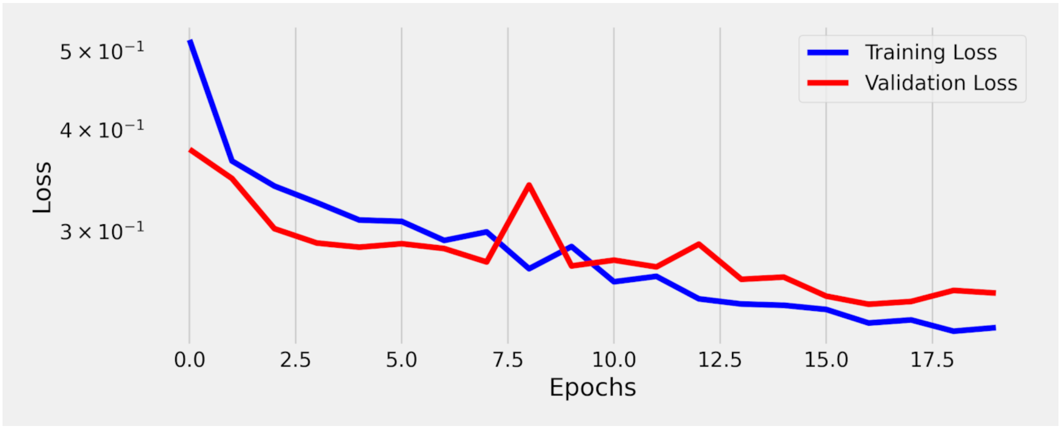


Figure 11.19 - Losses - Bag-of-Embeddings (BoE)

```
StepByStep.loader_apply(test_loader, sbs_emb.correct)
```

Output

```
tensor([[380, 440],
        [311, 331]])
```

That's 89.62% accuracy on the test set. Not bad, not bad at all!

"OK, but I don't want to use a `Sequential` *model, I want to use a* ***Transformer****…"*

I hear you.

Model II - GloVe + Transformer

We'll use a **Transformer Encoder** as a classifier once again, just like we did in the "*Vision Transformer*" section from Chapter 10. The model is pretty much the *same* except for the fact that we're using **pretrained word embeddings** instead of **patch embeddings**:

Model Configuration

```
class TransfClassifier(nn.Module):
    def __init__(self, embedding_layer, encoder, n_outputs):
        super().__init__()
        self.d_model = encoder.d_model
        self.n_outputs = n_outputs
        self.encoder = encoder
        self.mlp = nn.Linear(self.d_model, n_outputs)

        self.embed = embedding_layer                     ①
        self.cls_token = nn.Parameter(
            torch.zeros(1, 1, self.d_model)
        )

    def preprocess(self, X):
        # N, L -> N, L, D
        src = self.embed(X)
        # Special classifier token
        # 1, 1, D -> N, 1, D
        cls_tokens = self.cls_token.expand(X.size(0), -1, -1)
        # Concatenates CLS tokens -> N, 1 + L, D
        src = torch.cat((cls_tokens, src), dim=1)
        return src

    def encode(self, source, source_mask=None):
        # Encoder generates "hidden states"
        states = self.encoder(source, source_mask)       ②
        # Gets state from first token: CLS
        cls_state = states[:, 0]  # N, 1, D
        return cls_state

    @staticmethod
    def source_mask(X):                                    ②
        cls_mask = torch.ones(X.size(0), 1).type_as(X)
        pad_mask = torch.cat((cls_mask, X > 0), dim=1).bool()
        return pad_mask.unsqueeze(1)

    def forward(self, X):
        src = self.preprocess(X)
        # Featurizer
        cls_state = self.encode(src,
                                self.source_mask(X)) ②
        # Classifier
        out = self.mlp(cls_state) # N, 1, outputs
        return out
```

① The embedding layer is an argument now

② The encoder receives a source mask to flag the padded (and classification) tokens

Our model takes an instance of a **Transformer encoder**, a layer of **pretrained embeddings** (*not* an `EmbeddingBag` anymore!), and the desired **number of outputs** (logits) corresponding to the number of existing classes.

The `forward` method takes a mini-batch of tokenized sentences, pre-processes them, encodes them (featurizer), and outputs logits (classifier). It really works just like the Vision Transformer from Chapter 10 but now it takes a sequence of words (tokens) instead of image patches.

Let's create an instance of our model and train it in the usual way:

Model Configuration

```
torch.manual_seed(33)
# Loads the pretrained GloVe embeddings into an embedding layer
torch_embeddings = nn.Embedding.from_pretrained(
    extended_embeddings
)
# Creates a Transformer Encoder
layer = EncoderLayer(n_heads=2,
                     d_model=torch_embeddings.embedding_dim,
                     ff_units=128)
encoder = EncoderTransf(layer, n_layers=1)
# Uses both layers above to build our model
model = TransfClassifier(torch_embeddings, encoder, n_outputs=1)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

Model Training

```
sbs_transf = StepByStep(model, loss_fn, optimizer)
sbs_transf.set_loaders(train_loader, test_loader)
sbs_transf.train(10)
```

```
fig = sbs_transf.plot_losses()
```

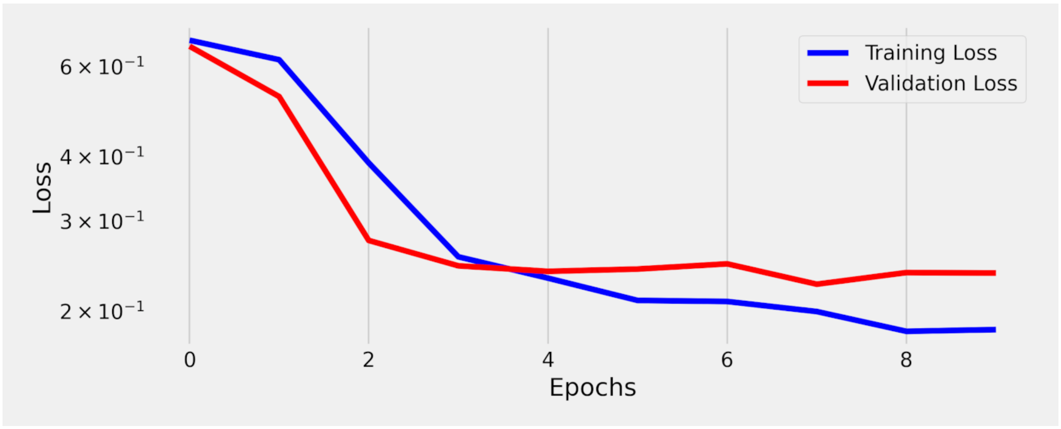


Figure 11.20 - Losses - Transformer + GloVe embeddings

Looks like our model started overfitting really fast since the validation loss barely improves, if at all, after the third epoch. Let's check its accuracy on the validation (test) set:

```
StepByStep.loader_apply(test_loader, sbs_transf.correct)
```

Output

```
tensor([[410, 440],
        [300, 331]])
```

That's 92.09% accuracy. Well, that's good, but *not so much better* than the simple bag-of-embeddings model as you might expect from a *mighty Transformer*, right?

Let's see what our model is actually **paying attention to**…

Visualizing Attention

Instead of using sentences from the validation (test) set, let's come up with **brand new, totally made-up**, sentences of our own:

```
sentences = ['The white rabbit and Alice ran away',
             'The lion met Dorothy on the road']
inputs = glove_tokenizer(sentences, add_special_tokens=False,
                         return_tensors='pt')['input_ids']
inputs
```

Output

```
tensor([[    2,   300, 12427,     7,  7101,  1423,   422],
        [    2,  6659,   811, 11238,    15,     2,   588]],
        device='cuda:0')
```

Yes, both sentences have the same number of tokens for our convenience :-) Even though they're made-up sentences, I wonder what our model will say about their source being either "*Alice's Adventures in Wonderland*" (positive class) or "*The Wonderful Wizard of Oz*" (negative class):

```
sbs_transf.model.eval()
out = sbs_transf.model(inputs)
# our model outputs logits, so we turn them into probs
torch.sigmoid(out)
```

Output

```
tensor([[0.9888],
        [0.0101]], device='cuda:0', grad_fn=<SigmoidBackward>)
```

The model really thinks that *only the first sentence* comes from "*Alice's Adventures in Wonderland*". To really understand *why* that is, we need to dig into its **attention scores**. The code below retrieves the attention scores for the **first (and only)** layer of our Transformer Encoder:

```
alphas = (sbs_transf.model
          .encoder
          .layers[0]
          .self_attn_heads
          .alphas)
alphas[:, :, 0, :].squeeze()
```

Output

```
tensor([[[2.6334e-01, 6.9912e-02, 1.6958e-01, 1.6574e-01,
          1.1365e-01, 1.3449e-01, 6.6508e-02, 1.6772e-02],
         [2.7878e-05, 2.5806e-03, 2.9353e-03, 1.3467e-01,
          1.7490e-03, 8.5641e-01, 7.3843e-04, 8.8371e-04]],

        [[6.8102e-02, 1.8080e-02, 1.0238e-01, 6.1889e-02,
          6.2652e-01, 1.0388e-02, 1.6588e-02, 9.6055e-02],
         [2.2783e-04, 2.1089e-02, 3.4972e-01, 2.3252e-02,
          5.2879e-01, 3.5840e-02, 2.5432e-02, 1.5650e-02]]],
        device='cuda:0')
```

*"Why are we **slicing the third dimension**? What **is** the third dimension again?"*

In the multi-headed self-attention mechanism, the **scores** have the following shape: **(N, n_heads, L, L)**. We have **two sentences** (`N=2`), **two attention heads** (`n_heads=2`), and our sequence has **eight tokens** (`L=8`).

*"I'm sorry, but our sequences have **seven tokens**, not eight…"*

Yes, that's true. But don't forget about the **special classification token** that was prepended to the sequence of embeddings. That's also the reason why we're slicing the third dimension: that zero index means **we're looking at the attention scores of the special classifier token**. Since we're using the output corresponding to that token to classify the sentences, it's only logical to check what it's paying attention to, right? Moreover, the **first value in each attention score tensor** above represents **how much attention the special classifier token is paying to itself**.

So, what *is* the model paying attention to then? Let's see:

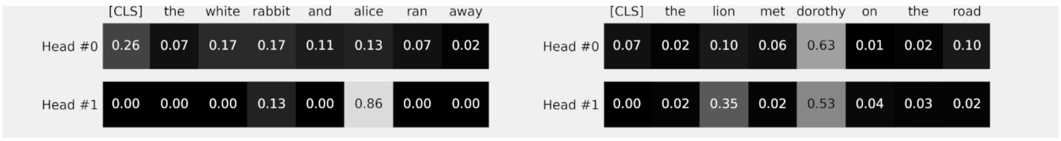


Figure 11.21 - Attention scores

Clearly, the model learned that "*white rabbit*" and "*Alice*" are **strong signs** that a given sentence belongs to "*Alice's Adventures in Wonderland*". Conversely, if there is a "*lion*" or "*Dorothy*" in the sentence, it's likely from "*The Wonderful Wizard of Oz*".

Cool, right? Looks like word embeddings are the best invention since sliced bread! That would indeed be the case if only actual languages were *straightforward and organized*… unfortunately, they are nothing like that.

Let me show you two sentences from "*Alice's Adventures in Wonderland*" (the highlights are mine):

- "*The Hatter was the first to break the silence. `What day of the month is it?' he said, turning to Alice: he had taken his* ***watch*** *out of his pocket, and was looking at it uneasily, shaking it every now and then, and holding it to his ear.*"
- "*Alice thought this a very curious thing, and she went nearer to* ***watch*** *them, and just as she came up to them she heard one of them say, `Look out now, Five! Don't go splashing paint over me like that!*"

In the first sentence, the word "*watch*" is a **noun** and it refers to the **object** The Hatter had taken out of his pocket. In the second sentence, "*watch*" is a **verb** and it refers to what Alice is **doing**. Clearly, two **very different meanings for the same word**.

But, if we look the "*watch*" token up in our vocabulary, we'll **always retrieve the**

same values from the word embeddings, **regardless of the actual meaning** of the word in a sentence.

Can we do better? Of course!

Contextual Word Embeddings

If a single token is not enough, why not take **the whole sentence**, right? Instead of taking a word by itself, we can take **its context too** in order to compute the vector that best represents a word. That was the whole point of **word embeddings**: finding **numerical representation for words** (or tokens).

> *"That's great, but it seems unpractical…"*

You're absolutely right! Trying to build a lookup table for every possible combination of word and context is probably not such a great idea… that's why **contextual word embeddings** won't come from a lookup table anymore but from the **outputs of a model** instead.

I want to introduce you to…

ELMo

Born in 2018, ELMo is able to understand that words may have different meanings in different contexts. If you feed it a sentence, it will give you back embeddings for each one of the words while taking the full context into account.

Embeddings from **L**anguage **Mo**dels (ELMo, for short) was introduced by Peters, M. et al. in their paper "*Deep contextualized word representations*"[181] (2018). The model is a **two-layer bi-directional LSTM encoder** using **4,096 dimensions in its cell states** and it was trained on a **really large corpus** containing **5.5 billion words**. Moreover, ELMo's representations are **character-based**, so it can easily handle unknown (out-of-vocabulary) words.

You can find more details about its implementation, as well as its **pretrained weights** at AllenNLP's ELMo[182] site. You can also check the ELMo section[183] of Lilian Weng's great post: *"Generalized Language Models"*[184].

"Cool, are we loading a pretrained model then?"

Well, we *could*, but ELMo embeddings can be conveniently retrieved using yet another library: flair[185]. Flair is an NLP framework built on top of PyTorch and it offers a **text embedding library** that provides **word embeddings** and **document embeddings** for popular muppets, oops, models like **ELMo** and **BERT**, as well as classical word embeddings like GloVe.

Let's use the two sentences containing the word "*watch*" to illustrate how to use flair to get contextual word embeddings:

```
watch1 = """
The Hatter was the first to break the silence. `What day of the
month is it?' he said, turning to Alice:  he had taken his watch out
of his pocket, and was looking at it uneasily, shaking it every now
and then, and holding it to his ear.
"""

watch2 = """
Alice thought this a very curious thing, and she went nearer to
watch them, and just as she came up to them she heard one of them
say, `Look out now, Five!  Don't go splashing paint over me like
that!
"""

sentences = [watch1, watch2]
```

In flair, every sentence is a `Sentence` object that's easily created using the corresponding text:

```
from flair.data import Sentence
flair_sentences = [Sentence(s) for s in sentences]
flair_sentences[0]
```

Output

```
Sentence: "The Hatter was the first to break the silence . ` What
day of the month is it ? ' he said , turning to Alice : he had taken
his watch out of his pocket , and was looking at it uneasily ,
shaking it every now and then , and holding it to his ear ."   [
Tokens: 58]
```

Our first sentence has 58 tokens. We can use either the `get_token` method or the `tokens` attribute to retrieve a given token:

```
flair_sentences[0].get_token(32)
```

Output

```
Token: 32 watch
```

The `get_token` method assumes indexing starts at **one** while the `tokens` attribute has the typical **zero-based** indexing:

```
flair_sentences[0].tokens[31]
```

Output

```
Token: 32 watch
```

To learn more about flair's `Sentence` object, please check flair's *"Tutorial 1: NLP Base Types"*[186].

Then, we can use these `Sentence` objects to retrieve **contextual word embeddings**. But, first, we need to actually **load ELMo** using `ELMoEmbeddings`:

```
from flair.embeddings import ELMoEmbeddings
elmo = ELMoEmbeddings()
```

```
elmo.embed(flair_sentences)
```

Output

```
[Sentence: "The Hatter was the first to break the silence . ` What
day of the month is it ? ' he said , turning to Alice : he had taken
his watch out of his pocket , and was looking at it uneasily ,
shaking it every now and then , and holding it to his ear ."   [
Tokens: 58],
 Sentence: "Alice thought this a very curious thing , and she went
nearer to watch them , and just as she came up to them she heard one
of them say , ` Look out now , Five ! Do n't go splashing paint over
me like that !"   [ Tokens: 48]]
```

There we go! Every token has its own `embedding` attribute now. Let's check the embeddings for the word "*watch*" in both sentences:

```
token_watch1 = flair_sentences[0].tokens[31]
token_watch2 = flair_sentences[1].tokens[13]
token_watch1, token_watch2
```

Output

```
(Token: 32 watch, Token: 14 watch)
```

```
token_watch1.embedding, token_watch2.embedding
```

Output

```
(tensor([-0.5047, -0.4183,  0.0910,  ..., -0.2228,  0.7794],
        device='cuda:0'),
 tensor([-0.5047, -0.4183,  0.0910,  ...,  0.8352, -0.5018],
        device='cuda:0'))
```

ELMo embeddings are **large**: there are **3,072 dimensions**. The first three values of both embeddings are the same but the last three are not. That's a good start - the same word was assigned two different vectors depending on the context it was found in.

Where do ELMo Embeddings come from?

The embeddings from ELMo are a combination of **classical word embeddings** and **hidden states** from the two-layer bidirectional LSTMs. Since both embeddings and hidden states have 512 dimensions each, it follows that, in each direction, there is **one 512 dimensions embedding** and **two 512 dimensions hidden states** (one for each layer). That's 1,536 dimensions in each direction, and 3,072 dimensions in total.

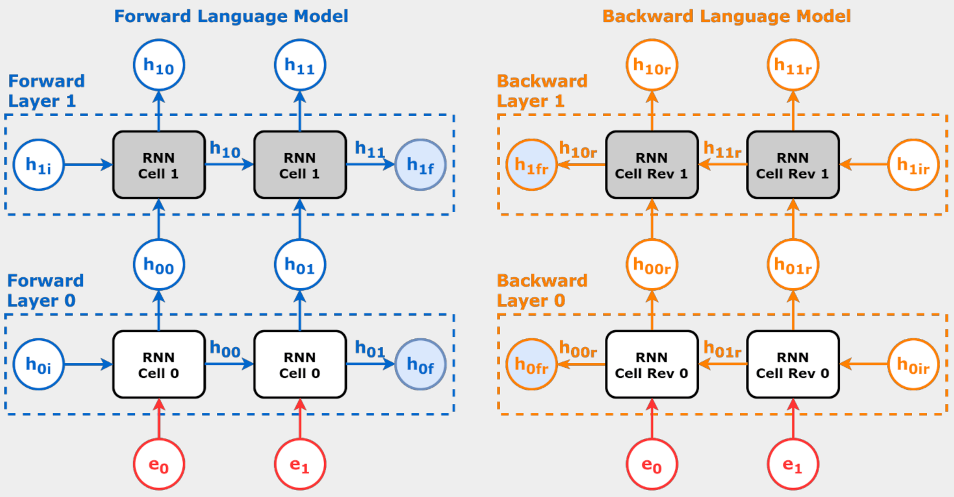


Figure 11.22 - ELMo's two-layer bidirectional LSTMs

The word embeddings are actually duplicated since both LSTMs use the same inputs. From the 3,072 dimensions, the first two chunks of 512 dimensions are actually identical:

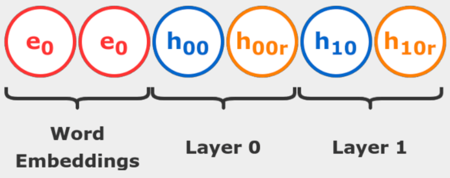


Figure 11.23 - ELMo embeddings

```
token_watch1.embedding[0], token_watch1.embedding[512]
```

Output

```
(tensor(-0.5047, device='cuda:0'), tensor(-0.5047, device
='cuda:0'))
```

Since the *classical word embeddings* are *context-independent*, it also means that both uses of "*watch*" have exactly the same values in their first 1,024 dimensions:

```
(token_watch1.embedding[:1024] ==
 token_watch2.embedding[:1024]).all()
```

Output

```
tensor(True, device='cuda:0')
```

If we'd like to find out *how similar* they are to each other, we can use **cosine similarity**:

```
similarity = nn.CosineSimilarity(dim=0, eps=1e-6)
similarity(token_watch1.embedding, token_watch2.embedding)
```

```
tensor(0.5949, device='cuda:0')
```

Even though the first 1,024 values were identical, it turns out that the two words are *not* so similar after all. Contextual word embeddings for the win :-)

To get word embeddings for all tokens in a sentence, we can simply **stack** them up:

Helper method to retrieve embeddings

```python
def get_embeddings(embeddings, sentence):
    sent = Sentence(sentence)
    embeddings.embed(sent)
    return torch.stack(
        [token.embedding for token in sent.tokens]
    ).float()
```

```python
get_embeddings(elmo, watch1)
```

Output

```
tensor([[-0.3288,  0.2022, -0.5940,  ...,  1.0606,  0.2637],
        [-0.7142,  0.4210, -0.9504,  ..., -0.6684,  1.7245],
        [ 0.2981, -0.0738, -0.1319,  ...,  1.1165,  0.6453],
        ...,
        [ 0.0475,  0.2325, -0.2013,  ..., -0.5294, -0.8543],
        [ 0.1599,  0.6898,  0.2946,  ...,  0.9584,  1.0337],
        [-0.8872, -0.2004, -1.0601,  ..., -0.0841,  0.0618]],
       device='cuda:0')
```

The returned tensor has 58 embeddings of 3,072 dimensions each.

For more details on ELMo embeddings, please check flair's *"ELMo Embeddings"*[187] and *"Tutorial 4: List of All Word Embeddings"*[188].

GloVe

GloVe embeddings are not contextual, as you already know, but they can also be easily retrieved using flair's `WordEmbeddings`:

```
from flair.embeddings import WordEmbeddings
glove_embedding = WordEmbeddings('glove')
```

Now, let's retrieve the word embeddings for our sentences but first, and that's **very important**, we need to create **new `Sentence` objects** for them:

```
new_flair_sentences = [Sentence(s) for s in sentences]
glove_embedding.embed(new_flair_sentences)
```

Output

```
[Sentence: "The Hatter was the first to break the silence . `
What day of the month is it ? ' he said , turning to Alice :
he had taken his watch out of his pocket , and was looking at
it uneasily , shaking it every now and then , and holding it
to his ear ."    [ Tokens: 58],
Sentence: "Alice thought this a very curious thing , and she
went nearer to watch them , and just as she came up to them
she heard one of them say , ` Look out now , Five ! Do n't go
splashing paint over me like that !"    [ Tokens: 48]]
```

Never reuse a `Sentence` object to retrieve **different word embeddings**! The `embedding` attribute may be **partially overwritten** (depending on the number of dimensions), and you may end up with **mixed embeddings** (e.g., 3,072 dimensions from ELMo, but the first 100 values being overwritten by GloVe embeddings).

Since GloVe is not contextual, the word "*watch*" will have the same embedding regardless of which sentence you retrieve it from:

```
torch.all(new_flair_sentences[0].tokens[31].embedding ==
          new_flair_sentences[1].tokens[13].embedding)
```

Output

```
tensor(True, device='cuda:0')
```

For more details on classic word embeddings, please check flair's *"Tutorial 3: Word Embeddings"*[189] and *"Classic Word Embeddings"*[190].

BERT

The general idea of obtaining **contextual word embeddings using a language model** introduced by ELMo still holds true for BERT. While ELMo is only a muppet, **BERT** is both **muppet and Transformer** (such a bizarre sentence to write!).

BERT, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers, is a model based on a **Transformer Encoder**. We'll *skip* more details about its architecture for now (don't worry, BERT has a full section of its own) and use it to get contextual word embeddings only (just like we did with ELMo).

First, we need to **load BERT** in flair using `TransformerWordEmbeddings`:

```
from flair.embeddings import TransformerWordEmbeddings
bert = TransformerWordEmbeddings('bert-base-uncased', layers='-1')
```

By the way, flair uses HuggingFace models under the hood :-) So, you can load **any pretrained model**[191] to generate embeddings for you.

In the example above, we're using the traditional `bert-base-uncased` to generate context word embeddings using BERT's **last layer (-1)**.

Next, we can use the same `get_embeddings` function to get the stacked embeddings for every token in a sentence:

```
embed1 = get_embeddings(bert, watch1)
embed2 = get_embeddings(bert, watch2)
embed2
```

Output

```
tensor([[ 0.6554, -0.3799, -0.2842,  ...,  0.8865,  0.4760],
        [-0.1459, -0.0204, -0.0615,  ...,  0.5052,  0.3324],
        [-0.0436, -0.0401, -0.0135,  ...,  0.5231,  0.9067],
        ...,
        [-0.2582,  0.6933,  0.2688,  ...,  0.0772,  0.2187],
        [-0.1868,  0.6398, -0.8127,  ...,  0.2793,  0.1880],
        [-0.1021,  0.5222, -0.7142,  ...,  0.0600, -0.1419]])
```

Then, let's compare the embeddings for the word "*watch*" in both sentences once again:

```
bert_watch1 = embed1[31]
bert_watch2 = embed2[13]
print(bert_watch1, bert_watch2)
```

Output

```
(tensor([ 8.5760e-01,  3.5888e-01, -3.7825e-01, -8.3564e-01,
          ...,
          2.0768e-01,  1.1880e-01,  4.1445e-01]),
 tensor([-9.8449e-02,  1.4698e+00,  2.8573e-01, -3.9569e-01,
          ...,
          3.1746e-01, -2.8264e-01, -2.1325e-01]))
```

Well, they look *more different* from one another now. But, are they, really?

```
similarity = nn.CosineSimilarity(dim=0, eps=1e-6)
similarity(bert_watch1, bert_watch2)
```

Output

```
tensor(0.3504, device='cuda:0')
```

Indeed, they have an even lower similarity now.

For more details on Transformer word embeddings, please check flair's "*Transformer Embeddings*"[192].

In the "*Pretrained PyTorch Embeddings*" section we **averaged (classical) word embeddings** to get a **single vector for each sentence**. We *could* do the same once again using **contextual word embeddings** instead. But we don't have to, we can use…

Document Embeddings

We can use pretrained models to generate embeddings for **whole documents** instead of single words, thus eliminating the need to average word embeddings. In our case, a *document* is a **sentence**:

```
documents = [Sentence(watch1), Sentence(watch2)]
```

To actually get the embeddings, we use `TransformerDocumentEmbeddings` in the same way as the others:

```
from flair.embeddings import TransformerDocumentEmbeddings
bert_doc = TransformerDocumentEmbeddings('bert-base-uncased')
bert_doc.embed(documents)
```

Output

```
[Sentence: "The Hatter was the first to break the silence . ` What
day of the month is it ? ' he said , turning to Alice : he had taken
his watch out of his pocket , and was looking at it uneasily ,
shaking it every now and then , and holding it to his ear ."   [
Tokens: 58],
 Sentence: "Alice thought this a very curious thing , and she went
nearer to watch them , and just as she came up to them she heard one
of them say , ` Look out now , Five ! Do n't go splashing paint over
me like that !"   [ Tokens: 48]]
```

Now, each **document** (a `Sentence` object) will have its **own, overall, embedding**:

```
documents[0].embedding
```

Output

```
tensor([-6.4245e-02,  3.5365e-01, -2.4962e-01, -5.3912e-01,
        -1.9917e-01, -2.7712e-01,  1.6942e-01,  1.0867e-01,
        ...
         7.4661e-02, -3.4777e-01,  1.5740e-01,  3.4407e-01,
        -5.0272e-01,  1.7432e-01,  7.9398e-01,  7.3562e-01],
       device='cuda:0',
       grad_fn=<CatBackward>)
```

Notice that the individual tokens don't get their own embeddings anymore:

```
documents[0].tokens[31].embedding
```

Output

```
tensor([], device='cuda:0')
```

We can leverage this fact to slightly modify the `get_embeddings` function so it works with both word and document embeddings:

Helper method to retrieve embeddings

```
def get_embeddings(embeddings, sentence):
    sent = Sentence(sentence)
    embeddings.embed(sent)
    if len(sent.embedding):
        return sent.embedding.float()
    else:
        return torch.stack(
            [token.embedding for token in sent.tokens]
        ).float()
```

