Getting Started with Deep Learning for Natural Language Processing

Learn How to Build NLP Applications with Deep Learning

SUNIL PATEL

bpb

Getting Started with Deep Learning for Natural Language Processing

Learn How to Build NLP Applications with Deep Learning

Sunil Patel

www.bpbonline.com

FIRST EDITION 2021

ISBN: 978-93-89898-11-8

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

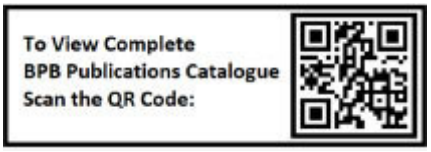
Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

My family

About the Author

Sunil Patel has completed his Master's in Information Technology from the Indian Institute of Information Technology-Allahabad, with a thesis focused on investigating 3D protein-protein interactions with deep learning. Sunil has worked with TCS Innovation Labs, Excelra, and Innoplexus before joining Nvidia. The main areas of research were using Deep Learning, Natural language processing in Banking, and healthcare domain. Sunil started experimenting with deep learning by implanting the basic layer used in pipelines and then developing complex pipelines for a real-life problem. Additionally, Sunil has participated in CASP-2014 in collaboration with SCFBIO-IIT Delhi to efficiently predict possible Protein multimer formation and its impact on diseases using Deep Learning. Currently, Sunil works as Data Scientist – III with Nvidia. In Nvidia, Sunil has expanded his area of interest to computer vision and simulated environments, and he extensively works in the banking, defense, and healthcare verticals areas. Sunil is currently focused on using GPUs for high-fidelity physics simulation. He has 3 pending US patents and 4 publications in the Deep Learning domain. To know more about his current research topic and interests, you can check out his LinkedIn profile:

About the Reviewer

Anurag Punia has 6 years of experience in data science and machine learning, with a special interest in topic modeling, information retrieval, and named entity recognition under the subfield of natural language processing. He has worked and delivered several data science projects across industry verticals, like insurance, asset management, marketing, tourism, and real estate. Currently, he is part of the center of excellence of a leading logistics company in Dubai, UAE. Anurag has a research-focused BS-MS dual degree from IISER Bhopal with a major in physics.

He can be reached at ***anurag.punia@gmail.com*** or ***https://www.linkedin.com/in/anurag-punia-data-scientist/***

Acknowledgements

First and foremost, I would like to thank God for giving me the courage to write this book. I would like to thank everyone at BPB Publications for helping me polish it and finally converting my writing to paperback.

I would also like to thank my parents, wife, and brother for their endless support and for helping me in numerous ways.

Lastly, I would like to thank my critics. Without their criticism, I would never be able to write this book.

Sunil Patel

Preface

"The world's most valuable resource is no longer oil but its data". Nowadays, titans and the most valued firm in the world like Amazon, Google, Apple, and Microsoft have similar concerns as were raised for oil a century ago. Data is changing the way we live, and the amount of data generated in the past few years is more than that generated since human beings have existed. The amount of data is expected to grow exponentially with the boom in connected devices, personal assistants, blockchain, and mobile devices.

The condition for the storage of data is getting favorable, as storage devices are getting cheaper 3X every 3 years. Hardware giants like Nvidia already claimed to have broken Moore's law, which also indicates the exponential growth in processing power. Today's world is highly favorable to the data-centric economy. And that's exactly why data is the next oil.

Unstructured and structured data is increasing at a similar rate. The former comes from a majority of sources, and algorithms are constantly being discovered to store and assimilate such data. Unstructured data can be anything, for example, scientific literature, randomly clicked selfies, chat messages, sensor data from self-driving vehicles, and voice/video over the Internet. It is rich in information, but processing such data and training a machine using such data is challenging. However, advancement has been made in gaining better understanding of unstructured data and using such a pre-trained network for supervised learning

in recent years. This technique is popularly known by the term "Transfer Learning".

Transfer learning decreases training time and also requires less amount of training data to achieve state-of-the-art results. Another type of data is structured data, which is majorly manually curated or generated semi-automatically. Actually, structured data is a bar of gold, an asset that costs millions and is capable of paying back in billions.

Machine learning is being extensively used in the field of medical diagnostics. Recently, the Food and Drugs Administration (FDA) developed a robot named IDx DR as the first autonomous AI-based diagnostic system. Yet another San Francisco startup developed a text recruit system called Automated Recruitment Interface (ARI), which is capable of holding a two-way conversation with candidates. It is also capable of posting job advertisements and openings, scheduling and conducting interviews, and maintaining all updates along the entire hiring funnel. Startups and firms are developing a system like Artificial Intelligence Virtual Artist (AIVA). Firms like Melodies and Google are generating music using artificial intelligence.

In a popular blog by Andrej Karpathy "The Unreasonable Effectiveness of Recurrent Neural Networks," he demonstrated that LSTM Models can easily generate lyrics. The days are not far when there will be robots making food in the restaurant and serving it while singing beautifully. This generated music will be rated by you and will be instantly sold live in another part of the world based on your ratings. The new wave has been created by

Google duplex—an AI engine that can make a call on your behalf to make reservations.

Machine learning has an equal number of applications in the fields of vision and text. Vision-related use cases exist in robotics, self-driving cars, self-flying vehicles, optical character readers, surveillance cameras, and security systems. The application of machine learning techniques on text is also known as Natural Language Processing (NLP), which can be applied to applications like text summarization, sentiment analysis, intend analysis, plagiarism detection, language translation, topic extraction, and audio language translation, text to speech and speech to text.

In the last 2 years, the GLUE score rose by almost 15 points from 64.7 to 80.4. The General Language Understanding Evaluation (GLUE) benchmark is a collection of resources for training, evaluating, and analyzing natural language understanding systems. Various state-of-the-art models like ELMo, ULMFiT, OpenAi transformer and Brat like models have come up and are constantly shaking up previous state-of-the-art models. This is an ImageNet movement for text.

This book is a comprehension of all the resources requires to not only learn NLP but to master it, and it is written keeping a beginner's skillset in mind. This book covers the entire spectrum, from understanding the basic concept of machine learning to the application of complex networks like generative networks, reinforcement learning, and speech processing in NLP.

In chapter we will learn about the basics of machine learning. The chapter includes basic concept like understanding data, when to apply machine learning, understanding various aspects of training a model, the founding principle of machine learning and AI, generalization, and dealing with overfitting and underfitting. This chapter will cover diagnostic concepts like bias-variance tradeoff, training and learning curves, generalization, and regularization concepts.

In chapter we will learn basic text processing. This chapter will cover the use as well as the implementation of techniques like stemming, lemmatization, and tokenization. This chapter covers basic operation and network building with Pytorch. Learning about Pytorch helps users quickly compile the network as per the desired thought process. As the scope of this book is focused toward NLP, we will also explore a utility called TorchText. It alleviates many problems related to text processing and also helps easily distribute data to multiple GPUs.

Chapter 3 is about converting/ representing our text into vectors so that it can be easily consumed by models. This chapter will cover various vital techniques like TF-IDF and Word2Vec. In addition to traditional techniques, also it will cover character-based vector embedding techniques like FastText.

Chapter 4 will cover Recurrent Neural Network (RNN), which is considered a milestone in sequence processing techniques. Going ahead of Vanilla RNN, this chapter will also help readers understand as well as implement the Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) Units. This chapter

will cover topics like a batch implementation of recurrent networks, attention architecture, and highway networks that enable us to train very deep sequence models.

Chapter 5 will take you through Convolution Neural Networks (CNN), which are heavily used in text processing nowadays. In this chapter, we will learn basic convolution operations and the effect of various parameters related to CNN to the accuracy of the concerned task. This chapter also covers concepts like Dropout and batch normalization, which help achieve greater accuracy with CNN. We will also cover advanced architectures like DenseNet. After covering everything required to get going, it's time to use the generated model in an unsupervised way or by someone else in our task.

Chapter 6 will explore vital topics required to apply transfer learning with text. This chapter will cover advanced architectures like ELMo-Bilm, sentence to vector, skip thought and InferSent.

All previous chapters make for a good foundation, and now we will apply a combination of all techniques to practical NLP tasks like sentiment analysis, implementing various approaches of topic modeling, text generation, building named entity recognition, building text summarization engine, and building language translation model.

Chapter 7 will provide hands-on experience regarding all the listed use cases.

Chapter 8 is all about complex networks and very recent techniques. It will take you through Recurrent Convolution Neural Network (RCNN) and Siamese Network. This chapter will cover advanced techniques like Random Multi-Model, Snapshot Ensemble techniques, CTC loss Recognition, and Sentence Piece. It will also explore a wonderful application of RNN and CNN in generating captions from images.

Chapter 9 will help you understand the fascinating world of Ian Goodfellow and concepts like Nash Equilibrium, KL-Divergence, KL-Divergence, JS-Divergence and KullbackLeibler Divergence to understand working on the Generative Adversarial Network. We will look at tips and tricks to solve the problem of an unstable gradient in the GAN. Finally, we will understand and code different types of GAN like Variational Autoencoder, and learn the application of GAN in generating images from text.

Chapter 10 will walk you through more advanced techniques of speech processing. It will cover how audio signals are captured and stored and look at a small use case of spoken digit recognition with an end-to-end model. This chapter will also cover advance frameworks, like deep speech and deep voice, and their usage is covered.

At the end, the book look at how to perform faster training and better deployment by utilizing the latest development in hardware and software.

This book covers all the necessary topics from the basics of machine learning to advance NLP techniques. That said, one

should know the basic concepts of machine learning to quickly grasp these topics. This book assumes that you have hands-on experience with the basics of machine learning and libraries like Numpy, NLTK, Matplotlib, PIL, and Scikit-Learn. Libraries like PyTorch deal with the differentiation required during the backpropagation of Deep Learning models and keeps users away from the mathematics required in building such models from scratch. We will use PyTorch, but understanding basic algebra, statistics, and vector space will aid easier grasping.

Downloading the code

bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/fxrpk

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Understanding the Basics of Learning Process

Structure

Objective

Pre-requisites

Learning from Data

Implementing the Perceptron Model

Generating and Understanding "Fake Image Data" and Binary Labels

Understanding Our First Tiny Machine Learning Model

Coding the Model with PyTorch

Confirming the Convergence of the Model

Error/Noise Reduction

Understanding Confusion Matrix and Derived Measures

Defining Weighted Loss Function

BLEU Score

Bias-Variance Problem

SciKit Learn Functions to Build Pipeline Quickly

Managing the Bias and Variance

Learning Curves

Loading Data, Pre-processing

Using Simple Regression

Using Random Forest Regression

Regularization

L1 Regularization (Lasso Regularization)

L2 Regularization (Ridge Regularization)

Implementing Lasso Regression

Implementing ElasticNet

Training and Inference

Software-based Accelerated Inferring
Hardware-based Accelerated Inferencing
The Three Learning Principles
Model related concepts
Data related Concepts
Conclusion

2. Text Processing Techniques
Structure
Objective
Pre-requisites
Understanding the Language Problem
Introduction to Data Retrieval and Processing
Scrapping the Web Page
Parsing Data from XML and JSON Format
Understanding Stemming
Understanding Snowball Algorithm
Understanding Lemmatization
Understanding Tokenization
Using NLTK Tokenizer
Using Spacy Tokenizer
Getting Familiarized with PyTorch
Installation
Using TorchText
Visualizing Using TensorBoard
Showing Scalar Values on TensorboardX
Projecting Images to TensorboardX
Showing Text on tensorboardX
Projecting Embedding Values on tensorboardX
Conclusion

3. Representing Language Mathematically
Structure
Objective
Prerequisite
Encompassing knowledge to numbers
Understanding the different approaches of converting a word/token to its embedding
Understanding co-occurrence matrix
Constructing a co-occurrence matrix
Understanding TF-IDF
Term frequency
Inverse document frequency
Constructing TF-IDF matrix
Understanding Word2Vec
Understanding methods to train Word2Vec
Implementation
Word2Vec improved version
Sub-sampling
Word pairs and phrases
Negative sampling
Understanding GloVe
Defining learnable parameters
Defining loss function
Many important components
Understanding character-based embedding
Character-based embedding generation
Conclusion

4. Using RNN for NLP
Structure

Objective
Pre-requisites
Understanding Recurrent Units
Rolling and Unrolling
Implementing the Concept of Embeddings
Downloading Dataset
Pre-processing
Training
Understanding Advance RNN Units
Gating Mechanism in LSTM
Modified LSTM Units
Understanding and Implementing GRU
GRU with PyTorch
Understanding the Sequence to Sequence Model
Implementing Sequence Encoder/Decoder
Encoder
Decoder
Actual Training
Evaluation
Understanding Batching with Seq2Seq
Decoder Phase
Encoder and Decoder with Batching
Decoder
The Loss Function for Sequence to Sequence
Translating in Batches with Seq2Seq
Implementing Encoder/Decoder Capable of Batch Processing
Encoder
Decoder
The Loss Function for Sequence to Sequence

Implementing Attention for Language Translation

Encoder
Attention Mechanism
Decoder
Conclusion

5. Applying CNN in NLP Tasks
Structure
Objective
Pre-requisites
Understanding CNN
Understanding Convolution Operations
Convolution Layers
Padding
Stride
Pooling layers
Fully Connected Layers
Convolution 1D
Convolution 2D
Pool Layers
Rectifier Linear Unit (Relu)
Using Word Level CNN
Pre-processing
Embedding
Convolution Layers
Using Character Level CNN
Understanding Character Representation
Network Architecture
Using Very Deep Convolution Network
The Convolution Block

Understanding the Network
Training Deeper Networks

ResNet
Highway Network
DenseNet
Fundamental Block of ResNet
Fundamental Block of Highway Network
DenseNet
Conclusion

6. Accelerating NLP with Transfer Learning
Structure
Objective
Pre-requisites
Introduction
Understanding the Transformer
Source and Target Masking
Positional Encoding
Converting Sentence to Vector
Sentence to Vector
Skip Thought
Getting to Know Contextual Vectors
Using the Pre-trained Model
Training Supervised Embedding
Playing with InferSent
Understanding and Using BERT
Conclusion

7. Applying Deep Learning to NLP Tasks
Structure

Objective
Technical Requirements

Topic Modeling
Applying LDA
Text generation
Understanding the Network
Building Text Summarization Engine
Abstractive Text Summarization
Building Language Translation Using a Transformer
Using a Transformer
Advancing Sentiment Analysis
Understanding Attention Mechanism
Building Named Entity Recognition
Word-level NER
Character-level NER
Conclusion

8. Application of Complex Architectures in NLP
Structure
Objective
Technical Requirements
Understanding SentencePiece
Understanding Random Multi-Model
Creating Flexible Networks
Using RMDL
Applying RMDL on Reuter Data
Ensembling by Taking a Snapshot
The Learning Rate Modifier
Recording Snapshots
Predicting Using Snapshots

Getting to Know Siamese Networks
Dataset Description
Loading and Pre-processing Data

Constructing a Sister Network
The Stem
Application of RCNN
Preparing the Dataset
Why Is It Difficult?
How Can It Be Solved?
Predicting Using CNN
Predicting Using RCNN
Understanding CTC Loss
The Simplest Choice
How Does CTC Work?
Loss Calculation
Understanding Decoding
Installation
Usage
Captioning Image
Downloading the Data
Implementation
Encoder Module
Decoder Module
Beam Search
Variants
Conclusion

9. Understanding Generative Networks
Structure
Objective

Technical Requirements
Understanding Unsupervised Pretraining
GAN Components
The Generator

The Discriminator
The GAN Architecture
The Loss Function
Implementing GAN for MNIST
The Understanding Theory behind GAN
Generating an Image from the Description
Conclusion

10. Techniques of Speech Processing
Structure
Objective
Technical Requirements
Learning about Docker
Getting to Know Phonemes
Loading an Audio File
Playing an Audio File
Visualizing the Signals
Feature Extraction
MFCC — Mel-Frequency Cepstral Coefficients
Spectral Centroid
Spectral Rolloff
Training a Small Network
Feature Extraction
Constructing the CNN Model
Training and Estimating Performance on the Test Set
Understanding Speech to Text

Installation
Datasets
Pretrained Model
Training
Visualizing Training

Dataset Augmentation
Checkpoints and Continuing from Checkpoint
Testing/Inference
Running a Server
Understanding Text to Speech
Grapheme to Phoneme Model
The Segmentation Model
Phoneme Duration and Fundamental Frequency Model
Audio Synthesis Model
Download Dataset
Installation
Preprocessing
Training
Monitoring using TensorBoard
Using the model for synthesis
Conclusion

11. The Road Ahead
Structure
Objective
Efficient Training
Parallel Data Loading
Utilizing Hardware Resources
Efficient Deployment
Hardware-related Optimizations

Conclusion

Index

CHAPTER 1

Understanding the Basics of Learning Process

This chapter covers the most basic aspects of machine learning. It will help you in understanding the basic mathematical representation of learning algorithm and teach you how to design a machine learning model from scratch. After constructing this model, we will understand the methods used to gauge the model's prediction accuracy using different accuracy metrics. Going further, we will coer the bias-variance problem and diagnose such a problem with a technique called learning curves. Once we get our model correct, we need it to generalize well on unknown datasets that can be understood through a chapter on regularization. After perfecting such a model, it must be efficiently deployed to help improve speed and accuracy.

Structure

In this chapter, we will cover the following topics:

Learning from data

Error/noise reduction

Bias-variance reduction

Learning curves

Regularization

Training and inference

The three learning principles

Objective

Building a simple model for efficient training and gauging the model's accuracy will be covered in this chapter. It will help you understand the usage of popular software and hardware acceleration for faster training and inference. We'll end this chapter with three learning principles that are extremely important to machine learning.

Pre-requisites

I have provided some of the examples through code, and the code for this chapter are present in ch1 folder at GitHub repository Basic know-ledge of the following Python packages is required to understand this chapter:

NumPy

Scikit-Learn

Matplotlib

Pandas

This chapter has one example that uses PyTorch for demonstration. If you don't know PyTorch, we will cover it in detail in *Chapter* Text Pre-Processing Techniques *in* NLP.

Learning from Data

In this data-centric world, a little improvement in the existing application can potentially help earn millions. We all remember a big prize (the $1,000,000) that Netflix gave to the winner for improving the algorithm's accuracy by 10.06%. A similar opportunity exists in financial planning, be it Forex forecasting or trade market analysis. Minute improvements in such use cases can provide beneficial results. One must explore the entire logic behind the process to improve something; this is what we call learning from data. Machine learning is an interesting area where one can use historical data to make a system capable of identifying observed patterns in the new data. However, machine learning cannot be applied to all problems; a rule of thumb is considered to decide whether machine learning should be applied to a given problem:

There must exist a hidden pattern

We cannot find such a pattern by applying simple mathematical approaches

There must be historical/relevant data about the task concerned

In this chapter, I will start with a basic perceptron model to give you a taste of the learning model. After building the perceptron

model, I will discuss Error/Noise and its detection using bias-variance and learning curves. In the learning curve, we will look at how the complexity of the algorithm helps mitigate a high-bias problem. Then, we will cover regularization techniques to achieve better generalization. All these techniques help get a better model, and then it's time to deploy such a model efficiently. Later in the chapter, we will cover techniques for faster and better inferencing and then look at the three learning principles that are not directly related to machine learning but help in giving state-of-the-art performance. Before going further, we will briefly discuss the mathematical formalization of any supervised learning problem.

Let's assume that we are talking about supervised learning paradigm. Supervised learning takes finite pairs of X and Y for learning. X and Y can be of the different types according to learning the goals. In the following table, we can see some examples with the nature of X and Y described for different learning problems:

problems:
problems: problems: problems: problems: problems:
problems: problems: problems: problems:
problems: problems: problems: problems:
problems: problems: problems: problems:
problems: problems:
problems: problems:
problems: problems: problems: problems: problems:

problems: problems: problems: problems: problems:
problems: problems:

Table 1.1

Here, Y is the label for X. Each X and Y is paired, as shown in the following equation:

=

Where are individual data points in and are individual data points in The main task of our hypothesis function *f* is to apply it over to predict = The goal is to predict so that it is the same as or near the original label and the Error between the predicted label and the original one(|Ŷ – Y|) tends to become 0. The function is = ..., The overall procedure to learn any function can be summarized as in the following flow diagram:

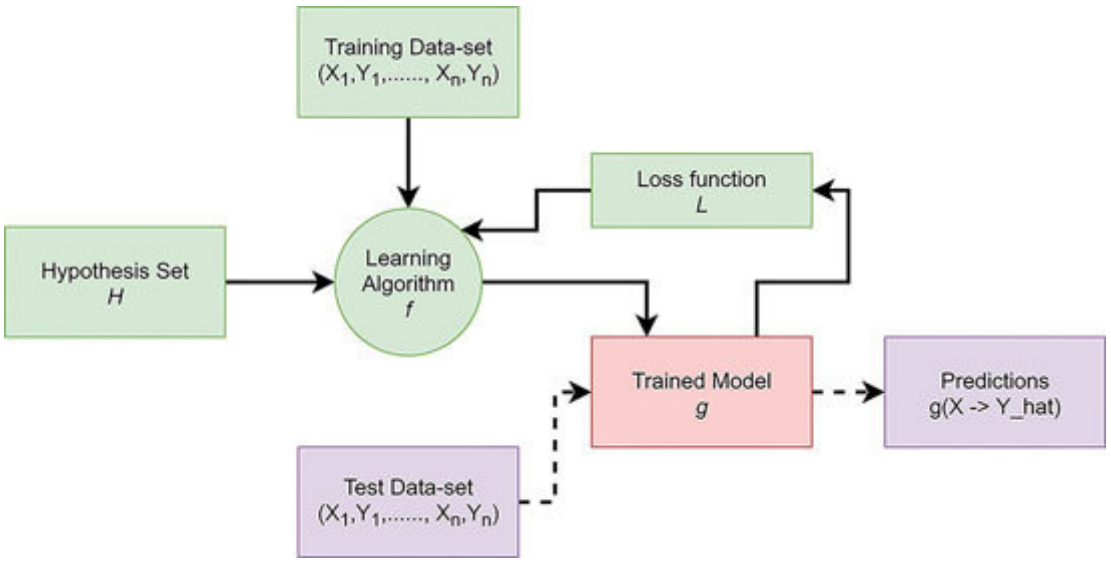


Figure 1.1: *Supervise learning paradigm with all major components involved in training and evaluation.*

As shown in the preceding figure, any of the hypotheses h is used for making predictions. Here, hypothesis can be anything like Linear, Logistic, Polynomial, SVM, RF, or Neural Network. Hypothesis h is also called function f in general terms.

There are also other types of learning techniques, like unsupervised learning and reinforcement learning. An unsupervised learning paradigm has no label attached to the data; it only has $\in$ and there is no Y label. In fact, unsupervised methodologies are gaining popularity nowadays and are responsible for pushing the state-of-the-art model in the field of vision and NLP even higher. Popular models like Bert and Megatron are examples of unsupervised models. On the other hand, reinforcement learning is a technique where an agent tries to maximize the immediate or cumulative reward by learning/adapting to a given environment. We will learn and apply unsupervised learning to NLP problems in the upcoming chapters.

Implementing the Perceptron Model

Well, this chapter is a little out of sync, but we will discuss the perceptron model. Perceptron with no activation function is a linear model. The perceptron algorithm was invented by **Frank Rosenblatt** in 1957 at the Cornell Aeronautical Laboratory. To date, this is the most important and widely used model in the course of the machine learning.

Let's take all the features, that is, and try to derive a hypothesis *h* that maps y ⇒ ŷ., where ŷ is the predicted label and y is the original label. Our hypothesis *h* can be one of the simplest possible perceptron models with a linear activation function. The preliminary hypothesis can be thought of as in the following equation:

$$h(x) = \sum_{(i=0)}^{N} (w_i x_i) b_i > Threshold$$

Where are learnable weights, changes as per the feedback signal received due to calculated loss, and is the bias term. Bias is used in the Perceptron network for phase shifting. The preceding equation can be thought of as the step function, a simple linear function if the predicted value is above the threshold then then the label is 1 else 0.

$$h(x) = Step\left(\sum_{(i=0)}^{N}(w_i x_i) + b_i\right) = Step(W^T X)$$

Where *X)* is the Matrix product. Matrix-based computations are faster than iteration-based computations, and GPU supports Matrix-based operations well. To see whether this hypothesis can solve our problem, I have taken a hypothetical dataset and applied the discussed hypothesis. In the present model, I will use the **Mean Squared Error** loss. MSE loss can be defined as:

$$MSE(y_i, \hat{y}_i) = 1/N \sum_{(i=0)}^{N} (y_i - \hat{y}_i)^2$$

We will use the **Stochastic Gradient Descent** as the optimizer that tries to find the best solution by exploring multi-dimensional terrain. We will learn about how SGD works while discussing the workings of RMS prop (another advanced optimizer) in the upcoming chapters. Although terms like SGD and MSE might seem intimidating, these are simple mathematical equations, the purpose of which will be better understood as we move ahead in the chapter.

Generating and Understanding "Fake Image Data" and Binary Labels

The following fake data with two classes—0 and 1—are generated to validate the preceding hypothesis:

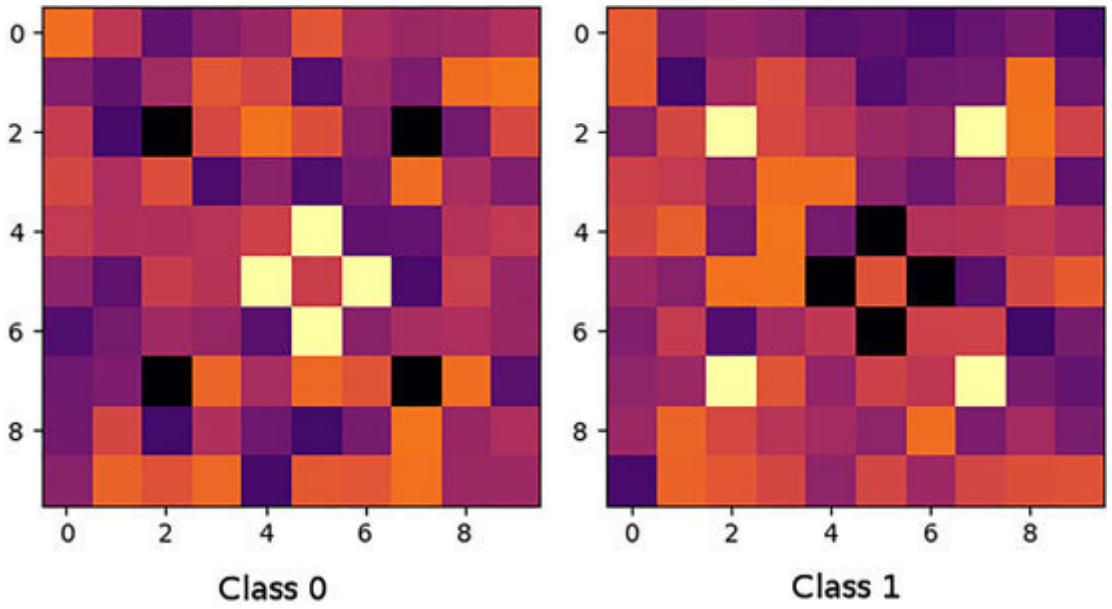


Figure 1.2: *An imaginary dataset to demonstrate learning using a simple perceptron model.*

This simple dataset is 10*10 matrix, with each location having a random float shown by variable color temperature. **Class 0** has **1** at locations and **0** at locations On the other hand, Class 1 has 0 at locations and **1** at locations Only 8 of 100 indicative points are present, and our algorithm must differentiate between these two classes based on these 8 points. The other 92 points are considered noise to the dataset, and the algorithm must be good enough to avoid this noise.

Understanding Our First Tiny Machine Learning Model

A simple model can be visualized as follows:

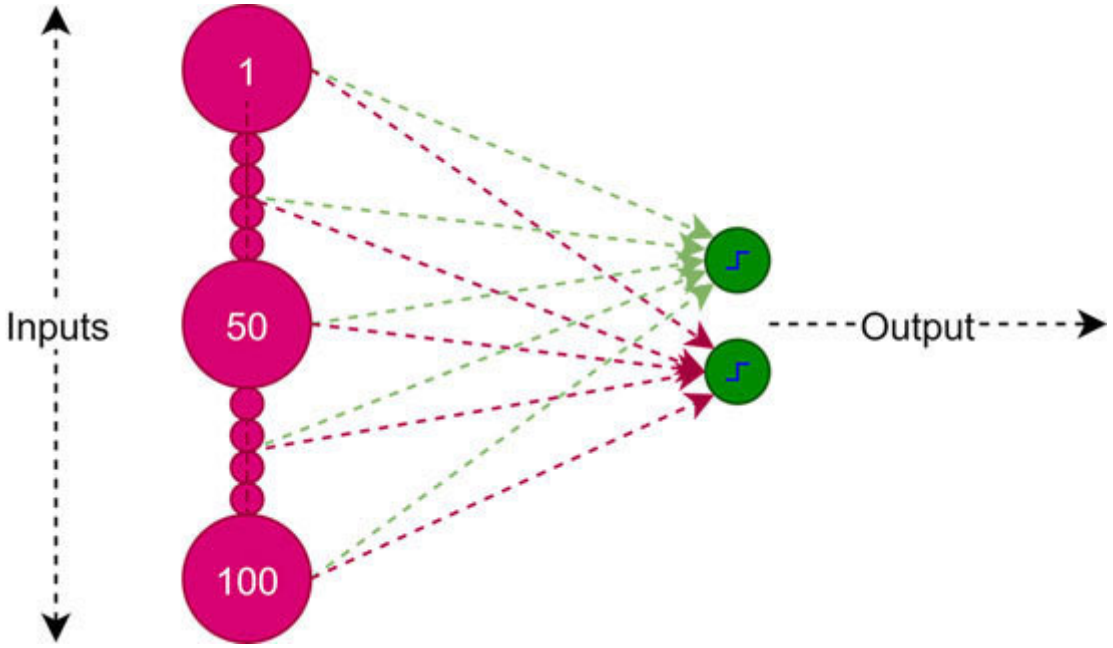


Figure 1.3: *Simple Perceptron model with the step function.*

As shown in the figure, our model accepts 100 input features (10*10 flattened). Each input is multiplied by weight to produce the final output. The input of size [1, 100] is multiplied with the weight matrix [100, 2]; multiplication yields [1, 2] output, on which the threshold (step) is applied. The main question is, why are there two outputs? Generally, the output is provided as one-hot encoded; 0 is encoded to [1, 0], and 1 is encoded to [0, 1]. Now, let's understand why one hot encoding. If we use only one output, where it can be 0 or 1, the model may predict 0.5 as the output every time and take this as the better alternative to accurately

predicting 0 and 1. This problem worsens when the number of classes increases, but one-hot encoding forces the network to predict well in extremes and minimizes loss.

Coding the Model with PyTorch

I have used PyTorch version 0.4.1 to design the preceding simple model. PyTorch lets us design the network as per our thought process, and the following is the simple network we just discussed. If you don't get this, we will cover these things in detail in *Chapter* Text Processing Techniques. Here, network can be designed simply by extending the torch.nn.module class. Then, you must implement two functions, namely __int()__ **and** The __init__ function initializes as well as defines various layers. In our network, we have defined a simple linear layer (nn.Linear) that takes two inputs: input dimension and output dimension. In our case, the input dimension is 100, and the output dimension is The following code block illustrates how to construct a simple network that takes a vector as the input and produces two-dimensional outputs:

```
class simple_module_1(nn.Module):
def __init__(self):
super(simple_module_1, self).__init__()
self.simple_linear = nn.Linear(100,2)
def forward(self, input):
return self.simple_linear(input)
```

We don't have to worry about the weight and its dimensions, as the framework will take care of that based on the input and output size.

Confirming the Convergence of the Model

After training for 200 epochs, the network determines the peculiarities between classes. As the epochs progresses, loss decreases and accuracy increases.

The following image shows the progress of **Loss/ Accuracy** as **Epochs** progresses:

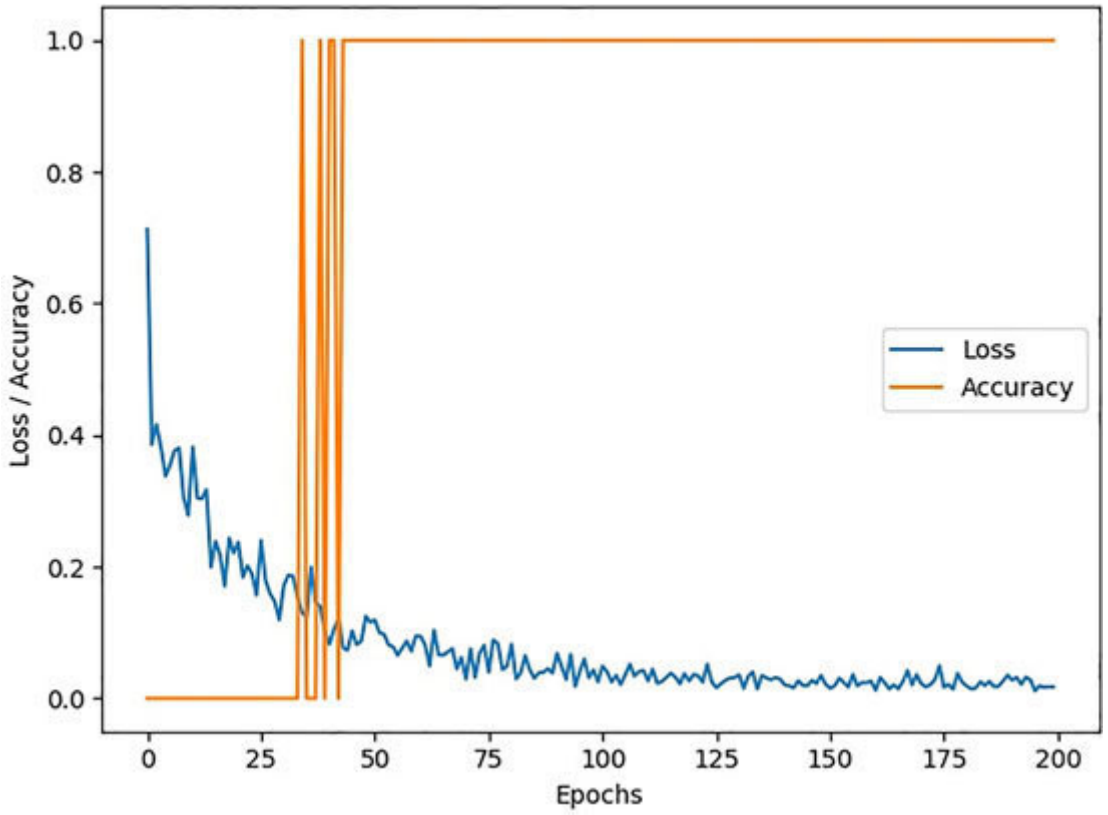


Figure 1.4: *The progress of loss/accuracy as Epochs progresses.*

To demonstrate the simplest learning model, I have generated dummy data; you can experiment the same with the dummy_data.py Python script. I used imaginary data generator function to produce dummy data and put it into the apply_perceptron.py script. In many other functions for data generation, PyTorch Model, Loss Calculator, Optimizer, Accuracy Calculator, and function to render plot are present.