```
get_embeddings(bert_doc, watch1)
```

Output

```
tensor([-6.4245e-02,  3.5365e-01, -2.4962e-01, -5.3912e-01,
        -1.9917e-01, -2.7712e-01,  1.6942e-01,  1.0867e-01,
        ...
         7.4661e-02, -3.4777e-01,  1.5740e-01,  3.4407e-01,
        -5.0272e-01,  1.7432e-01,  7.9398e-01,  7.3562e-01],
       device='cuda:0',
       grad_fn=<CatBackward>)
```

For more details on document embeddings, please check flair's *"Tutorial 5: Document Embeddings"*[193].

We can **revisit the `Sequential` model** from the "*Word Embeddings*" section and modify it to use **contextual word embeddings** instead. But, first, we need to *change a bit the datasets* as well.

Model III - Preprocessed Embeddings

Data Preparation

Before, the **features** were a sequence of **token ids**, which were used to **look embeddings up** in the embedding layer and return the corresponding **bag-of-embeddings** (that *was* a document embedding too, although less sophisticated).

Now, we're *outsourcing* these steps to BERT and getting **document embeddings** directly from it. It turns out, using a pretrained BERT model to retrieve document embeddings is a **preprocessing step** in this setup. Consequently, our model is going to be nothing else than a **simple classifier**.

"*Let the preprocessing begin!*"

Maximus Decimus Meridius

The idea is to use `get_embeddings` for each and every sentence in our datasets in order to retrieve their corresponding document embeddings. The HuggingFace's dataset allows us to easily do that using its `map` method to generate a new column:

Data Preparation

```python
train_dataset_doc = train_dataset.map(
    lambda row: {'embeddings': get_embeddings(bert_doc,
                                              row['sentence'])}
)
test_dataset_doc = test_dataset.map(
    lambda row: {'embeddings': get_embeddings(bert_doc,
                                              row['sentence'])}
)
```

Moreover, we need the embeddings to be returned as PyTorch tensors:

Data Preparation

```python
train_dataset_doc.set_format(type='torch',
                             columns=['embeddings', 'labels'])
test_dataset_doc.set_format(type='torch',
                            columns=['embeddings', 'labels'])
```

We can easily get the embeddings for all sentences in our dataset now:

```python
train_dataset_doc['embeddings']
```

Output

```
tensor([[-0.2932,  0.2595, -0.1252,  ...,  0.2998,  0.1157],
        [ 0.4934,  0.0129, -0.1991,  ...,  0.6320,  0.7036],
        [-0.6256, -0.3536, -0.4682,  ...,  0.2467,  0.6108],
        ...,
        [-0.5786,  0.0274, -0.1081,  ...,  0.0329,  0.9563],
        [ 0.1244,  0.3181,  0.0352,  ...,  0.6648,  0.9231],
        [ 0.2124,  0.6195, -0.2281,  ...,  0.4346,  0.6358]],
       dtype=torch.float64)
```

Next, we build the datasets the usual way:

Data Preparation

```python
train_dataset_doc = TensorDataset(
    train_dataset_doc['embeddings'].float(),
    train_dataset_doc['labels'].view(-1, 1).float())
generator = torch.Generator()
train_loader = DataLoader(
    train_dataset_doc, batch_size=32,
    shuffle=True, generator=generator
)
test_dataset_doc = TensorDataset(
    test_dataset_doc['embeddings'].float(),
    test_dataset_doc['labels'].view(-1, 1).float())
test_loader = DataLoader(
    test_dataset_doc, batch_size=32, shuffle=True)
```

Model Configuration & Training

We're using pretty much the **same `Sequential` model** as before, except for the fact that it doesn't have an embedding layer anymore, and we're using **only three hidden units** instead of 128:

Model Configuration

```
torch.manual_seed(41)
model = nn.Sequential(
    # Classifier
    nn.Linear(bert_doc.embedding_length, 3),
    nn.ReLU(),
    nn.Linear(3, 1)
)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

"Isn't that too few? Three?! Really?"

Really, it isn't too few... if you try using 128 like in the previous model, it will *immediately* overfit over a *single epoch*. Given the **embedding length** (768), the model gets **overparameterized** (a situation where there are **many more parameters than data points**) and it ends up **memorizing the training set**.

This is a **simple feed-forward classifier** with a **single hidden layer**. It doesn't get *much* simpler than that!

Model Training

```
sbs_doc_emb = StepByStep(model, loss_fn, optimizer)
sbs_doc_emb.set_loaders(train_loader, test_loader)
sbs_doc_emb.train(20)
```

```
fig = sbs_doc_emb.plot_losses()
```

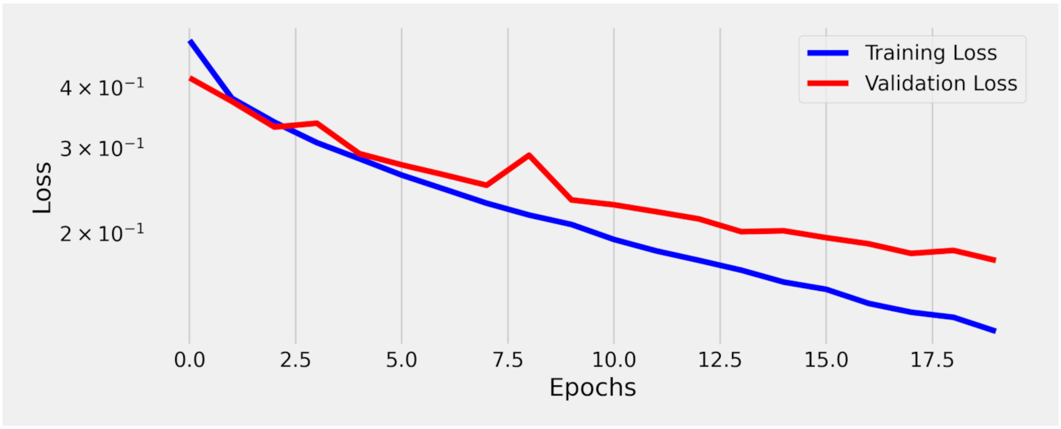


Figure 11.24 - Losses - Simple classifier with BERT embeddings

OK, it's still not overfitting… but can it deliver good predictions? You betcha!

```
StepByStep.loader_apply(test_loader, sbs_doc_emb.correct)
```

Output

```
tensor([[424, 440],
        [310, 331]])
```

That's 95.20% accuracy on the validation (test) set! Quite impressive for a model with only *three hidden units*, I might say.

Now, imagine what can be accomplished if we **fine-tune the actual BERT model** instead! Right? Right?

BERT

BERT, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers, is a model based on a **Transformer Encoder**. It was introduced by Devlin, J. et al. in their paper "*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*"[194] (2019).

The original BERT model was trained on two huge corpora: BookCorpus[195] (composed of 800M words in 11,038 unpublished books) and English Wikipedia[196] (2.5B words). It has **twelve "layers"** (the original Transformer had only six), **twelve attention heads**, and **768 hidden dimensions**, totaling **110 million parameters**.

If that's too big for your GPU, though, don't worry: there are **many** different versions of BERT for all tastes and budgets, and you can find them in Google Research's BERT repository[197].

You can also check BERT's documentation[198] and model card[199] available at HuggingFace for a quick overview of the model and its training procedure.

For a general overview of BERT, please check Jay Alammar's excellent posts on the topic: *"The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)"*[200] and *"A Visual Guide to Using BERT for the First Time"*[201].

Let's create our first BERT model by loading the pretrained weights for `bert-base-uncased`:

```
from transformers import BertModel
bert_model = BertModel.from_pretrained('bert-base-uncased')
```

We can inspect the pretrained model's configuration:

```
bert_model.config
```

Output

```
BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "type_vocab_size": 2,
  "vocab_size": 30522
}
```

Some of the items are easily recognizable: `hidden_size` (768), `num_attention_heads` (12), and `num_hidden_layer` (12). Some of the items will be discussed soon: `vocab_size` (30,522), and `max_position_embeddings` (512, the maximum sequence length). There are additional parameters used for training, like dropout probabilities and the architecture used.

Our model needs inputs, and those require…

AutoModel

If you want to quickly try *different models* without having to import their corresponding classes, you can use HuggingFace's `AutoModel` instead:

```
from transformers import AutoModel
auto_model = AutoModel.from_pretrained('bert-base-uncased')
print(auto_model.__class__)
```

Output

```
<class 'transformers.modeling_bert.BertModel'>
```

As you can see, it *infers* the correct model class based on the name of the model you're loading, e.g. `bert-base-uncased`.

Tokenization

The **tokenization** is a **pre-processing step** and, since we'll be using a **pretrained BERT model**, we need to use the **same tokenizer** that was used during pre-training. For each pretrained model available in HuggingFace there is an accompanying pretrained tokenizer as well.

Let's create our first **real** BERT tokenizer (instead of the fake ones we create to handle our own vocabulary):

```
from transformers import BertTokenizer
bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
len(bert_tokenizer.vocab)
```

Output

```
30522
```

It seems BERT’s vocabulary has **only 30,522 tokens**.

"Isn’t that too few?"

It *would* be if it weren’t for the fact that the **tokens are not (only) words**, but may also be **word pieces**.

"What is a "word piece"?"

It *literally* is a piece of a word. This is better understood with an example: let’s say that a particular word - "*inexplicably*" - is not so frequently used, thus not making it to the vocabulary. Before, we used the *special unknown token* to cover for words that were absent from our vocabulary. But that approach is **less than ideal**: every time a word is replaced by a `[UNK]` token, some **information gets lost**. We surely can do better than that.

So, what if we **disassemble** an unknown word into **its components (the word pieces)**? Our formerly unknown word, "*inexplicably*", can be disassembled into **five word pieces**: `inexplicably = in + ##ex + ##pl + ##ica + ##bly`.

Every word piece is prefixed with `##` to indicate that it does not stand on its own as a word.

Given enough word pieces in a vocabulary, it will be able to **represent every unknown word** using a **concatenation of word pieces**. Problem solved! That’s what BERT’s pretrained tokenizer does.

For more details on the **WordPiece** tokenizer, as well as other **subword tokenizers** like **Byte-Pair Encoding (BPE)** and **SentencePiece**, please check HuggingFace's Summary of the Tokenizers[202] and Cathal Horan's great post *"Tokenizers: How machines read"*[203] on FloydHub.

Let's tokenize a pair of sentences using BERT's word piece tokenizer:

```
sentence1 = 'Alice is inexplicably following the white rabbit'
sentence2 = 'Follow the white rabbit, Neo'
tokens = bert_tokenizer(sentence1, sentence2, return_tensors='pt')
tokens
```

Output

```
{'input_ids': tensor([[  101,  5650,  2003,  1999, 10288, 24759,
5555,  6321,  2206,  1996, 2317, 10442,   102,  3582,  1996,  2317,
10442,  1010,  9253,   102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1]])}
```

Notice that, since there are **two sentences**, the `token_type_ids` have two distinct values (zero and one) that work as the **sentence index** corresponding to the sentence each token belongs to. Hold this thought, because we're using this information in the next section.

To actually see the word pieces, it's easier to convert the input ids back into tokens:

```
print(bert_tokenizer.convert_ids_to_tokens(tokens['input_ids'][0]))
```

Output

```
['[CLS]', 'alice', 'is', 'in', '##ex', '##pl', '##ica', '##bly',
'following', 'the', 'white', 'rabbit', '[SEP]', 'follow', 'the',
'white', 'rabbit', ',', 'neo', '[SEP]']
```

There it is: "*inexplicably*" got disassembled into its word pieces, the separator token `[SEP]` got inserted between the two sentences (and at the end as well), and there is a classification token `[CLS]` at the start.

AutoTokenizer

If you want to quickly try *different tokenizers* without having to import their corresponding classes, you can use HuggingFace's `AutoTokenizer` instead:

```
from transformers import AutoTokenizer
auto_tokenizer = AutoTokenizer.from_pretrained('bert-base-
uncased')
print(auto_tokenizer.__class__)
```

Output

```
<class 'transformers.tokenization_bert.BertTokenizer'>
```

As you can see, it *infers* the correct model class based on the name of the model you're loading, e.g. `bert-base-uncased`.

Input Embeddings

Once the sentences are tokenized, we can use their tokens' ids to look the corresponding embeddings up as usual. These are the **word/token embeddings**. So far, our models used these embeddings (or a bag of them) as their **only input**.

But BERT, being a Transformer Encoder, also needs **positional information**. In Chapter 9 we used **positional encoding**, but BERT uses **position embeddings** instead.

"What's the difference between encoding and embedding?"

While the **position encoding** we used in the past had **fixed values** for each position, the **position embeddings** are **learned by the model** (like any other embedding layer). The number of entries in this lookup table is given by the **maximum length of the sequence**.

And there is more! BERT also adds a **third** embedding, namely, **segment embedding**, that is a **position embedding at the sentence level** (since inputs may have either one or two sentences). That's what the **`token_type_ids`** produced by the tokenizer are good for: they work as a *sentence index* for each token.