The following code uses mean square error as the measure of the error. It is one of the loss functions used according to learning goals. We will discuss the mathematics of these loss functions in the next section on error/noise Also, the SGD optimizer decreases loss by back-propagating error and adjusting weights:

```
define loss
objective = nn.MSELoss(reduce=True)
# define Optimizer
optimizer = optim.SGD(simple_module.parameters(),lr=0.01)
```

Error/Noise Reduction

Suppose we have a dataset, as shown in the following figure:

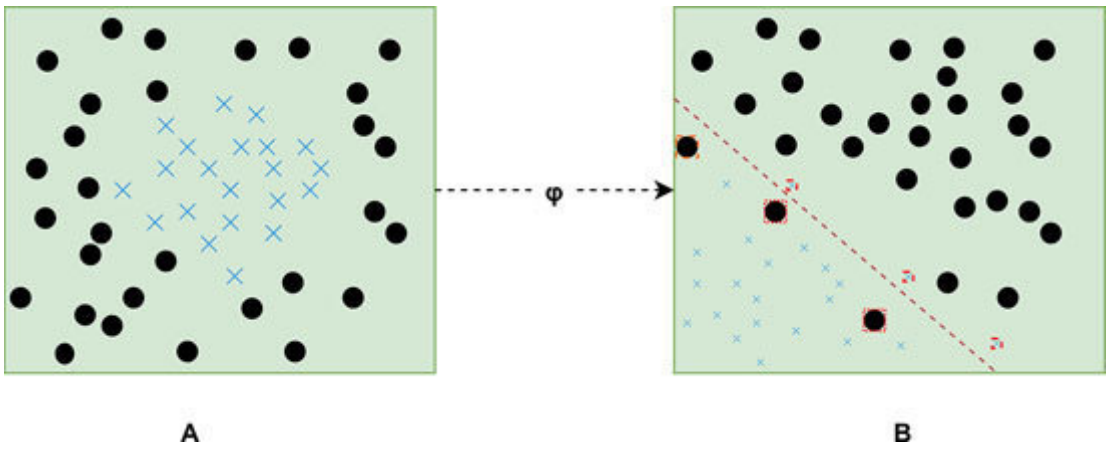


Figure 1.5: *Data before and after applying the non-linear transformation.*

Suppose we have the data shown in the preceding figure where each point belongs to superset in short ∈ To this data, we apply a transformation φ and the new dataset distribution, as shown in the same figure (B), can be given as = and ∈ Looking at the figure (B), we understand that the data is now linearly separable, and any linear model like linear regression or Perceptron model can separate such data. Transformations like φ are never perfect and often end up categorizing some of the points wrong. As shown in the preceding figure some of the points are wrongly categorized by the separating line. Now we need measures to quantify these errors.

Mathematically, we measure such errors by loss functions like the following:

Mean squared error:

$$MSE(y_i,(\hat{y}_i) = 1/N\sum_{(i=0)}^{N}(y_i - (\hat{y}_i)^2$$

Mean absolute error:

$$MSE(y_i,(\hat{y}_i) = 1/N\sum_{i=0}^{N}|\, y_i - \hat{y}_i\,|^2$$

Cross-entropy error:

$$CE(y_i,(\hat{y}_i) = -\sum_{x\in X} y_i(x)\log\hat{y}_i(x)$$

The loss function can also be problem-specific. In the next chapter, we will define sequence to sequence loss, which will be used particularly for language translation. The triple loss is a custom loss function used in style transfer applications. In fact, we must provide different penalties for different cases to meet specific requirements. We will discuss such a case in this chapter.

Understanding Confusion Matrix and Derived Measures

To quantify the goodness of our model, we generally use the confusion matrix. It is a table that allows the visualization of a comparison of our algorithm's performance. It has four values based on four cases:

True Positives These are cases in which we predicted yes, correctly

True Negatives We predicted no, correctly

False Positives We predicted yes, wrongly

False Negatives We predicted no, wrongly

The confusion matrix looks as follows:

follows: follows:
follows: follows: follows:
follows: follows: follows:

Some of the derived performance measures like accuracy, precision, sensitivity, and specificity can be calculated as follows:

Specificity, Selectivity or True Negative Rate(TNR)=TN\/(TN+FP)

Sensitivity, Recall, Hit Rate, or True Positive Rate(TPR)=TP\/(TP+FN)

Precision or Positive Predictive Value (PPV)=TP\/(TP+FP)

Accuracy=(TP+TN)\/(TP+TN+FP+FN

Apart from these measures, the confusion matrix helps derive other measures like F1-Score, Matthews Correlation Coefficient (MCC) and informedness.

Defining Weighted Loss Function

As discussed, loss function should be designed as per the end goal, and mistakes should be penalized accordingly by providing proper weights. To understand this concept, let's look at a few use cases:

Case 1 - A case of the grocery Suppose a grocery store decides to provide 25% off to their loyal customers, who are identified at the checkout counter by face recognition. Now, we gauge the accuracy of facial recognition based on the confusion matrix. In this case, our confusion matrix should be designed after considering the following:

TP and TN will have 0 penalty

FP means a customer was not very frequent but still got the discount

FN means a loyal customer's face was not identified correctly, and so they failed to get discount

In this case, more penalties should be applied to FN cases. We could not detect a loyal customer due to a system error, and we did not provide them with a discount. It leaves a bad impression, and we could lose the customer, so a higher penalty should be imposed on FN cases. FP cases do not do any harm; if a

customer accidentally gets the discount, they may like it and may visit again. FP cases may eventually enhance word-of-mouth promotion. In the loss function, if the penalty for FP cases is one, the penalty for FN cases should be 100.

Case 2 - An ATM security Let's say a company is designing the ATM feature where you press your thumb and get your money. The cases are as listed:

TN and TP are normal cases with 0 penalty

FP are cases where a person did not have an account, but got access to a random account and money when they put their thumb

FN are cases when a person with an account was not recognized

Here, FP cases should be taken very seriously. If the penalty for FN is 1, the penalty for FP should be 10,000.

In the upcoming chapters, we will discuss regularization to help us reduce errors in our algorithm and improve generalization to the test data; this is the reality of real-life data. Most often, you will deal with such data, and you'll rarely find clean data without noisy targets. The noisy target can be defined as the sum of deterministic target and some noise. Deterministic noise can be taken care of using various hypotheses or different feature sets. Noise is out of our control, and features to deal with such data

are not present in our example set. Also, confusion matrix is not the only type of matrix used to quantify learning.

BLEU Score

BLEU score is another measure for the quality of Language Translation and Text Summarization. The mean Average Precision (mAP) is used to measure the quality of the object detection algorithm. The GLUE benchmark is used to measure the quality of the language embedding task. There are many such measures, which we will discuss in detail.

Bias-Variance Problem

Our goal is clear—keep error, particularly test error, minimal. Two major sources of errors are bias and variance. To understand the logic behind the learning curve, we must understand bias-variance and the trade-off between them:

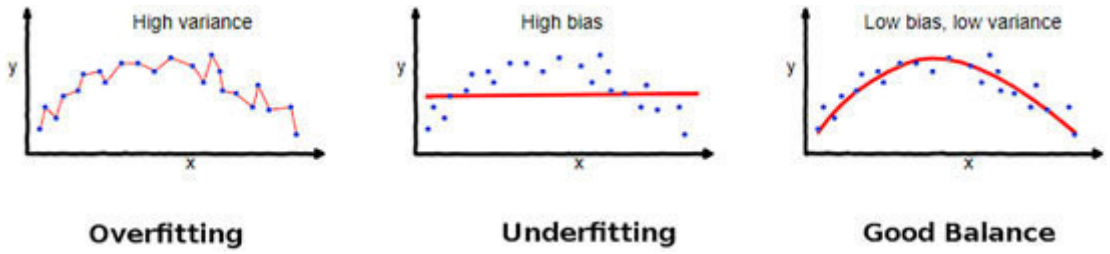


Figure 1.6: *Illustrating bias-variance trade-off in the data.*

Bias is a source of error, as a model with higher bias pays less attention to the training data and oversimplifies the model. Mathematically, bias can be formalized as follows.

Suppose we have training data and real-valued labels associated with each data point Let's say we define a function which effectively estimates taking but with minimum error Here, Here can be any learning algorithm like logistic regression, Perceptron, or any complex polynomial model. The mathematical formula for the same is $\hat{Y}$ = + where is the function which is very near to ideal function for minimal error, as shown in the following equations:

$$E = |yi - \hat{y}| or (yi - \hat{y})^2$$

Where if the error *E* is minimum and tends to become o in ideal cases. Bias can be mathematically represented as:

$$Bias\left[\hat{f}\right] = E\left[\hat{f}\right] + \hat{f}$$

There exist outliers in any distribution, and our function cannot be perfect enough to take care of all the points and still generalize well. So, there will always be an error *E* always, regardless of the hypothesis used. Bias is also known as under-fitting, where your model is incapable of learning from data and eventually ends up with high training and test error.

Let's look at the possible cause of the high bias:

No pattern The first assumption whereby we apply machine learning is that the data must have a pattern. If there is no pattern, no hypothesis can model it, so the model ends up with high training and test error. No model can solve this problem unless more representative data is provided. There can be an issue with the imbalanced dataset.

Modeling The model's architecture is incapable of learning the intricacies of the data. There can be multiple reasons like the use of the wrong activation function or a high learning rate. There can be an issue if the data is not evenly distributed among batches

and results in improper gradient propagation per batch and, so, no learning. Improper loss implementation can be one of the problems.

We can see a definite way to tackle the high bias problem:

Train Sometimes, algorithms take longer to learn higher-level abstraction and then converge quickly. This scenario is observed with image segmentation problems and more with deeper convolution neural network models. Allowing the algorithm to run for longer with a lower learning rate helps with convergence.

Train complex model / new model Another reason is that the model is not complex enough to learn the patterns in the data. For example, if the feed-forward model is applied to the text sentiment analysis problem, it may perform poorly as compared to the performance of the recurrent neural network applied to the same problem. Using non-linear kernels in SVM can help mitigate the problem.

Normalizing/increasing Adding more characteristic features can help. This can be understood with an example. One may never be able to predict the price bucket of the car by looking at some features like its color and length, but adding features like its maximum speed, gear shifting pattern, and pickup can help. In the case of the numerical feature, normalization of the feature between 0 and 1 can help with better convergence.

Adding This technique certainly helps and is widely used in the deeper convolution neural network models. Batch normalization is a well-known technique.

As we saw in the bias-related problem, the model was not flexible/complex enough to identify the patterns in the data. Variance is on the other side, where the model just mugs up the data. In other words, the model is specifically getting used to the training data and not to generalize the test data. Variance, also known as overfitting, can be mathematically denoted as .

High variance can be solved using the following techniques:

Increase We experience the high variance problem when there are lower data points and highly complex models. In this case, our model learns all the intricacies of the data and training error tends to reach zero, so no more learning occurs. Such a model provides poor performance when applied to test data belonging to a slightly different distribution than the training data.

Decrease the number of The trained data was so properly preprocessed that it has all necessary features, and the model takes thesedirect features and learns around them. When unprocessed test data is provided, the model cannot look at the feature and performs poorly.

The model may be fitting to noise in the data, due to which it is providing poor results on test data. We can reduce the model's flexibility by applying constraints, which is also known as

regularization. There is a definite solution in data science for this problem—the application of regularization techniques. Four types of regularization techniques can help overcome overfitting:

L1 regularization

L2 regularization

Elastic Net

Dropout

Reduce model Certain time-reducing model parameters also help a lot. Reducing the network by decreasing the layers in feed-forward network and CNN can prove helpful, and decreasing the hidden state size in the recurrent neural network can also help.

The We already learned about the bias-variance trade-off, so it's time to diagnose where the problem lies. Several methods can help identify such a problem, one of which is to plot the learning curve. After understanding the bias-variance problem, it is essential to understand, differentiate the problem, and solve it systematically. Mathematically, bias, variance, and some irreducible errors contribute to the overall error. This can be represented as:

$$E[(Y - \hat{f}(x))] = (Bias[\hat{f}(x)])^2 + (Var[\hat{f}(x)]) + \sigma^2$$

To practically model the bias-variance trade-off, we will take the help of polynomial equations and demonstrate the problem by taking polynomial equations of different degrees. Predictions are made on the held-out test set, and MSE is reported. To demonstrate the bias-variance problem, we will take the help of Scikit learn functions like Polynomial Features, Linear Regression, and Pipeline. The following code block defines a function that produces labels (Y) for a given value of X. This function will help us generate example data:

```
def true_fun(X):
“””
given X it will provide its mapping to Y by sing function
np.cos(1.5 * np.pi * X)
:param X:
:return:
“””

return np.cos(1.5 * np.pi * X)
```

We take 30 samples and calculate the polynomial of order and will generate random samples from a uniform distribution over The label for this function will be generated by the true_fun function:

```
n_samples = 30
degrees = [1, 3, 9, 15]
X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1
```

SciKit Learn Functions to Build Pipeline Quickly

We calculate polynomial features for each degree of the polynomial, and these polynomial features are used to perform linear regression. Such a model with polynomial features is fit for linear regression. Let's use the pipeline function to stack polynomial features and linear regression in one pipeline. Then, we'll use the pipeline function to predict text data:

```
polynomial_features = PolynomialFeatures (degree=degrees[i],
include_bias=False)
linear_regression = LinearRegression()
pipeline = Pipeline([("polynomial_features", polynomial_features),
("linear_regression", linear_regression)])
pipeline.fit(X[:, np.newaxis], y)
# Evaluate the models using cross-validation
scores = cross_val_score(pipeline, X[:, np.newaxis], y,
scoring="neg_mean_squared_error", cv=10)
X_test = np.linspace(0, 1, 100)
plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
```

We can see that polynomial with degree 1 is not fitting for the data, so it is called underfitting. A polynomial with a degree higher than 3 fits the data so well that it does not generalize to the test data and ends up with higher MSE, so it is called as overfitting. The following plot makes it clear that the MSE is

lowest with Degree = 3. It is the point of equilibrium between bias-variance and model complexity. You may visit the code bias_variance.py scripts to run or experiment with the code that produced the plots. The following image is generated by taking different degrees of polynomials to demonstrate bias-variance trade-off:

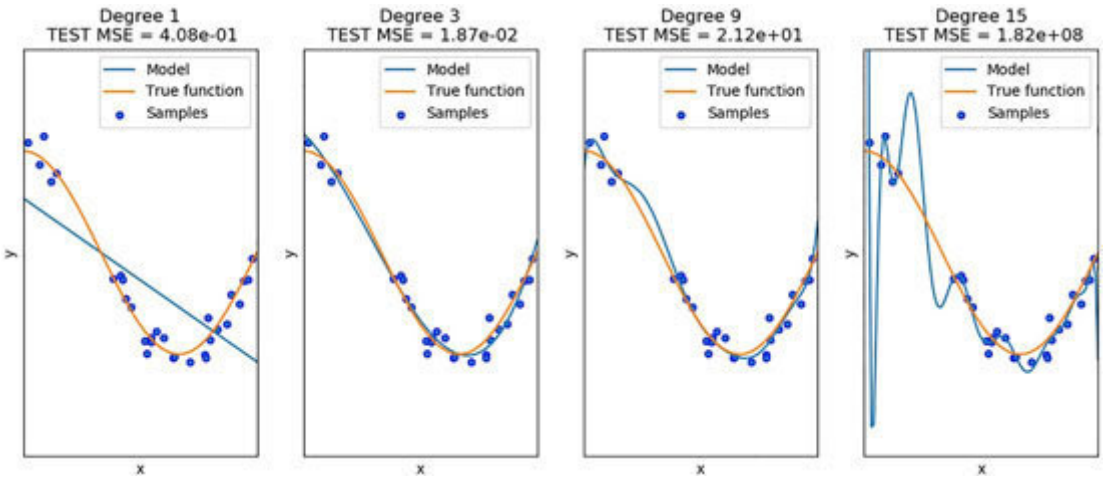


Figure 1.7: *Demonstrating bias-variance trade-off using a polynomial of different power.*

Ideally, the plot of error versus model complexity looks like this:

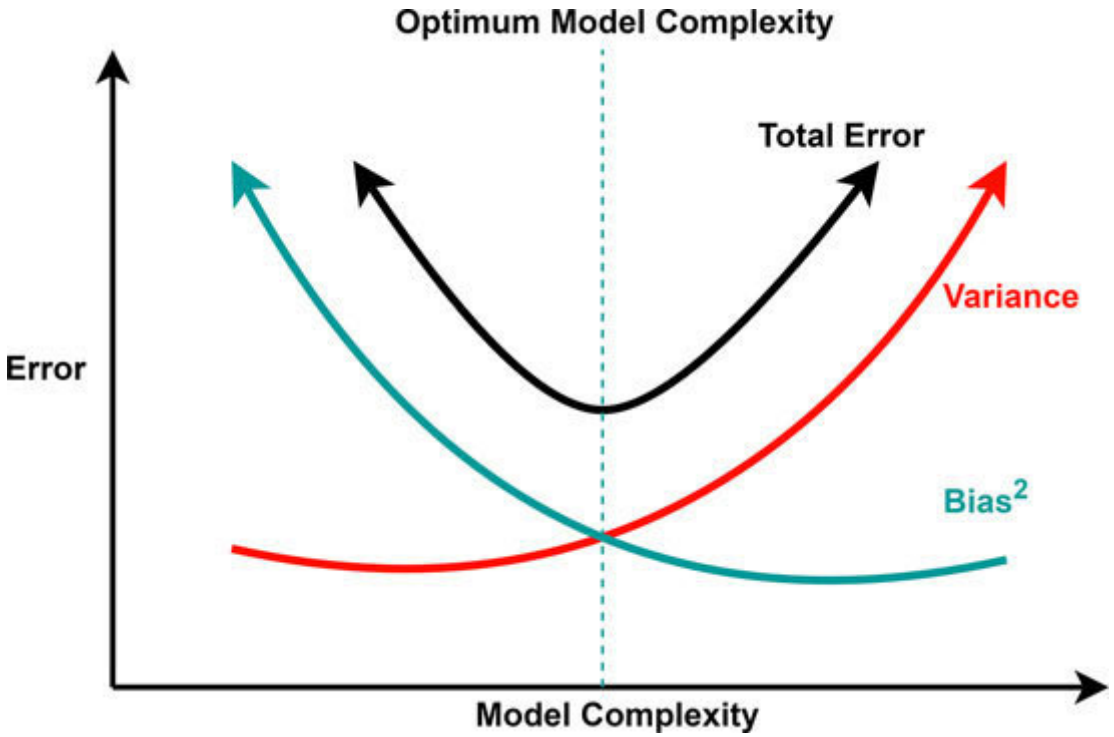


Figure 1.8: *Showing equilibrium between bias and variance.*

As the model's complexity increases, the bias increases, and so does the variance. The model with a complexity where the bias and variance are minimum should be selected for inference. Low complexity means the model is complex enough to absorb the complexity of data, so the problem of high variance occurs when we increase the model complexity. However, if we increase it gradually, given the model complexity, both the bias and variance are minimum at a point. As we further increase it, the model's complexity variance increases, also called model over-fitting problem.

Managing the Bias and Variance

Let's look at certain measures that can manage the bias and variance problem:

Bagging and Bagging refers to creating various partitions of the data and using random selection with replacement, and using each sample to train a model. Test time prediction is made using all these models, and the final prediction is made based on averaging or voting. This technique is also known as bagging or bootstrap aggregating. The Random Forest model makes good use of the bagging technique. Numerous trees are constructed based on each sample partition, and the entire model's bias is the same as that of a single tree. So, creating a forest of such trees and averaging them can significantly reduce the variance of the final model.

Fight Your Having a model that does not get trained or show convergence after certain epochs indicates that something is wrong with the model. Quick checks like lowering the learning rate, changes in momentum, using a different optimizer, using different loss functions, checking model implementation, and monitoring the progress for long enough can help a lot. One should always try to decrease bias at the cost of variance.

You can check the following link to know more on the Bias Variance problem: *http://scott.fortmann-roe.com/docs/BiasVariance.html*

Learning Curves

The learning curve is a diagnostic measure to identify the presence of a bias-related or variance-related problem. In the normal training test cycle, we just split the dataset into test a train, develop a model on the train data, and apply it on test data. To plot the learning curve, we follow a slightly different approach. We take an increasing number of training examples, such as 1, 2, 5, and 10, up to the entire training set and test it on test data. When we measure the training error for partition with an increasing number of examples, these are known as learning curves.

In a nutshell, the learning curve shows a change in training and validation error, as with the change in training set size. Initially, when the training sample is 1, the model fits it well, and while the training error is zero, the validation error is the highest. As we increase the training samples, the data cannot fit the training sample perfectly, so training error increases, but the validation error decreases. To demonstrate the concept of the learning curve, we will take the help of the dataset. Let's look at the local stability analysis of the 4-node star system (electricity producer is in the center) implementing the De-central Smart Grid Control concept. The electrical grid stability simulated dataset has four types of variables and stabf as target

Reaction time of participant

Nominal power consumed

Coefficient proportional to price elasticity

The maximal real part of the characteristic equation root (if positive - the system is linearly unstable)

The stability label of the system, which is the target variable

Loading Data, Pre-processing

We aim to produce a learning curve from this data. To quickly produce the learning curve, we will take the help of the learning_curve Scikit-learn function. This function determines cross-validated training and test scores for different training set sizes. The following code follows these steps:

Loading data.

Converting categorical string label to a numerical variable.

Defining training sample size as [1, 5, 10, 25, 20, 25, 50, 75]. The first iteration will use only one sample will be used for training, the next will use five samples, and so on.

Defining feature and target variables.

```
#loading data
data = pd.read_csv('Data_for_UCI_named.csv')
data.head()
# converting categorical to numerical
stav_int = []
for i in list(data["stabf"].values):
if i == "unstable":
stav_int.append(0)
else:
```

```
stav_int.append(1)
# assignnumerical variable to pandas data frame
data[“stav_int”] = stav_int
# defining various data fraction
train_sizes = [1, 5, 10, 25, 20, 25, 50, 75]

features =
[“tau1”,”tau2”,”tau3”,”tau4”,”p1”,”p2”,”p3”,”p4”,”g1”,”g2”,”g3”,”g4”,”stab”]
target = ‘stav_int’
```

Using Simple Regression

Let's now use a simple regression model and plot the learning curve using the following snippet:

```
train_sizes, train_scores, validation_scores =
learning_curve(estimator = LinearRegression(), X = data[features], y
= data[target], train_sizes = train_sizes,scoring =
'neg_mean_squared_error')
```

The following plot is the result of running the preceding script. The gap between training error and test error narrows, as the size of the training set increases; the bigger the gap between the two errors, the larger is the variance. In our case, the gap is extremely narrow, so our model does not have the variance problem.

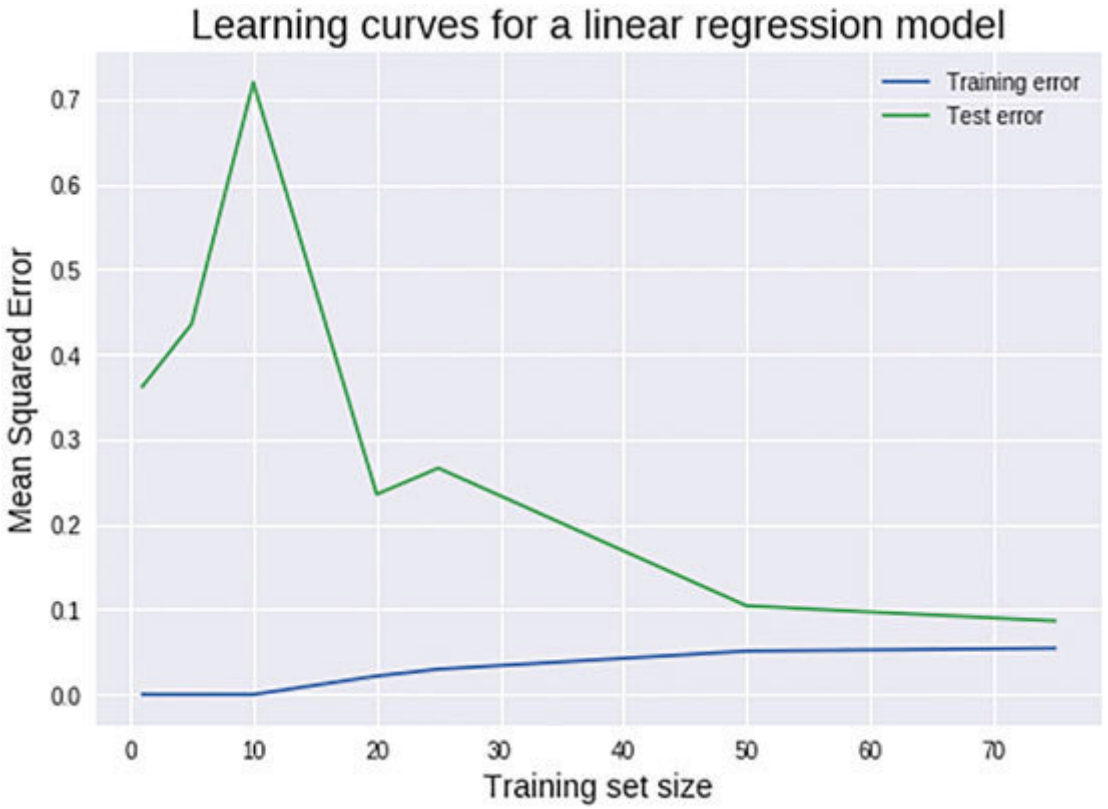


Figure 1.9: *Learning curves.*

The learning curve plotted using the linear regression model high training MSE and also represents low variance. In our case, the training error is already high, so we have low variance and high bias problems. Based on this, we can conclude that our model is underfitting the data, and adding more rows is highly unlikely to help it perform better using the current algorithm. We can try changing the algorithm to a more complex one. Increasing the features or moving to higher-order polynomial could also help. Besides, we can think of decreasing regularization to increase the flexibility of the algorithm and achieve a low-bias high-variance solution.

Using Random Forest Regression

The following code block applies complex algorithm to check if the variance decreases. I have applied the Random Forest algorithm to the same dataset using the RandomForest Regressor using the learning_curve function:

```
train_sizes, train_scores, validation_scores = learning_curve(estimator = RandomForestRegressor(), X = data[features], y = data[target], train_sizes = train_sizes, cv = 5, scoring = 'neg_mean_squared_error')
```

The resulting learning curve is as follows:

Figure 1.10: *Learning curves complex model*

This is the learning curve plotted using the Random Forest model. Now, it's clear that both the training and test error have decreased. It is a case of low bias and low variance. The gap between train error and validation error is also low, so we can conclude that the model can perform well on test data. You can look at the entire implementation in the learning_curves.py

We have probably arrived at a better solution than before. This model can be more complex by using a neural network like a model. You can experiment with the more complex model in the learning_curves.py script.

For further reference, you can check the following link:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html

Regularization

It's quite clear from the previous chapter of the bias-variance trade-off and learning curve that there is a higher chance that one may end up with higher bias and higher variance if no measures are taken. Regularization techniques are commonly used to deal with overfitting, and there are several regularization techniques in practice:

Lasso regularization

Ridge regularization

Elastic Net

Early Stopping

Drop-out

Ridge and Lasso are widely used in machine learning techniques, and Drop-out is used widely in deep learning techniques. There are many other regularization techniques, including DropConnect.

L1 Regularization (Lasso Regularization)

LASSO is short for Least Absolute Shrinkage and Selection Operator. L1 regularization adds a penalty equal to the absolute value of the coefficient and does both variable selection and regularization to increase the predictive accuracy of the model. Lasso regularization works by eliminating unwanted covariates and keeping only those that improve the model's efficiency. Before Lasso, step-wise regression was used, but it improves the efficacy only when some of the covariates have a strong correlation with the outcome. However, in a certain case, step-wise regression worsens the prediction.

Wherein feature sets . In , the subscript *i* represents the sample number and the super script 1 represents the feature number for that sample. Typically, in real-life data, each data point can have *n* features. is an intercept/bias coefficient, and is the coefficient, the liner regression model is defined as = + where represents the feature of the data point. The following is the formula for the Sum of Squared Error (RSS), where 0 to

$$RSS\ Loss\ Function = \sum_{i=0}^{N}(\hat{y}_i) - y_i)^2$$

And the changed function with penalty can be shown as:

$$RSS\ Loss\ Function(J(\Theta)) = \sum_{i=0}^{N}((\hat{y}_i) - y_i)^2 + \lambda \sum_{i=0}^{n} \beta_j$$

Here, is the cost function, and is the regularization term. The regularization coefficient penalizes all the parameters except intercept and prevents overfitting. As the model's complexity increases, L1 regularization adds the penalty for the higher term and decreases their importance, bringing the model toward a less complex solution and preventing it from overfitting. A tuning parameter, controls the strength of the L1 penalty. Here, λ is the amount of feature selection to be done:

In the equation, when λ = 0, no parameters are eliminated. The estimate equals to one which is found with the base model.

As λ increases, more and more coefficients are set to zero and eliminated. If λ = ∞, all coefficients are eliminated (practically, no one does that!)

Bias increases as λ increases.

Variance increases as λ decreases.

λ is sometimes regarded as Alpha (a).

L2 Regularization (Ridge Regularization)

L2 is another form of regularization widely used in machine learning. L2 regularization adds a square of the magnitude of coefficient as a penalty to the loss function. The following formula is for the sum of the square of error. L2 regularization only applies the regularization and does not make the variable selection. One thing peculiar about Ridge regularization is that it forces any coefficient to zero it. Still, with a higher λ value, it minimizes the effect of attributes on the trained model. Here's the changed loss function with the L2 penalty added to the original equation:

$$RSS\ Loss\ Function(J(\theta)) = \sum_{i=0}^{N} \left| \hat{y}_i - y_i \right|^2 + \lambda \sum_{i=0}^{n} \beta_j^2$$

Elastic Elastic Net is a convex combination of Ridge and Lasso regularizations. It emerges due to the drawback of L1 regularization, whose attribute selection is dependent on data, and so, very unstable. The solution is to combine both L1 and L2 to get the best of both worlds. Now, there are two parameters to choose from: α and λ. If α = 0 ridge regularization is used and if α = 1 for Lasso regularization, the intermediate value determines the balance between the two.

Implementing Lasso Regression

I have applied Lasso regression on the Boston house-prices dataset and applied the different values of λ. As we increase the value of λ, some of the variables are ignored. There are 13 variables, namely, CRIM, ZN, INDUS, CHAS, NOX, RM, AGE, DIS, RAD, TAX, PTRATIO, B, and LSTAT. I have taken the help of the sklearn function. Lasso, which is a linear model trained with L1 before regularizer. With different values of λ as [.0001, 0.25, .5, 0.75, 1.0]. The Lasso function denoted as λ and α both the notation are interchangeably used in literature. We can view it in the following code block:

```
# Run Three Lasso Regressions, Varying alphas Levels
# Create a function called lasso,
def lasso(alphas):
‘’’
Takes in a list of alphas. Outputs a dataframe containing the coefficients of lasso regressions from each alpha.
‘’’
# Create an empty data frame
df = pd.DataFrame()
# Create a column of feature names
df[‘Feature Name’] = names
# For each alpha value in the list of alpha values,
for alpha in alphas:
# Create a lasso regression with that alpha value,
lasso = Lasso(alpha=alpha)
```

```
# Fit the lasso regression
lasso.fit(X, Y)
# Create a column name for that alpha value

column_name = ' λ = %f' % alpha
# Create a column of coefficient values
df[column_name] = lasso.coef_
# Return the dataframe
return df
# Run the function called, Lasso
df = lasso([.0001, 0.25, .5, 0.75, 1.0])
```

As you can see, an increase in λ means some of the variables are given zero importance. So, by decreasing variable importance (or penalizing), l1 prevents model overfitting. You can run an experiment with the script

Figure 1.11: *L1 regularization with a different value of λ.*

Ridge The L2 equation makes it clear that if λ = 0, no regularization is applied. If λ is large, the model will not learn anything or underfit. I have applied Ridge Normalization to the Boston house price dataset. Let's view it in the following code block:

```
# Create a function called lasso,
def ridge(alphas):
‘’’
Takes in a list of alphas. Outputs a dataframe containing the coefficients of lasso regressions from each alpha.
‘’’
# Create an empty data frame
df = pd.DataFrame()
# Create a column of feature names
df[‘Feature Name’] = names
# For each alpha value in the list of alpha values,
for alpha in alphas:
# Create a lasso regression with that alpha value,
Ridge = Ridge(alpha=alpha)
# Fit the lasso regression
Ridge.fit(X, Y)
# Create a column name for that alpha value
column_name = ‘ λ = %f’ % alpha
# Create a column of coefficient values
df[column_name] = Ridge.coef_
# Return the datafram
return df
```

```
# Run the function called, Lasso
df = ridge([.0001, 25, 50, 75, 100])
```

As I increased the value of lambda from 0.0001 to 100, you can see that the penalization gradually increased. L2 increases penalty on attributes and prevents the model from overfitting. You can run an experiment with the l2_regularization.py script. We can see it in the following image:

Figure 1.12: *L2 Regularization with a different value of λ.*

Implementing ElasticNet

I applied ElasticNet on the same Boston house prices dataset but with different values of λ Setting α = 0.5. The following is the function of Elastic net optimizations. It is similar to the previous implementations and takes an additional l1_ratio parameter, which is to balance between l1 and l2 regularization:

```
# Create a function called Elastic,
def elastic(alphas):
‘’’
Takes in a list of alphas. Outputs a dataframe containing the coefficients of lasso regressions from each alpha.
‘’’
# Create an empty data frame
df = pd.DataFrame()
# Create a column of feature names
df[‘Feature Name’] = names
# For each alpha value in the list of alpha values,
for alpha in alphas:
# Create a lasso regression with that alpha value,
Ridge = ElasticNet(alpha=alpha)
# Fit the lasso regression
Ridge.fit(X, Y)
# Create a column name for that alpha value
column_name = ‘ λ = %f’ % alpha
# Create a column of coefficient values
df[column_name] = Ridge.coef_
```

```
# Return the datafram
return df
# Run the function called, Lasso

df = elastic([.0001, 0.25, 0.50, 0.75, 1.00])
```

I got the best of both worlds, as shown in the following plot. For some attributes, it applies penalties with an increasing value of λ, which is the effect of L2 regularization. Some attributes like NDX and RX are almost nullified, which is the effect of L1 regularization:

Figure 1.13: *ElasticNet regularization – The effect of different value of* λ.