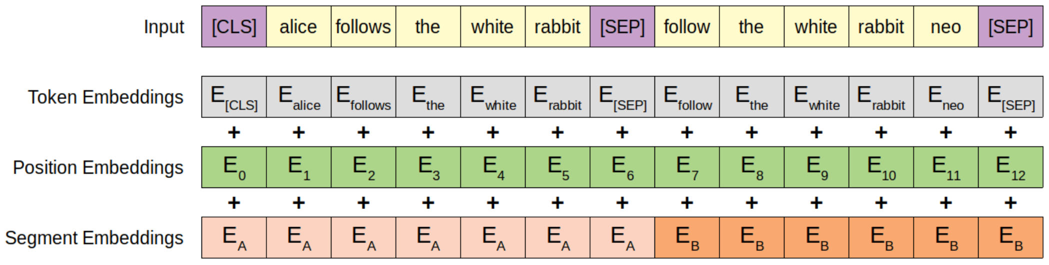

Input
[CLS]
alice
follows
the
white
rabbit
[SEP]
follow
the
white
rabbit
neo
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{alice} $E_{follows}$ E_{the} E_{white} E_{rabbit} $E_{[SEP]}$ E_{follow} E_{the} E_{white} E_{rabbit} E_{neo} $E_{[SEP]}$
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10} E_{11} E_{12}
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B E_B

Figure 11.25 - BERT's input embeddings

Talking (or writing) is cheap, though, so let's take a look under BERT's hood:

```
input_embeddings = bert_model.embeddings
```

Output

```
BertEmbeddings(
  (word_embeddings): Embedding(30522, 768, padding_idx=0)
  (position_embeddings): Embedding(512, 768)
  (token_type_embeddings): Embedding(2, 768)
  (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
```

The three embeddings are there: word, position, and segment (named `token_type_embeddings`). Let's go over each one of them:

```
token_embeddings = input_embeddings.word_embeddings
token_embeddings
```

Output

```
Embedding(30522, 768, padding_idx=0)
```

The word/token embedding layer has 30,522 entries, the size of BERT's vocabulary, and it has 768 hidden dimensions. As usual, embeddings will be returned by each token id in the input:

```
input_token_emb = token_embeddings(tokens['input_ids'])
input_token_emb
```

Output

```
tensor([[[ 1.3630e-02, -2.6490e-02, ..., 7.1340e-03,  1.5147e-02],
         ...,
         [-1.4521e-02, -9.9615e-03, ..., 4.6379e-03, -1.5378e-03]]],
       grad_fn=<EmbeddingBackward>)
```

Since each input may have up to 512 tokens, the position embedding layer has exactly that number of entries:

```
position_embeddings = input_embeddings.position_embeddings
position_embeddings
```

Output

```
Embedding(512, 768)
```

Each sequentially numbered position, up to the total length of the input, will return its corresponding embedding:

```
position_ids = torch.arange(512).expand((1, -1))
position_ids
```

Output

```
tensor([[  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
         ...
         504, 505, 506, 507, 508, 509, 510, 511]])
```

```
seq_length = tokens['input_ids'].size(1)
input_pos_emb = position_embeddings(position_ids[:, :seq_length])
input_pos_emb
```

Output

```
tensor([[[ 1.7505e-02, -2.5631e-02, ...,  1.5441e-02],
         ...,
         [-3.4622e-04, -8.3709e-04, ..., -5.7741e-04]]],
       grad_fn=<EmbeddingBackward>)
```

Then, since there can only be either one or two sentences in the input, the segment embedding layer has **only two entries**:

```
segment_embeddings = input_embeddings.token_type_embeddings
segment_embeddings
```

Output

```
Embedding(2, 768)
```

For these embeddings, BERT will use the `token_type_ids` returned by the tokenizer:

```
input_seg_emb = segment_embeddings(tokens['token_type_ids'])
input_seg_emb
```

Output

```
tensor([[[ 0.0004,  0.0110,  0.0037,  ..., -0.0034, -0.0086],
         [ 0.0004,  0.0110,  0.0037,  ..., -0.0034, -0.0086],
         [ 0.0004,  0.0110,  0.0037,  ..., -0.0034, -0.0086],
         ...,
         [ 0.0011, -0.0030, -0.0032,  ..., -0.0052, -0.0112],
         [ 0.0011, -0.0030, -0.0032,  ..., -0.0052, -0.0112],
         [ 0.0011, -0.0030, -0.0032,  ..., -0.0052, -0.0112]]],
       grad_fn=<EmbeddingBackward>)
```

Since the first tokens, up to and including the first separator, belong to the first sentence, they will all have the **same, first, segment embedding values**. The tokens after the first separator, up to and including the last token, will have the **same, second, segment embedding values**.

Finally, BERT adds up **all three embeddings** (token, position, and segment):

```
input_emb = input_token_emb + input_pos_emb + input_seg_emb
input_emb
```

Output

```
tensor([[[ 0.0316, -0.0411, -0.0564,  ...,  0.0044,  0.0219],
         [-0.0615, -0.0750, -0.0107,  ...,  0.0482, -0.0277],
         [-0.0469, -0.0156, -0.0336,  ...,  0.0135,  0.0109],
         ...,
         [-0.0081, -0.0051, -0.0172,  ..., -0.0103,  0.0083],
         [-0.0425, -0.0756, -0.0414,  ..., -0.0180, -0.0060],
         [-0.0138, -0.0138, -0.0194,  ..., -0.0011, -0.0133]]],
       grad_fn=<AddBackward0>)
```

It will still **layer normalize** the embeddings and apply **dropout** to them, but that's it, these are the **inputs** BERT uses.

Now, let’s take a look at its…

Pretraining Tasks

Masked Language Model (MLM)

BERT is said to be an **autoencoding model** because it **is a Transformer Encoder** and because it was **trained to "reconstruct" sentences from corrupted inputs** (it does *not* reconstruct the *entire input* but predicts the corrected words instead). That’s the **Masked Language Model (MLM)** pretraining task.

In the "*Language Model*" section we saw that the goal of a language model is to estimate the **probability of a token** or sequence of tokens or, simply put, to predict the tokens more likely to **fill in a blank**. That looks like a perfect task for a *Transformer Decoder*, right?

"*But BERT is an **encoder**…*"

Well, yeah, but who said the **blank must be at the end**? In the Continuous Bag-of-Words (CBoW) model, the **blank was the word in the center**, and the remaining words were the *context*. In a way, that’s what the **MLM task** is doing: it is **randomly choosing words to be masked as blanks** in a sentence. BERT then tries to predict the correct words that fill in the blanks.

Actually, it’s a bit more structured than that:

- in 80% of the time, it **masks 15% of the tokens** at random: "*Alice followed the [MASK] rabbit*"
- in 10% of the time, it **replaces 15% of the tokens** with some other random word: "*Alice followed the watch rabbit*"
- in the remaining 10% of the time, the **tokens are unchanged**: "*Alice followed the white rabbit*"

The **target** is the original sentence: "*Alice followed the white rabbit*". This way, the model effectively learns to reconstruct the original sentence from corrupted inputs (containing missing - masked - or randomly replaced words).

This is the perfect use case (besides *padding*) for the **source mask** argument of the **Transformer Encoder**.

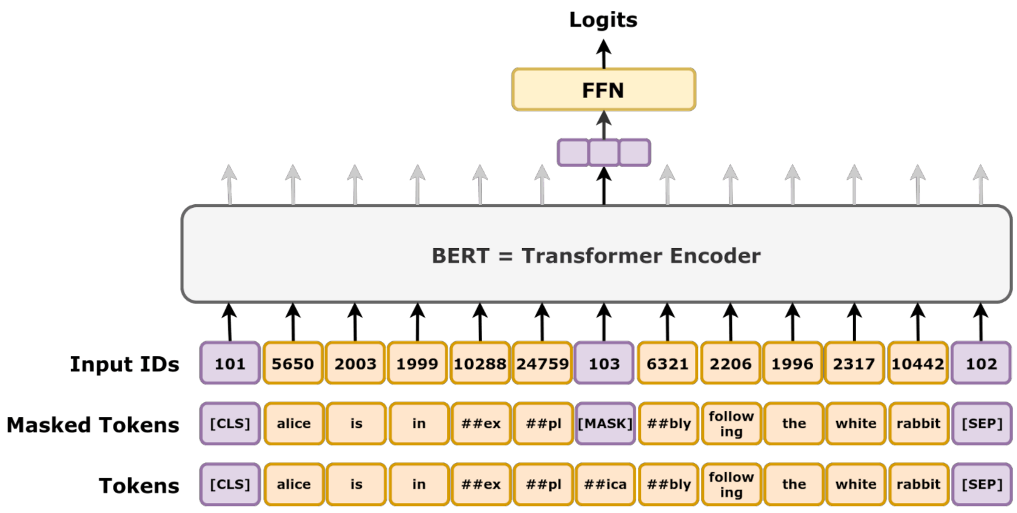


Figure 11.26 - Pretraining task - Masked Language Model (MLM)

Also, notice that BERT **computes logits** for the **randomly masked inputs only**. The remaining inputs are not even considered for computing the loss.

*"OK, but **how** can we randomly replace tokens like that?"*

One alternative, similar to the way we do data augmentation for images, would be to *implement a custom dataset* that performs the replacements on the fly in the `get_item` method. There is a **better alternative**, though: using a **collate function** or, better yet, a **data collator**. There's a data collator that performs the replacement procedure prescribed by BERT: `DataCollatorForLanguageModeling`.

Let's see an example of it in action, starting with an input sentence:

```
sentence = 'Alice is inexplicably following the white rabbit'
tokens = bert_tokenizer(sentence)
tokens['input_ids']
```

Output

```
[101, 5650, 2003, 1999, 10288, 24759, 5555, 6321, 2206, 1996, 2317,
10442, 102]
```

Then, let's create an instance of the data collator and apply it to our mini-batch of one:

```
from transformers import DataCollatorForLanguageModeling
torch.manual_seed(41)
data_collator = DataCollatorForLanguageModeling(
    tokenizer=bert_tokenizer, mlm_probability=0.15
)
mlm_tokens = data_collator([tokens])
mlm_tokens
```

Output

```
{'input_ids': tensor([[  101,  5650,  2003,  1999, 10288, 24759,
103,  6321,  2206,  1996, 2317, 10442,   102]]),
 'labels': tensor([[-100, -100, -100, -100, -100, -100, 5555, -100,
-100, -100, -100, -100, -100]])}
```

If you look closely, you'll see that the **seventh token** (`5555` in the original input) was **replaced** by some **other token** (`103`). Moreover, the **labels** contain the **replaced tokens in their original positions** (and `-100` everywhere else to indicate these tokens are irrelevant for computing the loss). It's actually easier to visualize the difference if we convert the ids back to tokens:

```
print(bert_tokenizer.convert_ids_to_tokens(tokens['input_ids']))
print(bert_tokenizer.convert_ids_to_tokens(
    mlm_tokens['input_ids'][0]
))
```

Output

```
['[CLS]', 'alice', 'is', 'in', '##ex', '##pl', '##ica', '##bly',
'following', 'the', 'white', 'rabbit', '[SEP]']
['[CLS]', 'alice', 'is', 'in', '##ex', '##pl', '[MASK]', '##bly',
'following', 'the', 'white', 'rabbit', '[SEP]']
```

See? The seventh token (`##ica`) got **masked**!

We're **not using collators** in our example but they can be used together with HuggingFace's `Trainer` (more on that in the "*Fine-Tuning with HuggingFace*" section) if you're into training some BERT from scratch on the MLM task :-)

But that's *not* the only thing that BERT is trained to do…

Next Sentence Prediction (NSP)

The second pretraining task is a **binary classification task**: BERT was **trained to predict if a second sentence is actually the next sentence** in the original text or not. The purpose of this task is to give BERT the ability to understand the **relationship between sentences**, which can be useful for some of the tasks BERT can be **fine-tuned** for, like **question answering**.

So, it takes **two sentences** as inputs (with the *special separator token* `[SEP]` between them):

- in 50% of the time, the **second sentence is indeed the next sentence** (the positive class)

- in 50% of the time, the **second sentence is a randomly chosen one** (the negative class)

This task uses the **special classification token `[CLS]`**, taking the values of the corresponding **final hidden state** as features for a classifier. For example, let's take two sentences and tokenize them:

```
sentence1 = 'alice follows the white rabbit'
sentence2 = 'follow the white rabbit neo'
bert_tokenizer(sentence1, sentence2, return_tensors='pt')
```

Output

```
{'input_ids': tensor([[  101,  5650,  4076,  1996,  2317, 10442,
102,  3582,  1996,  2317, 10442,  9253,   102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1]])}
```

If these two sentences were the input of the NSP task, that's how BERT's inputs and outputs would look like:

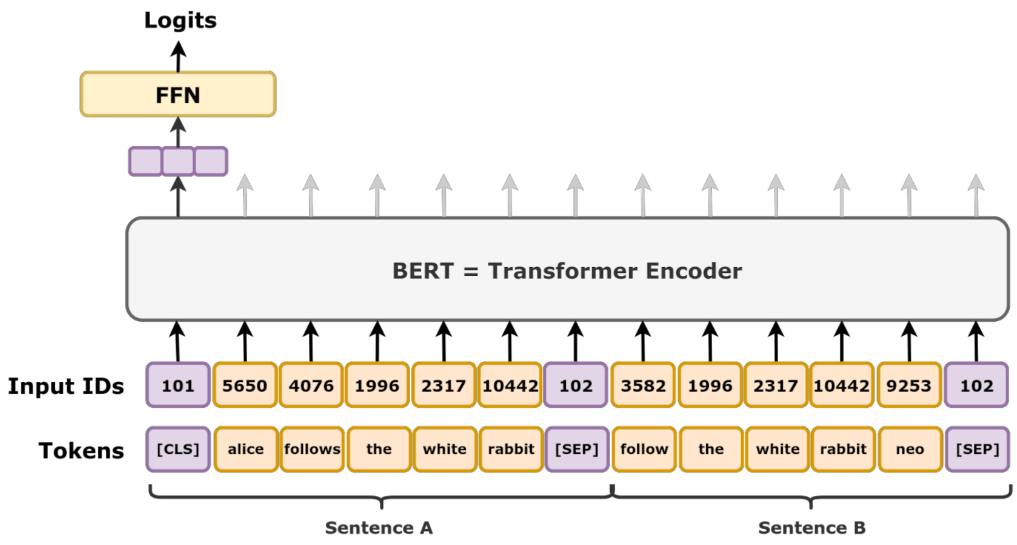


Figure 11.27 - Pretraining task - Next Sentence Prediction (NSP)

The final hidden state is actually *further processed* by a **pooler** (composed of a linear layer and a hyperbolic tangent activation function) before being fed to the classifier (FFN, feed-forward network, in the figure above):

```
bert_model.pooler
```

Output

```
BertPooler(
  (dense): Linear(in_features=768, out_features=768, bias=True)
  (activation): Tanh()
)
```

Outputs

We've seen the many embeddings BERT uses as inputs, but we're more interested in its **outputs**, like the **contextual word embeddings**, right?

By the way, BERT's outputs are always **batch-first**, that is, their shape is **`(mini-batch size, sequence length, hidden_dimensions)`**.