The main question is how to select the optimal value of variables a and λ in Lasso, Ridge, or ElasticNet regression. Methods like using cross-validation and Information-criteria based model selection are used to do this, but the details of these methods are out of the scope of this book. Techniques like Early Stopping and DropOut are widely used in the field of deep learning. We will discuss Early Stopping and DropOut in great detail in chapters related to CNN and RNN later in this book.

You can check the following links for reference:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

Elastic *https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html*

http://yann.lecun.com/exdb/publis/pdf/wan-icml-13.pdf

https://arxiv.org/abs/1605.06465

Training and Inference

There is a great correlation between how a child learns and how a machine learns. Machine learning techniques are highly dominated by supervised learning models like Random Forest, logistic regression, **Gradient Boosted Machine** perceptron model, and other architecture like **Convolution Network Network** and **Long-Short Term Memory**

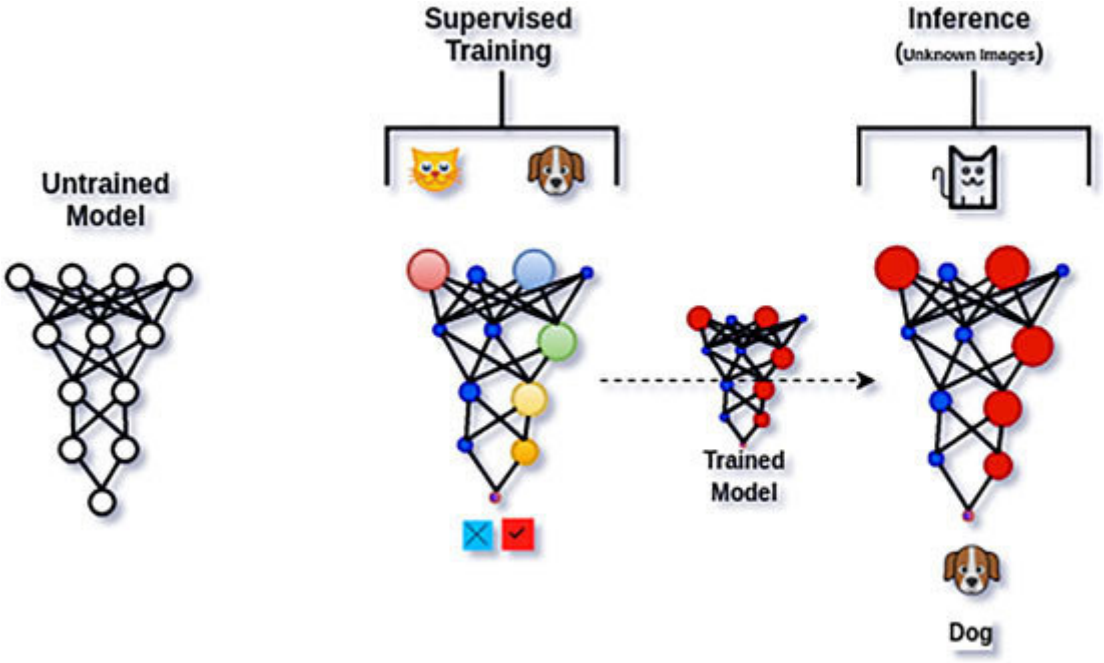


Figure 1.14: *A flow diagram to dictate how a model is built and used in the inference pipeline.*

Inference is a process where capabilities learned during deep learning training are put to work. Generally, the desired training to inference ratio is 70:30, but this ratio does not hold to be true

for all sizes of the dataset; so, we must consider different ratios for different data sizes. As the size of the training data increases, the fraction of test data is decreased. The most important thing to remember is in the split; there should not be a distribution difference between the inference and training data.

Around 90% of machine learning models do not end up in production due to their efficiency and productivity. The production efficacy of the model can be optimized to achieve the maximum number of inference with low energy consumption in the production environment. Several techniques can be used to ensure the production viability of the model. All major frameworks like Tensorflow, Theano, PyTorch, and MxNet use dynamic or static graphs internally to optimize the overall training process. However the network graph is formed for training is not optimized for inferencing. There is a great need to improve the speed and efficiency of inferencing. We all know of Netflix's collaborative filtering competition. On June 26, 2009, the team named **BellKor's Pragmatic Chaos** achieved a 10.05% improvement over the next best team and won the competition. However, very few people know about what happened after—the model never went to production owing to its complexity. It was an ensemble model, and Netflix could not scale it. Keeping this experience in mind, we must think of model scalability and inferencing efficiency from the beginning of the experimentation.

Software-based Accelerated Inferring

Software-based inferencing can be done using TensorRT by freezing the graph:

TensorRT was released by Nvidia that optimizes the neural network model. TensorRT is used by many companies in their production codes.

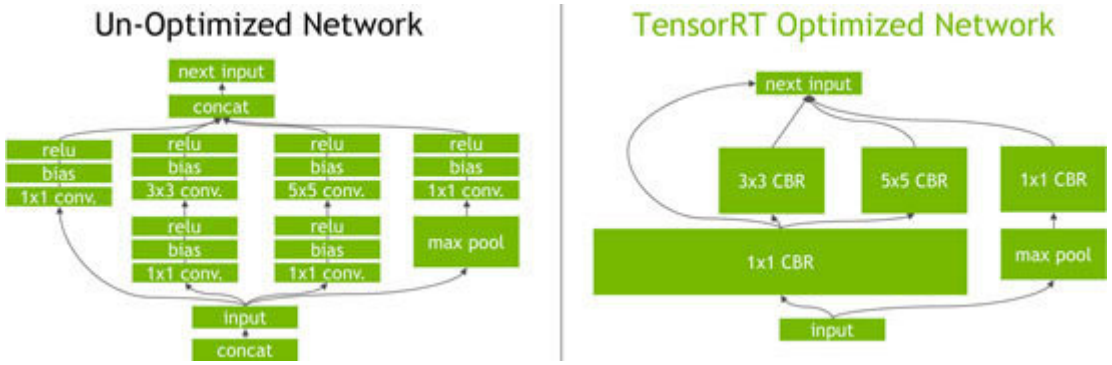


Figure 1.15: *Showing how TensorRT works and converts trained model into inference ready engine.*

TensorRT performs multiple series of predefined operations to optimize the neural network model. Many a time, models can have multiple outputs, which are often made to regularize and stabilize the model. Multiple outputs in the neural network also help learn very deep models.

Convolutional bias and relu are fused to form one layer.

Parallel layers with the same operations and similar parameters are merged.

Backpropagation parameters are removed.

Original FP - 32 (double-precision)-based Model is changed to single-precision (FP - 16 and INT - 8) based model.

These operations do not change the overall accuracy of the model to a great extent, and the model is only reorganized to make it faster. You can refer to the image given at This image illustrates how TensorRT optimizes Google net or Inception/ GoogLeNet module.

TensorRT takes a model graph in ONNX format and generates a TensorRT-optimized model. ONNX is an open format to represent deep learning models. With ONNX, AI developers can easily move models between state-of-the-art tools and choose the combination best suitable for them.

Freezing the TensorFlow uses this technique for faster inference. In the original graph generated by TensorFlow, various parameters are related to back-propagation, updates of weights, the queuing and decoding of inputs, and information related to checkpoints. These are no longer required during inferencing. Sometimes, it has been observed that the model, which is of size in a few GBs, diminished to a size in MBs after freezing the graph.

Hardware-based Accelerated Inferencing

With the advent of an edge computing device like self-driving vehicles and personal assistance like Alexa, Google Home, and Siri, it is essential that our model be capable of running on edge nodes. Generally, the edge node is the device with low RAM and compute capacity. The model must be small, fast, and energy-efficient to be accommodated into this device. Hardware-based inferencing device is already available in the market to help achieve this. Some of the computing devices optimized for inferencing are:

Nvidia Turing and Jetson ecosystem

Movidius 2450 by Intel

Kirin 970 (Huawei)

Qualcomm 660

Some of the required properties needed in such inferencing devices are as follows:

Low power consumption

Low cost (many such devices will be required; one at each edge)

Ease of deployment

Inference speed (Most I. M. P.!)

An inference can be made on the following type of the device:

CPU

GPU

FPGA

DPU

ASIC

The list is in decreasing order of flexibility and increasing order of inference efficiency. These devices work by using various techniques like fusing layers, memory optimization, and efficient allocation.

The Three Learning Principles

There are certain principles, knowing which can help avoid far-sighted issues, although they do not help in the actual implementation. These principles are not related to ML in general, but learning about them helps us understand machine learning better.

Model related concepts

Occam's It's a common thing in life "simpler is better." The Occam's Razor theory says, *An experiment should be made as simple as possible but no* In a common terms, one should slice the data with a razor until it provides a fully logical explanation. The simple model that fits the data is also the most plausible. There are questions to be answered:

What does it mean for a model to be simple?

How do we know it is a simple model?

What does it mean model to be simple?

The simplicity of a model can be measured in the following ways:

By measuring the complexity of a single hypothesis (we often refer to this as

By measuring the complexity of an entire class of the hypotheses (we refer to this as a group of H)

The complexity of *h* can be measured as follows:

If one says 70th order polynomial or 100th order polynomial, which one is more complex?

Minimum Description Length in minimum bits can be explained. It can be better explained by Kolmogorov complexity, but MDL is a friendlier one because it isn't affected by compute capability and such.

Two methods can be used to measure the complexity of a group of hypotheses

Entropy can be used to measure the complexity of the group.

The **Vapnik–Chervonenkis** dimension can be used. It looks at the hypothesis and gives one number that describes the diversity of the group.

Sometimes, we think that the data is complex, but taking the same data in a higher dimension can make it simpler to deal with. This is the working principle behind support vector machines. It is better to have a simple fever hypothesis than a complex one. Occam's Razor silently says that we must understand whatever we do, as blindly picking a complex hypothesis may make our analysis vulnerable to overfitting. Let's take an example of the stock market oracle to understand the complexity of the algorithm. Let's say you get daily emails with a prediction of whether the share market will go up or down. You received this prediction for the next four days, and the received

predictions were right. On the sixth day, you get the mail that says give $50 for the next prediction.

There is a higher chance that you end up paying, but it's important to understand how this is happening. So the oracle has figured out a complex algorithm to predict share market, or something else is happening. In reality, the person sending this email has drafted 32 emails and sent them to 32 people. Half of the emails state that the market will go down, and the other half state that it will rise. At the end of the day, they have 16 people to whom they sent an email that turned out to be correct. On the second day, he repeats the experiment with 16 people, and everyday people get reduced, and on the sixth day, only one person is left who has received all the predictions correctly. On this day, that person gets the email asking for $50. This is what happened behind the scene. The person sending the emails saw that the prediction ability of this algorithm is great because that person does not know about its failures. There is one hypothesis, and it was totally correct. In reality, there are 32 hypotheses, so the prediction capability of this group is approximately 1/32* 100%, so this hypotheses set is useless.

Data related Concepts

Sampling bias occurs when data collection is done in a biased way, as the learning will also produce biased outcomes. We should always think of test data/real-life data distribution while collecting training data. It often happens that trained data performs poorly on test data due to distribution-related issues. Sometimes, this distribution can be corrected by fine-tuning a small amount of data with a similar distribution as test data. Let's say a personal assistant, let's assume Alexa, was trained using an American accent. When anyone says **Alexa,** it wakes up and follows the voice command. Will this system work with the same efficiency for a person speaking in an Indian accent? I doubt. To correct this distribution, the model should be retrained or may be fine-tuned on Indian-accent related data. The following figure shows the meaning of the difference in distribution between **Train** and **Test** data:

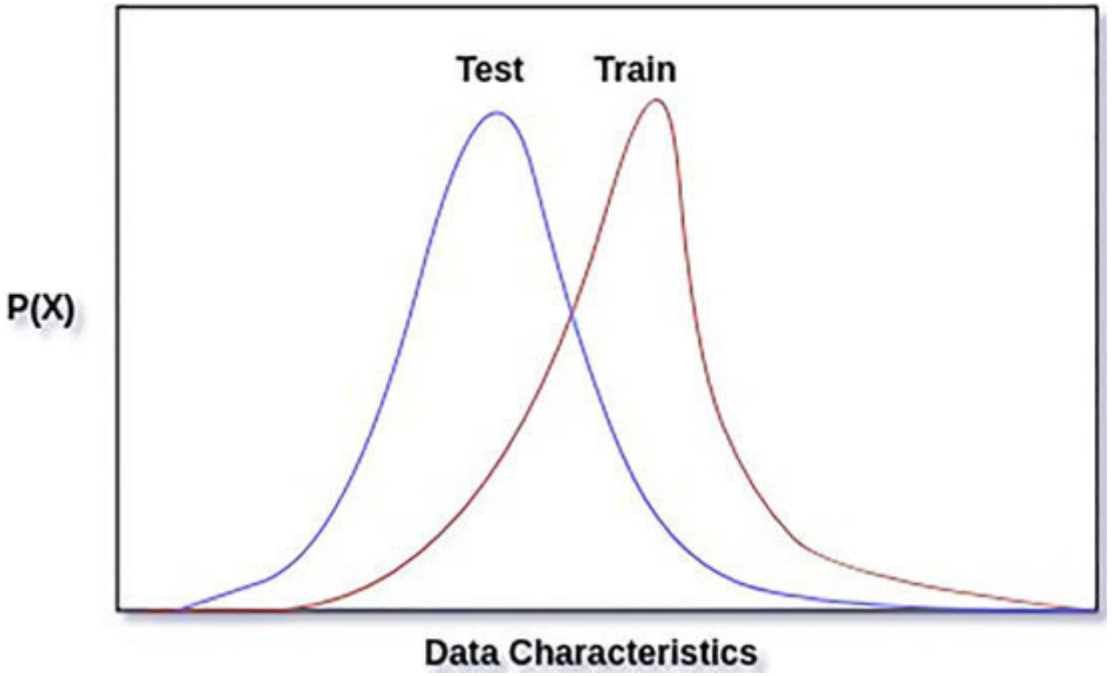


Figure 1.16: *The meaning of the difference in distribution between Train and Test data.*

Let's take one more example to demonstrate how data bias can affect the final model. In an Indian election between candidate X and Y, one of the news channels conducted an online poll after the elections to predict who will win before the results were out. The news channel got thousands of replies and the channel declared that X won over Y based on the statistical analysis. When the actual results were declared, it was flipped and candidate Y won with a large margin. Now, where did things go wrong? They went wrong with the online poll. The age group highly influenced the result. The majority of the people who voted in the online poll were youngsters who favored X over Y, which reflected in the results. But this poll does not cover the majority of the population including older adults, those who don't have access to the Internet, or others who favored Y over X. This is an

example of how a change in the distribution of train and test data can affect predictions.

Data snooping is the statistical inference that researchers decide after looking at the data. It is also known as data finishing, data dredging, p-hacking, and data butchery. *If you torture your data long enough, it will confess anything* - **Ronald** Yes, the model will confess anything, but it won't be able to predict correctly on test data. For example, let's say we have the following data:

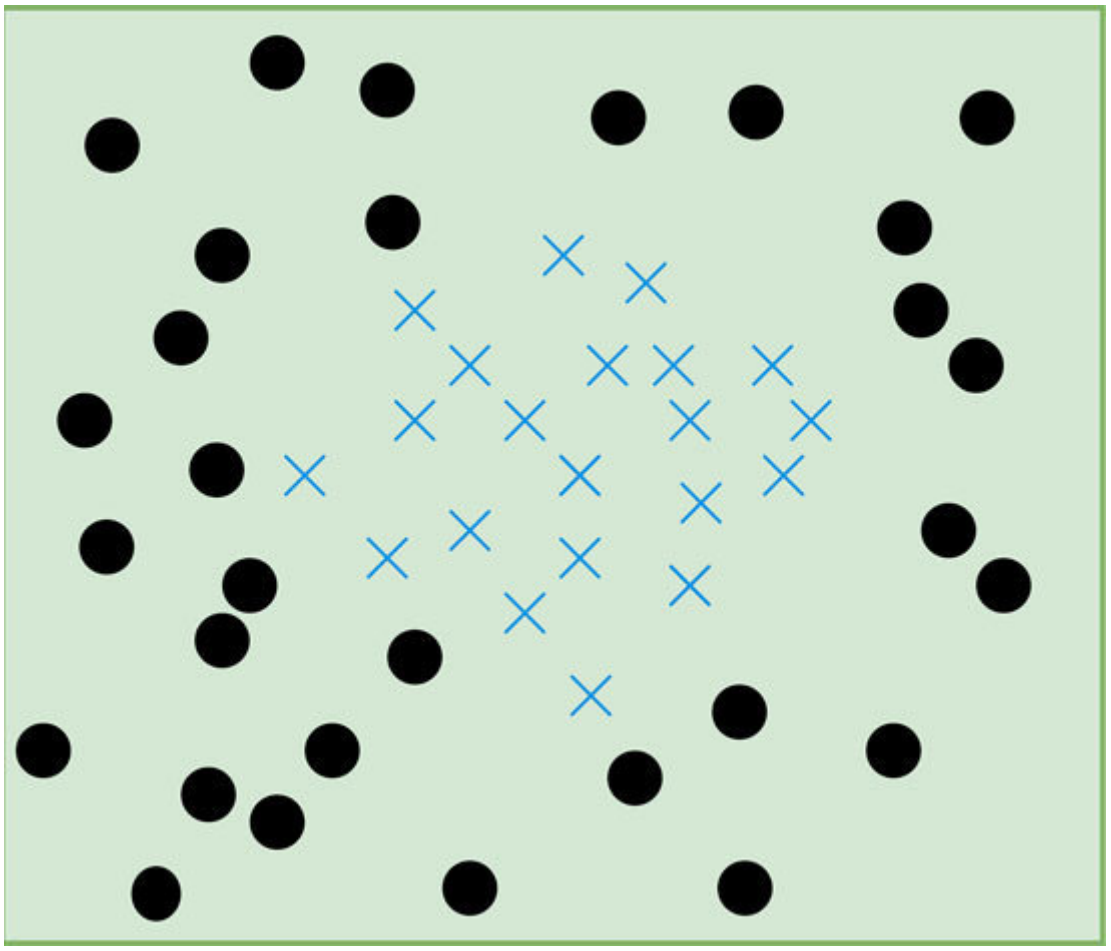


Figure 1.17: *Example data distribution to understand data snooping.*

Ideally, your equation should be: 1 + + + + + By the kick of elementary math, you already knew that the only part that would be useful is + By doing this, you get good accuracy on the train data, but this model will never generalize because it is learning the data in a biased manner. Unintentional data-snooping can be prevented by performing certain checks:

The main idea is not to develop an algorithm by looking at every point of the data but by looking at the pattern/properties of the data.

Don't restrict your algorithm based on the data point you observed. If you do so, your algorithm will perform worst on the test data.

Before collecting or using data, think about the hypothesis you will use. The hypothesis should be chosen considering the characteristics of the data, so a hypothesis that performed well in the past with similar data characteristics should be selected first for experimentation.

Always divide your dataset into random Train, Validation, and Test partitions. The training sample is used for Training, and the validation sample is used for checking the generalization capability of the hypothesis on the validation while training is in progress. A validation sample is also used for early stopping. Early stopping is one of the techniques to prevent overfitting. After the Training sample is completed, the resultant hypothesis is tested on the test dataset. If Train, Test and validation sample belong to the same

distribution, the hypothesis performing well on the validation sample is expected to perform equally well on the test sample.

Conclusion

This chapter provided the base of what we will be building upon throughout this book. This chapter covered the very first and the simplest perceptron model that works well on our example data. We also went through some of the basic methods to identify problem with models, like bias-variance tuning and methods to generalize the model to unknown data using regularization. We have also seen what the possible matrices can be to gauge the progress and quality of the model. This chapter is very useful and will always be at the core of any machine learning or deep learning workflow.

In the next chapter, we will learn text processing techniques.

CHAPTER 2

Text Processing Techniques

This chapter starts with making you aware of the challenges related to **Natural Language Processing** This chapter helps you understand why dealing with NLP is difficult by providing various examples, and it covers methods to get data either by web scraping or parsing data from different formats. We will also explore basic text processing techniques like Stemming and Lemmatization, to convert the inflected token to stem and help decrease vocabulary size. We will tokenize and compare sentences with Spacy and **Natural Language Toolkit** libraries in the recipe related to tokenization. One of the most important recipes of the book is covered in this chapter, which is an introduction to PyTorch. The PyTorch-related recipe will help you implement a basic neural network model on your own, and recipe related to TorchText will help you with hand on text manipulation and processing it with PyTorch. The last recipe is related to visualizing our model progress by plotting scalar, images, text, and embedding produced during training, helping track the learning progress.

Structure

In this chapter, we will cover the following recipes:

Understanding the language problem

Introduction to data retrieval and processing

Understanding stemming

Understanding lemmatization

Understanding tokenization

Getting familiarized with PyTorch

Using TorchText

Visualizing using TensorBoard

Objective

This chapter will help readers learn about common text processing techniques used in the natural language processing pipelines. This chapter covers the fundamentals of the workings of stemming, canonicalization, and tokenization. Additionally, it explores topics like PyTorch essential, understanding PyTorch text for text processing, and using TensorBoard with PyTorch to visualize the training process, embedding, and network graph.

Pre-requisites

I have provided some of the examples through code, and the code for this chapter is in the ch2 folder in the GitHub repository

To understand this chapter, you require basic knowledge of the following Python packages:

PyTorch

Numpy

TensorFlow_gpu

Requests

NLTK

Scipy

Spacy

PyTorchvision

BeautifulSoup4

TensorboardX

TensorFlow

You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r

Understanding the Language Problem

Language is the mechanism for connecting with those around us. It evolves constantly, and it takes many years for humans to understand it.

There's a paradox; language, for us, is something we understand, while language, for the machine, is purely mathematical. Human language and mathematics are not evolved according to each other, so finding a straight line between them is difficult. Humans would take a considerable amount of time if they are asked to calculate 100 order polynomial, but a machine would do it easily. On the other side, machines are less intelligent than kids in understanding and generalizing language, and NLP bridges this gap:

"In NLP, No Rule applies the indefinite rule applies"

Understanding language is difficult, and there are around 6,500 languages in the world. Each language has its grammar, syntax, and semantics, so there are certain known challenges in understanding language. I have discussed a few challenges here to give you a sense of challenges we will face ahead of this chapter.

The following are the complexities involved in understanding the nuances of natural language. Details regarding these will be discussed in the next section:

Contextual When a given word is used in different contexts and carries a different meaning in each case, it is also known as polysemy.

Sentence I went to the bank to withdraw money.

Sentence I saw an alligator at the bank.

Here, the meaning of the bank is different in both the sentences. ≠ When methods like word2vec or glove generates the word vector, the context of the word is not considered, so vector *both the word will be* = But when contextual embedding is used, a **bank** in both cases will have a different meaning, so the vector for the same word in different sentences will be different.

Pronoun It is related to identifying a subject or object with respect to the given verb.

children stole the guns. They were later

Here, the pronoun ***They*** may refer children or guns. To solve this problem, we may use techniques called **anaphora** It is a problem to identify how the noun or pronoun refers to the verb.

Ambiguity in proposition Before we go forward, let me give you an example:

"The policeman saw the thief with the telescope."

It is difficult to find whether the past participle (with the telescope) is attached to the noun phrase (the when the interpreted reader goes as the thief has a telescope and is seen the policeman. On the other hand, it could be attached to the verb phrase (see); so, the policeman had a telescope, an instrument for seeing the thief. At the same time, relative clause attachment connected at least two phrases that composed a sentence.

Humor/sarcasm Humor and sarcasm are very difficult for the machine to identify. The main reason is the lack of diversity and the training data not having enough coverage. Understanding humor requires a fine sense of language as well as attention. Present machine learning techniques lack both, so it is difficult to make a system that can identify humor or sarcasm.

Native language-related Well, this is a huge challenge when it comes to classifying tweets or sentiments. Generally, the available training dataset belongs to one distribution, whereas the test data belongs to another distribution. The way a Native American writes English is very different than the writings of a native Indian. It is one of the spots where a lot of research is required so that the model is generalized so well that retraining or fine-tuning is not required.

Long-range Machine learning techniques are not well suited for capturing long-range dependencies.

Nvidia's platform will allow automakers to integrate full self-driving autopilot capabilities, including lane detection, lane change, lane splits, signboard detection, and emergency response system into will also help them easily deploy intelligent cockpit assistance systems to automakers.

Here, the word ***It*** refers to It is difficult for many algorithms to figure out this relation; this problem is also known as named entity disambiguation. Due to these problems, natural language processing and natural language understanding are still difficult tasks to accomplish.

Speech Nowadays, many personal assistants like Alexa, Google Home, and Siri are facing challenges related to speech recognition. Let's say your assistant is trained on a Native American accent, and it fails in the Indian market. It essentially requires retraining or fine-tuning. Fine-tuning will work if the device was trained on a Native American accent and needed to be used in India, but it will not work if such a device needs to be used in China with the Chinese Language, the device would require retraining using a new dataset. There are many challenges in speech recognition, like noise, channel variability, multiple speakers, speaking style, and homophones. We will discuss these challenges in *Chapter* Techniques and Advance Framework of Speech Processing.

Realeyes

Realize

Real lies

All the preceding, when spoken, are perceived as similar, which makes it difficult to differentiate between them. This example is from a tweet by Christopher Syre Smith (an american rapper).

Based on difficulty, NLP problems can be broadly categorized as follows:

Easy or mostly Spam detection, part of speech tagging, named entity recognition

Intermediate or making good Sentiment analysis, machine translation, sentence/query parsing or natural language understanding, word sense disambiguation, information translation - converting unstructured data to structured data.

Hard or still need a lot of Text summarization, machine dialogue system

This book will cover many of these tasks, and we will cover tasks that are easy, medium, and hard. In the upcoming chapters, we will design models that are state-of-the-art or very close to it. Stay tuned!

You can visit to the following link: to understand the current trend and challenges in NLP

Natural Language Processing: State of The Art, Current Trends and Challenges: *https://arxiv.org/pdf/1708.05148.pdf*

Introduction to Data Retrieval and Processing

Modern algorithms are data-hungry. A huge amount of data is required to generalize the given task, and such labeled data can be gathered from two sources:

Data made and published by someone

Data to be gathered yourself

The first option is very lucrative but limited, as labeled data cannot be available for each learning task. Sometimes, available data is not useful due to the difference in the distribution of train and test data. It is related to the generalization problem we discussed in *Chapter 1, Understanding the Basics of Learning* One thing can be done to have better accuracy on the test data. Initial training is done with the labeled data we received from datasets that are already available, and fine-tuning of such a model can be done using domain-related data.

Let's take an example to make you aware of the distribution difference between the train and test data. Let's say a carmaker is interested in analyzing the sentiments of a customer who comes to their service centers. Since a majority of cars are not sold online, there is no proper feedback loop defined, so historical data is less or not available. In this condition, one may take Amazon sentiment analysis data. Still, the model's distribution is different

from the test data, and this model will perform poorly on the car's service sentiment data. In this case, a model can be trained on the Amazon sentiment analysis data and fine-tuned on a small amount of the available car's service sentiment analysis to perform better.

We need to think how to get such custom data. It can be gathered from the following sources:

Scrapping the web page

Extracting data from XML and JSON

Web scraping is the process of extracting data from websites. Web scrapers are the program used to collect specific information from the web pages, and scraping can be done when you know that data is available on web pages. Different websites have different structures, so one scrapper script cannot work for all websites; we need different scrappers for different websites. A web scrapper performs specific tasks to get the desired information:

It sends a get request to the specified URL.

Upon receiving the page, it parses the **Document Object Model (DOM)** of the HTML and extracts the required information as per the tag UD, class, or property targeted.

DOM is a cross-platform and language-independent application programming interface. A scoring to DOM file formats like HTML,

XHTML, or XML has a tree structure. In this tree structure, each node is an object representing a part of the document. The DOM model represents a document with a logical tree.

Scrapping the Web Page

Let's look at an example to understand how to use scrappers. Assume that I want to build a scraper that scraps comments and corresponding star ratings related to a particular vehicle. I will be scrapping comments from Let's say I am scrapping information related to Mahindra Marazzo, which is located at Python has many scrapping packages available, including Scrappy, BeautifulSoup, and Selenium. I will be using BeautifulSoup for this example. If you inspect the HTML elements of this web page using your favorite browser, you will see the following HTML elements. Inspect elements can be opened by selecting an option after right-clicking on any part of the HTML or by pressing the *Ctrl + Shift* key combination.

The following is the source code of the page, in particular to comments. This source code is taken from

id="userReviewListing">
style="margin-top: 20px;">
style="font-size: 14px; font-weight: bold;" href="/mahindra-cars/marazzo/userreviews/59854/" data-cwtccat="ReviewsListingsPage" data-cwtcact="TitleClick" data-cwtclbl="modelid=1098|source=1">Worst car

class="text-grey" style="margin-top: 3px;">class="rating-sprite one-rating"> by sagar, 2 months ago, > 0 Comments

style="margin-top: 10px;">Just a face lift of renault lodgy nothing else. No value for money, quite expensive for 16.5L. I bought M8 variant which has started making noises..... href="/mahindra-cars/marazzo/userreviews/59854/" data-cwtccat="ReviewsListingsPage" data-cwtcact="ReadMoreClick" data-cwtclbl="modelid=1098|source=1">read complete review

Out of the entire page, I have only shown the HTML elements that have user review-related information. To extract this information, we must target the DIV with an ID equal to Under this DIV, each user review is organized as a nested DIV, and there are two SPAN elements under each inner DIV: one has **star** information, and the second has **user**

Here's an example of scripts to extract comments from the page:

```
# Getting page Content
page = requests.get('https://www.carwale.com/mahindra-cars/marazzo/userreviews/', verify=False)
contents = page.content
# parsing DOM using Soup
soup = BeautifulSoup(contents, 'html.parser')
# getting all data under userReviewListing
mydivs = soup.findAll(id="userReviewListing")
# getting All DIV under userReviewListing
for i in mydivs:
for j in i.find_all("div"): # Getting individual DIV under previous DIV
```

try:

m, n = j.find_all("span") # Getting SPAN under DIV One span is having rating info one SPAN is full Comment
pprint({"Rating": m.get('class', []), "Full_Comment": n.text}) # Making Dict for Rating and Full Text
except:
""

After running the code from you will receive the following dictionary as an output for each user review. One of the output is shown as follows:

{'Full_Comment': '1. I am not buying it but i am thinking to buy this car\n' '2. Driving experience is Awesome i personally drive this ' 'car. Car are awesome in all...', 'Rating': ['rating-sprite', 'three-rating']}

This code is not complete yet, and we can improve it in many ways:

The existing code is not scrapping full comments and returns only the first few words. We can improve the code to extract full comments.

We can have improvement logic that can handle the pagination of user reviews.

By the time you try the preceding code, the website's structure might have changed, and possibly the code might

not run. So, instead of following the code, try to understand how to select HTML tags using Python and extract the information content later.

Parsing Data from XML and JSON Format

JSON and XML are the standard formats for web services and data storage. JSON has a dictionary-like format and can have nested elements, while XML is a markup language that encodes documents in a format that is both human-and machine-readable. All higher-level languages are equipped with an inbuilt parser for JSON and XML. Python also has complete support for XML and JSON. One can use the XML or JSON package in Python to deal with XML formatted or JSON formatted files.

It's always not as simple as in the preceding example. There are many genuine reasons for the web developer not wanting to allow you to scrap the website. Some of the reasons are as follows:

No one wants their data to be used without their permission.

Web scrapper on multiple CPU threads can throw millions of requests per second and makes the server unusable for other users. Besides, the original developer needs to pay for outbound bandwidth consumed by scrappers.

Web developers often use some of these techniques to not allow web scrappers:

Password-based blocking

Captcha implantation

Server-side implementing leeches protection

IP based blocking

All these techniques can be fully or partially bypassed. There are good frameworks available to bypass these blocking mechanisms.

Understanding Stemming

Different forms of the word often communicate the same meaning. There is no difference if we search for Shirt or Shirts. Syntactic differences between different word forms are known as which are often found to be blamed for the problem in query understanding. In NLP, stemming is the technique of replacing the derived or inflected word with its base form stem. Algorithms for stemming are practiced since 1960. Stemming-related algorithms were used for search engines, where such algorithms convert the search word to its base form. In the search, the entered word is searched along with its stem from the word, treating all its modified forms as synonyms.

For example, a stemming algorithm should convert the words fished, fishing, and fisher to their base form fish. There are various implementations of the stemming algorithm, like snowball and porter stemmer. These algorithms operate on a set of rules:

The production This is a reverse lookup-based technique. First of all, various modifications are generated; for example, for the word "run," all its modifications are generated like "ran," "running," "runs," and "runny." Then, reverse lookup is performed on all these modifications to get the base word "run."

Suffix stripping In these techniques, unlike the production techniques, a suffix keeping algorithm with a set of rules is employed instead of keeping all the modification and bootstrapping. Common rules can be words ending with ed, ing, or ly; such suffixes are removed to bring the words back to their base form. Though it is a simple algorithm, it does not do well in practice, and it does not affect words like run and its past participle for ran. It is required to note that all languages do not use suffix and prefixes.

Let's take another example of the Italian language:

Mandargi = Mandare + gli means "to send him"

Mandarglielo = Mandare + gli + lo means "to send it to him"

In Portuguese, hyphen (-) is used to separate the base form from the suffix, so it is easy to deal with the Portuguese language in this case. After many years of research, there is nothing like a versatile stemming algorithm. Each language has its stemming algorithm operating with a set of rules.

Understanding Snowball Algorithm

We will discuss the most-used stemming technique—the snowball algorithm. It operates on a certain rule created after closely studying the pattern of suffixes and replacement to be considered to bring it to the base form. The following explanation aligns with the snowball explanation given at

A, E, I, O, and U are vowels, and all other characters are consonants. A vowel is designated by V, and a consonant is designated by C. Based on this, any word can be represented in these four basic forms:

V ..... V: Starting with a vowel and ending with a vowel

C ..... C: Starting with a consonant and ending with a consonant

C ..... V: Starting with a consonant and ending with a vowel

V ..... C: Starting with a vowel and ending with a consonant

It can be shortly represented as which represents all word that start either with a vowel or consonant. It can have a VC repeating pattern and end with a vowel or consonant.

consonant.
consonant. consonant. consonant. consonant.
consonant. consonant. consonant.
consonant. consonant.
consonant.

Table 2.1

Remember that the following short-forms will help you understand the algorithm better:

The stem ends with S (and similarly for the other letters, for example, m).

The stem contains a vowel.

The stem ends with a double consonant (for example, -TT, -SS).

The stem ends CVC, where the second c is not W, X or Y (for example, -WIL, -HOP).

The following are some of the rules that say its base form will replace the suffix. These rules repeatedly apply until no

substitution is possible. In **Step -** certain rules are defined to change some of the suffixes to their base forms:

forms:
forms: forms: forms: forms: forms: forms: forms: forms: forms: forms: forms: forms: forms:

Table 2.2

The rules defined in **Step – 1B** are applied in suffix reduction, in combination with the preceding short-forms and VC terms:

terms:
terms: terms: terms: terms: terms: terms:

Table 2.3

If 1B – 2 or 1B – 3 is successful, the following replacement is applied after removing a suffix.

suffix.
suffix. suffix. suffix.
suffix. suffix. suffix.
suffix. suffix. suffix.

suffix. suffix. suffix.
suffix. suffix. suffix.

Table 2.4

Step 1C is the rule to convert words with the character 'y' at the end:

end:
end: end: end: end: end: end:

Table 2.5

It is all about the entire algorithm following these steps; one can design a production-ready stemmer. Various steamers are present in the NLTK package. In *chapter* Understanding Lemmatization, we will test how a particular word is stemmed using different stemming algorithms.

Several other techniques are used for stemming, like stochastic algorithms, hybrid approaches, matching algorithms, and affix stemmers. Let's look at them in brief:

Stochastic A stochastic algorithm is based on probabilistic models, which are usually trained on the pair of infected form and its stem token. Such a model internally represents complex language rules. After training, when a word is given

to such a model, it will turn it to its base form. Going a step ahead, such an algorithm can be trained considering the parts of speech and the context of the word, in addition to the original word.

Affix In linguistics, 'affix' means both suffix and prefix. Affix stemmers work by removing both and suffixes and prefixes to convert the token to stem. For example, the word indefinitely will be converted to its stem definitely.

Hybrid The hybrid approach uses many combinations of the preceding approaches. Typically, hybrid approaches operate using a set of rules that call any of the preceding methods.

You can refer to the following links for further understanding on stemming:

Snowball algorithm:

Affix stemming: *http://www.aclweb.org/anthology/P/P09/P09-1017.pdf*

Understanding Lemmatization

In lemmatization, the root word is called the lemma. The main difference between Stemming and Lemmatization is that lemmatization takes into consideration the context of the word, while stemming does not. Stemming, by default, operates on the suffix and does not consider the removal of the prefix. Stemming just removes a few characters and often ends up with the incorrect meaning.

Stemming is popular nowadays, and lemmatization is of lesser importance. These two algorithms were of great importance when we used to employ vectorization techniques based on TF-IDF and the co-occurrence matrix. With a large vocabulary, these techniques are given unnecessary larger sparse matrix. Such a matrix with a million * million elements is very difficult to accommodate into the main memory.

The Stemming and Lemmatization algorithm is still under research and not close to perfect. The following example uses stemmers, and lemmatization functions form NLTK. NLTK has various types of steamers, like Lancaster, Porter, and Snowball.

The following code tests a different type of Steamers and Lemmatizer on the same set of tokens:

```
lancaster = nltk.stem.lancaster.LancasterStemmer()
porter = nltk.stem.porter.PorterStemmer()
snowball = nltk.stem.snowball.EnglishStemmer()
WordNetLemmatizer = nltk.stem.WordNetLemmatizer()
def differnt_stemmars_and_lemmatizer(word):
print("Word : ", word)
print("Lancaster Stemmer : ", lancaster.stem(word))
print("Porter Stemmer : ", porter.stem(word))

print("Snowball Stemmer : ", snowball.stem(word))
print("Snowball Lemmatizer : ",
WordNetLemmatizer.lemmatize(word))
word_list = ["maximum", "cats", "seventy-one", "cacti", "geese",
"better", "Agreed", "Plastered", "Motoring"]
```

The following table of results shows a comparison between the different types of Steamers and

Table 2.6

As the preceding table illustrates, the Lancaster Stemmer is more aggressive, while porter and snowball produce similar results. WordNetLemmatizer does not produce any change and give the same word as the result. You may reproduce or experiment with different stemmers and lemmatization functions from NLTK by running the stemming_lemmatize.py

script. In addition, NLTK has a stemming algorithm for other languages like while **Isri** is the stemmer for Arabic, and **Cistem** is the stemmer for German.

Things have changed with the invention of character-based embeddings like FastText or Elmo embedding. The embeddings of the word fish and its forms like fishing, fisher, fished will be closely placed in an n-dimensional space. It also means that cosine distance between the vectors of all these words will be toward minimum. So, modern algorithm remains largely unaffected by the modified words.

The advantage of Stemming and Lemmatization is that it reduces the vocabulary size and makes tasks memory-efficient. Also, it avoids unnecessary similar words and enables faster convergence.

Just like NLTK, Spacy has support for lemmatization. You can experiment with Spacy lemmatization tools; usage document for the same is described at

Understanding Tokenization

To the computer, any string is a continuous memory allocation with some encoding. The process of separating such string chunks into linguistically significant and methodologically useful units like words or sentences is called

In English, words are separated by spaces, but all individual words do not constitute a linguistically significant token. For example, the words **American Express** make up the name of a company, and separating these two words into **American** and **Express** while tokenizing does not carry any meaning.

Standard word tokenizers are made up of some of the rules, like separating by white space. Building a tokenizer seems easy, but it is a challenging yet most interesting task. Care must be taken while performing this task because any mistakes propagate to your entire pipeline. Tokenization is challenging in the biological domain, as many words are multi-word, and hyphen and slash are often found in a single token.

There can be two levels of tokenization, namely:

Low-level which can be considered as tokenizing based on in-between spaces or by applying some additional rules

High-level tokenization is regrouping words and restoring the linguistic meaning.

The tokenization process can have the following steps (mostly rule-based):

Tokenization by spaces

Handling hyphenated words, for example, seventy-one should be one token

Handling abbreviations, for example, in Dr., the "." should not be treated as a full stop.

Numeric and special numeric formats should not be broken

Dates with hyphen and slash

Phone number with bracket or hyphen

Email IDs and URLs

Sentence tokenization also works on a similar set of rules. We won't go into details, as we have a readymade tool with the maximum possible accuracy. Tokenizers are being researched for years, but the accuracy is still not up to the mark. In the following paragraphs, you will see an example

where I have applied a sentence as well as a word tokenizer using NLTK.

With the same paragraph, I have applied sentence as well as word tokenizer using Spacy, an open-source library for advance NLP written in Cython and Python programming languages. The following sentences are taken from Town Geology by Charles Kingsley, an openly available book from the Gutenberg project.

Using NLTK Tokenizer

Sentence Thus, and I believe thus only, can we explain the facts connected with these boulder pebbles. WORDS: [‘Thus’, ‘, ‘, ‘and’, ‘I’, ‘believe’, ‘thus’, ‘only’, ‘, ‘, ‘can’, ‘we’, ‘explain’, ‘the’, ‘facts’, ‘connected’, ‘with’, ‘these’, ‘boulder’, ‘pebbles’, ‘.’]

Sentence No agent known on earth can have stuck them in the clay, save ice, which is known to do so still elsewhere. WORDS: [‘No’, ‘agent’, ‘known’, ‘on’, ‘earth’, ‘can’, ‘have’, ‘stuck’, ‘them’, ‘in’, ‘the’, ‘clay’, ‘, ‘, ‘save’, ‘ice’, ‘, ‘, ‘which’, ‘is’, ‘known’, ‘to’, ‘do’, ‘so’, ‘still’, ‘elsewhere’, ‘.’]

Sentence No known agent can have scratched them as they are scratched, save ice, which is known to do so still elsewhere. WORDS: [‘No’, ‘known’, ‘agent’, ‘can’, ‘have’, ‘scratched’, ‘them’, ‘as’, ‘they’, ‘are’, ‘scratched’, ‘, ‘, ‘save’, ‘ice’, ‘, ‘, ‘which’, ‘is’, ‘known’, ‘to’, ‘do’, ‘so’, ‘still’, ‘elsewhere’, ‘.’]

Using Spacy Tokenizer

Sentence Thus, and I believe thus only, can we explain the facts connected with these boulder pebbles. WORDS: ['Thus', ' ', ' ', 'and', 'I', 'believe', 'thus', 'only', ' ', ' ', 'can', 'we', 'explain', 'the', 'facts', 'connected', 'with', 'these', 'boulder', 'pebbles', '.', ' ']

Sentence No agent known on earth can have stuck them in the clay, save ice, which is known to do so still elsewhere. WORDS: ['No', 'agent', 'known', 'on', 'earth', 'can', 'have', 'stuck', 'them', 'in', 'the', 'clay', ' ', ' ', 'save', 'ice', ' ', ' ', 'which', 'is', 'known', 'to', 'do', 'so', 'still', 'elsewhere', '.']

Sentence No known agent can have scratched them as they are scratched, save ice, which is known to do so still elsewhere. WORDS: ['No', 'known', 'agent', 'can', 'have', 'scratched', 'them', 'as', 'they', 'are', 'scratched', ' ', ' ', 'save', ' ', 'ice', ' ', ' ', 'which', 'is', 'known', 'to', 'do', 'so', 'still', 'elsewhere', '.']

Sentence and word tokenization performed using NLTK and Spacy have a significant difference, and each tokenizer is making some or the other mistake. You may try with more complex text, including a web URL and the mail address, to

know about the mistakes made by tokenizers. However, one would require custom tokenizers for a custom domain, for example, tokenizers required in the financial domain and life science. These are generalized tokenizers, but custom tokenizers are often made for the domain of application. For example, the medical domain requires identifying the complex multiword disease name as one token. Such tokenizers are made using NLP techniques, and **Named Entity Resolution** is mostly used in addition to the English tokenizer. We will learn about NER in the upcoming chapter.

Apart from the basic tokenizer discussed earlier, NLTK has task-specific tokenizers, some of which are as follows:

Twitter-aware The Twitter-aware tokenizer is designed to adapt the tweet structure better. It has an additional function like and the tokens will be in lower case if this is set to false. strip_handles removes the names of the handle from the if set to true, and then it truncates repeating characters, for example, "!!!!" will be converted to the ['!', '!'] tokens.

Here's an example of how to the tweet tokenizer works:

```
from nltk.tokenize import TweetTokenizer
tknzr = TweetTokenizer()
example_string = """@mjcavaretta It's great that you've created a new
Machine Learning algorithm, but you're not done until you've released it on
#github, preferably using #rstats or
#python.#ai#machinelearning#bigdata#iot#ml#tech
#artificialintelligence
```

```
#datascience"""
print(" With Default Parameter :
",tknzr.tokenize(example_string))
tknzr = TweetTokenizer(strip_handles=True)


print(" With Default Parameter :
",tknzr.tokenize(example_string))
>>> With Default Parameter : ['@mjcavaretta', "It's", 'great',
'that',
"you've", 'created', 'a', 'new', 'Machine', 'Learning', 'algorithm',
',',
'but', "you're", 'not', 'done', 'until', "you've", 'released', 'it', 'on',
'#github', ',', 'preferably', 'using', '#rstats', 'or', '#python', '.',
'#ai', '#machinelearning', '#bigdata', '#iot', '#ml', '#tech',
'#artificialintelligence', '#datascience']
>>> With Custom Parameter (lower case | striping handles) :
["it's",
'great', 'that', "you've", 'created', 'a', 'new', 'machine', 'learning',
'algorithm', ',', 'but', "you're", 'not', 'done', 'until', "you've",
'released', 'it', 'on', '#github', ',', 'preferably', 'using', '#rstats',
'or', '#python', '.', '#ai', '#machinelearning', '#bigdata', '#iot',
'#ml',
'#tech', '#artificialintelligence', '#datascience']
```

Multi-Word Expression Tokenizer This is used to retokenize (or merge) multiple words into one token. Many a time, breaking by space also breaks the meaning; for example, the name of the company Goldman Sachs is tokenized as and this probably may not preserve the original meaning. Such cases can be handled using

The following example shows how the MWETokenizer tokenizer works:

```
from nltk.tokenize import MWETokenizer
example_string = """Goldman Sachs's Jan Hatzius expects the Fed to hike
interest rates at least once in 2019."""
tokenizer = MWETokenizer([('Goldman','Sachs'),('Jan', 'Hatzius')])
print (tokenizer.tokenize(example_string.split()))
>>> ['Goldman_Sachs', ''s', 'Jan_Hatzius', 'expects', 'the', 'Fed', 'to',
'hike', 'interest', 'rates', 'at', 'least', 'once', 'in', '2019.']
```

Regular Expressions Regular Expressions tokenizers split the string based on Regular Expressions. For example, such a tokenizer forms tokens out of money expressions, alphabetic sequences, and any other non-whitespace sequences. A subclass of Regular Expression tokenizers is the Blackline Tokenizer, which tokenizes based on the newline.

There are many other types of tokenizers like space tokenizer, TabTokenizer, StringTokenizer, RegularExpression Tokenizer, and Punkt Sentence Tokenizer so on. The details of all these tokenizers, which are made for specific purposes, can be found at NLTK Tokenizer package.

You can refer to the following links for more information:

Spacy *https://spacy.io/usage/spacy-101*

NLTK *https://www.nltk.org/api/nltk.tokenize.html*

Getting Familiarized with PyTorch

PyTorch is Python-friendly PyTorch implementation, actively developed and maintained by Facebook. PyTorch has something unique to offer; it has an intuitive, Python-friendly development approach and is a little different and advanced than the existing frameworks like Keras and TensorFlow.

Francois Chollet originally writes Keras. Keras is a high-level wrapper that supports TensorFlow, Theano, and MXNet. Keras is, indeed, the most lucid and powerful framework and is well suited for newbies for converting ideas to a working model. It is designed to enable fast experimentation with deep neural networks, with the main aim being to provide faster implementation by abstracting some of the layers, leading to faster development. However, Keras is not deployment-friendly and does not support the freezing graph to this date; it has no ONNX support, so TensorRT cannot be used to optimize graphs and for faster inference. Keras does allow custom layer implementation, but it is highly dependent on the back-end engine. Many times, the model that runs successfully on one back-end returns an error when it is changed.

TensorFlow was developed by Google and placed in the open-source domain. It is the best tool for high-performance computing, and it has recently started supporting new

hardware, a Tensor Processing Unit, (TPU). TPU is claimed to be faster than the GPU. TPUs are less flexible hardware than GPU implementation-wise, but they are specifically designed for processing tensors. TensorFlow's development is quite counter-intuitive and less Pythonic. The method by which TensorFlow construct graphs is not quite pythonic, so you cannot use great debuggers like pdb, ipdb, and PyCharm debugger. You will have to use a specific tool like TensorFlow debugger or tfdbg On the other hand, you can use any of your favorite debuggers with PyTorch. TensorFlow is also supported by an independent tool like TensorBoard, which allows us to visualize the training process and supports visualizing images histograms and embeddings. TensorboardX, which is an open-source connector to connect any frameworks output to TensorBoard, has all the benefits that you get using TensorFlow. We will frequently use TensorboardX to visualize our training progress using PyTorch. In many cases, PyTorch is faster than other available frameworks, like TensorFlow or Keras.

A majority of the research community prefers PyTorch because of some of its features:

The PyTorch development style is quite Pythonic

Debugging the graph is easier

An active community is present for rescue whenever you get stuck

Cross-platform support tools like ONNX and TensorBoard enable efficient deployment and visualizations

This recipe will help you understand some of the basic syntax and operations that will, eventually, help you catch-up easily and understand various model developments using PyTorch.

Installation

If you are a native Numpy user and have used a higher-level library like Keras, you will find PyTorch easy. You can easily download and install PyTorch with We will use TorchText frequently, and it can be installed very easily:

pip install PyTorch==0.4.1
pip install torchtext

PyTorch is a group of packages, including a tensor library with strong GPU support, a neural network library, and an auto differentiation library.

It has a total of five vital components, as listed in the following table:

table: table: table: table: table: table: table: table: table:
table: table: table: table: table:
table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table: table: table: table: table:

table: table: table: table: table: table: table: table: table: table: table:

Table 2.7

We will begin with basic tensor manipulation. PyTorch Tensor manipulation is similar to Numpy. Also, the PyTorch has strong integration with the GPU. So any operation that can be done with CPU can be done with GPU with very little time complexity:

import PyTorch # importing PyTorch

PyTorch Module

Initializing 3 * 2 matrix with a random number in PyTorch:

mat1 = PyTorch.randn([2,3])
mat1 >> tensor([[0.8992, -1.0515, 0.6143], [0.3743, -0.4616, 1.1338]])

Initializing another 2 * 3 matrix with a random number:

mat2 = PyTorch.randn([3,2])
mat >> tensor([[-0.6860, -1.6029], [-1.0089, -0.9182], [-0.3778, -0.3748]])

Taking the Dot product of mat1 and mat2 . :

```
PyTorch.matmul(mat1, mat2)
tensor([[0.2119, -0.7061], [-0.2194, -0.6010]])
```

These tensors, by default, were present in RAM, and all the calculations were carried out by CPU. What if we want to carry out the operation with GPU? First, your machine must have GPU hardware. PyTorch supports Nvidia GPUs. To enable GPU computations on your data, you must pass your data to GPU VRAM, which is analogous to RAM in the computer but is faster. Plus, as it is within the device, it allows faster computation with hundreds or thousands of GPU cores with low latency.

To transfer data to GPU, it needs to get GPU's address, and that can be done as follows:

```
cuda = PyTorch.device('cuda')
```

More specifically, if you have multiple GPUs attached to the same motherboard, you can transfer your data to any specific GPU:

```
cuda1 = PyTorch.device('cuda:0') # GPU 1
cuda2 = PyTorch.device('cuda:1') # GPU 2 (GPUs are 0-indexed)
```

Generally, I prefer to use the following code, as it takes care of any environment we want to run our code in:

device = PyTorch.device("cuda:0" if PyTorch.cuda.is_available() else "cpu")

With an increase in the size of the model with billions of parameters, it has been difficult to keep the required parameters in the VRM of one GPU. Additionally, GPUs connected through PCI express do not see each other directly, so direct communication between GPUs in the same motherboard was not possible. To resolve this problem, Nvidia introduced a new architecture called With NVLink, GPUs can see each other, and direct parameter/ data sharing became possible. NVLink was first introduced in the NVIDIA Pascal™ architecture. NVLink on V100 GPU card with Tesla architecture has increased the signaling rate from 20 to 25 GB/second in both directions. It can be used for GPU-to-CPU or GPU-to-GPU communication, as in

Taking a little detour, PyTorch has the following data types and their respective CUDA data types. Knowing these data types will help you in error debugging. The following given data types are PyTorch.org:

PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:

PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:
PyTorch.org: PyTorch.org:

Table 2.8

The PyTorch cannot combine operations between CPU tensors and GPU tensors. To carry out operations between two tensors, both tensors must be present in either GPU or CPU. Whenever you get an error related to an operation between CUDA and CPU tensors, checking the type of tensor may help resolve it.

For example, let's check the type of

mat1

```
>> tensor([[0.8992, -1.0515, 0.6143],[0.3743, -0.4616, 1.1338]])
```

Now, let's check the type of mat1 after transferring it to GPU VRAM. If one has a tensor present in RAM and wants to transfer it to GPU VRAM, they can use .to operator. It transfers any tensor to the defined GPU:

```
mat1.to(device)
>> tensor([[0.8992, -1.0515, 0.6143], [0.3743, -0.4616, 1.1338]],
device='cuda:0')
```

One may directly initialize tensor using device parameters while making tensor. If you don't have a GPU device or proper drivers, this will throw a runtime error:

```
mat1_cuda = PyTorch.randn([2,3],device = device)
mat1_cuda
>> tensor([[0.8247, 0.0689, 1.0346], [0.5418, 0.1267, 0.5000]], device='cuda:0')
```

So far, we have used only three functions, namely:

To generate random numbers

To send tensor to device

To calculate the dot product between two matrices

Hundreds of similar functions are present in PyTorch sub-packages. Each function has a particular use and scientific relevance, and GPU and CPU fully support all these functions. We will discuss other similar functions in the upcoming chapters.

In *Chapter 3, Representing Language* we will define more complex objects by extending the PyTorch.nn module. After understanding that concept, we will see how to reference such an object with GPU so that the computation can be lifted over to GPU.

Then, we will discuss another has all the layers required to define any type of architecture, including LSTM, CNN, and Feed Forward Network. Let's look at defining a simple feed-forward layer that takes ten input and gives two outputs:

```
linear1 = PyTorch.nn.Linear(in_features=10, out_features=2) # defining a linear layer
dummy_tensor = PyTorch.randn([1,10]) # initialising a tensor with 10 features.Size of such tensor will be [1,10]
linear_output = linear1(dummy_tensor) # passing dummy_input to linear transformation layer
print (linear_output.shape) # output>> PyTorch.Size([1, 2])
```

Now, if we look at the linear layer closely, it has two parameters attached: weights and bias. Weight is of size [2, and bias is of size [2] (attached to each output). Both bias and weights are trainable, so the values of both change with every iteration.

The following method is used to see the parameters in the various PyTorch layers:

```
print (" All Parameters : ",linear1.parameters)
print (" Shape of Weights : ",linear1.weight.shape)
print (" Shape of Bias : ",linear1.bias.shape)
>> All Parameters : method Module.parameters of Linear(in_features=10, out_features=2, bias=True)
>> Shape of Weights : PyTorch.Size([2, 10])
```

>> Shape of Bias : PyTorch.Size([2])

Let's take an example of the **Recurrent Neural Networks** layer. **Gated Recurrent Unit** is a popular unit used in designing RNN architectures. If you don't understand RNN architecture, don't worry! We will discuss GRU in detail in the upcoming chapters. GRU can be defined and used in PyTorch as follows:

```
GRU = PyTorch.nn.GRU(input_size=10, hidden_size=20,
num_layers = 2) # defining GRU
dummy_input = PyTorch.randn(5, 3, 10) # making dummy
input
hidden_state = PyTorch.randn(2, 3, 20) # hidden state
output, hidden_state = GRU(dummy_input, hidden_state) #
applying GRU to the input shape
```

Convolution Neural Network is another kind of architecture supported by PyTorch. CNN stands popular for vision-related applications, but CNN is gaining popularity in-text analytics-related applications like entity recognition, embedding generation, and language translation. Nowadays, it is common to use CNN with RNN to get state-of-the-art results. We will deal with CNN and hybrid network with CNN and RNN in the upcoming chapters:

```
Conv_1 = PyTorch.nn.Conv1d(in_channels=16, out_channels=33,
kernel_size=3, stride=2)
dummy_input = PyTorch.randn(20, 16, 50)output =
Conv_1(dummy_input)
```

Apart from these architecture layers, PyTorch has various loss functions. Loss functions measure the extent to which the predicted output differs from the actual output. Loss functions are present under the torch.nn package, and the choice of loss function always depends on the objective of the training. You can define a custom loss function; we will use custom loss functions in the upcoming chapters while making a language transliteration model. PyTorch supports many loss functions by default, including the ones listed here:

L1Loss

Mean Squared Error Loss

Negative Log Likelihood (NLLLoss)

CrossEntropyLoss

Poisson Negative Log Likelihood (PoissonNLLLoss)

Kullback–Leibler divergence (KLDivLoss)

Binary Cross Entropy Loss (BCELoss)

Sigmoid layer and the BCELoss in one single function (BCEWithLogitsLoss)

Each one is used for a specific purpose. Let's see how to use **Mean Squared Error Loss** with some predicted output and actual labels. In the new PyTorch release 1.0.0, losses are moved under the PyTorch.nn.functional package:

```
predicted_output = PyTorch.randn(3, 5)
actual_label = PyTorch.randn(3, 5)
loss = PyTorch.nn.MSELoss()
output = loss(predicted_output, actual_label)print(output)>>
tensor(3.1610)
```

Once the loss is calculated, various trainable parameters must be adjusted to minimize loss. To do this, we use mathematical functions called optimizers, and PyTorch optimizers are placed under the PyTorch.nn.optim package. PyTorch supports the **Stochastic Gradient Descent** ADAM, AdaDelta, AdaGrad, and RMSProp optimizers by default. Optimizer accepts a few of the parameters:

Trainable model parameters

Learning rate

Momentum

Other parameters like b1, b2 required for specific optimizers like Adam optimizer

For input, target in dataset:

```
optimizer.zero_grad()
output = model(input)
loss = loss_fn(output, target)
loss.backward() optimizer.step()
```

All optimizers implement a step() method that updates the parameters. It can be used in two ways:

```
optimizer.step()
```

The gradients are computed using, for example, backward()

In PyTorch, network architecture classes are made by extending One architecture can have multiple layers attached differently. Such a class must extend and implement two methods:

```
init()
```

```
forward()
```

The init() method generally houses the initialization of various layers, while the forward() method connects the layers initialized in init() and forms a logical flow. A class can have additional methods depending on the use case. Let's take an example of the model we discussed in *Chapter 1, Understanding the Basics of Learning* Here I am initializing the linear layer, which takes 100-dimensional input and converts it to 2-dimensional output. Forward methods take this predefined layer, apply it to the input, and provide output.

The following is the simple network architecture with a linear layer:

```
class simple_module_1(nn.Module):

def __init__(self):
super(simple_module_1, self).__init__()
self.simple_linear = nn.Linear(100,2)
def forward(self, input):
“”
return self.simple_linear(input)
```

If we want to transfer our simple module to GPU, we can use the .to operator as follows:

```
SM = simple_module_1(nn.Module) # making object for simple_module_1 class
cuda = PyTorch.device(‘cuda’) # defining device
SM = SM.to(cuda)
```

The three basic building blocks to execute any learning tasks are network architecture, loss function, and optimizers. PyTorch also allows you to design custom architecture, loss function, or optimizer functions. We will use these components to construct a simple model in *Chapter 3, Representing Language*

You can visit the following links to learn more:

https://torchtext.readthedocs.io/en/latest/

TorchText

https://media.readthedocs.org/pdf/torchtext/latest/TorchText.pdf

Using TorchText

TorchText is a powerful tool with data processing utility and a popular dataset for natural language processing. Text processing is a headache, and we must change pre-processing and plugging mechanisms every time we change algorithms. It is better if we use a kind of standard mechanism for text processing, and decrease movable components as much as possible.

The following image illustrates the requirement of TorchText in the text processing pipeline:

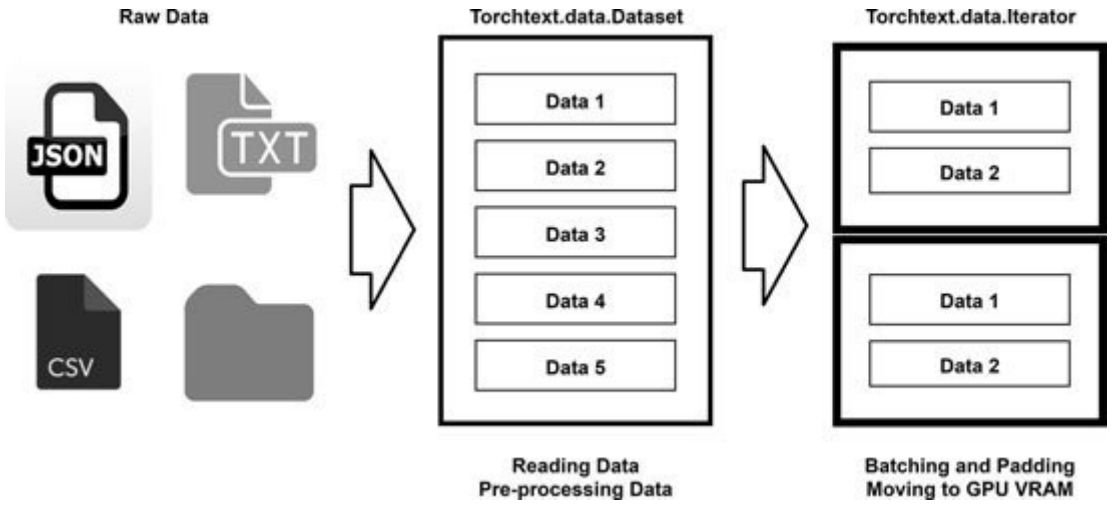


Figure 2.1: *The requirement of TorchText in the text processing pipeline.*

TorchText has three basic components:

TorchText.data

TorchText.datasets

TorchText.vocab

Let's look at these basic components in brief:

The data sub-package provides flexible methods to define preprocessing pipelines like batching, padding, and numericizing datasets. In addition, the data package houses additional functionality like splitting datasets, tokenizer, random pooling, and so on.

The dataset sub-package houses some of the standard datasets for tasks related to sentiment analysis, question classification, entailment, language modelling, machine translation, sequence tagging, and question answering. By providing nascent support for this dataset, TorchText provides an easy way to benchmark an experiment and helps in the easier publication of novelty.

the vocab sub-package helps define vocabulary and load predefined vectors like Glove, Word2Vec, FastText, and CharNGram.

Having understood the basic underlined packages of TorchText, I will walk you through some of the examples utilized with TorchText. This example will help you understand and use some of the utilities of TorchText in the upcoming chapters:

```
import PyTorch
import torchtext
import json
from torchtext import data
from torchtext import datasets
```

Loading the TorchText supports loading data from JSON, **Tab Separated Values** and **Comma Separated Values** Generally, JSON is preferred as CSV/ TSV due to following reasons:

CSV/TSV cannot store data as a list, but JSON can.

TSV/ CSV data are error-prone as erroneous parsing occurs if tab or comma is present in the data itself. This type of parsing error is impossible in JSON.