Let's retrieve the embeddings for the words in the first sentence of our training set:

```
sentence = train_dataset[0]['sentence']
sentence
```

Output

```
'And, so far as they knew, they were quite right.'
```

First, we need to **tokenize** it:

```
tokens = bert_tokenizer(sentence,
                        padding='max_length',
                        max_length=30,
                        truncation=True,
                        return_tensors="pt")
tokens
```

Output

```
{'input_ids': tensor([[ 101, 1998, 1010, 2061, 2521, 2004, 2027,
2354, 1010, 2027, 2020, 3243, 2157, 1012,  102,    0,    0,    0,
0,    0,    0,    0,    0,    0, 0,    0,    0,    0,    0,    0]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

The tokenizer is **padding** the sentence up to the maximum length (only 30 in this example to more easily visualize the outputs) and this is reflected on the `attention_mask` as well. We'll use both `input_ids` and `attention_mask` as inputs to our BERT model (the `token_type_ids` are irrelevant here because there is only one sentence).

The BERT model may take many other arguments, and we're using three of them to get richer outputs:

```
bert_model.eval()
out = bert_model(input_ids=tokens['input_ids'],
                 attention_mask=tokens['attention_mask'],
                 output_attentions=True,
                 output_hidden_states=True,
                 return_dict=True)
out.keys()
```

Output

```
odict_keys(['last_hidden_state', 'pooler_output', 'hidden_states',
'attentions'])
```

Let's see what's inside each one of these four outputs:

- `last_hidden_state` is returned by default and it is the most important output of all: it contains the **final hidden states** for each and every token in the input and they can be used as **contextual word embeddings**.

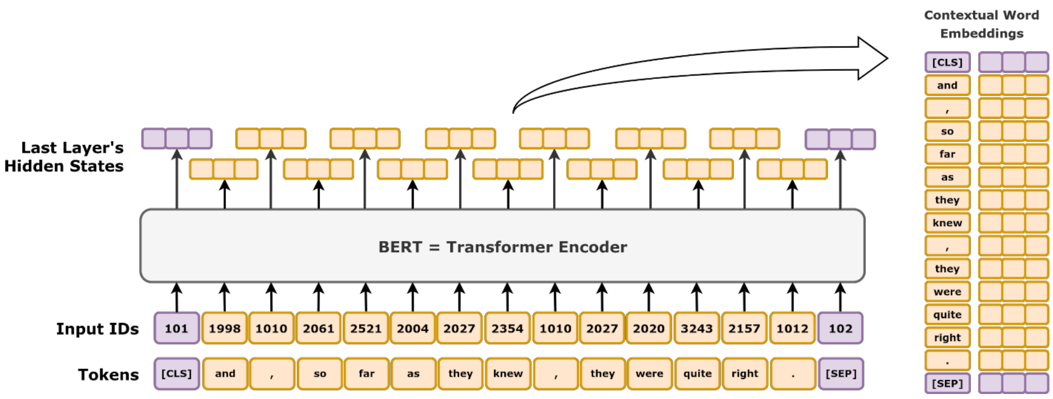

Last Layer's Hidden States
BERT = Transformer Encoder
Input IDs
101 1998 1010 2061 2521 2004 2027 2354 1010 2027 2020 3243 2157 1012 102
Tokens
[CLS] and , so far as they knew , they were quite right . [SEP]
Contextual Word Embeddings

Figure 11.28 - Word embeddings from BERT's last layer

Don't forget that the **first token** is the **special classification token** `[CLS]` and that there may be padding (`[PAD]`) and separator (`[SEP]`) tokens as well!

```
last_hidden_batch = out['last_hidden_state']
last_hidden_sentence = last_hidden_batch[0]
# Removes hidden states for [PAD] tokens using the mask
mask = tokens['attention_mask'].squeeze().bool()
embeddings = last_hidden_sentence[mask]
# Removes embeddings for the first [CLS] and last [SEP] tokens
embeddings[1:-1]
```

Output

```
tensor([[ 0.0100,  0.8575, -0.5429,  ...,  0.4241, -0.2035],
        [-0.3705,  1.1001,  0.3326,  ...,  0.0656, -0.5644],
        [-0.2947,  0.5797,  0.1997,  ..., -0.3062,  0.6690],
        ...,
        [ 0.0691,  0.7393,  0.0552,  ..., -0.4896, -0.4832],
        [-0.1566,  0.6177,  0.1536,  ...,  0.0904, -0.4917],
        [ 0.7511,  0.3110, -0.3116,  ..., -0.1740, -0.2337]],
       grad_fn=<SliceBackward>)
```

The flair library is doing exactly that under its hood! We can use our `get_embeddings` function to get embeddings for our sentence using flair's wrapper for BERT:

```
get_embeddings(bert_flair, sentence)
```

Output

```
tensor([[ 0.0100,  0.8575, -0.5429,  ...,  0.4241, -0.2035],
        [-0.3705,  1.1001,  0.3326,  ...,  0.0656, -0.5644],
        [-0.2947,  0.5797,  0.1997,  ..., -0.3062,  0.6690],
        ...,
        [ 0.0691,  0.7393,  0.0552,  ..., -0.4896, -0.4832],
        [-0.1566,  0.6177,  0.1536,  ...,  0.0904, -0.4917],
        [ 0.7511,  0.3110, -0.3116,  ..., -0.1740, -0.2337]],
       device='cuda:0')
```

Perfect match!

The **contextual word embeddings** are the **hidden states** produced by the **encoder "layers" of the Transformer**. They can either come from **the last layer only** like in the example above, or from a concatenation of hidden states produced by several out of the twelve layers in the model.

- `hidden_states` returns hidden states for **every "layer"** in BERT's encoder architecture, including the last one (returned as `last_hidden_state`), and the **input embeddings** as well:

```
print(len(out['hidden_states']))
print(out['hidden_states'][0].shape)
```

Output

```
13
torch.Size([1, 30, 768])
```

The **first** one corresponds to the **input embeddings**:

```
(out['hidden_states'][0] ==
 bert_model.embeddings(tokens['input_ids'])).all()
```

Output

```
tensor(True)
```

And the **last** one is redundant:

```
(out['hidden_states'][-1] == out['last_hidden_state']).all()
```

Output

```
tensor(True)
```

- `pooler_output` is returned by default and, as already mentioned, it's the output of the **pooler** given the last hidden state as its input:

```
(out['pooler_output'] ==
 bert_model.pooler(out['last_hidden_state'])).all()
```

Output

```
tensor(True)
```

- `attentions` returns the **self-attention scores** for each attention head in each "layer" of BERT's encoder:

```
print(len(out['attentions']))
print(out['attentions'][0].shape)
```

Output

```
12
torch.Size([1, 12, 30, 30])
```

The returned tuple has **twelve elements**, one for each "layer", and each element has a tensor containing the scores for the sentences in the mini-batch (only one in our case) considering each one of BERT's **twelve self-attention heads**, each head indicating **how much attention** each one of the **thirty tokens** is paying to all **thirty tokens** in the input. Are you still with me? That's 129,600 attention scores in total! No, I'm not even *trying* to visualize that :-)

Model IV - Classifying using BERT

We'll use a **Transformer Encoder** as a classifier once again (like in "*Model II*") but it will be *much easier* now because **BERT will be our encoder** and it already handles the special classification token by itself:

Model Configuration

```python
class BERTClassifier(nn.Module):
    def __init__(self, bert_model, ff_units,
                 n_outputs, dropout=0.3):
        super().__init__()
        self.d_model = bert_model.config.dim
        self.n_outputs = n_outputs
        self.encoder = bert_model
        self.mlp = nn.Sequential(
            nn.Linear(self.d_model, ff_units),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ff_units, n_outputs)
        )

    def encode(self, source, source_mask=None):
        states = self.encoder(
            input_ids=source, attention_mask=source_mask)[0]
        cls_state = states[:, 0]
        return cls_state

    def forward(self, X):
        source_mask = (X > 0)
        # Featurizer
        cls_state = self.encode(X, source_mask)
        # Classifier
        out = self.mlp(cls_state)
        return out
```

Both `encode` and `forward` methods are roughly the same as before, but the classifier (`mlp`) has hidden and dropout layers now.

Our model takes an instance of a **pretrained BERT model**, the number of units in the hidden layer of the classifier, and the desired **number of outputs** (logits) corresponding to the number of existing classes. The `forward` method takes a **mini-**

batch of token ids, encodes them using BERT (featurizer), and outputs logits (classifier).

> "*Why does the model compute the* ***source mask*** *itself instead of using the output from the tokenizer?*"

Good catch! I know that's less than ideal, but our `StepByStep` class can only take a single mini-batch of inputs and no additional information like the attention masks. Of course, we *could* modify our class to handle that, but **HuggingFace has its own trainer** (more on that soon!), so there's no point in doing so.

This is actually the *last time* we'll use the `StepByStep` class since it requires too many adjustments to the inputs to work well with HuggingFace's tokenizers and models.

Data Preparation

To turn the sentences in our datasets into mini-batches of token ids and labels for a binary classification task, we can create a helper function that takes an **HF's `Dataset`**, the names of the fields corresponding to the sentences and labels, and a **tokenizer**, and **builds a `TensorDataset`** out of them:

From HF's `Dataset` *to tokenized* `TensorDataset`

```python
def tokenize_dataset(hf_dataset, sentence_field,
                     label_field, tokenizer, **kwargs):
    sentences = hf_dataset[sentence_field]
    token_ids = tokenizer(
        sentences, return_tensors='pt', **kwargs
    )['input_ids']
    labels = torch.as_tensor(hf_dataset[label_field])
    dataset = TensorDataset(token_ids, labels)
    return dataset
```

First, we create a **tokenizer** and define the parameters we'll use while tokenizing the sentences:

Data Preparation

```
auto_tokenizer = AutoTokenizer.from_pretrained(
    'distilbert-base-uncased'
)
tokenizer_kwargs = dict(truncation=True,
                        padding=True,
                        max_length=30,
                        add_special_tokens=True)
```

"Which BERT is that? DistilBERT?!"

DistilBERT is a **smaller**, **faster**, **cheaper**, and **lighter** version of BERT, introduced by Sahn, V. et al. in their paper *"DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter"*[204]. We're not going into any details about it here, but we're using this version because it's also **friendlier** for fine-tuning in low-end GPUs.

We need to change the labels to **float** as well, so they are compatible with the `nn.BCEWithLogitsLoss` we'll be using:

```python
train_dataset_float = train_dataset.map(
    lambda row: {'labels': [float(row['labels'])]}
)
test_dataset_float = test_dataset.map(
    lambda row: {'labels': [float(row['labels'])]}
)

train_tensor_dataset = tokenize_dataset(train_dataset_float,
                                        'sentence',
                                        'labels',
                                        auto_tokenizer,
                                        **tokenizer_kwargs)
test_tensor_dataset = tokenize_dataset(test_dataset_float,
                                       'sentence',
                                       'labels',
                                       auto_tokenizer,
                                       **tokenizer_kwargs)
generator = torch.Generator()
train_loader = DataLoader(
    train_tensor_dataset, batch_size=4,
    shuffle=True, generator=generator
)
test_loader = DataLoader(test_tensor_dataset, batch_size=8)
```

"*Batch size **FOUR**?!*"

Yes, four! DistilBERT is still kinda big, so we're using a *very small batch size* such that it will fit a low-end GPU with 6 GB RAM. If you have more powerful hardware at your disposal, by all means, try larger batch sizes :-)

Model Configuration & Training

Let's create an instance of our model using DistilBERT and train it in the usual way:

Model Configuration

```
torch.manual_seed(41)
bert_model = AutoModel.from_pretrained("distilbert-base-uncased")
model = BERTClassifier(bert_model, 128, n_outputs=1)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-5)
```

Model Training

```
sbs_bert = StepByStep(model, loss_fn, optimizer)
sbs_bert.set_loaders(train_loader, test_loader)
sbs_bert.train(1)
```

You probably noticed that it **takes quite some time** to train for a single epoch… but then again, there are more than **66 million parameters** to update:

```
sbs_bert.count_parameters()
```

Output

```
66461441
```

Let's check the accuracy of our model:

```
StepByStep.loader_apply(test_loader, sbs_bert.correct)
```

Output

```
tensor([[424, 440],
        [317, 331]])
```

That’s 96.11% accuracy on the validation set, nice! Of course, our dataset is tiny and the model is huge, but still :-)

Well, you probably don’t want to go through all this trouble - adjusting the datasets and writing a model class - to fine-tune a BERT model, right?

Say no more!

Fine-Tuning with HuggingFace

What if I told you that **there is a BERT model for every task**, and you just need to fine-tune it? Cool, isn’t it? Then, what if I told you that you can use a **trainer** that does most of the fine-tuning work for you? Amazing, right? The HuggingFace library is **that good**, really!

There are BERT models available for many different tasks:

- pretraining tasks
  - Masked Language Model (`BertForMaskedLM`)
  - Next Sentence Prediction (`BertForNextSentencePrediction`)
- typical tasks (also available as `AutoModel`)
  - Sequence Classification (`BertForSequenceClassification`)
  - Token Classification (`BertForTokenClassification`)
  - Question Answering (`BertForQuestionAnswering`)
- BERT (and family) specific:
  - Multiple Choice (`BertForMultipleChoice`)

We’re sticking with the **sequence classification task** using **DistilBERT** instead of the regular BERT to make the fine-tuning faster.

Sequence Classification (or Regression)

Let's **load the pretrained model** using its corresponding class:

```
from transformers import DistilBertForSequenceClassification
torch.manual_seed(42)
bert_cls = DistilBertForSequenceClassification.from_pretrained(
    'distilbert-base-uncased', num_labels=2
)
```

It comes with a warning:

Output

```
You should probably TRAIN this model on a down-stream task to be
able to use it for predictions and inference.
```

Makes sense!

Since ours is a **binary classification task**, the `num_labels` argument is two, which happens to be the default value. Unfortunately, at the time of writing, the documentation is *not* as explicit as it should be in this case. There is **no mention** of `num_labels` as a possible argument of the model, and it's only referred to in the documentation of the `forward` method of `DistilBertForSequenceClassification` (highlights are mine):

- **`labels`** (`torch.LongTensor` of shape `(batch_size,)`, optional) – Labels for computing the sequence classification/regression loss. Indices should be in `[0, ..., config.num_labels - 1]`. **If `config.num_labels == 1` a regression loss is computed (Mean-Square loss), If `config.num_labels > 1` a classification loss is computed (Cross-Entropy)**.

Some of the returning values of the `forward` method also include references to the `num_labels` argument:

- **`loss`** (`torch.FloatTensor` of shape `(1,)`, optional, returned when `labels` is provided) – Classification (or regression if config.num_labels==1) loss.
- **`logits`** (`torch.FloatTensor` of shape `(batch_size, config.num_labels)`) – Classification (or regression if config.num_labels==1) scores (before SoftMax).

That's right… `DistilBertForSequenceClassification` (or any other `ForSequenceClassification` model) **can be used for regression too** as long as you **set `num_labels=1` as argument**.

If you want to learn more about the **arguments** the pretrained models may take, please check the documentation on Configuration: `PretrainedConfig`.

To learn more about the **outputs** of several pretrained models, please check the documentation on Model Outputs.

The `ForSequenceClassification` models add a **single linear layer** (classifier) on **top of the pooled output** from the underlying base model to produce the `logits` output.

We already have a model, let's look at our dataset…

More AutoModels

If you want to quickly try *different fine-tuning models* without having to import their corresponding classes, you can use HuggingFace's `AutoModel` corresponding to a fine-tuning task:

```
from transformers import AutoModelForSequenceClassification
auto_cls = AutoModelForSequenceClassification.from_pretrained(
    'distilbert-base-uncased', num_labels=2
)
print(auto_cls.__class__)
```

Output

```
<class 'transformers.modeling_distilbert
.DistilBertForSequenceClassification'>
```

As you can see, it *infers* the correct model class based on the name of the model you're loading, e.g. `distilbert-base-uncased`.

Tokenized Dataset

The training and test datasets are HF's `Datasets` and, *finally*, we'll **keep them like that** instead of building `TensorDatasets` out of them. We *still* have to **tokenize** them, though:

Data Preparation

```python
auto_tokenizer = AutoTokenizer.from_pretrained(
    'distilbert-base-uncased'
)
def tokenize(row):
    return auto_tokenizer(row['sentence'],
                          truncation=True,
                          padding='max_length',
                          max_length=30)
```

We load a **pretrained tokenizer** and build a simple function that takes **one row from the dataset** and **tokenizes it**. So far, so good, right?

IMPORTANT: the **pretrained tokenizer** and **pretrained model** must have **matching architectures** - in our case, both are pretrained on `distilbert-base-uncased`.

Next, we use the `map` method of HF's `Dataset` to **create new columns** by using our **`tokenize` function**:

Data Preparation

```python
tokenized_train_dataset = train_dataset.map(
    tokenize, batched=True
)
tokenized_test_dataset = test_dataset.map(tokenize, batched=True)
```

The `batched` argument **speeds up** the tokenization but the **tokenizer must return lists** instead of tensors (notice the missing `return_tensors='pt'` in the call to `auto_tokenizer`):

```python
print(tokenized_train_dataset[0])
```

Output

```
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'input_ids': [101, 1998, 1010, 2061, 2521, 2004, 2027, 2354, 1010,
2027, 2020, 3243, 2157, 1012, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0],
 'labels': 0,
 'sentence': 'And, so far as they knew, they were quite right.',
 'source': 'wizoz10-1740.txt'}
```

See? Regular Python lists, not PyTorch tensors. It created new columns (`attention_mask` and `input_ids`) and kept the old ones (`labels`, `sentence`, and `source`).

But we don't need *all* these columns for training, we need only the first three. So, let's **tidy it up** by using the `set_format` method of the `Dataset`:

Data Preparation

```
tokenized_train_dataset.set_format(
    type='torch',
    columns=['input_ids', 'attention_mask', 'labels']
)
tokenized_test_dataset.set_format(
    type='torch',
    columns=['input_ids', 'attention_mask', 'labels']
)
```

Not only we're specifying the columns we're actually interested in, but we're also telling it to return PyTorch tensors:

```
tokenized_train_dataset[0]
```

Output

```
{'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'input_ids': tensor([ 101, 1998, 1010, 2061, 2521, 2004, 2027,
2354, 1010, 2027, 2020, 3243, 2157, 1012,  102,    0,    0,    0,
0,    0,    0,    0,    0,    0, 0,    0,    0,    0,    0,    0]),
 'labels': tensor(0)}
```

Much better! We're done with our datasets and we can move on to the…

Trainer

Even though every pretrained model on HuggingFace can be fine-tuned in native PyTorch as we've done in the previous section, the library offers an easy-to-use interface for training and evaluation: `Trainer`.

As expected, it takes **a model** and **a training dataset** as required arguments, and that's it:

```
from transformers import Trainer
trainer = Trainer(model=bert_cls,
                  train_dataset=tokenized_train_dataset)
```

We only need to call the `train` method and our model will be trained! YES, **this `train` method** actually **trains the model**! Thank you, HuggingFace :-)

*"Awesome! But… does it train for **how many epochs**? Which **optimizer** and **learning rate** does it use for training?"*

We can find it all out looking at its `TrainingArguments`:

```
trainer.args
```

Output

```
TrainingArguments(output_dir=tmp_trainer, overwrite_output_dir=
False, do_train=False, do_eval=None, do_predict=False,
evaluation_strategy=IntervalStrategy.NO, prediction_loss_only=False,
per_device_train_batch_size=8, per_device_eval_batch_size=8,
gradient_accumulation_steps=1, eval_accumulation_steps=None,
learning_rate=5e-05, weight_decay=0.0, adam_beta1=0.9,
adam_beta2=0.999, adam_epsilon=1e-08, max_grad_norm=1.0,
num_train_epochs=3.0, max_steps=-1, lr_scheduler_type=SchedulerType
.LINEAR, warmup_ratio=0.0, warmup_steps=0, logging_dir=runs/Apr21_20
-33-20_MONSTER, logging_strategy=IntervalStrategy.STEPS,
logging_first_step=False, logging_steps=500, save_strategy
=IntervalStrategy.STEPS, save_steps=500, save_total_limit=None,
no_cuda=False, seed=42, fp16=False, fp16_opt_level=O1, fp16_backend
=auto, fp16_full_eval=False, local_rank=-1, tpu_num_cores=None,
tpu_metrics_debug=False, debug=False, dataloader_drop_last=False,
eval_steps=500, dataloader_num_workers=0, past_index=-1, run_name
=tmp_trainer, disable_tqdm=False, remove_unused_columns=True,
label_names=None, load_best_model_at_end=False,
metric_for_best_model=None, greater_is_better=None,
ignore_data_skip=False, sharded_ddp=[], deepspeed=None,
label_smoothing_factor=0.0, adafactor=False, group_by_length=False,
length_column_name=length, report_to=['tensorboard'],
ddp_find_unused_parameters=None, dataloader_pin_memory=True,
skip_memory_metrics=False, _n_gpu=1, mp_parameters=)
```

The `Trainer` creates an instance of `TrainingArguments` by itself, and the values above are the arguments' default values. There is the `learning_rate=5e-05`, and the `num_train_epochs=3.0`, and **many, many others**. The *optimizer* used, even though it's not listed above, is the `AdamW`, a variation of Adam.

We can create an instance of `TrainingArguments` ourselves to get at least *a bit of control* over the training process. The *only* required argument is the `output_dir`, but we'll specify some other arguments as well:

Training Arguments

```python
from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir='output',
    num_train_epochs=1,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=8,
    evaluation_strategy='steps',
    eval_steps=300,
    logging_steps=300,
    gradient_accumulation_steps=8,
)
```

"*Batch size **ONE**?! You gotta be kidding me…*"

Well, I *would*, if it were not for the **`gradient_accumulation_steps`** argument. That's how we can make the **mini-batch size larger** even if we're using a **low-end GPU** that is capable of handling **only one data point** at a time.

The `Trainer` can **accumulate the gradients** computed at every training step (which is taking only one data point) and, **after eight steps**, it uses the accumulated gradients to **update the parameters**. To all intents and purposes, it is **as if** the mini-batch had **size eight**. Awesome, right?

Moreover, let's set the `logging_steps` to three hundred, so it prints the **training losses** every three hundred mini-batches (and it counts the mini-batches as having size eight due to the gradient accumulation).

"*What about **validation losses**?*"

The `evaluation_strategy` argument allows you to run an evaluation **after every `eval_steps` steps** (if set to `steps` like in the example above) or **after every epoch** (if set to `epoch`).

*"Can I get it to print **accuracy** or other metrics too?"*

Sure, you can! But, first, you need to define a function that takes an instance of `EvalPrediction` (returned by the internal validation loop), **computes the desired metrics**, and **returns a dictionary**:

Method for computing accuracy

```python
def compute_metrics(eval_pred):
    predictions = eval_pred.predictions
    labels = eval_pred.label_ids
    predictions = np.argmax(predictions, axis=1)
    return {"accuracy": (predictions == labels).mean()}
```

We can use a simple function like the one above to compute accuracy and pass it as the `compute_metrics` argument of the `Trainer` along with the remaining `TrainingArguments` and datasets:

Model Training

```python
trainer = Trainer(model=bert_cls,
                  args=training_args,
                  train_dataset=tokenized_train_dataset,
                  eval_dataset=tokenized_test_dataset,
                  compute_metrics=compute_metrics)
```

There we go - we're 100% ready to call the **glorious `train` method**:

Model Training

```python
trainer.train()
```

Output

```
Step Training Loss Validation Loss Accuracy  ...
 300      0.194600        0.159694 0.953307  ...

TrainOutput(global_step=385, training_loss=0.17661244776341822,
metrics={'train_runtime': 80.0324, 'train_samples_per_second':
4.811, 'total_flos': 37119857544000.0, 'epoch': 1.0,
'init_mem_cpu_alloc_delta': 0, 'init_mem_gpu_alloc_delta': 0,
'init_mem_cpu_peaked_delta': 0, 'init_mem_gpu_peaked_delta': 0,
'train_mem_cpu_alloc_delta': 5025792, 'train_mem_gpu_alloc_delta':
806599168, 'train_mem_cpu_peaked_delta': 0,
'train_mem_gpu_peaked_delta': 96468992})
```

It's printing the training and validation losses, and the validation accuracy, every three hundred mini-batches as configured. To check the **final validation figures**, though, we need to call the `evaluate` method which, guess what, **actually runs a validation loop**! Thank you, HuggingFace :-)

```
trainer.evaluate()
```

Output

```
{'eval_loss': 0.142591193318367,
 'eval_accuracy': 0.9610894941634242,
 'eval_runtime': 1.6634,
 'eval_samples_per_second': 463.51,
 'epoch': 1.0,
 'eval_mem_cpu_alloc_delta': 0,
 'eval_mem_gpu_alloc_delta': 0,
 'eval_mem_cpu_peaked_delta': 0,
 'eval_mem_gpu_peaked_delta': 8132096}
```

That's 96.11% accuracy on the validation set after one epoch, roughly the same as our own implementation ("*Model IV*"). Nice!

Once the model is trained, we can **save it to disk** using the `save_model` method from `Trainer`:

```
trainer.save_model('bert_alice_vs_wizard')
os.listdir('bert_alice_vs_wizard')
```

Output

```
['training_args.bin', 'config.json', 'pytorch_model.bin']
```

It creates a *folder* with the provided name, and it stores the trained model (`pytorch_model.bin`) along with its configuration (`config.json`), and training arguments (`training_args.bin`).

Later on, we can easily **load the trained model** using the `from_pretrained` method from the corresponding `AutoModel`:

```
loaded_model = (AutoModelForSequenceClassification
                .from_pretrained('bert_alice_vs_wizard'))
loaded_model.device
```

Output

```
device(type='cpu')
```

The model is loaded to the CPU by default, but we can send it to a different device in the usual PyTorch way:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
loaded_model.to(device)
loaded_model.device
```

Output

```
device(type='cuda', index=0)
```

Predictions

We can **finally** answer the most important question of all: where does the sentence in the title, "*Down the Yellow Brick Rabbit Hole*", come from? Let's ask BERT:

```
sentence = 'Down the yellow brick rabbit hole'
tokens = auto_tokenizer(sentence, return_tensors='pt')
tokens
```

Output

```
{'input_ids': tensor([[  101,  2091,  1996,  3756,  5318, 10442,
4920,   102]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1]])}
```

After tokenizing the sentence, we need to make sure the tensors are in the **same device as the model**.

> ? *"Do I need to send each tensor to a device, really?"*

Not really, no. It turns out, the **output of the tokenizer** isn't *just* a dictionary, but an instance of `BatchEncoding`, and we can easily call its `to()` method to send the tensors to the same device as the model:

```
print(type(tokens))
tokens.to(loaded_model.device)
```

Output

```
<class 'transformers.tokenization_utils_base.BatchEncoding'>
{'input_ids': tensor([[  101,  2091,  1996,  3756,  5318, 10442,
4920,   102]], device='cuda:0'),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1]], device
='cuda:0')}
```

That was easy, right? :-)

Let's call the model using these inputs!

Even though the model is loaded in *evaluation mode* by default, it is always a good idea to explicitly **set the model to evaluation mode** using the PyTorch model's `eval` method during evaluation or test phases:

```
loaded_model.eval()
logits = loaded_model(input_ids=tokens['input_ids'],
                      attention_mask=tokens['attention_mask'])
logits
```

Output

```
SequenceClassifierOutput(loss=None, logits=tensor([[ 2.7745, -
2.5539]], device='cuda:0', grad_fn=<AddmmBackward>), hidden_states
=None, attentions=None)
```

The **largest logit** corresponds to the **predicted class** as usual:

```
logits.logits.argmax(dim=1)
```

Output

```
tensor([0], device='cuda:0')
```

BERT has spoken: the sentence "*Down the Yellow Brick Rabbit Hole*" is more likely coming from "*The Wonderful Wizard of Oz*" (the negative class of our binary classification task).

Don't you think that's a lot of work to get predictions for a single sentence? I mean, tokenizing, sending it to the device, feeding the inputs to the model, getting the largest logit, that's a lot, right? I think so, too.

Pipelines

The pipelines can handle all these steps for us, we just have to choose the appropriate one. There are *many* different pipelines, one for each task, like the `TextClassificationPipeline` and the `TextGenerationPipeline`. Let's use the former to run our tokenizer and trained model at once:

```
from transformers import TextClassificationPipeline
device_index = (loaded_model.device.index
                if loaded_model.device.type != 'cpu'
                else -1)
classifier = TextClassificationPipeline(model=loaded_model,
                                        tokenizer=auto_tokenizer,
                                        device=device_index)
```

Every pipeline takes at least **two** required arguments: **a model** and **a tokenizer**. We can also send it straight to the same device as our model, but we need to use the **device index** instead (-1 if it's on a CPU, `0` if it's on the first or only GPU, 1 if it's on the second one, and so on).