Suppose we have the following JSON content and want to read it using TorchText:

```
example_json = """{"name": "Sunil", "location": "United Kingdom", "age": 27, "quote": ["i", "love", "the", "united kingdom"]} {"name": "Kinara", "location": "United States", "age": 0, "quote": ["i", "want", "more", "toys"]}"""
```

With the train.json file we use in our example, the dataset will look like this:

{“name”: “Aaayash”, “location”: “United Kingdom”, “age”: 12, “quote”: [“i”, “want”, “the”, “stun granade”]}{“name”: “Mayuresh”, “location”: “United States”, “age”: 20, “quote”: [“i”, “want”, “Guns”]}

Now, we define the field for each key in the JSON format. Fields determine how data is pre-processed and converted to a numeric format. Let’s not specify any pre-processing function and use our data as is:

```
NAME = data.Field()
```

```
PLACE = data.Field()
```

```
QUOTE = data.Field()
```

All individual fields are now collected into one variable field, with appropriate mapping to JSON keys. For example, the JSON key name is mapped to the field and so on:

```
fields = {‘name’: (‘n’, NAME), ‘location’: (‘p’, PLACE), ‘quote’: (‘q’, QUOTE)}
```

Now, we will use the tabularDataset.split function to split our dataset into train and test data:

```
train_data, test_data = data.TabularDataset.splits(path =
'NLP_Cookbook/Chapter2/data/', train = 'train.json', test =
'test.json', format = 'json', fields = fields)
print(vars(train_data[0])) # printing the first data in train_data
>> {'n': ['Aaayash'], 'p': ['United', 'Kingdom'], 'q': ['i', 'want',
'the', 'stun granade']}
```

Earlier in the Data Fields, we did not define any pre-processing function. Now, we will define a function to convert the quote field to capitals:

```
def capitalize(word_array):
return [i.upper() for i in word_array]
NAME = data.Field()
PLACE = data.Field()
QUOTE = data.Field(preprocessing=capitalize)
#using the same data
setsfields = {'name': ('n', NAME), 'location': ('p', PLACE),
'quote': ('q', QUOTE)}
train_data, test_data = data.TabularDataset.splits(path =
'NLP_Cookbook/Chapter2/data/', train = 'train.json', test =
'test.json', format = 'json', fields = fields)
print(vars(train_data[0])) # printing the first data in train_data
>> {'n': ['Aaayash'], 'p': ['United', 'Kingdom'], 'q': ['I', 'WANT',
'THE', 'STUN GRANADE']}
```

Now, you can see that all the words in the Quote fields are converted to upper case. For example, in pre-processing parameters, we can define the tokenization function to tokenize sentences to words:

```
QUOTE.build_vocab(train_data)


vocab = QUOTE.vocab
print("vocab : ", vocab)
print("Token strings to numerical identifiers : ",
QUOTE.vocab.stoi)
print("Token numerical to strings identifiers : ",
QUOTE.vocab.itos)
print("Token Frequency : ", QUOTE.vocab.freqs) # frequency of
the original vocabulary created by Field
print("First 3 token of first train example :
",train_data.examples[0].q[:3])
>> vocab : <torchtext.vocab.Vocab object at 0x7fd282c615c0
>> Token strings to numerical identifiers :
defaultdict(_default_unk_index at 0x7fd282d048c8>, {'': 0, '': 1,
'I': 2, 'WANT': 3, 'GUNS': 4, 'STUN GRANADE': 5, 'THE': 6})
>> Token numerical to strings identifiers : ['', '', 'I', 'WANT',
'GUNS', 'STUN GRANADE', 'THE']
>> Token Frequency : Counter({'I': 2, 'WANT': 2, 'THE': 1,
'STUN GRANADE': 1, 'GUNS': 1})
>> First 3 token of first train example : ['I', 'WANT', 'THE']
```

Similarly, we can use a CSV or TSV file as input. Let's take an example by loading CSV files. The example train.csv file looks like this:

this:

this: this: this: this: this:
this: this: this: this:

Table 2.9

Following block shows how to quickly read file using TorchText

```
# defining tokenizers to convert sentence to words
def tokenizer(sentence):return
sentence.split() # quick and dirty splitting
# defining fields
NAME = data.Field()
PLACE = data.Field()
QUOTE = data.Field(tokenize=tokenizer)
```

While processing CSV or TSV, we must omit the fields that we are not using. You can use (None, None) to omit these columns. At this moment, we will omit the Age column using (None, None) in the third position. One more thing is that I have used the tokenizer for the QUOTE column, so the sentences are also tokenized in the final output. Additionally, validation data is also included:

```
#using the same datasets
fields = [('n', NAME), ('p', PLACE), (None, None), ('q',
QUOTE)]
```

```
train_data, valid_data, test_data =
data.TabularDataset.splits(path =
'NLP_Cookbook/Chapter2/data/', train = 'train.tsv', validation =
'valid.tsv', test = 'test.tsv', format = 'tsv', fields = fields,
skip_header = True)
print(vars(train_data[0])) # printing the first data in train_data
>> {'n': ['Karry'], 'p': ['United', 'Kingdom'], 'q': ['i', 'have',
'money']}
```

Vectorization is the process of converting a word to an n-dimensional dense vector of floats. We will learn about techniques to generate embeddings in *Chapter 3, Representing Language* With TorchText, it's very easy to vectorize tokens; one can do it easily by specifying a type of vector they want to use while building vocab. PyTorch text automatically downloads the specified vectors inthe ./.vector_cache temp directory. These are all vectors supported by TorchText, and a few of them are and

```
embeddings = QUOTE.build_vocab(train_data,
vectors="glove.6B.100d")
```

We have six different words in our vocabulary {'': 0, '': 1, 'have': 2, 'i': 3, 'money': 4, 'power': 5}, so QUOTE.vocab.vectors will give a 100-dimensional vector for each word. One can easily take such vectors and feed it to any machine learning algorithm:

```
QUOTE.vocab.vectors.shape # will provide output shape
```

```
>> PyTorch.Size([6, 100])
```

Vectors for any word can be obtained using the syntax in the following code block:

```
have_vector = QUOTE.vocab.vectors[QUOTE.vocab.stoi['have']]
print(have_vector)
>> tensor([0.1571, 0.6561, .., -0.6061, 0.7100, 0.4147])
```

TorchText is good at handling the unknown word. Other techniques may give an unknown word index or may require explicit try and catch syntax. However, TorchText handles everything wonderfully. The following code snippet shows how TorchText handles previously unseen words:

```
unknown_word = "humbahumba"
print("Index for unknown word %s: %d" %(unknown_word,
vocab.stoi[unknown_word]))
print("Token for unknown word: ",
vocab.itos[vocab.stoi[unknown_word]])
>> Index for unknown word humbahumba: 0
>> Token for unknown word:
```

After training valid and test partition are generated, Iterator allows us to specify the batch size. We have only two examples, so I have kept the batch size as 2, and we get one batch to iterate over:

```
train_iter, val_iter, test_iter = data.Iterator.splits((train_data,
valid_data, test_data), batch_sizes=(2,-1,-1),
```

sort_within_batch=True, repeat=False)

train_data.examples
>> [at 0x7fd2803f7f28>, <;torchtext.data.example.Example at 0x7fd2b668e8d0>]'

You can explore the TorchText document at Some of the functionalities of TorchText are now merged with PyTorch and can be found at

Visualizing Using TensorBoard

TensorBoard is an independent project by Google and is tightly integrated with TensorFlow. Machine learning models are getting complex, and visualization helps track the progress. TensorBoard was developed for TensorFlow, and TensorboardX is an open-source tool that helps connect another framework like PyTorch, MXNet, and Keras to TensorBoard.

The following image shows TensorboardX as an intermediate tool that connects frameworks like PyTorch, MXNet, and Keras to TensorboardX:

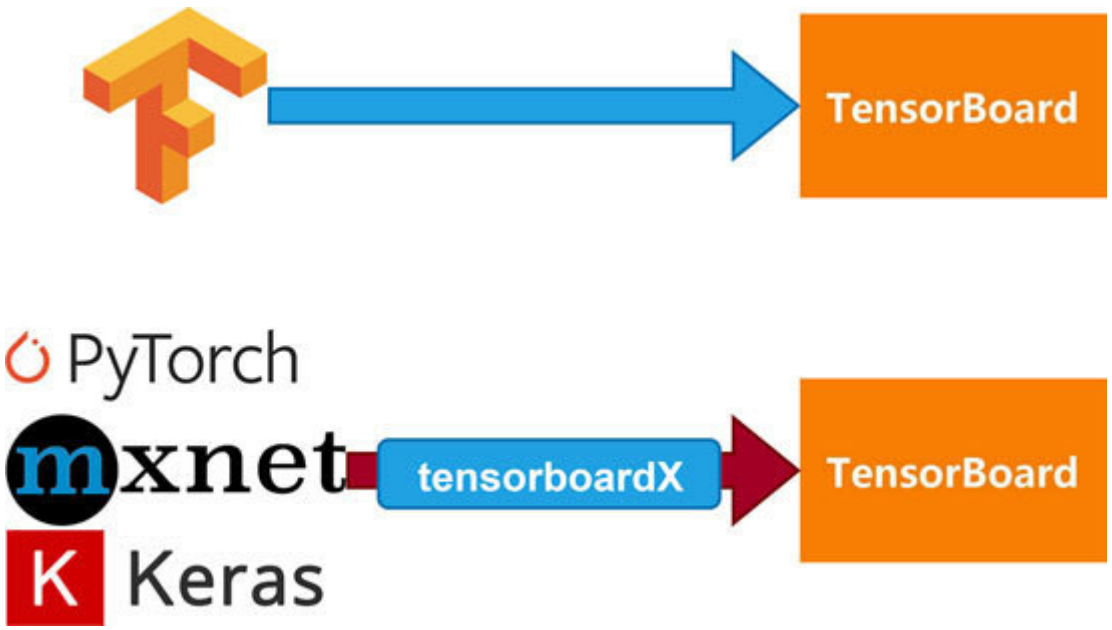


Figure 2.2: *TensorBoard, a look at web-based UI.*

The following is an illustration of how to install and get started with TensorBoard and TensorboardX.

Install TensorBoard and TensorboardX using out the preferred installer,

```
pip install tensorboardX==0.15.4
pip install tensorboard==1.12.2
pip install --user --upgrade TensorFlow # TensorBoard depends on TensorFlow
```

Once installed, start TensorBoard by issuing the following command, where path/to/log-directory is the directory where TensorboardX sends the output:

```
tensorboard --logdir=path/to/log-directory
```

It will bring you to a web-based GUI that looks as follows:

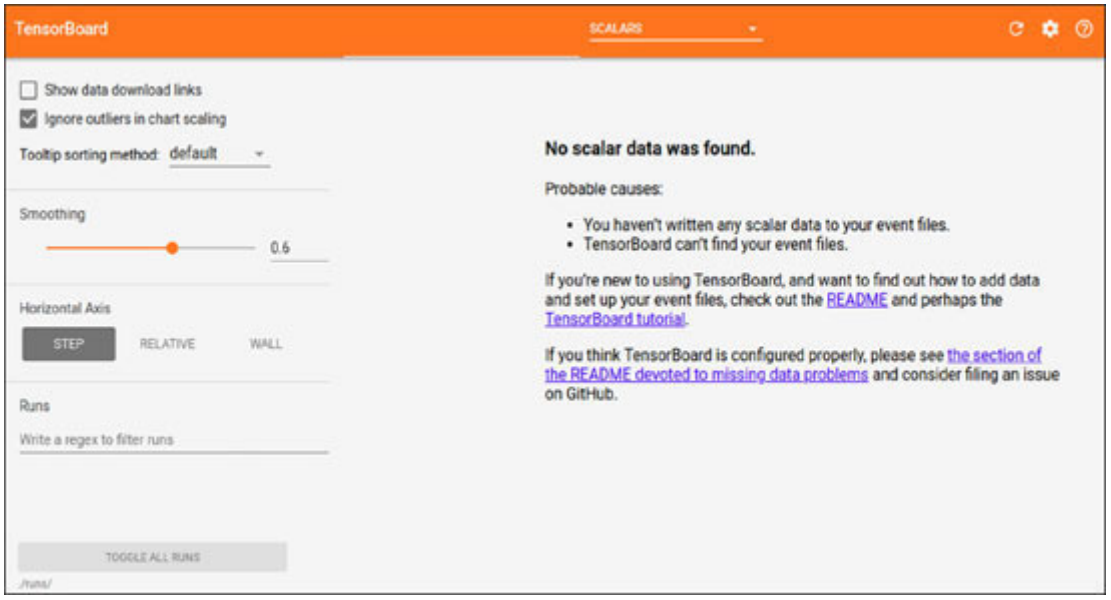


Figure 2.3: *TensorBoard, a look at web-based UI.*

Showing Scalar Values on TensorboardX

TensorBoard supports various types of visualizations, including scalar, histogram, images, video, text, and embedding. First, the summary writer object is created, and this object has various options like adding scalar, image, histogram, figure, graph, audio, and embedding. Here, I am using the add scalar option.

The add scalar takes three inputs, namely:

The name of a scalar

The value on the Y-axis

Value on X-axis (generally iteration/ epoch count)

The following code snippet shows how to project scalars on the TensorBoard using

```
from tensorboardx import SummaryWriter # init
import math
import random
writer = SummaryWriter()
# writing both to separately
for i in range(0,100):
```

```
writer.add_scalar('sin',math.sin(i*0.001) + random.random(), i)
writer.add_scalar('cos', math.cos(i*0.001) + random.random(),
i)
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```

After running the preceding tfboardExperiments/scalar.py script, it will create the runs directory in the same location as the code. Then, start your TensorBoard server with log directory pointing at the ./runs directory - TensorBoard Go to TensorBoard and refresh the view; you will get a plot for sin and You will see the following diagram showing random values:

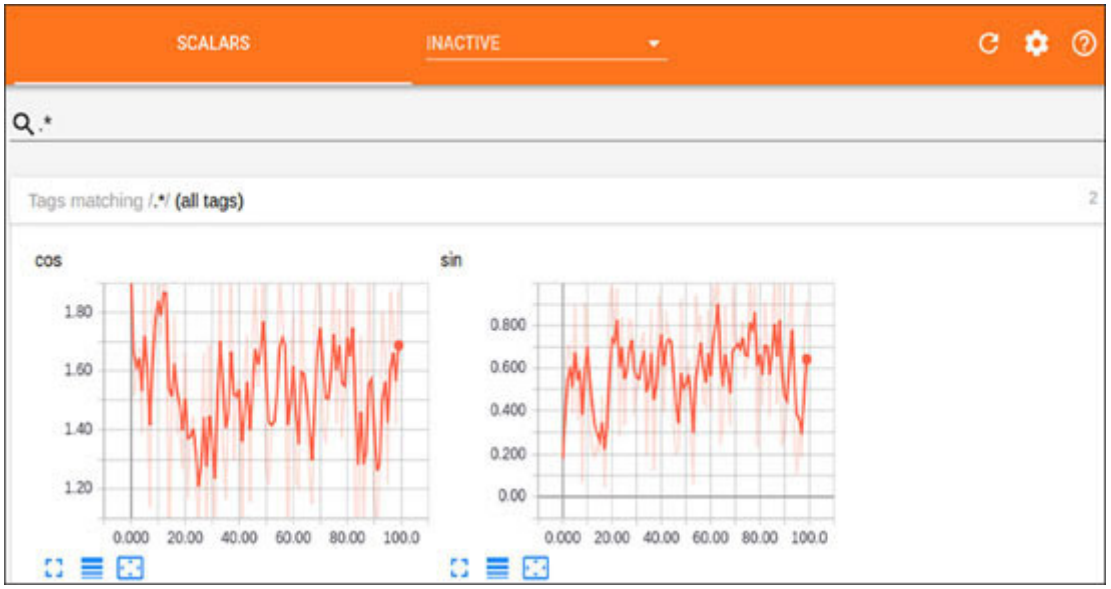


Figure 2.4: *How to add scalars to tensorboard using TensorboardX.*

Add scalar can be used to plot quantities like loss or accuracy on the Y-axis versus epochs on X-axis can be

plotted.

Projecting Images to TensorboardX

When we work with image datasets, we must also keep track of what output is generated at what epoch. TensorBoard provides good support for the images. In the following code, I am generating random images of size [3, 256, 256] and adding them to a writer with the add_image function.

Here, add image takes three parameters:

Names of the images

Image as a Numpy array or PyTorch array

Iteration number

Here's the code snippet to show scalar to the TensorBoard using TensorboardX:

```
from TensorboardX import SummaryWriter
import numpy as np
writer = SummaryWriter()
for i in range(0, 10):
dummy_img = np.random.random([3, 256,256])
# output from network
writer.add_image('Image', dummy_img, i)
```

```
writer.export_scalars_to_json("./all_scalars_2.json")
writer.close()
```

After successful execution of the tfboardExperiments/images.py script, you will get the following output in the TensorBoard:

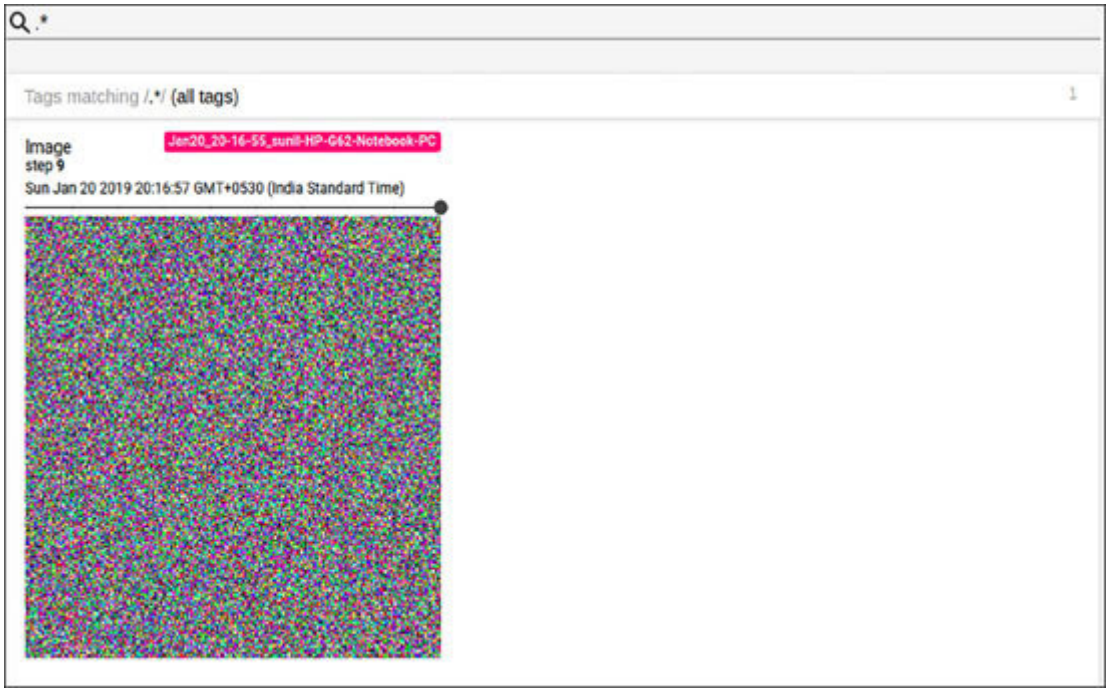


Figure 2.5: *How to add images to the tensorboard using tensorboardX.*

As shown in the preceding diagram, a slider/knob is provided to help you see images at a given step/ epoch/ time as specified during image writing.

Showing Text on tensorboardX

We may require text while working with the text, and TensorBoard has text support. While writing data, you must specify the identifier, text, and iteration number. The TensorBoard will show text grouped by the identifier in the UI.

The following code snippet shows text content in the TensorBoard with the help of tensorboardX:

```
from TensorboardX import SummaryWriter
writer = SummaryWriter()
for i in range(0, 10):
writer.add_text('mytext', 'this is a pen_' + str(i), i)
writer.export_scalars_to_json("./all_scalars_2.json")
writer.close()
```

After executing the tfboardExperiments/text.py script, you will see the following output:

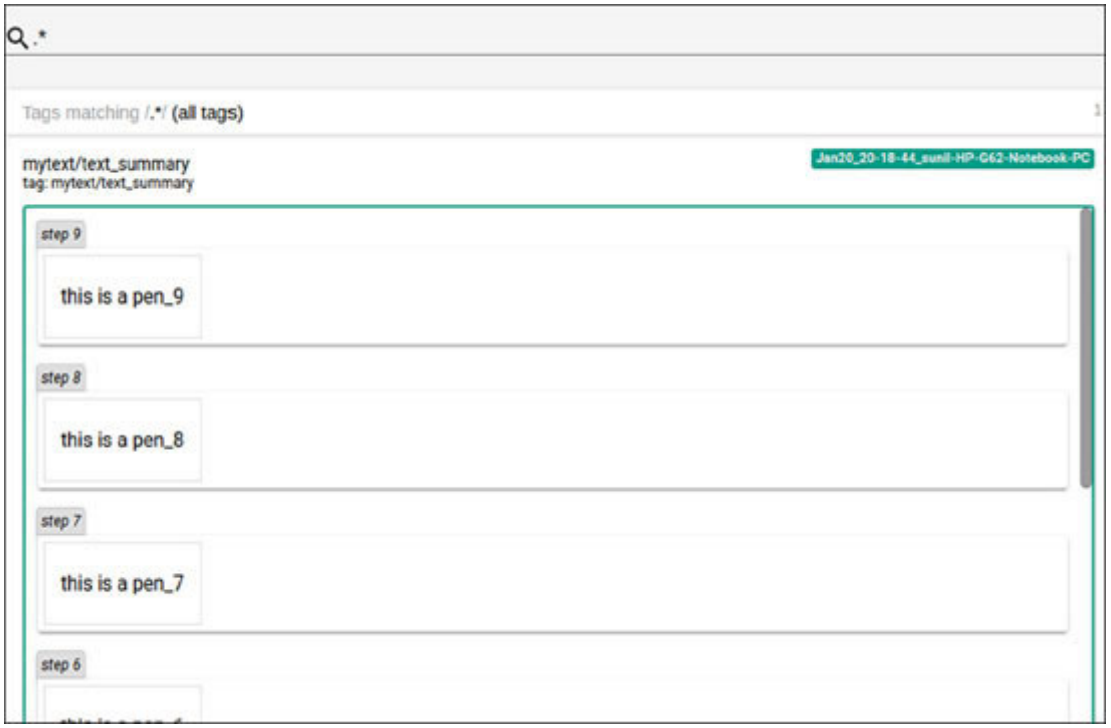


Figure 2.6: *How to add text to the Tensorboard using tensorboardX.*

Projecting Embedding Values on tensorboardX

The last function that we will use extensively is embedding—a higher dimensional representation. Visualizing such embeddings in lower dimensions (generally 2D or 3D) provides an idea about the quality of the embedding. Here, I have generated random embeddings of dimension 20 for 32 numbers to demonstrate the functionality. Add the Embedding function to take this high-dimensional vector array label as input.

The following code shows embeddings to the TensorBoard using TensorboardX :

```
from tensorboardx
import SummaryWriter
import numpy as np
writer = SummaryWriter()
i = [np.random.randint(0,100)
for i in range(0,32):
writer.add_embedding(np.random.random([32,20]), i)
writer.export_scalars_to_json("./all_scalars_2.json")
writer.close()
```

After running the preceding tfboardExperiments/embeddings.py you will get the following output. TensorFlow generates 3D projection by PCA or TSNE and helps in the inspection. Upon

selecting any of the points in TensorBoard, it shows other nearby points:

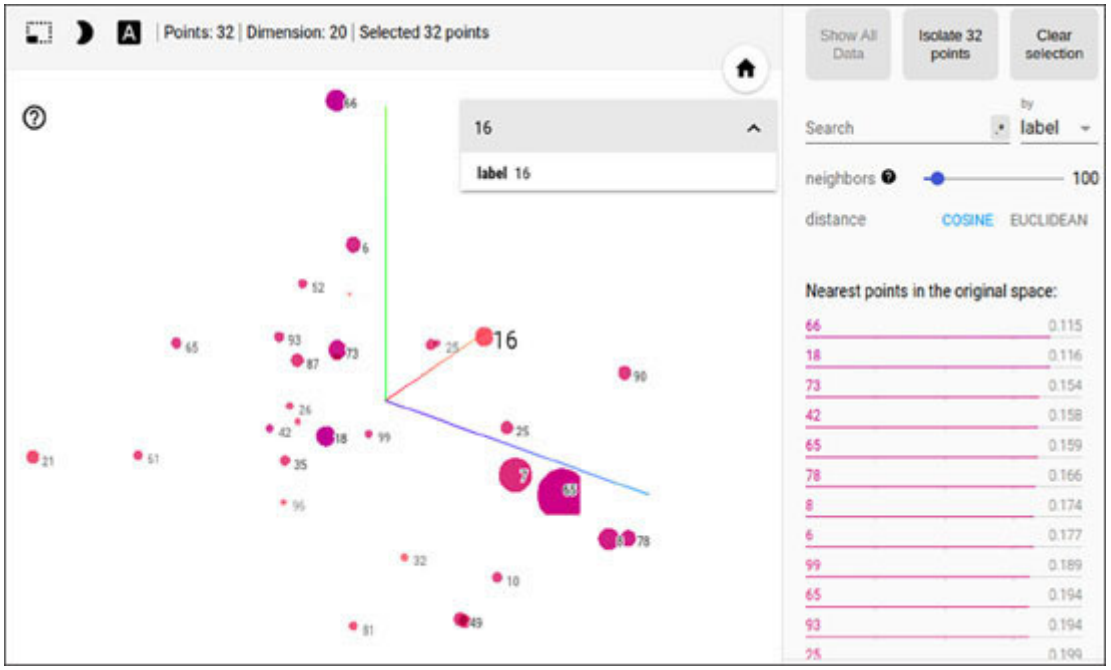


Figure 2.7: *How to add Embedding to the Tensorboard using TensorboardX.*

To practically see how TensorBoard is helpful, I have provided a tfboardExperiments/MNIST.py script, which uses the network with one linear hidden layer and optimizes the network for 5000 epochs to identify digit in the MNIST dataset. This script will write scalar, histogram, and images to TensorboardX. One of the benefits of using TensorBoard is that you can always monitor progress/parameters and see whether the network is converging.

You can visit the following link:

TensorboardX:
https://tensorboardX.readthedocs.io/en/latest/tutorial.html

Conclusion

In this chapter, we went one step ahead in our goal to master the domain of natural language processing. We started with data retrieval, and processing knowledge of such technique is essential to collect application-specific data. Remember, generally available data cannot satisfy all business needs, so one will always need a custom dataset for custom problems, which can be gathered from various sources. Stemming and Lemmatization techniques are, although not perfect, used for machine learning applications and significantly help reduce/concentrate the vocabulary size. We covered the basic tokenization techniques. That said, modern techniques like Bert are using far more advanced and memory-efficient tokenization techniques, which we will cover in the upcoming chapters. We moved on to learn a basic snippet of PyTorch—the framework that we will use throughout this book. PyTorch text is the package that greatly simplifies the preprocessing pipeline, so we will use TorchText and PyTorch data loaders—a more generalized class for data loading that comes out of the box with PyTorch.

In the next chapter, we will learn how to represent language mathematically.

CHAPTER 3

Representing Language Mathematically

In previous chapters, we have covered all that required to understand and implement basic machine learning models. We have also seen how to utilize TorchText and TensoboardX to process the text better and visualize the results as well as the training process. This chapter is about getting started with the mathematical representation of the text. Representing text mathematically so that it offers better meaning to the machine. This an active area of research, and many big players such as Google, Microsoft, and Facebook are constantly working and producing state of the art models. This chapter will walk you through the basic count-based vectorization approaches such as Co-occurrence Matrix and TF-IDF. This chapter in-depth covers the predictive dense vector generation approaches such as Word2Vec, GloVe, and FastText.

Structure

In this chapter, we will cover the following recipes:

Prerequisite

Encompassing knowledge to numbers

Understanding co-occurrence matrix

Understanding TF-IDF

Understanding Word2Vec

Understanding GloVe

Understanding character-based embedding

Objective

This chapter mainly focusses on how to convert language to mathematical representation. Such mathematical representation of the human language can be consumed by machine learning/deep learning algorithm to perform the different natural language processing tasks.

Prerequisite

I have provided some of the examples through codes. Codes for this chapter are present in the folder Ch3 at GitHub repository To understand this chapter, you require to have some basic knowledge about the following python packages:

Numpy

Scipy

Matplotlib

FastText

Torch

TensorboardX

Nltk

Tqdm

Matplotlib

You may install these requirements by installing all the packages listed in It can be simply done by issuing pip install -r

Encompassing knowledge to numbers

Computers are made for the purpose to compute with numbers. Mathematical logic can only be applied to numbers. Natural language is used by humans, and it has very different fundamentals then the numbers. Language is meant to communicate and has basic building blocks, such as characters and words. Mathematics is known as the language of the universe, and each representation has an absolute never changing meaning, whereas natural language has fuzziness. In natural language, one word or character can have a different meaning as per its context.

Language has a different level of organization. Such as a basic entity is called a character; many characters come together to form a word. Many words in a linguistic fashion, if arranged, form a meaningful sentence. Such a sentence, if arranged in peculiar order around a particular topic, such a bunch of sentences is known as In literature, sometimes sentence and paragraphs are referred to as A word is often referred to as a Some languages are character based such as Chinese and Japanese, wherein each character has a similar meaning as words in other languages. Generally, words are separated by spaces, but it's often not true; in Chinese and Japanese, there is no rule for token separation. A word in one language has a different meaning in another language. A

word surrounded by some tokens has a different meaning than the same word surrounded by different tokens in a sentence. This is also known as **polysemy** of the word. Polysemy means a word having a different meaning in a different context. These are a few peculiarities of languages to make you aware of the challenges hidden in the language understanding. To represent a word in the form of a number so that the number carries all these inherent characteristics of the language is even a bigger task to accomplish. Modern development has taken care of many of the aspects of language representation; still, much has to be accomplished to reach the human level performance.

Embedding is a high dimensional representation of the word or token. Here, high dimensional representation is a dense vector, with a floating number of a size from 100 to 1000. Let us say if we have any word, then it can be represented in a vector of size 100.

Understanding the different approaches of converting a word/token to its embedding

According to the previous discussion, a character or word or a sentence can be represented in the form of a number. I am keeping a little secret by not talking about the characters and sentences. Let us go ahead with representing word to numbers. Following are the ways in the Chronological order that comes to our mind when we think of representing word to numbers:

Id based If we have two words, **Cat** and Let us say we label id 123 to the word cat and id 345 to panther. These two ids don't show any relations between cat and panther. Both belong to the same family in classification, and both are mammals. Now, let us say, I take another word **pigeon,** then we cannot calculate the relation between the three words. Ideally, Cat and Panther are more related then Pigeon. In language, the way a word is written or spoken, every word is related to each other. If your simplest approach cannot be able to show the inherent relationship between the two words, then it is obviously going to fail. Another problem is if you have billions of words, then you have to number them from one to billion. Now, this seems a terrible idea. What if you have received a word which is a typo mistake, and the actual token was **each** In this case, your id-based approaches will not be able to find any id for such word, and it fails.

The one hot One hot encoding is representing categorical variables as binary vectors. This technique is inspired by digital circuits. Continuing with the preceding example. If we have vocabulary size as 2, the cat will be represented as [0, 1], and the panther will be represented as [1, 0]. This suffers from the same problem of being sparse. If we have billions of words, this technique would fail as we run out of memory. Additionally, there is no correlation between the two vectors.

Co-occurrence It is inspired by the contextual meaning of the words. *A* word is characterized by the company it keeps. This technique will be discussed in the next recipe of the chapter.

Term Frequency based measures: These methods provide weight to the word according to the occurrence of the word. Techniques like TF-IDF falls into this category. This technique will be discussed in detail in the upcoming recipes of this chapter. Co-occurrence and TF-IDF fall into a single category known as the **count-based**

Predictive approaches: This is the wider most class to generate embeddings. Methods belonging to this class usually take help of a neural network or deep learning algorithm such as LSTM and CNN to the meaningful dense representation of the word. These techniques operate at the level of characters, words, and sentences. We will be discussing these techniques in the upcoming recipes. Word level embeddings where the dense vector is generated based on word-level features, whereas character level embeddings where the dense vector is

generated based on the character level features. The word and character level embeddings will be discussed in the upcoming recipes. These embeddings convert a word into a fixed vector of size 100, 200, or 300. This vector is then taken as input to the subsequent machine learning tasks. The sentence level embeddings will be discussed in *Chapter*

Now, we know little about the embeddings, and the obvious question is why we require embeddings. The first thing that comes to mind is we are unnecessarily increasing the data-size by converting a single word in to a float vector of size 100. This type of high dimensional representational is also known as dense vector representation. Well, this seems to be a storage intensive solution, but this is the best we can do. In addition, if such vectors generated by the predictive approaches are properly built, then it produces meaningful relationships as follows:

Vectors are mathematically represented with double bars on both sides of the vector. Vector is which is represented as Vectors are many times represented by single bars on both the sides In the upcoming recipe, we will train our own word and character-based embeddings and practically see if the preceding mentioned facts hold true or not.

Refer to the following links:

Neural information retrieval: a literature review
https://arxiv.Org/abs/1611.06792

Symbolic, distributed, and distributional representations for natural language processing in the era of deep learning is as follows: a survey https://arxiv.Org/pdf/1702.00764.Pdf

Understanding co-occurrence matrix

A single word has no meaning in language, but when it is combined with context, it gives meaning. This is the main intuition behind this technique. Often in image processing, the co-occurrence of pixel is taken as a prominent feature. Similarly, in language, the co-occurrence of the word can also be considered a useful feature.

Generally speaking, the co-occurrence matrix is a square matrix with an equal number of entries in rows and columns. The value of the element for a given row and column represents the number of times both entities co-occur. Hence, to make such matrix, you need to define your number of entities as well as the context window in advance. Generally, the context window is kept between 5 and 10 as follows:

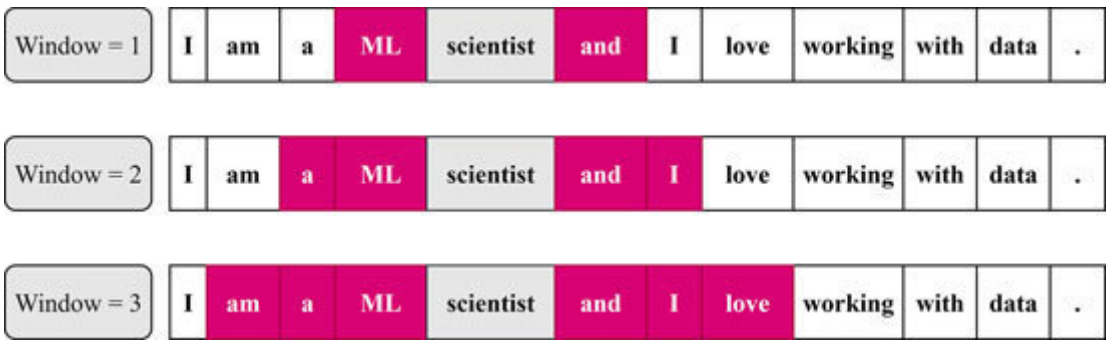


Figure 3.1: *Pictorial representation of target and context word by taking context windows 1 to 3*

The preceding figure is a pictorial representation of target and context word by taking context windows from 1 to 3.

Constructing a co-occurrence matrix

To give you an example, I am taking a context window as 1. It means that each word is defined by a word left to it and a word right to it. Mathematically, this relationship is defined as the co-occurrence matrix for each window. To construct a co-occurrence matrix, consider the following example text, ***I am an ML scientist, and I love working with***

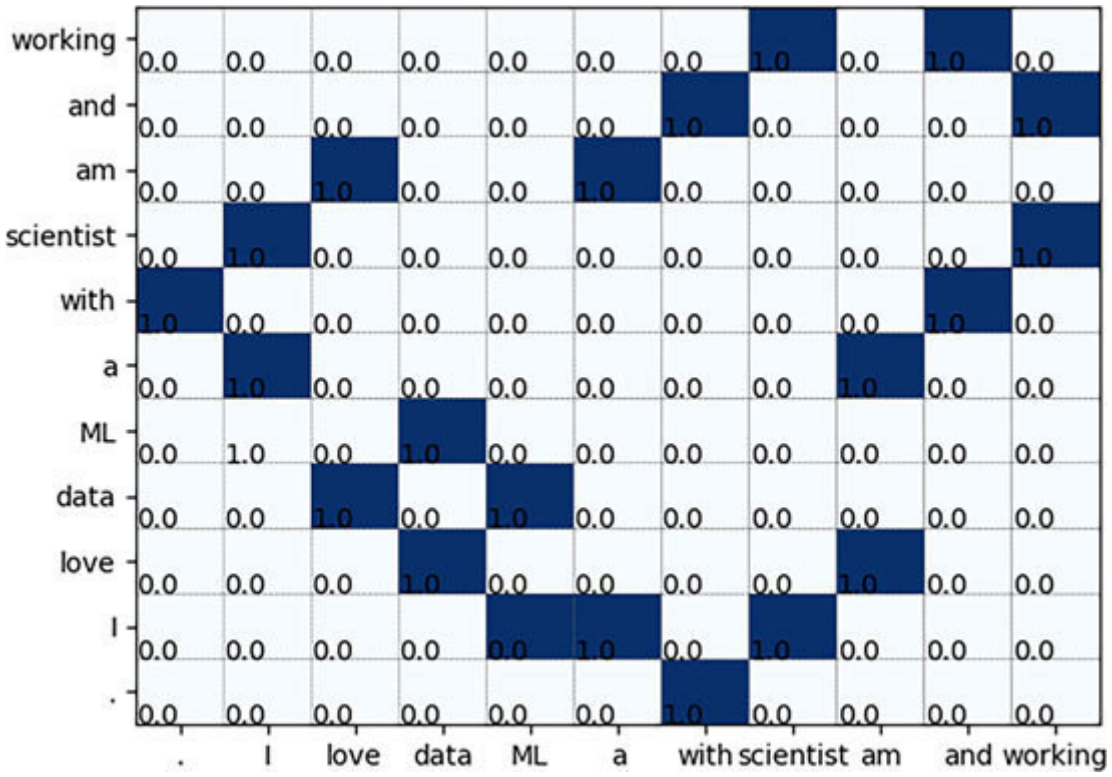


Figure 3.2: *Co-occurrence matrix for an example statement.*

Preceding is the co-occurrence matrix for an example sentence. By default, the co-occurrence for the word is not counted with self, and hence, the diagonal in such a matrix is always zero. The co-occurrence matrix usually has bilateral symmetry, and such a matrix remains highly sparse. Here, I have just put 1 wherever co-occurrence exists; there can be other variation where number count of co-occurrence frequency two words occur can be counted in some of the cases where the relative distance between two words can be calculated, and the co-occurrence matrix is constructed. Here, the nearby word gets the most weight, and this weight decays as the distance between the two words increases in the context windows. We will be using this kind of co-occurrence matrix while constructing GloVe embeddings. I have implemented a basic version of such a matrix, and it is shown as follows:

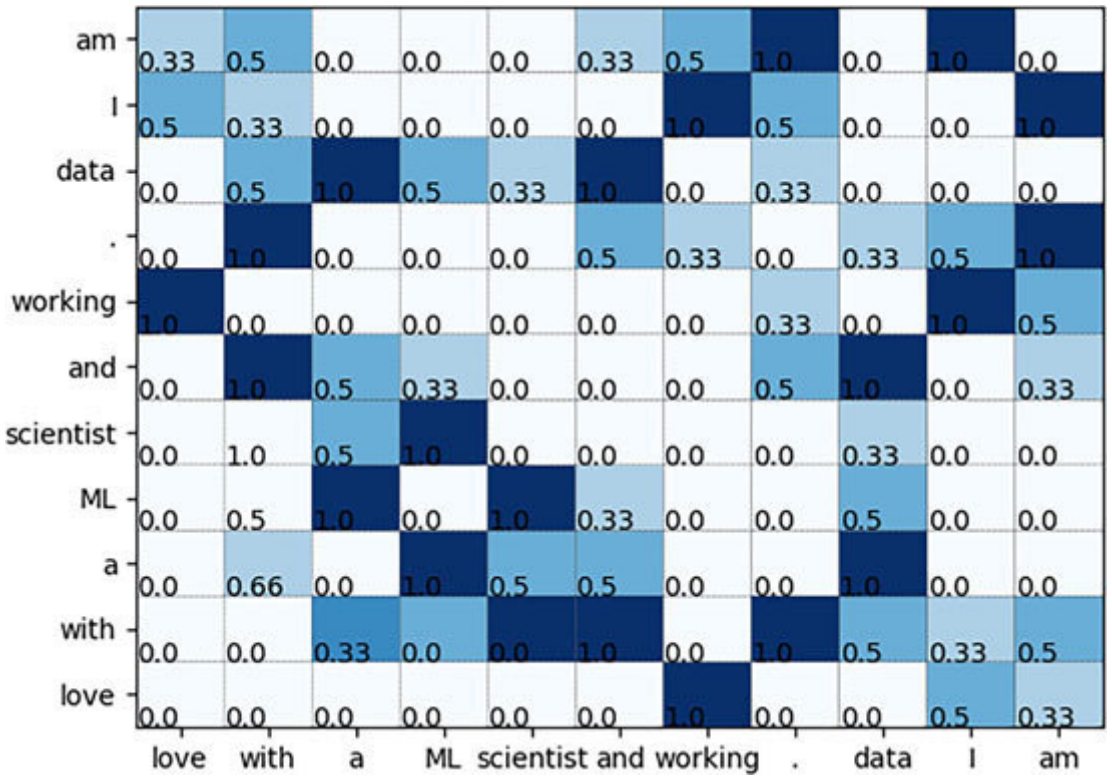


Figure 3.3: *Co-occurrence matrix with distance-based weight decay.*

You may experiment with script A function _create_cooccurrence_matrix_ is the main function in the script as follows:

```
def create_cooccurrence_matrix(text, context_size):
word_list = text.split()
vocab = np.unique(word_list)
w_list_size = len(word_list)
vocab_size = len(vocab)
w_to_i = {word: ind for ind, word in enumerate(vocab)}
comat = np.zeros((vocab_size, vocab_size))

for i in range(w_list_size):
for j in range(1, context_size + 1):
ind = w_to_i[word_list[i]]
if i - j > 0:
lind = w_to_i[word_list[i - j]]
comat[ind, lind] += round(1.0 / j, 2)
if i + j < w_list_size:
rind = w_to_i[word_list[i + j]]
comat[ind, rind] += round(1.0 / j,2)
return comat
samples = 'I am a ML scientist and I love working with data
.' cooccurrence_matrix = create_cooccurrence_matrix(samples,
context_size=3)
```

By keeping the context size between 1 and 3 in the preceding script, you will see a different type of co-occurrence matrix, as shown in the preceding figure. The co-occurrence matrix

seems to be one step ahead of our simplest id-based approach because it takes context into consideration, and hence, makes it more sound fundamental.

A co-occurrence matrix is used in constructing GloVe and Word2Vec embeddings. There are certain disadvantages in using a co-occurrence matrix, which is eventually restricting its usage; these disadvantages are listed as follows:

Once the matrix is constructed, adding new word requires recalculation of the entire matrix.

Change in the context windows also requires recalculation of the entire matrix.

This matrix tends to become voluminous owing to its sparse and symmetric nature. Handling a co-occurrence matrix with vocabulary size in million requires a distributed system like HDFS. It requires a huge memory to store the co-occurrence matrix.

GloVe embedding extends the concept of the co-occurrence matrix to one step further by applying the statistical learning on top of it. We will learn in detail about GloVe embedding in the upcoming chapters.

As we discussed at the beginning of this chapter that the use of co-occurrence matrix is not restricted to text processing, but it is used in image processing too. **Grey-level co-**

occurrence matrix is used in vision processing to characterize the texture of an image by calculating how often the pairs of a pixel with specific values and in a specified spatial relationship occur in an image.

Please take a look at the following reference:

Grey level co-occurrence matrices: generalization and some new features: *https://arxiv.Org/pdf/1205.4831.pdf*

Understanding TF-IDF

Term Frequency-Inverse Document Frequency has two following parts into it:

Term frequency

Inverse document frequency

The TF-IDF technique was historically used by search engines to rank the domains and to calculate cost per clicks (with some additional features). TF-IDF is mostly used in recommended systems. TF-IDF is not used much in deep learning model to a major extent, but it is used in machine learning.

Term frequency

TF refers to the frequency of the word appearing in the document divided by a number of words in the document. TF helps in identifying those words which occur more often in the document. Such words with high frequency are usually stop-words, such as or TF can be mathematically designated as follows:

$$TF = {f_{t,d}}/{\sum_{t' \in d} f_{t',d}}$$

Where is the frequency f of the given term *t* in the document and it is divided by sum of all other terms present in the document Consider a document containing 1000 words wherein the word **cat** appears three times. The term frequency (that is, TF) for cat is then (3/1000) = 0.003.

Inverse document frequency

The term frequency is the measure that takes into account the frequency of the word in a document. What about those words that are more important but are not more frequently used? The inverse document frequency is here to take care of the important but less frequent words and nullify the most frequent words, which are usually known as stop-words. IDF can be mathematically defined as follows:

$$IDF = \log \frac{N}{N_t}$$

Where IDF is log of N (total number of words in document) divided by (the number of documents where the term is present). Now, assume we have 100 million documents, and the word Stephen appears in one thousand of these. Then, the inverse document frequency (that is, IDF) is calculated as log(100,000,000/1,000) = 5. If a more frequent word like **the** occurs in almost all documents, then such word decreases, and hence, the importance of such word is reduced as follows:

TF – IDF = TF * IDF

Thus, the TF-IDF weight is the product of these two quantities: *0.003 * 4 =*

Constructing TF-IDF matrix

To create TF-IDF, one may use TfidfVectorizer function from Scikit-learn. To get TF-IDF of any corpus, we need to provide each corpus as a separate element of the array. Here, this is the one document.', 'this is the second document.', and 'and this is the third one.' are considered to be individual documents as follows:

```
corpus = [
'this is the one document.',
'this is the second document.',
'and this is the third one, which is very similar to first one.',
'is this the first document relates to politics?',]
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())
print(csr_matrix.todense(X))
```

Run the script Ch3/tfidf.py, and with the help of the plotting library Matplotlib, the TF-IDF can be visualized as follows:

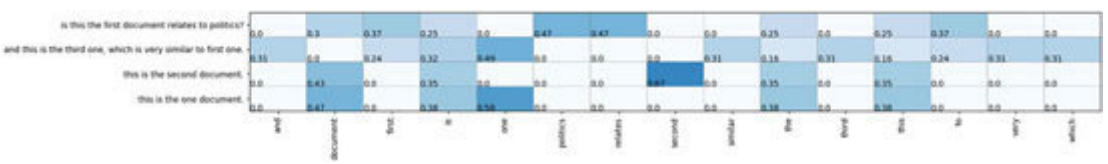


Figure 3.4: *TF-IDF applied to example sentences*

From the figure, you can see that the TF-IDF value for words like **second** and **first** is higher than the more frequent words like **the** and In the present matrix, the difference between the TF-IDF value of a more and less frequent word is minor. This difference will grow as the corpus size increases, so it is very important that one should take a sufficiently large corpus to make sense with TF-IDF. After TF-IDF, the first document 'this is the document one.' can be represented by the resultant TF-IDF. This vector can be used in the subsequent machine learning pipeline.

TF-IDF can be applied to many other areas like its usage to determine the word relevance. TF-IDF calculates values for each word in a document through an inverse proportion of the frequency of the word in a particular document to the percentage of documents the word appears in. Words with high TF-IDF numbers imply a stronger relationship with the document they are present with, suggesting that if that word were to appear in a query, the document could be of interest to the user.

Please refer to the following link:

Using TF-IDF to determine word relevance in document queries: citeseerx. Ist. Psu. Edu/viewdoc/download? Doi=10.1.1.121.1424&rep=rep1&type=pdf

Understanding Word2Vec

The word to vector technique is popularly known as Word2Vec. Word2Vec is basically taking the concept of co-occurrence-based information to the next level by applying a single hidden layered neural network to it. Word2Vec was originally proposed by a Google researcher **Tomas Mikolov** in 2013. Word2Vec belong to the category of **Vector Space Models** These models usually represent a word into a multi-dimensional vector; such a vector is developed so that the similar or most often co-occurrence word is placed nearby in the vector space. The different approaches that leverage this principle are divided into two categories as follows:

Count-based method

Predictive approaches

As we have already discussed, count-based methods learn co-occurrence of different words with a certain window size and generate the matrix. Such a matrix with high dimensions is usually converted into a dense vector by using **Principle Component Analysis** or **Singular Vector Decomposition** Whereas in predictive approaches hidden representation is learned by forcing the network to predict the context of the

word from target word or another way around. Word2Vec falls into this category. Word2Vec is a computationally reliable and mathematically stable approach that learns vector representation by using either **Continuous Bag of Words** or SkipGram technique. These two techniques are different ways by which a Word2Vec model can be trained. Before we go into the technicality and details of these two techniques, let us see how in general a Word2Vec model is trained. A Word2Vec model is a single hidden layered neural network with linear or no activation (or linear activation). It has three layers, namely, an **Input** a **Hidden** and an **Output Layer** as shown in the following figure:

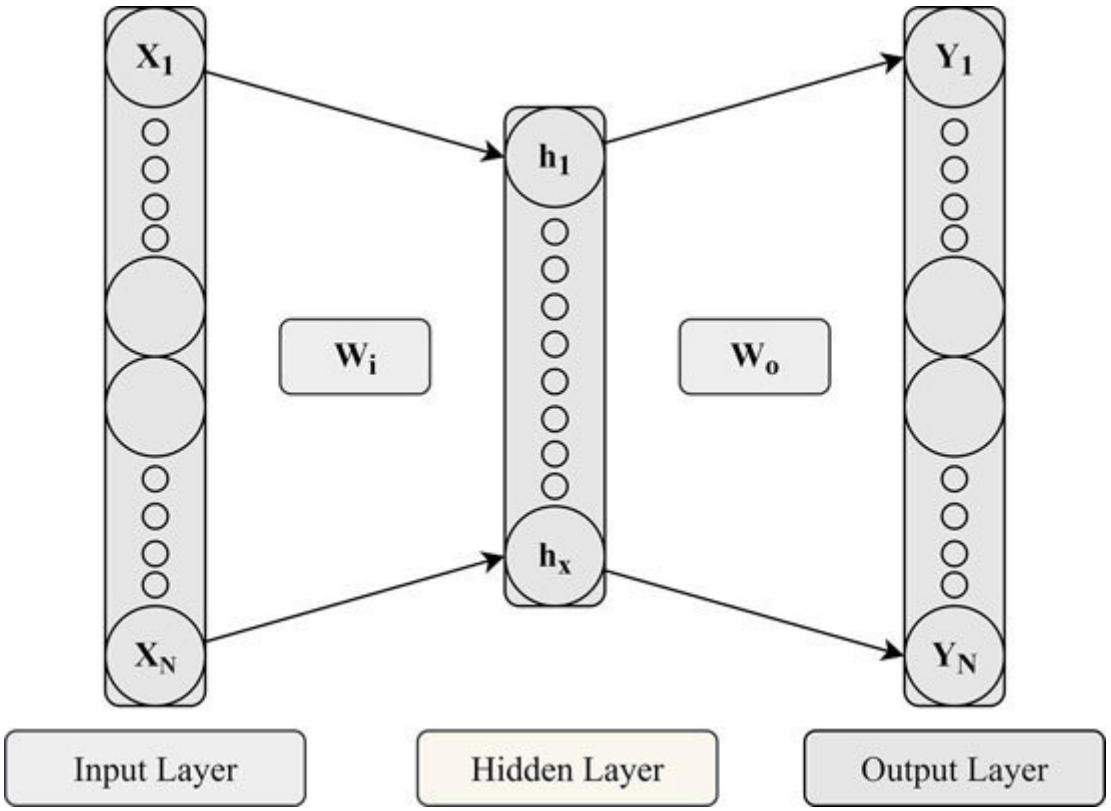


Figure 3.5: *A generic neural network architecture to train Word2Vec model*

Mathematically, it can be given as follows:

$$\underbrace{X_{(1,500)} * W_{i(500,300)}} \rightarrow H_{(1,300)} * \underbrace{W_{O(300,500)} \rightarrow \hat{y}_{(1,500)}} \rightarrow Softmax \rightarrow Argmax \rightarrow Error$$

Here, in the preceding equation, we are having a vocabulary size of 500, so each token *X* can be given as one hot vector of size (1, 500). We want to keep our embeddings *H* dimension as 300, so we multiply input *X* with the weight matrix of dimension (500, 300). Now, this embedding vector is multiplied with another weight matrix of size (300, 500) to convert it to a target vector representation, which is mostly in a float number showing the likelihood for each vocab token. Softmax operation is applied to such output to calculate the probability distribution. In this distribution, the target token is one that is having the highest probability. If this predicted token $\hat{Y}$ is the same as then error is zero else error back propagates, and weights are adjusted accordingly.

The torch is used to design our three layered neural networks. tensorboardX for to write output so that TensorBoard can plot it. SummaryWriter writes the output to disk and will be read by tensorboardX as

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from tensorboardX import SummaryWriter
import numpy as np
```

```
writer = SummaryWriter(log_dir='runs/')
torch.manual_seed(1)
```

Understanding methods to train Word2Vec

In this section, we will be learning the methods that are required to train Word2Vec:

Continuous Bag of Words CBOW is one of the approaches to train the Word2Vec model. The CBOW model takes the context word and tries to predict the Target word. In the following figure, considering the window size, 2 are the context words, and the word is the target word.

Figure 3.6: *Showing target and context words with co-occurrence window = 2*

It is showing the target and context words with the co-occurrence window = 2. The Network consists of an input layer that takes all these contextual words and tries to predict the target word. Let us say we have a small vocabulary size of 500. Each input word *X* will be represented by one-hot encoding so that each context word will be represented by a vector of size [1, 500]. Let us decode that we want our embedding size to be N = 300. Considering this, the size of input weight will be [500, 300]. The Dot product of *X* with produces a hidden vector of size 300. The hidden dimension

is multiplied with another weight of size [300] to produce a target token of dimension [1, 500].

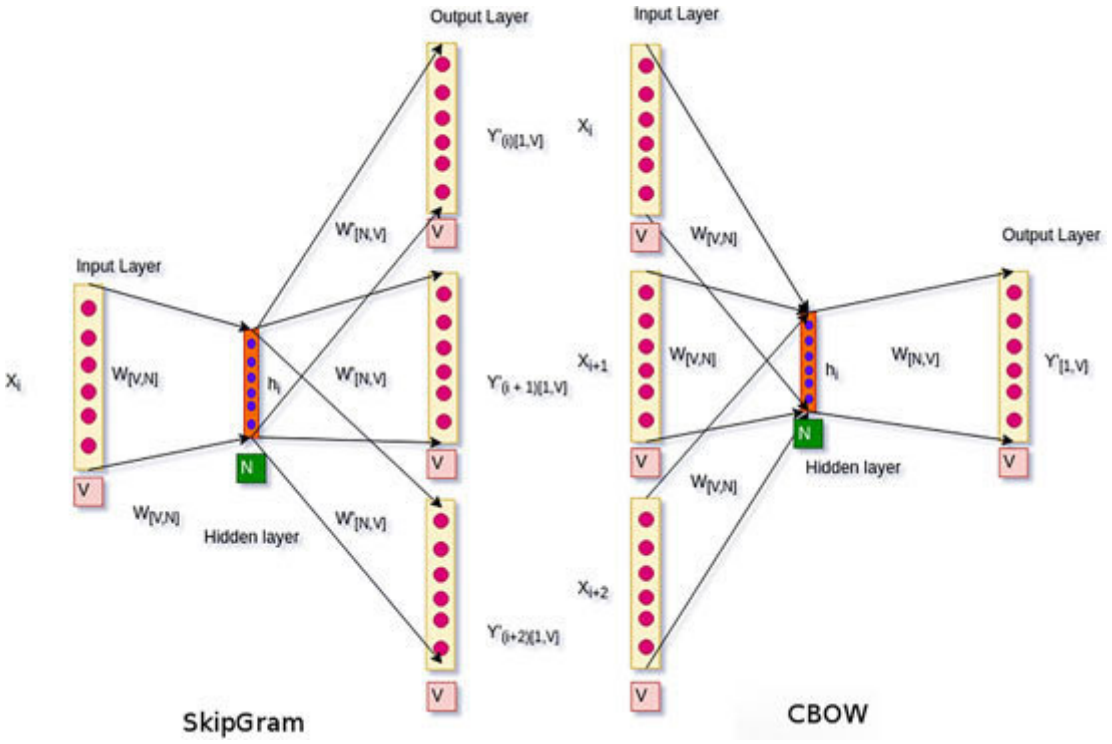


Figure 3.7: *showing SkipGram and CBOW network architecture*

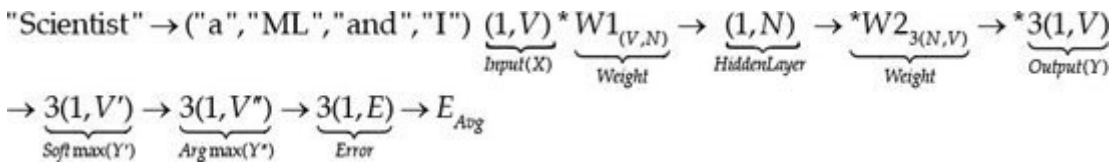
The entire CBOW algorithm can be, mathematically, summarized as follows:

$$("a","ML","and","I") \rightarrow "Scientist" \underbrace{3(1,V)}_{Input(X)} * \underbrace{W_{i(V,N)}}_{Weight} \rightarrow \underbrace{3*(1,N)}_{HiddenLayer(H)} \rightarrow \underbrace{*Wo(N,V)}_{Weight} \rightarrow$$

$$\underbrace{*3(1,V)}_{Output(Y)} \rightarrow Softmax \rightarrow Argmax \rightarrow Error$$

Using a similar model to the CBOW. In fact, SkipGram is the reverse of the CBOW model. Here, the task is to predict a context from the target word. As per the preceding figure with window size 2, the context words will be ["a", "ML,"

“and,”, “I”] which are predicted using the target word “Scientist.” Dimensional explanation is just the reverse of the CBOW approach.



Having understood both the approaches, we will now be implementing both the approaches and visualizing the generated vectors using TensorBoard. In the recipe, I will be walking you through some of the essential components of the CBOW and SkipGram. If you want to run end-to-end examples, please refer to script By running this script end-to-end, you will get an idea about the following aspect of the Word2Vec:

Designing the pre-processing pipeline

Writing simple models like CBOW and SkipGram in PyTorch

Defining the loss and optimizer function

Tracking the loss and ensuring convergence

Examining the embeddings with TensorBoard

Implementation

In the following code implementation, I have covered an implementation related to CBOW only. In the script, I have also implemented the SkipGram related portion too.

Includes removing stop words by using the NLTK library. This module takes text and iterates over all the sentences, removes stop-words, and again returns the sentences with filtered words as follows:

```
def remove_stop_words(text):
all_sentence = []
stop_words = set(stopwords.words('english'))
for each_sentence in text:
word_tokens = word_tokenize(each_sentence)
filtered_sentence = [w for w in word_tokens if not w in
stop_words]
all_sentence.append(' '.join(filtered_sentence))
return all_sentence
```

Reading data, vocabulary Here, we are defining the context size as 2 that means two words to the left and right to the target word will be selected as context. Next is reading the text. Here, I am using a small dataset made-up by taking different abstracts from Wikipedia. You may look into the file Ch3/data/testdata_en.txt to have an idea about how our

dataset looks like. Next is generating vocabulary, identifying all unique tokens, and making a word to index and index to word dictionary as follows:

```
CONTEXT_SIZE = 2 # 2 words to the left, 2 to the right
text = open("Change this").read().split()
text = remove_stop_words(text)

vocab = set(text)
vocab_size = len(vocab)
print('vocab_size:', vocab_size)
w2i = {w: i for i, w in enumerate(vocab)}
i2w = {i: w for i, w in enumerate(vocab)}
```

Create a dataset for As shown in the preceding figure, CBOW takes the context words and tries to predict the target word. The dataset is prepared accordingly by using the following function:

```
def create_cbow_dataset(text):
"""
Create data for CBOW
"""
data = []
for i in range(2, len(text) - 2):
context = [text[i - 2], text[i - 1],
text[i + 1], text[i + 2]]
target = text[i]
data.append((context, target))
return data
```

You can execute a preceding function on to our text data, and you will observe that the prepared dataset for CBOW is having pairs of the context words and the respective target word as follows:

```
cbow_train = create_cbow_dataset(text)
>>>cbow sample (['First', 'Citizen :', '', 'proceed'], 'Before')
```

Creating CBOW As illustrated in the preceding figure that the CBOW model has three layers, namely, input, output, and a hidden layer. As per the PyTorch convention, the network class has two functions. In the __init__() function, we define all the layers that will be needed to construct the network. In the forward() function, we connect all these previously defined functions to construct a proper architecture as follows:

```
class CBOW(nn.Module):
def __init__(self, vocab_size, embd_size, context_size,
hidden_size):
super(CBOW, self).__init__()
self.embeddings = nn.Embedding(vocab_size, embd_size)
self.linear1 = nn.Linear(2*context_size*embd_size, hidden_size)
self.linear2 = nn.Linear(hidden_size, vocab_size)
def forward(self, inputs):
embedded = self.embeddings(inputs).view((1, -1))
hid = F.Linear(self.linear1(embedded))
out = self.linear2(hid)
log_probs = F.log_softmax(out)
return log_probs, hid
```

As you might have observed that the preceding model is outputting the hidden representation along with the log probabilities. This hidden representation is the crux of the model's learning. In *Chapter 4, Using RNN for* it will be very clear that the majority of the model that is being used to generate a character or word or sentence embedding are having two parts in the network, namely, an encoder and a decoder. Here, the first half of the network consists of an input layer and weights connecting to input, and the hidden layer is considered to be equivalent to the encoder. The later part weight connects to the hidden and output layer, and the output layer itself can be considered to be decoder. The crux of learning is learned by So, in the preceding network, log probability will be used in the training phase to converge the network, and the hidden value is of no importance here. Whereas in the test/inference phase, hidden weights are used as the dense representation of the word.

Training Here, we have defined our embedding's size as 100. So, at the end of the training, we will get a dense vector of size 100 for each word. This network will be trained for 10 epochs with a **negative log likelihood** loss function and **stochastic gradient descent** optimizer as follows:

```
embd_size = 100
learning_rate = 0.01
n_epoch = 10
def train_cbow():
losses = []
loss_fn = nn.NLLLoss()
```

```
model = CBOW(vocab_size, embd_size, CONTEXT_SIZE,
hidden_size).to(device)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
for epoch in tqdm(range(n_epoch)):
total_loss = .0
for context, target in cbow_train:
ctx_idxs = [w2i[w] for w in context]

ctx_var = Variable(torch.LongTensor(ctx_idxs).to(device))
model.zero_grad()
log_probs, _ = model(ctx_var)
loss = loss_fn(log_probs,
Variable(torch.LongTensor([w2i[target]]).to(device)))
loss.backward()
optimizer.step()
total_loss += loss.data[0]
losses.append(total_loss)
return model, losses
```

Examining quality on the Ideally, one should use a left-out portion of the dataset for the test, but in this case, I have used the training data because this dataset is too small, where the previously generated model will be used for the inference. In the inference process, the vector for all unique words in the dataset will be generated using the trained model. Post inference, all the dense vectors are dumped to disk for to visualize using TensorboardX as follows:

```
def test_cbow(test_data, model):
vector_array = []
predicted_word_array = []
```

```
correct_ct = 0
for ctx, target in test_data:
ctx_idxs = [w2i[w] for w in ctx]
ctx_var = Variable(torch.LongTensor(ctx_idxs).to(device))
model.zero_grad()
log_probs, hidden = model(ctx_var)


_, predicted = torch.max(log_probs.data, 1)
predicted_word = i2w[int(predicted[0])]
if (predicted_word not in predicted_word_array):
vector_array.append(np.array(hidden.to('cpu').detach().numpy())
[0])
predicted_word_array.append(str(predicted_word))
if predicted_word == target:
correct_ct += 1
if correct_ct == 10000:
break
# for visualization using tensorboardX
writer.add_embedding(torch.Tensor(vector_array),metadata=predic
ted_word_array,global_step=2)
writer.export_scalars_to_json("all_scalars.json")
writer.close()
```

To demonstrate the effectiveness of the Word2Vec techniques, I have used a custom corpus and fed it to the Word2Vec algorithm. This custom corpus has information from the open domain. You may have a look at the corpus that is placed at Finally, if we plot the 3D projection of the resulting vectors using TensorboardX, then it will look like as follows:

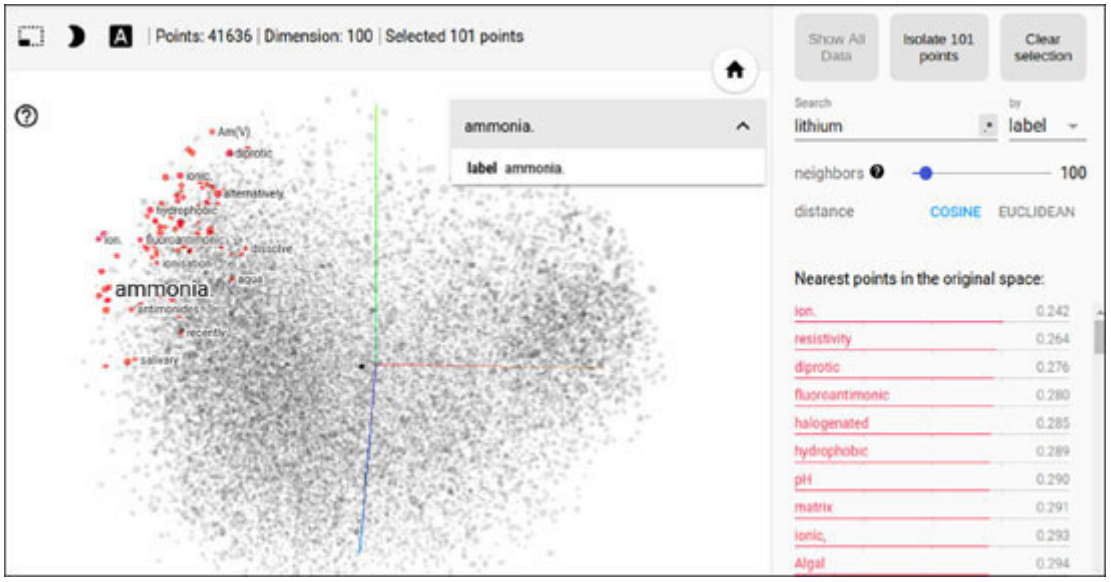


Figure 3.8: *Showing 2D projection of word vector learned by using Word2Vec, showing similar word getting clustered nearby.*

I have selected the word ammonia and all its probable neighbors such as ion, resistivity, diprotic, and so on are shown. The result was based on the basic implementation as per the first paper of the Word2Vec. Moving ahead in the next section, we will be discussing the improvement in Word2Vec as published in the second part of the paper.

Word2Vec improved version

Tomas Mikolov published a second version of the Word2Vec to introduce some of the techniques that makes training the model easier. The publication was entitled as Distributed Representations of Words and Phrases and their Compositionality. These three improvements are as follow:

Sub-sampling

Combining word pairs and phrases

Negative sampling

Sub-sampling

The stop-words such as “and,” “a,” and “I” are nowhere related to the word such as “Scientist” and do not help in relating the context to target. Word2Vec addresses this problem by introducing the sub-sampling techniques. Sub-sampling can be effectively implemented by the equation shown as follows.

$$P(w_i) = \sqrt{z(w_i), 0.001} + 1 * (0.001 / z(w_i))$$

Where is the probability of keeping the word, is the word, and is the fraction of the total word. For example, if the word “and” appears 1, 00, 000 times in 1 million words, then = 0.1. The constant 0.001 is known as “sample,” and it controls how much sub-sampling occurs. I have used the following code to reproduce the meaning of the sub-sampling approach:

```
def get_chances_of_being_kept(fraction):
sample = 0.001
return (math.sqrt(fraction/sample)+1)*(sample/fraction)
fractions = np.arange(0.01,1.0,0.001)
chances_of_kept = [get_chances_of_being_kept(i) for i in fractions]
plt.title(“Chances being kept v/s Occurance in Fraction”)
```

```
x = chances_of_kept
plt.ylabel("Chances being kept")
plt.xlabel("Occurance in Fraction")
plt.plot(x, fractions);
```

In the following plot, the 0.05% of words in the vocabulary are having 100% chance of being kept at the threshold of 0.05%. The plot having an exponential decay shows that a large number of words are repeating, and a very high fraction of words has a high number of chances being rejected as follows:

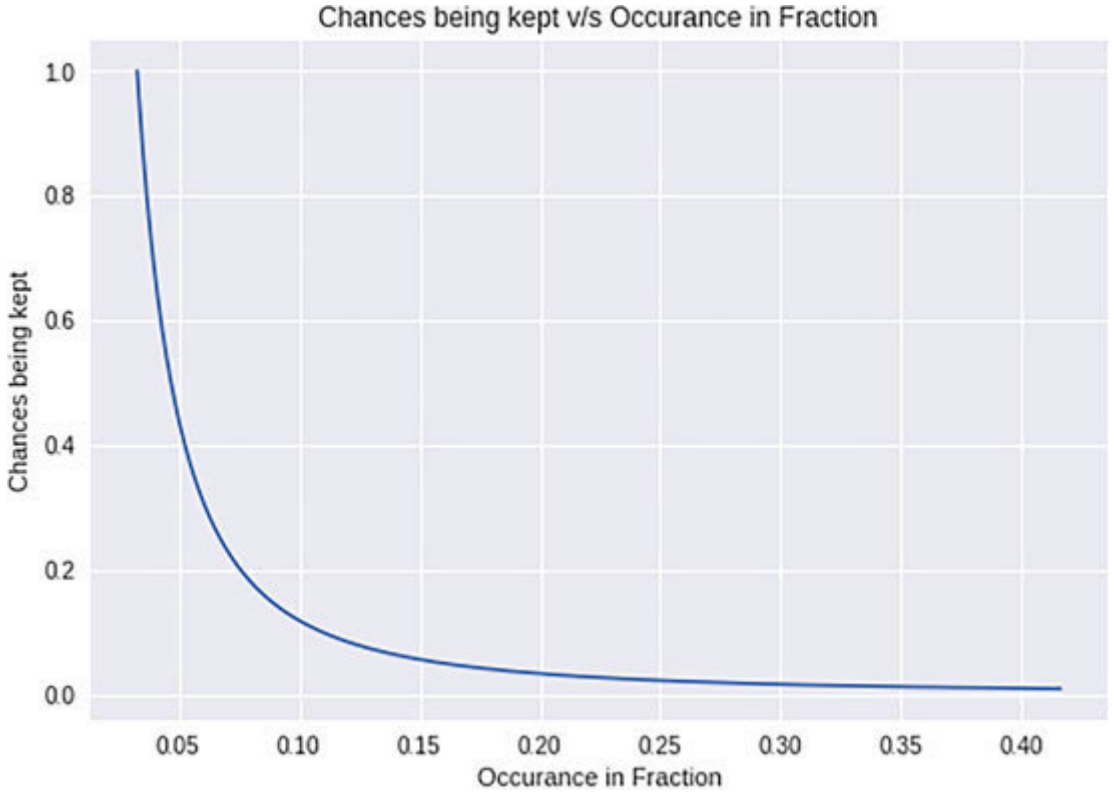


Figure 3.9: *Demonstrating Sub-sampling logic in to improve Word2Vec*

Word pairs and phrases

The author has pointed out that some word pair carries meaning and tearing apart such phrases breaks the overall meaning. For example, if the word **New York Times,** the name of a newspaper, is broken in tokens, it will lose its meaning. To solve this, the author recommends finding such pairs that are having a high occurrence in the vocabulary and considers them as a single token. Well, this seems a primitive idea, but when Google implemented this, the 100 billion vocabulary size decreased by 3 million words.

Negative sampling

Word2Vec has two weight matrices to tune. Let us say we have a hidden size of 300 and a vocab size of 10000, we have a very high number of weights attached to it to be changed in each epoch *300*10, 000 = 30, 000,* This size becomes unimaginable when the vocab size further increases. In this case, the network is trained by taking one positive sample, and several negative samples are those that lie far apart, and we want our network to give *0* output for such words. We generally mix one positive pair with five negative words. Now, the overall network will be trained such that there are six words momentarily, and the loss is calculated. So, we are updating weights for only six samples, and (300*6) = 1, 800 weight values total. That's only 0.06% of the 3M weights in the output layer. This is a huge improvement, isn't it? Such pairs of Negative and positive words are provided at each iteration, and it greatly reduces the computational complexity.

Refer to the following links:

Efficient estimation of word representations in vector space: *https://arxiv.Org/abs/1301.3781*

Distributed representations of words and phrases and their compositionality: *https://papers.Nips.Cc/paper/5021-distributed-*

representations-of-words-and-phrases-and-their-compositionality.Pdf

Have a look at Google's vocabulary used to train Word2Vec: *https://github.com/chrisjmccormick/inspect_word2vec/tree/master/vocabulary*

Understanding GloVe

GloVe or Global Vectors for Word Representation is another technique to train word embeddings in an unsupervised way. GloVe was proposed by Stanford University researchers Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe is a much more principled approach then Word2Vec. As the GloVe is said to be much more principled, then there must be some drawbacks in the previous approach Word2Vec. These limitations are as follows:

As we have seen in the previous recipe, Word2Vec takes into account only local contexts of the words. It does not take into account global co-occurrence of the word into account.

Word2Vec is trained using the back-propagation, and hence, it will not be able to learn embeddings of rare words correctly.

GloVe vector implementation exists to achieve two following goals:

Creating a vector that carries meaning in vector space

Take in to account global count statistics not just local meaning

There is a differentiating factor between GloVe and Word2Vec implementation. Unlike Word2Vec that operates by streaming sentences, GloVe operates by a co-occurrence matrix. In GloVe, the loss is based on the word frequency. GloVe and Word2Vec both are having different approaches, but often their end results are similar. They generate vectors of similar quality; in some cases, GloVe wins in some Word2Vec. Here, in the following figure, we have taken window size = Here, the **scientist** is the target word and are the context word.

Figure 3.10: *Showing target and context words with co-occurrence window = 2*

In GloVe, we start off with building the co-occurrence matrix. We refer the co-occurrence matrix as Such that each element represents how many times a token i is appearing with a token J. Such a matrix will be bilaterally asymmetric. The co-occurrence matrix is constructed by keeping the window of some size. Unlike SkipGram techniques, we don't give constant weights to all the words in the window. In GloVe, less weight is given to the distant words. This weight change is defined by the following formula:

$$decay = \frac{1}{offset}$$

Offset means the distance of context word from the target word. As the offset increases, the decay in weight will be proportionally more.

Defining learnable parameters

Next, is defining learnable weight and bias for each pair of words:

+ + =

Where, are scalar biases for the main and context words.

Defining loss function

$$J = \sum_{i=i}^{V} \sum_{i=1}^{V} f(X_{ij})(w_i^T w_j + b_i + b_j - logX_{ij})^2$$

Here f is the function to prevent the influence of extremely frequent word on the embeddings. The original authors choose the following function to reduce the effect of such words.

$$weight(x) = min\left(1, \left(\frac{x}{x_{\{max\}}}\right)^a\right)$$

The preceding equation may seem to be confusing, but once we plot, its meaning will be very clear. Here, a is another constant whose value is taken 0.75 by default. The following code can be used to demonstrate the weight function:

```
import matplotlib.pyplot as plt
import math
xmax = 1000
a = 0.75
weight = [min(1,math.pow(x/xmax,a)) for x in range(1,2000)]
plt.plot([x for x in range(1,2000)], weight)
plt.ylabel("weight")
plt.xlabel("Xij")
```

plt.title("Weighting Function")

The function looks like as follows when plotted. As shown in the following figure, after the fragment grows beyond 1, the weight for such tokens no more increases and applies the same weight to all the frequent words.

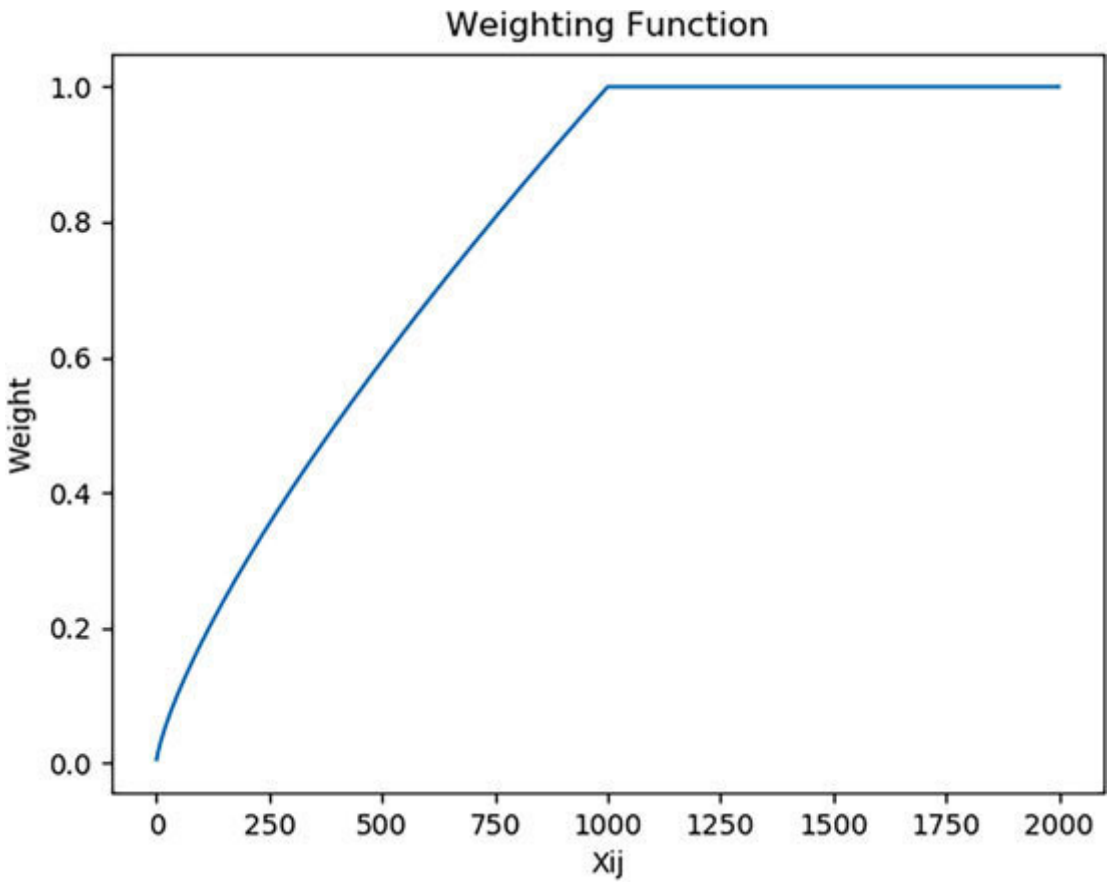


Figure 3.11: *Weight function to prevent the influence of extremely frequent word on the algorithm*

Defining Defining the context window 3 words left, and right will be chosen as a context word. As this is a very small corpus, I am defining xmax to be very small as As we are using a very small corpus, I have kept the Embedding size as

50 only. You may try running this code with a higher embedding size.

```
# Set parameters
context_size = 3
embed_size = 50
xmax = 2
alpha = 0.75
batch_size = 20

l_rate = 0.001
num_epochs = 10
```

Data It includes reading the file, constructing vocab, counting vocab, and constructing a word to index mapping as follows:

```
# Open and read in text
text_file = open('TorchGlove/short_story.txt', 'r')
text = text_file.read().lower()
text_file.close()
# Create vocabulary and word lists
word_list = word_tokenize(text)
vocab = np.unique(word_list)
w_list_size = len(word_list)
vocab_size = len(vocab)
# Create word to index mapping
w_to_i = {word: ind for ind, word in enumerate(vocab)}
'''
```

Constructing co-occurrence Weighted co-occurrence matrix is constructed as discussed in Recipe 1 of the present chapter.

In the end, the np.nonzero returns the indices of the elements that are non-zero. This helps in shrinking the matrix and decreasing the main memory space occupancy as follows:

```
# Construct co-occurence matrix
comat = np.zeros((vocab_size, vocab_size))
for i in range(w_list_size):
ind = w_to_i[word_list[i]]

for j in range(1, context_size + 1):
if i - j > 0:
lind = w_to_i[word_list[i - j]]
comat[ind, lind] += 1.0 / j
if i + j < w_list_size:
rind = w_to_i[word_list[i + j]]
comat[ind, rind] += 1.0 / j
# Non-zero co-occurrences
coocs = np.transpose(np.nonzero(comat))
```

Many important components

A weight function as discussed previously that helps in dealing with stop words/Words that are encountered more often and may diverge the model building

Weight and bias Weight and bias are initialized with the normal distribution. These parameters will change throughout the training and will be holding a crux of the learning. The weight matrix will be of size *embedding_size * vocabulary* The bias will be of a size equal to the vocabulary size.

Here, we are using the Adam optimizer to train the model.

```
# Weight function
def weight_function(x):
if x < xmax:
return (x/xmax)**alpha
return 1
# Set up word vectors and biases
left_embed, right_embed =
[[Variable(torch.from_numpy(np.random.normal(0, 0.01,
(embed_size, 1))),
requires_grad = True) for j in range(vocab_size)] for i in
range(2)]
left_biases, right_biases =
[[Variable(torch.from_numpy(np.random.normal(0, 0.01, 1)),
```

```
requires_grad = True) for j in range(vocab_size)] for i in
range(2)]
# Set up optimizer


optimizer = optim.Adam(left_embed + right_embed +
left_biases + right_biases, lr = l_rate)
```

PyTorch is very flexible and allows you to define your own architecture with different trainable weights and bias. As shown in the preceding code, you can add your custom parameters to optimizers so that the optimizer considers it as a trainable parameter and considers them in the back propagation graph.

Train the GloVe This function iteratively takes data from gen_batch function. Loss is calculated and back propagated to adjust the weights/bias with Adam.

```
# Train model
for epoch in range(num_epochs):
num_batches = int(w_list_size/batch_size)
avg_loss = 0.0
for batch in range(num_batches):
optimizer.zero_grad()
l_vecs, r_vecs, covals, l_v_bias, r_v_bias = gen_batch()
loss = sum([torch.mul((torch.dot(l_vecs[i].view(-1),
r_vecs[i].view(-1)) + l_v_bias[i] + r_v_bias[i] -
np.log(covals[i]))**2,weight_function(covals[i])) for i in
range(batch_size)])
avg_loss += loss.data[0]/num_batches
loss.backward()
```

```
optimizer.step()
print("Average loss for epoch "+str(epoch+1)+": ", avg_loss)
```

Here again, I am using TensorBoard to visualize our dense vectors generated by GloVe.

```
word_array = []
embed_array = []
word_inds = np.random.choice(np.arange(len(vocab)), size=100, replace=False)
for word_ind in word_inds:
w_embed = (left_embed[word_ind].data + right_embed[word_ind].data).numpy()
word_array.append(vocab[word_ind])
embed_array.append(torch.transpose(torch.Tensor(w_embed),0, 1).numpy())
writer.add_embedding(np.asarray(embed_array).reshape(-1,50), word_array)
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```

To demonstrate the effectiveness of our GloVe implementation, I will be using the same corpus as used to demonstrate the effectiveness of Word2Vec. This corpus can be found at When plotted, such vectors looks like as follows:

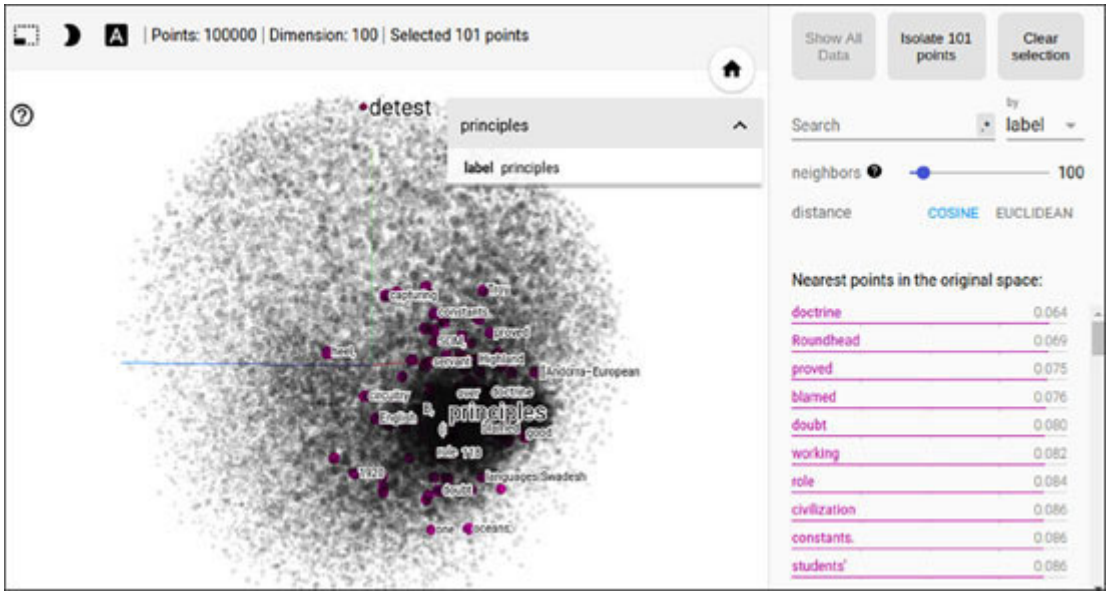


Figure 3.12: *Showing 2D projection of word vector learned by using GloVe, showing similar word getting clustered nearby.*

I have selected a word principle and all its probable neighbors such as and so on, as shown in the preceding figure. I have run the algorithm for 10 epochs and received these results; you may try with higher epochs.

Word vector so made with Word2Vec or GloVe needs to be evaluated. Evaluation of such vectors can be made by the following techniques:

Intrinsic Word pair are manually scored with the same dataset with which the training was done. This score is then matched with the cosine similarity score provided between word vector pairs generated by GloVe. The minimum is the difference between the two scores, the better is the word vector.

Extrinsic Word vector so made are used in various tasks. Performance of word vector is measured on such tasks and compared with the previous state of the art word embeddings. To perform extrinsic evaluation various standard datasets are used, such as CoLA, SST-2, MRPC, STS-B, QQP, MNLI-m, and so on. The overall score is calculated on as per the glue benchmark and the rank of embedding is determined.

Please refer to the following links for more details:

GloVe: Global vectors for word Representation:
https://nlp.stanford.edu/projects/glove/

A survey of word embeddings evaluation:
https://arxiv.org/pdf/1801.09536.pdf

A survey of word embeddings evaluation methods:
https://arxiv.Org/abs/1801.09536

Understanding character-based embedding

So far, we have seen that the embeddings are constructed using the word-based features. Word-based features are excellent but not efficient in many ways. Going a little deeper can provide us another dimension to train the vectors. Character-based embeddings are constructed using character n-gram as a feature. Xiang Zhang and Yann LeCun realized the potential of character-based embedding in their paper published in 2016. In this work, they conclude that the character-based features work better than the word-based features. A similar conclusion is derived from Googles paper entitled *Exploring the Limits of Language* The main advantages with character-based embedding are as follows:

You can better handle emoticons, new words, and misspelling words.

Character-based embeddings are better at handling rare-words. Word2Vec do not do better justice at rare words.

In character-based embeddings, it applies softmax on the vector of dimension equal to the unique character in vocabulary. On the other hand, word-based embedding needs to apply softmax on the vector of dimension equal to unique words. As the character set will be always smaller than the

word set, character-based embeddings are computationally less expensive.

Vectors can be formed if the word was not present while training the model.

Learning character embeddings can be summarized in the following steps:

Define a list of the character, including English characters, numeric characters, and special characters

Represent each character into on hot encoding, keeping empty vector for blank for unknown characters and spaces. Take one hot encoding for defined max length of the word if the word exceeds the max length, then chop-out extra character or pad with the blank token if smaller.

Use three 1D CNN layers (configurable) to learn the sequence

Similar to Word2Vec, the FastText can also be trained using SkipGram or CBOW approach.

Character embedding is a brilliant way that takes care of an unknown word as long as the word contains a character considered while building the model. FastText is the state-of-the-art charter-based model developed and pushed to the community by **Facebook AI Research** FAIR went one step ahead and included sub-word information such as **Facebook**

word can have “F”, “Fa”, “Fac” and so on as sub word to train the word. To demonstrate the charter-based embeddings techniques, I will be using FastText module in python.
FastText can be imported in the following ways:

```
import fasttext
```

Character-based embedding generation

One can use FastText to train the model either by CBOW or by SkipGram. To demonstrate the effectiveness of our FastText, I will be using the same corpus as used to demonstrate the effectiveness of Word2Vec and GloVe. This corpus can be found at

```
import fasttext
from tensorboardX import SummaryWriter
import numpy as np
writer = SummaryWriter()
# Skipgram model
model = fasttext.skipgram('data/testdata_en.txt', 'model')
words = model.words # list of words in dictionary
print ("words present in the model : ", words)
#visualizing using tensorboard
all_vectors = []
for eachword in words:
all_vectors.append(model[eachword])
writer.add_embedding(np.asarray(all_vectors), words)
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```

I have applied code from Here first the model is built using Fasttext. Following screenshot is the 2D projection of resultant vectors:

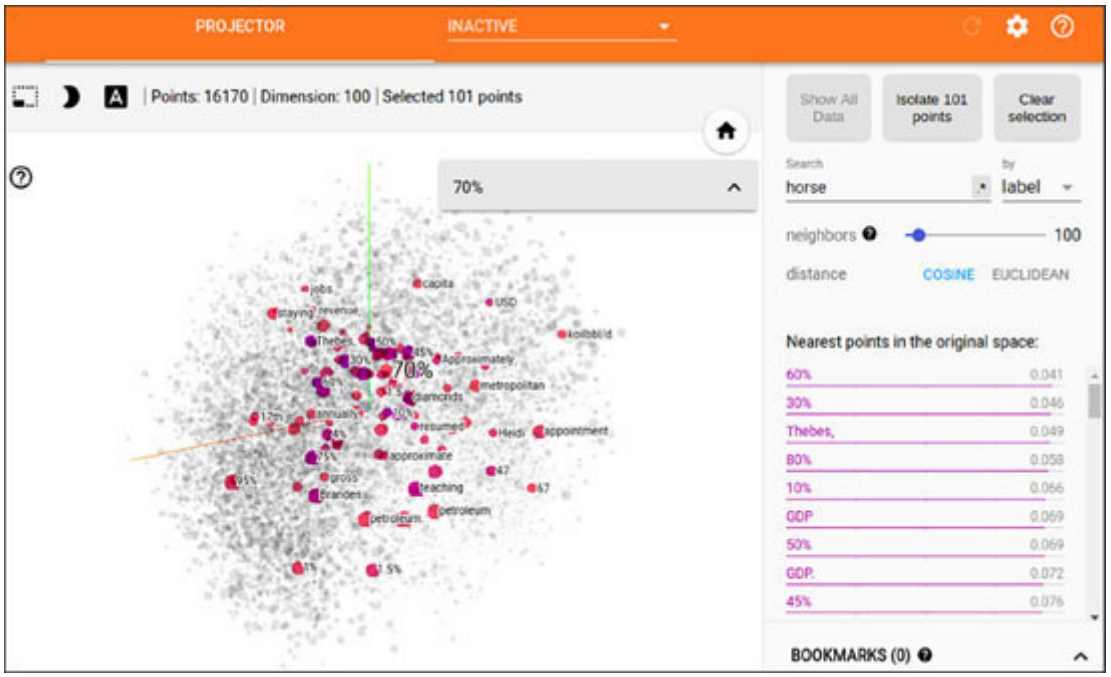


Figure 3.13: *Showing 2D projection of word vector learned by using Fasttext, showing similar word getting clustered nearby.*

FastText can also be used for the supervised text classification task. To demonstrate this capability, I will be using SMS spam collection dataset.

FastText requires a particular format to perform a supervised classification. The text must be formatted as: __label__ Where __label__ is an identifier followed by an actual label (in our case, it is either spam or ham). Then followed by some spaces, the text is provided on which the classification is to be done. The following are few lines from the dataset to give you a clear idea about the dataset.

Free entry in 2 a weekly comp to win FA Cup final tickets 21st May 2005. Text FA to 87121 to receive entry question (std txt rate) T&C's apply 08452810075over18's.

I can't believe how attached I am to seeing you every day. I know you will do the best you can to get to me, babe. I will go.

I have divided the SMS spam collection data set into two parts, namely, train and test. This dataset can be found under Ch3/dataset folder. To classify, the following steps are needed to be performed:

```
import fasttext
# train classifier
classifier = fasttext.supervised('data/SMSSpamCollection.train', 'model')
# check performance on test
result = classifier.test('data/SMSSpamCollection.test')
print ('Precision:', result.precision)
print ('Recall:', result.recall)
print ('Number of examples:', result.nexamples)
Precision: 0.9760239760239761
Recall: 0.9760239760239761
Number of examples: 1001
```

In our case, with a very small dataset, it provided close to 95% precision and recall on the test dataset. FastText is an extremely handy and powerful algorithm. Fasttext can be used to quickly prototype an application.

One need not build the Fasttext vectors every-time from scratch. You can download Fasttext vector trained on Wikipedia and common crawl from here This site has FastText vector available in 157 languages. This site also has the pre-trained model on various open source datasets such as ag news, Amazon review full, Amazon review polarity, DBpedia, Sogou news, yahoo answers, yelp review polarity, and yelp review full. These models can serve as a good starting point for training your text classifier. You may take one of the relevant models and fine tune your own dataset to see if you can get a better model with a fewer data.

The following are the references:

Text understanding from Scratch:
https://arxiv.org/pdf/1502.01710v5.pdf

SMS spam collection data set:
https://archive.ics.uci.edu/ml/datasets/sms+spam+collection

Conclusion

This chapter is of paramount importance in the entire book. Almost all your natural language processing pipeline will need to use the techniques discussed and demonstrated in this chapter. It is one of the biggest areas of research in the domain of NLP. We started with the understanding of the very first state the art word-based model Word2Vec. Word2Vec laid down the path forward on how the shallower perceptron model can be used to generate the uncompressed language representation.

Earlier, the embeddings were generated using the mathematical techniques like TF-IDF with or without techniques like PCA and SVD. Nowadays, research focusses on the deeper model for embedding generations like Bert and Elmo. We also realized the problem with the word-based model and the potential advantages of character and contextual models.

In the next chapter, we will be learning about using RNN for NPL.

CHAPTER 4

Using RNN for NLP

This chapter is the most important in this book. It covers the most-used and required techniques like vanilla recurrent neural network, gated recurrent units, and long short-term memories. This chapter covers how to use pre-trained embeddings or train embeddings from scratch. The difference in accuracy when the same task is carried out by different RNN units like Vanilla RNN, GRU, and LSTM is explored by implementing a network from scratch. This chapter also covers sequence to sequence learning tasks that have been extensively used in application like language translation and text summarization. We will also use batching with the sequence to sequence to process a batch of examples simultaneously for higher speed and better convergence. This chapter will also take you through the most important topics used in all recent NLP-related tasks.

Structure

The following topics will be covered in this chapter:

Understanding recurrent units

Implementing the concept of embeddings

Understanding advance RNN units

Understanding and implementing GRU

Understanding the sequence to sequence model

Understanding batching with Seq2Seq

Translating in batching with Seq2Seq

Implementing attention for language translation

Objective

Using RNN for sequence processing and understanding the concept of vanilla RNN. Practically understanding which one is better: GRU or LSTM. Although there are sub-libraries in Pytorch that have already implemented ready-to-go LSTMs, we will implement one from scratch.

Understanding how batch processing with sequence works and applying the same in implementing attention networks.

Implementing a simple seq-to-seq model with and without batching.

Using highways to build deeper architecture.

Understanding the *Transformer* - how holy grail of the sequence to sequence learning works.

Learning to use contextual embedding and implementing the concept of highways in the deeper network.

Pre-requisites

The code for this chapter can be found in the Ch4 folder at GitHub repository This chapter requires the following packages to be installed to execute the code:

Nltk

Pandas

Torch

Numpy

Matplotlib

Tqdm

Torchtext

You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r

This chapter uses *Ipython Notebook/ Jupyter Notebook* for easy execution and connecting thoughts with the implementation.

This chapter onward, we will use medium-size dataset and GPUs for processing. GPU is alternative computer hardware with hundreds to thousands of computing cores. GPU provides great speed, so multiple experiments can be carried out in a small amount of time. If you don't have GPU hardware, don't worry. You can use free GPUs available at Google Collaboratory (https://colab.research.google.com). Google Collaboratory provides free GPUs and TPUs to run experiments. You can simply clone this book's repository to Google Collab and start experimenting on the code provided.

Understanding Recurrent Units

In the previous chapter, we look at the implementation of GloVe and Word2Vec. Both utilize a feed-forward network to train the model, wherein there is no relationship mapping between the current and the previous input. Feed-forward network architecture is good but cannot be used to solve all kinds of tasks. This recipe is all about understanding the concept of RNN. You are already using applications with recurrent units in your daily life on your mobile. RNN powers the type-ahead feature in your keyboard. Most voice-assisted systems like *Google* and *Alexa* use some advanced form of RNN.

To illustrate the use of RNN, let me walk you through one of the examples. Let's say you want to find the sentiment of the given sentence using a feed-forward network. To do this, the logical steps to be followed are as listed:

Split the sentence into words (let's say we have 10 words)

Get dense vector for each word (let's say we have 100-dimensional vector)

Pass it to neural network

The nature of the problem is sequential, and passing each word to the feed-forward network will not perform as in FNN; there is no mapping of the relation between two words, and there is no mechanism by which it can give you sentiment based on the entire sentence. Passing the entire sentence instead of words seems to be a choice, but it's an expensive choice for 10 words, each word with a 100-dimensional vector that, when combined, make 10000 (10100) as the input feature size. If the second layer in the feed-forward network has half the perceptron, then the input (i.e.5000), the weight matrix between two will be of [10000, 5000] dimension. It is a huge matrix to train. To solve this problem and establish the relationship the between previous and current input, a different type of architecture is required, which is known as RNN.

Nowadays, RNN refers to the group of techniques, including Vanilla RNN, GRU, and LSTM. Vanilla RNN is the simplest form of RNN that suffers from many shortcomings. One of them is vanishing gradient, which we will discuss in detail in the upcoming topics. Nowadays, Vanilla RNN and RNN are used interchangeably. This recipe is most precisely about Vanilla RNN.

The RNN structure is derived from the FNN, but it has internal loops. The typical FNN with a single input, single output, and single hidden perceptron in the hidden layer can be visualized as given below. RNN has the concept of sequential or internal memory. It has a mechanism to store information related to previous outputs. The FNN with

recurrent component described earlier looks like the following diagram:

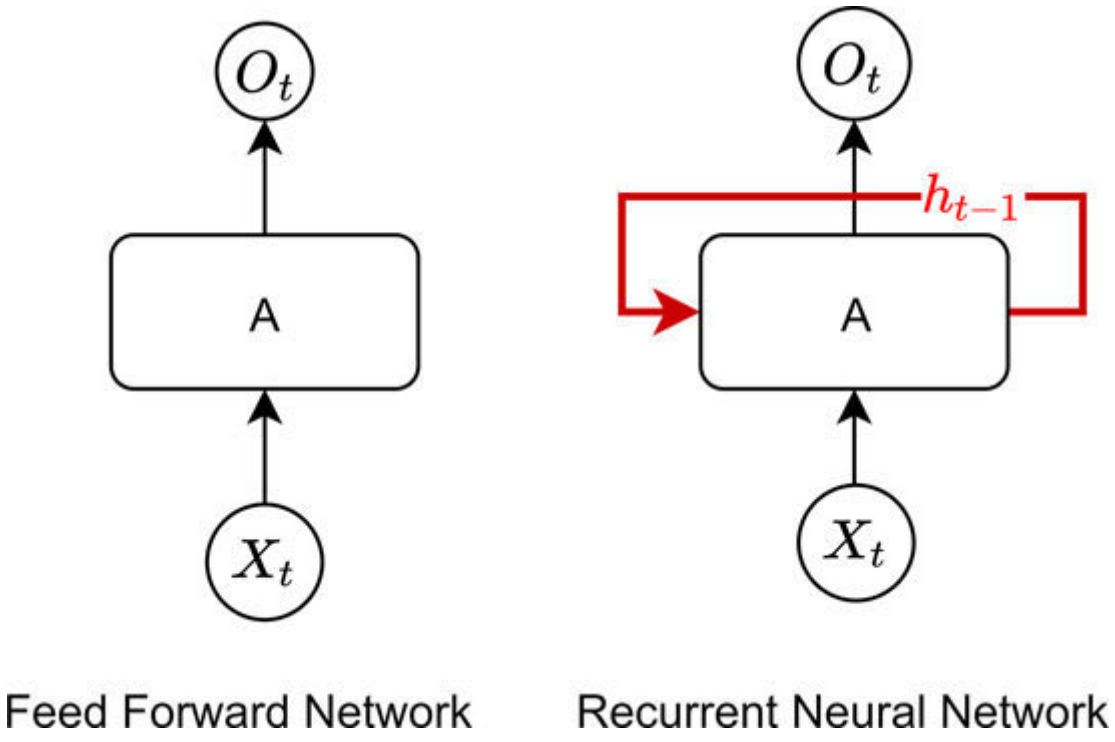


Figure 4.1: *RNN in the rolled form.*

As shown in the figure, the RNN has the looping mechanism to act as a highway between the previous and current information. This highway between the previous and the current information is known as the hidden state of the RNN. Moving ahead with our earlier example, let's say we have a movie review as *had a terrible* Let's see how RNN will help here. RNN kind of netwrorks will eventually help predict the sentiment of the review. A unit with the random hidden state is fed with the word the unit gives out output and hidden state. The updated hidden state so produces passed to the next unit. This hidden state is combined with previous learning and current learning. The word *'was'* again influences

this hidden state and gives out updated hidden state and output. At the end of all the words, the hidden state will hold crux of the feedback, and one can classify this sentence as positive or negative with it.

Rolling and Unrolling

To understand the RNN, *figure 4.1* shows that there are individual units for each word. The above-shown units are only one unit, but its unrolling is shown in the next figure for better understanding. In reality, it is only one unit that takes input and gives output and hidden state. The hidden state is also utilized in the next step, and this continues until the end of the sequence. The unrolling of the same network is shown as follows:

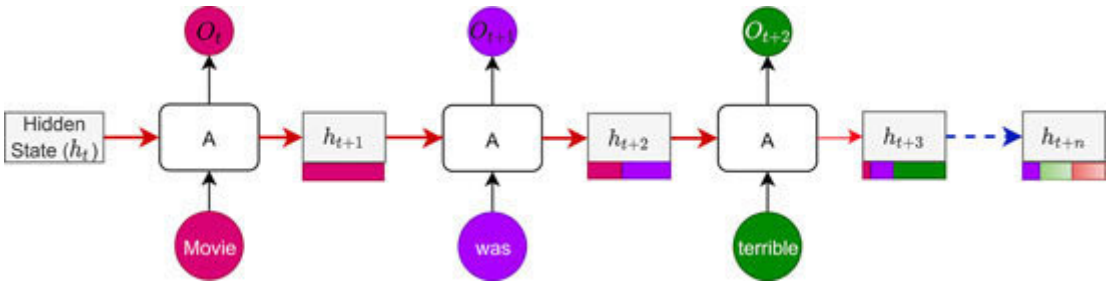


Figure 4.2: *RNN in an unrolled form.*

RNN unit unrolled from time *t* = *t* = *o* to time *t* = In RNN, each input is referred to as a time step. In our example, each word can be referred to as a time step. Time steps are generally given as *t* + 1, *t* + 2,..., *t* + *t* + 1, *t* + + Here, *n* is the total number of time-steps. In the forward pass, we move forward with each time step, and we perform *through* or BPTT in the backward pass. BPTT seems to be the fancy name for a method to backpropagate gradient while training

RNN. The comprehensive network with all the associated weights and hidden layers can be given as follows:

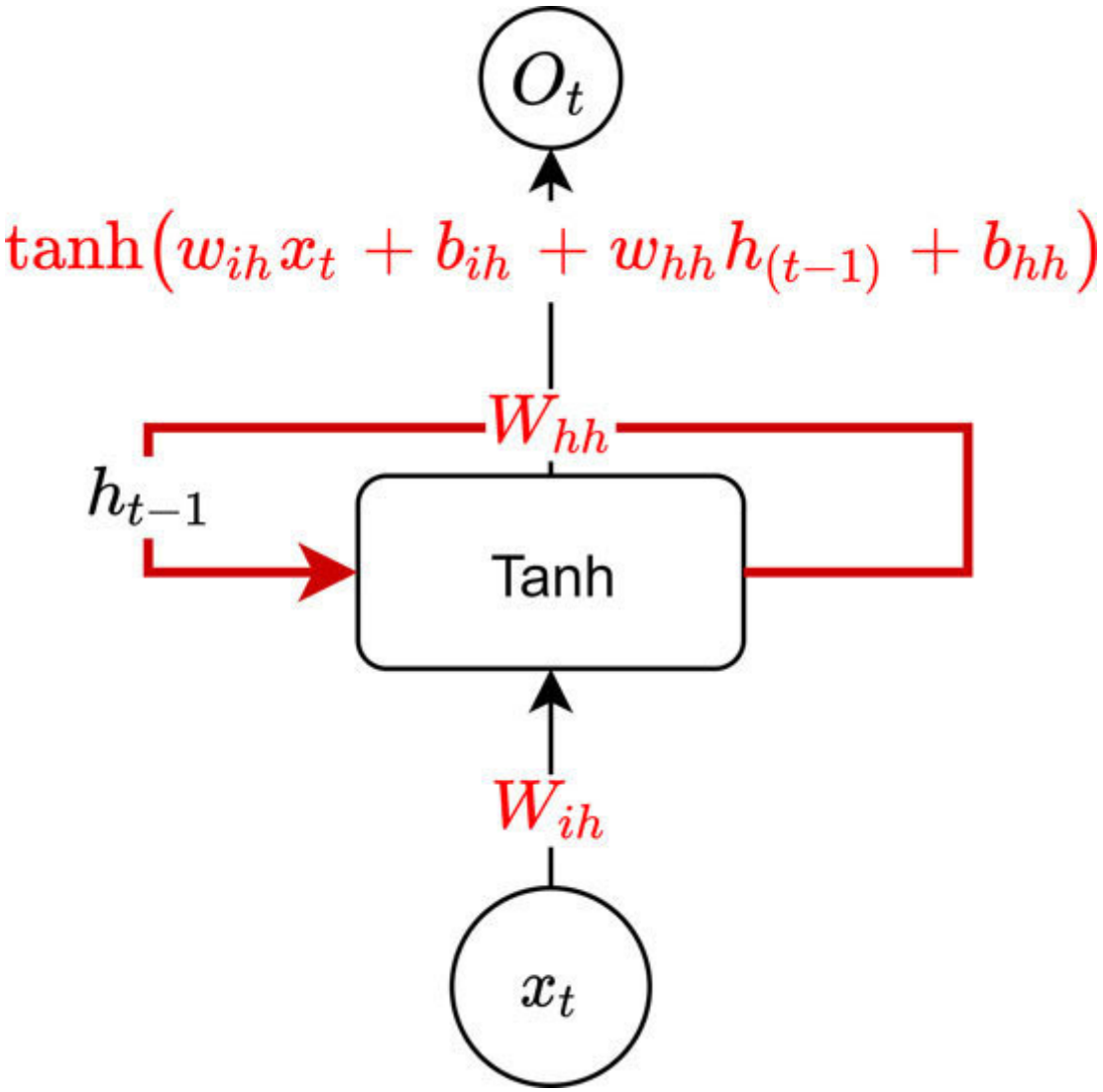


Figure 4.3: *Mathematical details regarding how RNN functions.*

The basic equation for RNN can be given as:

= + + _ +

Where is the hidden state at the time t, is the hidden state of the previous layer at a time or the initial hidden state at a time $t = 0$, and is the input at a time Also, is the weight from input to hidden, is the weight from previously hidden to current hidden, and and are bias to input and hidden, respectively.