Now we can make predictions using **the original sentences**:

```
classifier(['Down the Yellow Brick Rabbit Hole', 'Alice rules!'])
```

Output

```
[{'label': 'LABEL_0', 'score': 0.9951714277267456},
 {'label': 'LABEL_1', 'score': 0.9985325336456299}]
```

The model seems pretty confident that the **first sentence** is from "*The Wonderful Wizard of Oz*" (negative class) while the **second sentence** is from "*Alice's Adventures in Wonderland*" (positive class).

We can make the output a bit more intuitive, though, by **setting proper labels** for each one of the classes using the `id2label` attribute of our model's configuration:

```
loaded_model.config.id2label = {0: 'Wizard', 1: 'Alice'}
```

Let's try it again:

```
classifier(['Down the Yellow Brick Rabbit Hole', 'Alice rules!'])
```

Output

```
[{'label': 'Wizard', 'score': 0.9951714277267456},
 {'label': 'Alice', 'score': 0.9985325336456299}]
```

That's much better!

More Pipelines

It's also possible to use pretrained pipelines for **typical tasks** like **sentiment**

analysis without having to fine-tune your own model:

```
from transformers import pipeline
sentiment = pipeline('sentiment-analysis')
```

That's it! The **task defines which pipeline is used**. For sentiment analysis, the pipeline above loads a `TextClassificationPipeline` like ours, but one that's pretrained to perform that task.

For a complete list of the available tasks, please check HuggingFace's `pipeline` documentation.

Let's run the first sentence of our training set through the sentiment analysis pipeline:

```
sentence = train_dataset[0]['sentence']
print(sentence)
print(sentiment(sentence))
```

Output

```
And, so far as they knew, they were quite right.
[{'label': 'POSITIVE', 'score': 0.9998356699943542}]
```

Positive indeed!

If you're curious about **which model** is being used under the hood, you can check the `SUPPORTED_TASKS` dictionary. For sentiment analysis, it uses the `distilbert-base-uncased-finetuned-sst-2-english` model:

```
from transformers.pipelines import SUPPORTED_TASKS
SUPPORTED_TASKS['sentiment-analysis']
```

Output

```
{'impl': transformers.pipelines.text_classification
.TextClassificationPipeline,
 'tf': None,
 'pt': types.AutoModelForSequenceClassification,
 'default': {'model': {'pt': 'distilbert-base-uncased-finetuned-sst-
2-english',
   'tf': 'distilbert-base-uncased-finetuned-sst-2-english'}}}
```

"*What about **text generation**?*"

```
SUPPORTED_TASKS['text-generation']
```

Output

```
{'impl': transformers.pipelines.text_generation
.TextGenerationPipeline,
 'tf': None,
 'pt': types.AutoModelForCausalLM,
 'default': {'model': {'pt': 'gpt2', 'tf': 'gpt2'}}}
```

That's the **famous GPT-2** model, and we'll discuss it briefly in the next, and last, section of this chapter.

GPT-2

The **G**enerative **P**retrained **T**ransformer **2**, introduced by Radford, A. et al. in their paper *"Language Models are Unsupervised Multitask Learners"*[205] (2018), made headlines with its impressive ability to **generate text** of high quality in a variety of contexts. Just like BERT, it is a **language model**, that is, it is trained to **fill in the blanks** in sentences. But, while BERT was trained to fill in the blanks in the middle of sentences (thus correcting corrupted inputs), **GPT-2** was trained to **fill in blanks**

at the end of sentences, effectively **predicting the next word in a given sentence**.

Predicting the **next element in a sequence** is exactly what a **Transformer Decoder** does, so it should be no surprise that **GPT-2 actually is a Transformer Decoder**.

It was trained on more than 40 GB of Internet text spread over 8 million web pages. Its largest version has **48 "layers"** (the original Transformer had only six), **twelve attention heads**, and **1,600 hidden dimensions**, totaling **1.5 billion parameters**, and it was released in November 2019[206].

"*Don't train this at home!*"

On the other end of the scale, the smallest version has *only* twelve "layers", twelve attention heads, and 768 hidden dimensions, totaling 117 million parameters (the smallest GPT-2 is still a bit larger than the original BERT!). This is the version automatically loaded in the `TextGenerationPipeline`.

You can find the models and the corresponding code in Open AI's GPT-2 repository [207]. For a demo of GPT-2's capabilities, please check AllenNLP's Language Modeling Demo[208], which uses GPT-2's medium model (345 million parameters).

You can also check GPT-2's documentation and model card available at HuggingFace for a quick overview of the model and its training procedure.

For a general overview of GPT-2, see this great post by Jay Alammar: "*The Illustrated GPT-2 (Visualizing Transformer Language Models)*"[209].

To learn more details about GPT-2's architecture, please check "*The Annotated GPT-2*"[210] by Aman Arora.

There is also Andrej Karpathy's **minimalistic implementation** of GPT, minGPT[211], if you feel like trying to train a GPT model from scratch.

Let's load the **GPT-2-based text generation pipeline**:

```
text_generator = pipeline("text-generation")
```

Then, let's use the **first two paragraphs** from "*Alice's Adventures in Wonderland*" as base text:

```
base_text = """
Alice was beginning to get very tired of sitting by her sister on
the bank, and of having nothing to do:  once or twice she had peeped
into the book her sister was reading, but it had no pictures or
conversations in it, `and what is the use of a book,'thought Alice
`without pictures or conversation?' So she was considering in her
own mind (as well as she could, for the hot day made her feel very
sleepy and stupid), whether the pleasure of making a daisy-chain
would be worth the trouble of getting up and picking the daisies,
when suddenly a White Rabbit with pink eyes ran close by her.
"""
```

The generator will produce a text of size `max_length`, including the base text, so this value has to be **larger** than the length of the base text. By default, the model in the text generation pipeline has its **`do_sample`** argument set to `True` to generate words using **beam search** instead of greedy decoding:

```
text_generator.model.config.task_specific_params
```

Output

```
{'text-generation': {'do_sample': True, 'max_length': 50}}
```

```
result = text_generator(base_text, max_length=250)
print(result[0]['generated_text'])
```

Output

```
...
Alice stared at the familiar looking man in red, in a white dress,
and smiled shyly.

She saw the cat on the grass and sat down with it gently and
eagerly, with her arms up.

There was a faint, long, dark stench, the cat had its tail held at
the end by a large furry, white fur over it.

Alice glanced at it.

It was a cat, but a White Rabbit was very good at drawing this,
thinking over its many attributes, and making sure that no reds
appeared
```

I've removed the base text from the output above, so that's **generated text** only. Looks decent, right? I tried it several times, and the generated text is usually consistent, even though it *digresses* some times and, on occasion, generates some really weird pieces of text.

"*What is this **beam search**? That sounds oddly familiar…*"

That's true, we have briefly discussed **beam search** (and its alternative, greedy decoding), in Chapter 9, and I reproduce it below for your convenience:

> "...**greedy decoding** because **each prediction** is deemed **final**. "*No backsies*": once it's done, it's *really* done, you just move along to the next prediction and never look back. In the context of our sequence-to-sequence problem, a regression, it wouldn't make much sense to do otherwise anyway.
>
> But that **may not** be the case for other types of sequence-to-sequence problems. In machine translation, for example, the decoder outputs **probabilities** for the **next word** in the sentence at each step. The **greedy** approach would simply take the **word with the highest probability** and move on to the next.
>
> However, since **each prediction** is an input to the **next step**, taking the **top word at every step** is **not necessarily the winning approach** (translating from one language to another is not exactly "linear"). It is probably wiser to keep a **handful of candidates** at every step and **try their combinations** to choose the best one: that's called **beam search**...."

By the way, if you try using *greedy decoding* instead (setting `do_sample=False`), the generated text simply and annoyingly repeats the same text over and over again:

```
'What is the use of a daisy-chain?'
'I don't know,' said Alice, 'but I think it is a good idea.'

'What is the use of a daisy-chain?'
'I don't know,' said Alice, 'but I think it is a good idea.'
```

For more details on the different arguments that can be used for text generation, including a more detailed explanation of both **greedy decoding** and **beam search**, please check HuggingFace's blog post *"How to generate text: using different decoding methods for language generation with Transformers"*[212] by Patrick von Platen.

*"Wait a minute… aren't we **fine-tuning GPT-2** so it can write text in a given style?"*

I thought you would never ask… yes, we are. It's the final example, and we're covering it in the next section…

Putting It All Together

In this chapter, we **built a dataset** using two books and explored many preprocessing steps and techniques: sentence and word tokenization, **word embeddings**, and much more. We used HuggingFace's `Dataset` and **pretrained tokenizers** extensively for the data preparation step, and leveraged the power of **pretrained models** like **BERT** to **classify sequences (sentences)** according to their source. We also used HuggingFace's `Trainer` and `pipeline` classes to easily **train models** and **deliver predictions**, respectively.

Data Preparation

In order to capture the *style* of Lewis Carroll's "*Alice's Adventures in Wonderland*", we need to use a dataset containing sentences from that book alone, instead of our previous dataset that included "*The Wonderful Wizard of Oz*" as well.

Data Preparation

```
dataset = load_dataset(path='csv',
                       data_files=['texts/alice28-1476.sent.csv'],
                       quotechar='\\', split=Split.TRAIN)

shuffled_dataset = dataset.shuffle(seed=42)
split_dataset = shuffled_dataset.train_test_split(test_size=0.2,
                                                  seed=42)
train_dataset = split_dataset['train']
test_dataset = split_dataset['test']
```

Next, we **tokenize** the dataset using **GPT-2**'s pretrained tokenizer. There are *some* differences from BERT, though:

- first, GPT-2 uses a **Byte-Pair Encoding (BPE)** instead of WordPiece for tokenization
- second, we're **not padding** the sentences since we're trying to **generate text** and it wouldn't make much sense to **predict the next word after a bunch of padded elements**
- third, we're **removing the original columns** (`source` and `sentence`) such that only the output of the tokenizer (`input_ids` and `attention_mask`) remains

Data Preparation

```
auto_tokenizer = AutoTokenizer.from_pretrained('gpt2')
def tokenize(row):
    return auto_tokenizer(row['sentence'])

tokenized_train_dataset = train_dataset.map
    (tokenize, remove_columns=['source', 'sentence'], batched
=True
)
tokenized_test_dataset = test_dataset.map(
    tokenize, remove_columns=['source', 'sentence'], batched=True
)
```

Maybe you've already realized that, without padding, the **sentences have varied lengths**:

```
list(map(len, tokenized_train_dataset[0:6]['input_ids']))
```

Output

```
[9, 28, 20, 9, 34, 29]
```

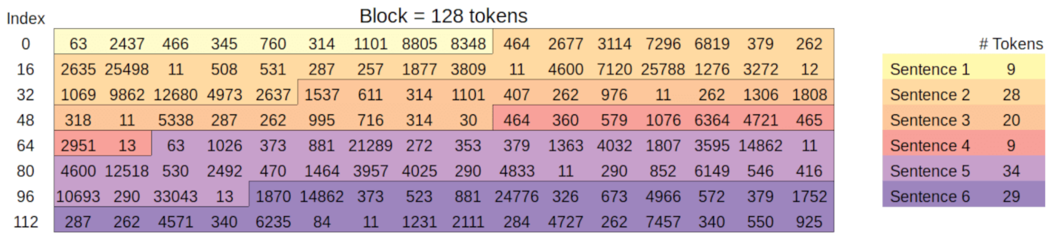
These are the **first six** sentences, and their length ranges from **nine** to **thirty-four tokens**.

*"Can't we just **pack the sequences** using* `rnn_utils.pack_sequence` *like in Chapter 8?"*

You got the *gist* of it: the general idea is to "*pack*" sequences together, indeed, but in a *different way*…

"Packed" Dataset

The "*packing*" is actually simpler now, it is simply **concatenating** the inputs together and then **chunking them into blocks**:



Block = 128 tokens

Index																
0	63	2437	466	345	760	314	1101	8805	8348	464	2677	3114	7296	6819	379	262
16	2635	25498	11	508	531	287	257	1877	3809	11	4600	7120	25788	1276	3272	12
32	1069	9862	12680	4973	2637	1537	611	314	1101	407	262	976	11	262	1306	1808
48	318	11	5338	287	262	995	716	314	30	464	360	579	1076	6364	4721	465
64	2951	13	63	1026	373	881	21289	272	353	379	1363	4032	1807	3595	14862	11
80	4600	12518	530	2492	470	1464	3957	4025	290	4833	11	290	852	6149	546	416
96	10693	290	33043	13	1870	14862	373	523	881	24776	326	673	4966	572	379	1752
112	287	262	4571	340	6235	84	11	1231	2111	284	4727	262	7457	340	550	925

	# Tokens
Sentence 1	9
Sentence 2	28
Sentence 3	20
Sentence 4	9
Sentence 5	34
Sentence 6	29

Figure 11.29 - Grouping sentences into blocks

The function below was adapted from HuggingFace's language modeling fine-tuning script `run_clm.py`[213], and it "*packs*" the inputs together:

Method for grouping sentences into blocks

```python
# Adapted from https://github.com/huggingface/transformers/blob/
# master/examples/pytorch/language-modeling/run_clm.py
def group_texts(examples, block_size=128):
    # Concatenate all texts.
    concatenated_examples = {k: sum(examples[k], [])
                             for k in examples.keys()}
    total_length = len(
        concatenated_examples[list(examples.keys())[0]]
    )
    # We drop the small remainder, we could add padding
    # if the model supported it instead of this drop, you
    # can customize this part to your needs.
    total_length = (total_length // block_size) * block_size
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + block_size]
            for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result
```

We can apply the function above to our datasets in the usual way and then set their output formats to PyTorch tensors:

Data Preparation

```
lm_train_dataset = tokenized_train_dataset.map(
    group_texts, batched=True
)
lm_test_dataset = tokenized_test_dataset.map(
    group_texts, batched=True
)
lm_train_dataset.set_format(type='torch')
lm_test_dataset.set_format(type='torch')
```

Now, the **first data point** actually contains the **first 128 tokens** of our dataset (the *first five sentences* and *almost all tokens from the sixth*):

```
print(lm_train_dataset[0]['input_ids'])
```

Output

```
tensor([   63,  2437,   466,   345,   760,   314,  1101,  8805,
         8348,   464,  2677,  3114,  7296,  6819,   379,   262,
         2635, 25498,    11,   508,   531,   287,   257,  1877,
         3809,    11,  4600,  7120, 25788,  1276,  3272,    12,
         1069,  9862, 12680,  4973,  2637,  1537,   611,   314,
         1101,   407,   262,   976,    11,   262,  1306,  1808,
          318,    11,  5338,   287,   262,   995,   716,   314,
           30,   464,   360,   579,  1076,  6364,  4721,   465,
         2951,    13,    63,  1026,   373,   881, 21289,   272,
          353,   379,  1363,  4032,  1807,  3595, 14862,    11,
         4600, 12518,   530,  2492,   470,  1464,  3957,  4025,
          290,  4833,    11,   290,   852,  6149,   546,   416,
        10693,   290, 33043,    13,  1870, 14862,   373,   523,
          881, 24776,   326,   673,  4966,   572,   379,  1752,
          287,   262,  4571,   340,  6235,   284,    11,  1231,
         2111,   284,  4727,   262,  7457,   340,   550,   925])
```

Consequently, the **datasets got smaller**, since they do not contain *sentences* anymore but **sequences of 128 tokens** instead:

```
len(lm_train_dataset), len(lm_test_dataset)
```

Output

```
(239, 56)
```

The dataset is ready! We can move on to the…

Model Configuration & Training

GPT-2 is a model for **causal language modeling** and that's the `AutoModel` we use to load it:

Model Configuration

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained('gpt2')
print(model.__class__)
```

Output

```
<class 'transformers.modeling_gpt2.GPT2LMHeadModel'>
```

GPT-2's tokenizer **does not** include a **special padding token** by default but you *may add it* if needed. If you **do add any tokens** to the vocabulary, though, you also need to **let the model know it** using `resize_token_embeddings`:

```
model.resize_token_embeddings(len(auto_tokenizer))
```

Output

```
Embedding(50257, 768)
```

In our example, it **doesn't make a difference**, but it's a good idea to add the line above to the code anyway to be on the safe side.

The training arguments are roughly the same ones we used to train BERT, but there is an additional one: `prediction_loss_only=True`. Since GPT-2 is a **generative model**, we won't be running any additional metrics during training or validation, and there's no need for anything else but the loss.

Model Training

```python
training_args = TrainingArguments(
    output_dir='output',
    num_train_epochs=1,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=8,
    evaluation_strategy='steps',
    eval_steps=50,
    logging_steps=50,
    gradient_accumulation_steps=4,
    prediction_loss_only=True,
)

trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=lm_train_dataset,
                  eval_dataset=lm_test_dataset)
```

After configuring the `Trainer`, we call its `train` method, and then its `evaluate` method:

Model Training

```python
trainer.train()
```

Output

```
Step Training Loss Validation Loss  ...
  50      3.587500        3.327199  ...

TrainOutput(global_step=59, training_loss=3.5507330167091498,
metrics={'train_runtime': 22.6958, 'train_samples_per_second': 2.6,
'total_flos': 22554466320384.0, 'epoch': 0.99,
'init_mem_cpu_alloc_delta': 1316954112, 'init_mem_gpu_alloc_delta':
511148032, 'init_mem_cpu_peaked_delta': 465375232,
'init_mem_gpu_peaked_delta': 0, 'train_mem_cpu_alloc_delta':
13103104, 'train_mem_gpu_alloc_delta': 1499219456,
'train_mem_cpu_peaked_delta': 0, 'train_mem_gpu_peaked_delta':
730768896})
```

```
trainer.evaluate()
```

Output

```
{'eval_loss': 3.320632219314575,
 'eval_runtime': 0.9266,
 'eval_samples_per_second': 60.438,
 'epoch': 0.99,
 'eval_mem_cpu_alloc_delta': 151552,
 'eval_mem_gpu_alloc_delta': 0,
 'eval_mem_cpu_peaked_delta': 0,
 'eval_mem_gpu_peaked_delta': 730768896}
```

There we go, GPT-2 was fine-tuned on "*Alice's Adventures in Wonderland*" for one epoch.

How good is it at being Lewis Carroll now? Let's check it out!

Generating Text

The GPT-2 model has a `generate` method with *plenty of options* for generating text (e.g. greedy decoding, beam search, and more). We won't be delving into these details but going the **easy way** instead: assigning our fine-tuned model and pretrained tokenizer to a **pipeline** and using most of its default values:

```
device_index = (model.device.index
                if model.device.type != 'cpu'
                else -1)
gpt2_gen = pipeline('text-generation',
                    model=model,
                    tokenizer=auto_tokenizer,
                    device=device_index)
```

The only parameter we may have to change is, once again, the `max_length`:

```
result = gpt2_gen(base_text, max_length=250)
print(result[0]['generated_text'])
```

Output

```
Alice was beginning to get very tired of sitting by her sister on
the bank, and of having nothing to do:  once or twice she had peeped
into the book her sister was reading, but it had no pictures or
conversations in it, `and what is the use of a book,'thought Alice
`without pictures or conversation?' So she was considering in her
own mind (as well as she could, for the hot day made her feel very
sleepy and stupid), whether the pleasure of making a daisy-chain
would be worth the trouble of getting up and picking the daisies,
when suddenly a White Rabbit with pink eyes ran close by her.

The rabbit was running away quite as quickly as it had jumped to her
feet.She had nothing of the kind, and, as she made it up to Alice,
was beginning to look at the door carefully in one thought.`It's
very curious,'after having been warned,`that I should be talking to
Alice!'`It's not,'she went on, `it wasn't even a cat,' so very very
quietly indeed.'In that instant he began to cry out aloud.Alice
began to sob out, 'I am not to cry out!'`What
```

This time, I've kept the whole thing, the **base**, and the **generated text**. I tried it out several times and, in my humble opinion, the output **looks more "*Alice-y*"** now.

What do *you* think?

Recap

In this chapter, we took a *deep dive* into the **Natural Language Processing** world. We built our own dataset from scratch using two books, "*Alice's Adventures in Wonderland*" and "*The Wonderful Wizard of Oz*", and performed **sentence and word tokenization**. Then, we built a **vocabulary** and used it with a **tokenizer** to generate the primary input of our models: **sequences of token ids**. Next, we created **numerical representations** for our tokens, starting with a basic **one-hot encoding** and working our way to using **word embeddings** to train a model for classifying the source of a sentence. We also learned about the limitations of classical embeddings, and the need for **contextual word embeddings** produced by **language models** like **ELMo** and **BERT**. We got to know our muppet friend in detail: input embeddings, pretraining tasks, and hidden states (the actual embeddings). We leveraged the HuggingFace library to **fine-tune** a pretrained model using a **`Trainer`** and to deliver predictions using a **`pipeline`**. Lastly, we used the famous **GPT-2** model to **generate text** that, hopefully, looks like it was written by Lewis Carroll. This is what we've covered:

- using `NLTK` to perform **sentence tokenization** on our text corpora
- converting each book into a CSV file containing **one sentence per line**
- building a dataset using HuggingFace's **`Dataset`** to load the CSV files
- creating new columns in the dataset using `map`
- learning about **data augmentation** for **text data**
- using `Gensim` to perform **word tokenization**
- building a **vocabulary** and using it to **get a token id** for each word
- adding **special tokens** to the vocabulary, like `[UNK]` and `[PAD]`
- loading our own vocabulary into a HuggingFace's **tokenizer**
- understanding the **output** of a tokenizer: `input_ids`, `token_type_ids`, and `attention_mask`
- using the tokenizer to **tokenize two sentences** as a **single input**
- creating **numerical representations** for each token, starting with **one-hot encoding**
- learning about the simplicity and limitations of the **Bag-of-Words (BoW)** approach
- learning that a **language model** is used to estimate the **probability of a token**, pretty much like **filling the blanks** in a sentence
- understanding the general idea behind the **Word2Vec** model and its common implementation, the **CBoW (Continuous Bag-of-Words)**

- learning that **word embeddings** are basically a **lookup table** to retrieve the vector corresponding to a given token
- using **pretrained embeddings** like **GloVe** to perform **embedding arithmetic**
- loading **GloVe embeddings** and using them to **train a simple classifier**
- using a **Transformer Encoder** together with **Glove embeddings** to classify sentences
- understanding the importance of **contextual word embeddings** to distinguish between different meanings for the same word
- using `flair` to retrieve **contextual word embeddings from ELMo**
- getting an overview of **ELMo's architecture** and its **hidden states** (the embeddings)
- using `flair` to **preprocess sentences** into **BERT embeddings** and **train a classifier**
- learning about **WordPiece tokenization** used by BERT
- computing BERT's **input embeddings** using **token**, **position**, and **segment** embeddings
- understanding BERT's pretraining tasks: **Masked Language Model (MLM)** and **Next Sentence Prediction (NSP)**
- exploring the different **outputs** from BERT: **hidden states**, **pooler output**, and **attentions**
- training a classifier using **pretrained BERT as a layer**
- **fine-tuning BERT** using HuggingFace's models for **sequence classification**
- remembering to **always** use **matching pretrained model and tokenizer**
- exploring and using the `Trainer` class to **fine-tune large models** using **gradient accumulation**
- combining **tokenizer and model** into a **pipeline** to easily deliver predictions
- loading **pretrained pipelines** to perform typical tasks, like **sentiment analysis**

- learning about the famous **GPT-2** model, and fine-tuning it to **generate text** like Lewis Carroll :-)

Congratulations! You survived an intense *crash course* on (almost) all things NLP, from basic **sentence tokenization** using NLTK all the way up to **sequence classification** using **BERT** and **causal language modeling** using **GPT-2**. You're now equipped to handle **text data** and train or **fine-tune models using HuggingFace**.

In the next chapter, we'll… oh, wait, there is *no* next chapter. We've actually **finished** our *long* journey…

Thank You!

I really hope you enjoyed reading and learning about all these topics as much as I enjoyed writing (and learning, too!) about them.

If you have any suggestions, or if you find any errors, please don't hesitate to contact me through GitHub, Twitter, or LinkedIn.

I'm looking forward to hearing back from you!

Daniel Voigt Godoy, April 24th, 2021

☺ "*You're still here? It's over. Go home!*"

Ferris Bueller

Sorry, but I **had to** end with a silly joke :-)

[153] https://github.com/dvgodoy/PyTorchStepByStep/blob/master/Chapter11.ipynb

[154] https://colab.research.google.com/github/dvgodoy/PyTorchStepByStep/blob/master/Chapter11.ipynb

[155] https://ota.bodleian.ox.ac.uk/repository/xmlui/handle/20.500.12024/1476

[156] https://ota.bodleian.ox.ac.uk/repository/xmlui/handle/20.500.12024/1740

[157] https://ota.bodleian.ox.ac.uk/repository/xmlui/

[158] https://ota.bodleian.ox.ac.uk/repository/xmlui/bitstream/handle/20.500.12024/1476/alice28-1476.txt

[159] https://ota.bodleian.ox.ac.uk/repository/xmlui/bitstream/handle/20.500.12024/1740/wizoz10-1740.txt

[160] https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

[161] https://nlp.stanford.edu/IR-book/

[162] https://huggingface.co/docs/datasets/quicktour.html

[163] https://huggingface.co/docs/datasets/exploring.html

[164] https://huggingface.co/docs/datasets/loading_datasets.html

[165] https://huggingface.co/datasets

[166] https://huggingface.co/docs/datasets/processing.html

[167] https://radimrehurek.com/gensim/

[168] https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

[169] https://nlp.stanford.edu/IR-book/

[170] https://wordnet.princeton.edu/

[171] https://github.com/QData/TextAttack

[172] https://bit.ly/2ObRNvH

[173] https://bit.ly/3ehmLxk

[174] https://lena-voita.github.io/nlp_course.html

[175] https://arxiv.org/abs/1301.3781

[176] https://jalammar.github.io/illustrated-word2vec/

[177] https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html

[178] https://machinelearningmastery.com/develop-word-embeddings-python-gensim/

[179] https://github.com/RaRe-Technologies/gensim-data

[180] https://www.aclweb.org/anthology/D14-1162/

[181] https://arxiv.org/abs/1802.05365

[182] https://allennlp.org/elmo

[183] https://bit.ly/2OIfmNb

[184] https://lilianweng.github.io/lil-log/2019/01/31/generalized-language-models.html

[185] https://github.com/flairNLP/flair

[186] https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_1_BASICS.md

[187] https://github.com/flairNLP/flair/blob/master/resources/docs/embeddings/ELMO_EMBEDDINGS.md

[188] https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_4_ELMO_BERT_FLAIR_EMBEDDING.md

[189] https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_3_WORD_EMBEDDING.md

[190] https://github.com/flairNLP/flair/blob/master/resources/docs/embeddings/CLASSIC_WORD_EMBEDDINGS.md

[191] https://huggingface.co/transformers/master/pretrained_models.html

[192] https://github.com/flairNLP/flair/blob/master/resources/docs/embeddings/TRANSFORMER_EMBEDDINGS.md

[193] https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_5_DOCUMENT_EMBEDDINGS.md

[194] https://arxiv.org/abs/1810.04805

[195] https://yknzhu.wixsite.com/mbweb

[196] https://en.wikipedia.org/wiki/English_Wikipedia

[197] https://github.com/google-research/bert

[198] https://huggingface.co/transformers/model_doc/bert.html

[199] https://huggingface.co/bert-base-uncased

[200] https://jalammar.github.io/illustrated-bert/

[201] https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

[202] https://huggingface.co/transformers/tokenizer_summary.html

[203] https://blog.floydhub.com/tokenization-nlp/

[204] https://arxiv.org/abs/1910.01108

[205] https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[206] https://openai.com/blog/gpt-2-1-5b-release/

[207] https://github.com/openai/gpt-2

[208] https://demo.allennlp.org/next-token-lm

[209] http://jalammar.github.io/illustrated-gpt2/

[210] https://amaarora.github.io/2020/02/18/annotatedGPT2.html

[211] https://github.com/karpathy/minGPT

[212] https://huggingface.co/blog/how-to-generate

[213] https://github.com/huggingface/transformers/blob/master/examples/pytorch/language-modeling/run_clm.py