Implementing RNN in PyTorch is very easy. PyTorch takes care of many abstractions related to implementing and initializing weights. The RNN function requires three parameters:

The number of expected features in input

The number of features in the hidden state

The number of recurrent layers. For example, num_layers=2 would mean stacking two RNNs in layers together to form a stacked RNN. The second layer of RNN derives input from the first layer.

The shape of the RNN input is [batch_size, max_time_steps, Let's say we are dealing with sentiment analysis tasks. To make the dimension of the input the same, we assume that one sentence can have a maximum of 20 words. Padding is applied if the sentence is smaller, and longer sentences are truncated to make the word size 20. Each word has 100-dimensional embeddings. Each batch has 64 sentences, so the resultant shape of input would be [64, 10, Here, 10 is our max time step. You can simply initialize an RNN, as

given below. The hidden unit shape in all the RNN units is [number of layers*directions, batch_size, Here, the number of layers represents how many layers of RNN are stacked on top of each other, and direction represents whether the RNN is unidirectional or bidirectional. Let's say I have bidirectional RNN with three layers having the same batch size and 100 dimensions; then, the hidden shape will be [2*3, 64,

```
import torch
rnn = torch.nn.RNN(10, 20, 2) input = torch.randn(5, 3, 10)
h0 = torch.randn(2, 3, 20) output, hn = rnn(input, h0)
>>>torch.Size([5, 3, 20]) #output
```

Consider the preceding diagram of hidden state; as the sequence elongates, the influence of early time steps decreases and eventually becomes zero. RNN is unable to remember longer sequences due to this problem. This phenomenon with RNN is known as **Short-Term** We will discuss the root cause of and the solution to this problem in the upcoming topics.

You can learn more on the fundamentals of the **Recurrent Neural Network** and **Long Short-Term memory** network at

Implementing the Concept of Embeddings

We generated embeddings based on words and characters in the previous chapter. The help of embeddings is not limited to just providing a numerical output of the token. Embedding is the way to transfer learning in natural language processing. In image processing, the concept of transfer learning is quite mature and is getting stronger for NLP. In this recipe, I will demonstrate how embeddings can help in training. We'll take an example to illustrate how transfer learning can help. Let's say Tom stays in the USA and is a native English speaker. He has got an internship opportunity in France and will be traveling after 3 months. Tom does not know French; he was smart, so he started listening to the French radio channel for 3 months. Although he could not understand anything initially, his brain started making sense of the language gradually. By the time he reached France, he had somewhat pre-trained his brain. Now, this low-level understanding of the French language helped him learn it faster. It is similar to how embeddings work. Embeddings are formed by forcing the network to learn from context. These learnings are passed to RNN Like network; which can learn that rather than learning from scratch.

As discussed in the recipe related to pre-trained embeddings are available and can be easily used in our model. I have found a module on GitHub that lets you easily download the

required embeddings. This package is known as *Chakin* and is a simple downloader for pre-trained word vectors. You can install Chakin using pip, and you can use it as follows:

```
import chakin

chakin.search(lang='English') # this will list all availanbleembeddigs
```

availanbleembeddigs availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs
availanbleembeddigs

Table 4.1

You can select any of the vectors to be used in your training pipeline:

```
chakin.download(number=2, save_dir='./') # select fastText(en)
```

From this chapter onward, we will use the popular regularization technique - Dropout. Most of today's deep learning implementations use dropout. As the name suggests, it drops the percentage of connection between two layers randomly. This can be diagrammatically shown as follows:

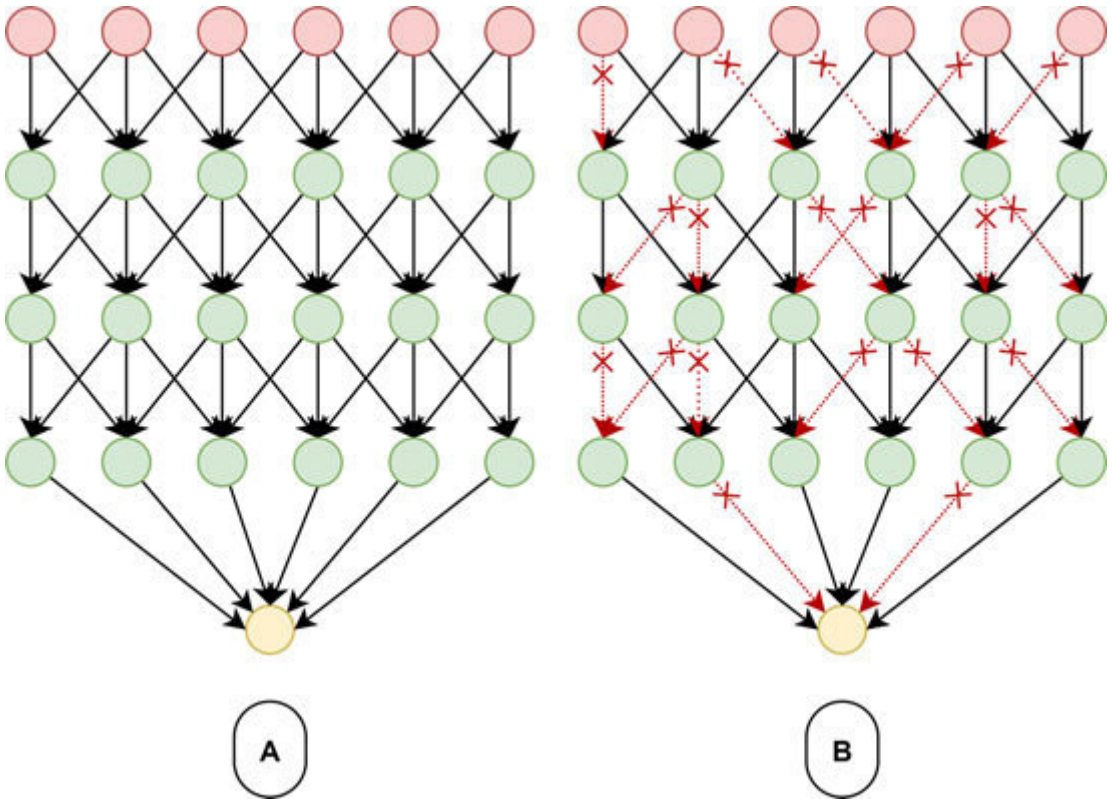


Figure 4.4: *A network diagram A. Without dropout B. With Dropout.*

As shown, all layers remain fully connected to the previous and next layers in the normal network. The dropout drops the connection randomly between these layers as per the defined percentage. If the dropout percentage/probability defined is 0.2, 20% of the connection will be dropped. This dropping of connection occurs at each iteration and induces variability in the graph. Due to this variability, the network does not rely on strong and biased features and tries to generalize based on weaker features as well. This is how dropout is helpful in better generalization.

Downloading Dataset

To demonstrate how embeddings can help, we will experiment with sentiment analysis tasks. I have used a movie review dataset with 5331 positive and 5331 negative processed sentences. The entire experiment is divided into five sections. The dataset is available at

Pre-processing

I am using *TorchText* to preprocess the downloaded data. Pre-processing includes following steps:

Reading and parsing data

Defining sentiment and label fields

Dividing data into train, valid, and test subsets

Downloading embedding

Forming the train, valid, and test iterators

Training

Training will be conducted for two models, one with no pre-trained embedding and one with *FastText* embeddings. I am using *FastText* embeddings trained on Wikipedia corpus for 1 billion words with a vector size of 300. The network with no pre-trained embeddings can be defined as follows:

```
class SCRATCH_RNN(nn.Module):
def __init__(self, vocab_size, embedding_dim, hidden_dim,
output_dim, n_layers, bidirectional, dropout, sentiment_vocab):
super(SCRATCH_RNN, self).__init__()
self.embedding = nn.Embedding(vocab_size, embedding_dim)
self.rnn = nn.RNN(embedding_dim, hidden_dim,
num_layers=n_layers, birectional=bidirectional, dropout=dropout)
self.fc = nn.Linear(hidden_dim * 2, output_dim)
self.dropout = nn.Dropout(dropout)
def forward(self, x):
embedded = self.dropout(self.embedding(x))
output, hidden = self.rnn(embedded)
# concat the final forward (hidden[-2,:,:]) and backward
(hidden[-1,:,:]) hidden layers
# and apply dropout
hidden = self.dropout(torch.cat((hidden[-2, :, :], hidden[-1, :, :]),
dim=1))
return torch.softmax(self.fc(hidden.squeeze(0)),dim = 1)
```

The SCRATCH_RNN class builds embeddings from scratch using the torch embedding function. The Embedding function is very frequently used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings. Parameters of the embeddings functions are trainable, so the weights change constantly throughout training and help generate better word vectors. Such embedding vectors are passed to the RNN function to get hidden and output tensor. The hidden tensor has the crux of the learning, so the hidden output is passed through a linear transformation. After the application of softmax, the predicted output is calculated.

The other network is one where we are passing pre-trained embeddings. This network looks like the previous one, except for the change in one line, as indicated in bold:

```
class FT_RNN(nn.Module):
def __init__(self, vocab_size, embedding_dim, hidden_dim,
output_dim, n_layers, bidirectional, dropout, sentiment_vocab):
super(GLOVE_RNN, self).__init__()
self.embedding = nn.Embedding(vocab_size, 300)
self.embedding.weight.data.copy_(sentiment_vocab.vectors)
self.embedding.weight.requires_grad = True
self.rnn = nn.RNN(embedding_dim, hidden_dim,
num_layers=n_layers, bidirectional=bidirectional,
dropout=dropout)
self.fc = nn.Linear(hidden_dim * 2, output_dim)
self.dropout = nn.Dropout(dropout)
def forward(self, x)
```

```
embedded = self.dropout(self.embedding(x))
output, hidden = self.rnn(embedded)
# concat the final forward (hidden[-2,:,:]) and backward
(hidden[-1,:,:]) hidden layers

# and apply dropout
hidden = self.dropout(torch.cat((hidden[-2, :, :], hidden[-1, :, :]),
dim=1))
return self.fc(hidden.squeeze(0))
```

The changed line (as highlighted above) copies pre-trained vectors from the loaded *FastText* model vectors. So, in the FT_RNN class, we are not training the embeddings from scratch. I allowed to train this network for 100 epochs and plotted the accuracy progress of both models. The plot is as follows:

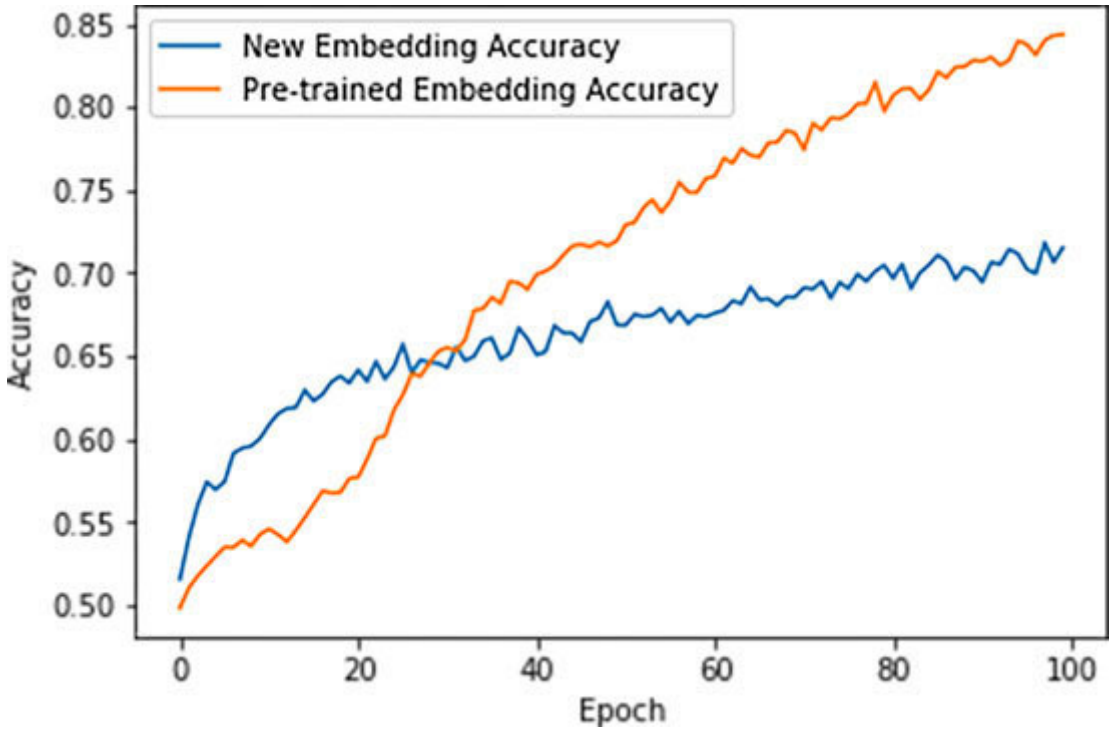


Figure 4.5: *Difference in accuracy when embeddings are trained from scratch and when pretrained FastText embeddings are used.*

It is very clear that pre-trained embeddings help in learning. Training from scratch resulted in ~70% accuracy, whereas training using *FastText* embeddings provided 88% accuracy.

In the preceding network, I am reporting accuracy only on the train data. The entire code has both tests, and validation partitions are available. As part of the exercise, you can try with the following changes in the network to reinforce your learning:

Try with different embeddings and note the change in accuracy/ loss.

Try with different datasets to see whether the trend differs by the change in the dataset (different datasets are provided along with *TorchText* in

Check the accuracy by iterations on validation and test data

Refer to the following reference links:

A simple way to prevent neural networks from overfitting: https://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf

downloader for pre-trained word vectors: *https://github.Com/chakki-Works/chakin*

Movie review *www.cs.cornell.edu/people/pabo/movie-Review-Data/*

Understanding Advance RNN Units

In the previous chapter, we discussed basic RNN or Elman RNN or vanilla RNN units. Vanilla RNN units have some shortcomings, so we will discuss some of the advanced RNN units, like LSTM and **Gated Recurrent Units** in this chapter.

The following image looks familiar; it is the hidden tensor representation we have used in recipe 1 while understanding RNN units:

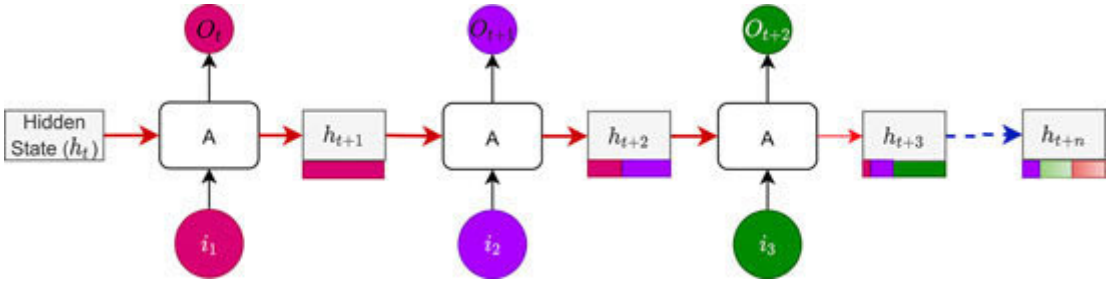


Figure 4.6: *An Illustration of the vanishing gradient problem.*

The color bar below the hidden state indicates the information content of the hidden state. As the hidden state sees more and more tokens, it is influenced by all and stores information about those tokens. One more thing to see is that as the hidden state moves forward, it tends to forget information about the token seen in the previous time steps. This hidden state has a problem. As you can see, as the sequence elongates, the contribution of the very first time

step input becomes zero or deleted. When BPTT is applied to such networks, the gradient for those early time steps is not calculated, and learning to the weights is not imparted. If I show you the same thing in the unrolled version of RNN, the gradient decreases while backpropagation and eventually vanishes.

To solve this problem, *Jürgen Schmidhuber* coworker in 1997 proposed LSTM. In this chapter, we will discuss LSTM in detail.

Gating Mechanism in LSTM

LSTM units have a very intuitive structure. It has two internal states, whereas vanilla RNN has only one hidden state. The cell state in LSTM is like a conveyor belt that runs on the top of the unit, as shown in the following diagram. The cell state is highly regulated by the gates attached to it, and these gates are the way to let the information through. LSTM has three gates to control the information flow.

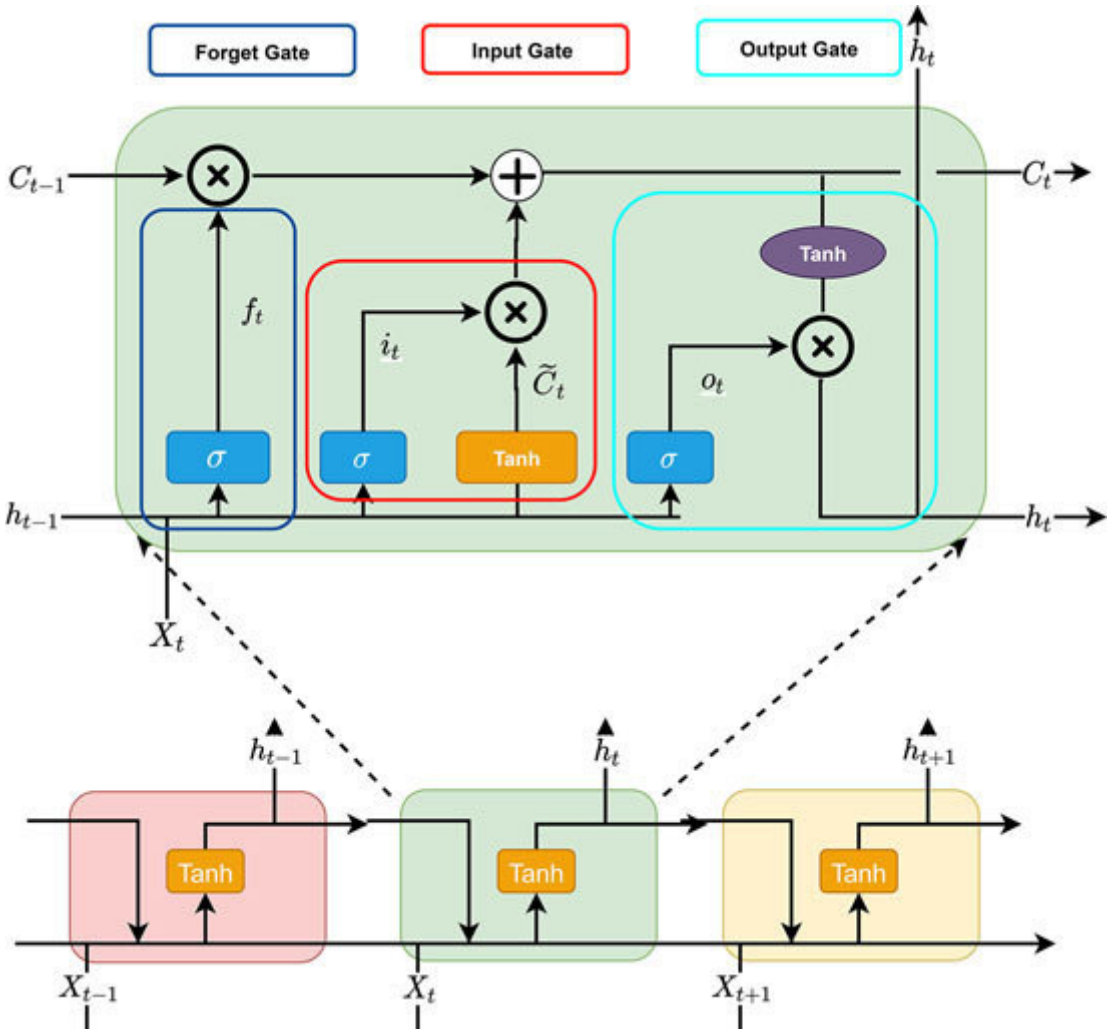


Figure 4.7: *The various gates present in LSTM.*

Forget It regulates the information flow. A sigmoid gate looks at the input and previous hidden state The sigmoid output value of 1 means let everything go through, and 0 means nothing should get through:

$$= +$$

To keep or not is gradually learned by the weights and bias attached to the forget gate.

Input Next is the input gate that decides what information we will keep in the cell state. The input gate has two inputs: one is controlled by sigmoid, and another is controlled by The following equations define the input gate:

$$= \cdot +$$

$$C\ t = \cdot +$$

Output It decides what information to let through according to the cell state and hidden state. A sigmoid gate decides what information from the hidden state goes to the output, and tanh decides what information from the cell state goes to the output gate. The output gate can be mathematically represented as follows:

$$= \bullet +$$

$$= *$$

The information controlled by this gate then merges into the cell state, as shown in the following equation:

$$= \circ + \circ$$

LSTM can be easily implemented using PyTorch. PyTorch has an LSTM function, which takes a similar input shape as described in the case of vanilla RNN. It can be used as follows:

```
rnn = nn.LSTM(15, 25, 2)
input = torch.randn(5, 3, 15) h0 = torch.randn(2, 3, 25)
c0 = torch.randn(2, 3, 25)
output, (hn, cn) = rnn(input, (h0, c0))
```

The cell's state and hidden state are taken along with the input by LSTM, and these two will always be of the same shape.

In theory, LSTM seems to be a jaguar; let's test it on the ground. Here, to prove the effectiveness of the LSTM, I am using the same dataset, pre-processing the pipeline, Vanilla

RNN, as in the previous recipe. The only thing new is the use of LSTM. The modified function looks like the following:

```
class LSTM_RNN(nn.Module):
def __init__(self, vocab_size, embedding_dim, hidden_dim,
output_dim, n_layers, bidirectional, dropout, sentiment_vocab):
super(LSTM_RNN, self).__init__()
self.embedding = nn.Embedding(vocab_size, embedding_dim)
self.rnn = nn.LSTM(embedding_dim, hidden_dim,
num_layers=n_layers, bidirectional=bidirectional,
dropout=dropout)
self.fc = nn.Linear(hidden_dim * 2, output_dim)
self.dropout = nn.Dropout(dropout)
def forward(self, x):
embedded = self.dropout(self.embedding(x))
output, (hidden, cell)= self.rnn(embedded)
# concat the final forward (hidden[-2,:,:]) and backward
(hidden[-1,:,:]) hidden layers
# and apply dropout
hidden = self.dropout(torch.cat((hidden[-2, :, :], hidden[-1, :, :]),
dim=1))
return self.fc(hidden.squeeze(0))
```

When the sentiment analysis test was run for 100 epochs, I found LSTM's performance recommendable.

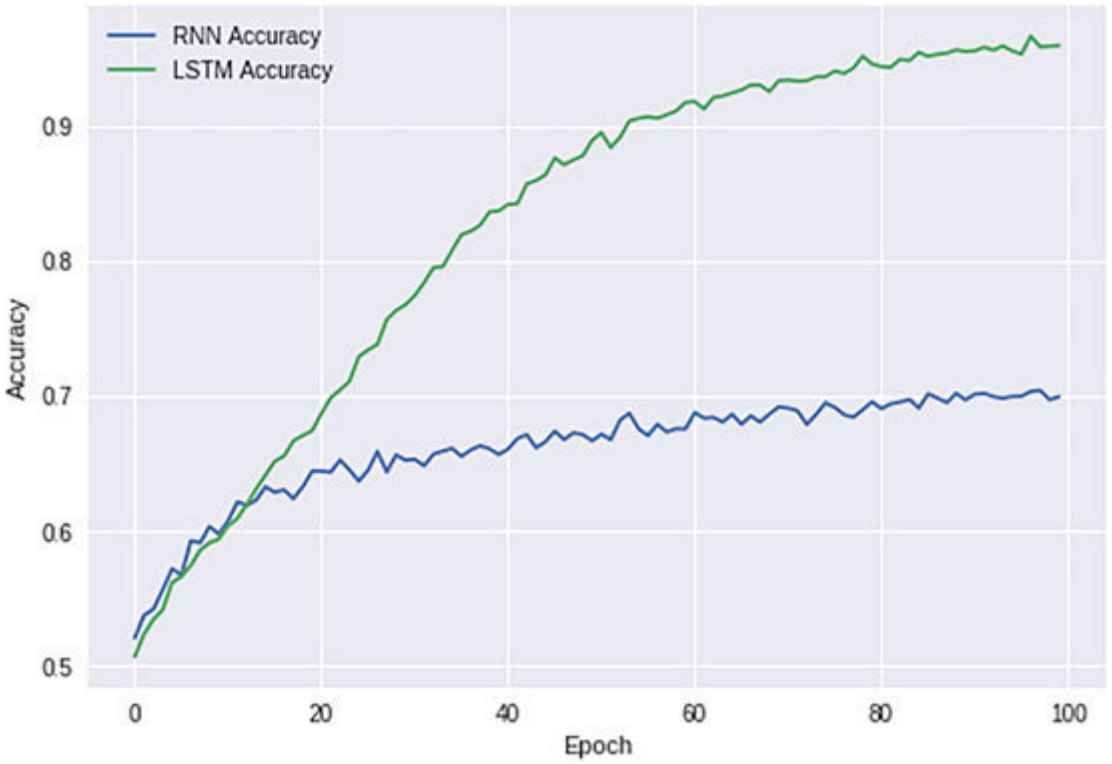
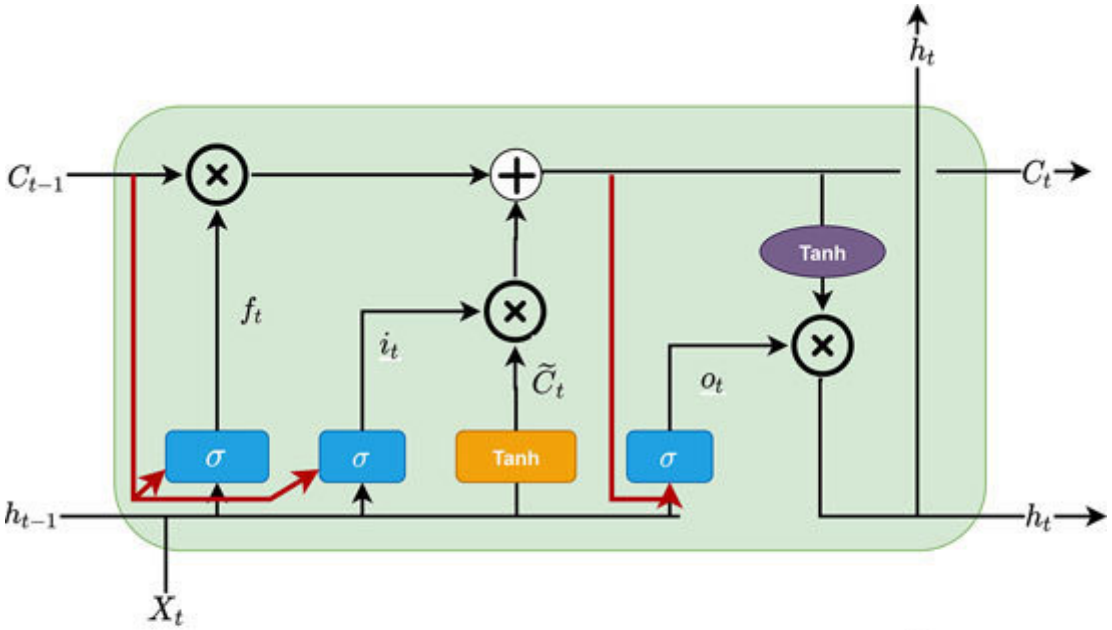


Figure 4.8: *Difference between accuracy when LSTM and RNN are used for text classification.*

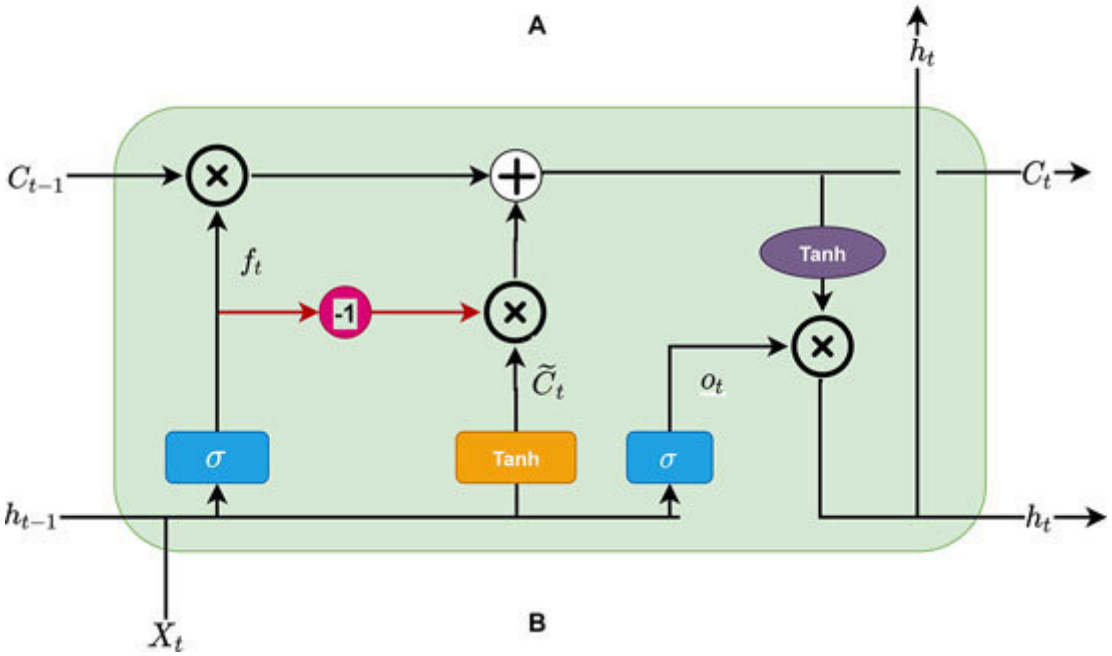
The accuracy of train data was over 95% with LSTM and was around 70% with RNN. The complete implementation with supporting code is given at

Modified LSTM Units

After the success of LSTM, many researchers came up with its variants. Here, we will briefly discuss two of the variants, as shown in the following diagrams:



A



B

Figure 4.9: *LSTM variants designed by making extra connections between various gates.*

Figure A was designed by *Gers &Schmidhuber (2000)* and has a peephole whereby the sigmoid layer can see the cell state. In figure B, a connection is added between the forgetting and input gate. The intuition is to let them forget, and input decides what to let go and what to add. There are several such variations, and many perform great in some tasks. Discussing the variants may provide an additional intuition and broaden your vision about LSTM. In the preceding experiment, I am reporting accuracy on the train data only. The entire code has both tests, and validation partitions are available. As part of the exercise, you can try making the following changes in the network to reinforce your learning:

Try with different datasets to see if the trend differs by a change in the dataset (different datasets are provided along with *torchtext* in

Check the accuracy by iterations on validation and test data

Take a look at the following links for more details:

Recurrent nets that time and count:
ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf

Depth-gated recurrent neural networks: https://arxiv.org/pdf/1508.03790v2.pdf

Understanding and Implementing GRU

In the experimentation carried out in the previous recipe, the gating mechanism can be a solution to achieve state-of-the-art results and alleviate the problem of vanishing gradients. **Gated Recurrent Units** are also inspired by the design of LSTM. Gated recurrent units were published in 2014 by *Cho et* in a research paper named **Learning Phrase Representations** using RNN Encoder-Decoder for statistical machine translation.

As we had gated in LSTM, GRU has two gates: update and reset gate. These two gates decide what information should be discarded and what information should be passed through. Learnable parameters in these two gates can be trained to change the information content and make continuous updates. The flow diagram for GRU looks like this:

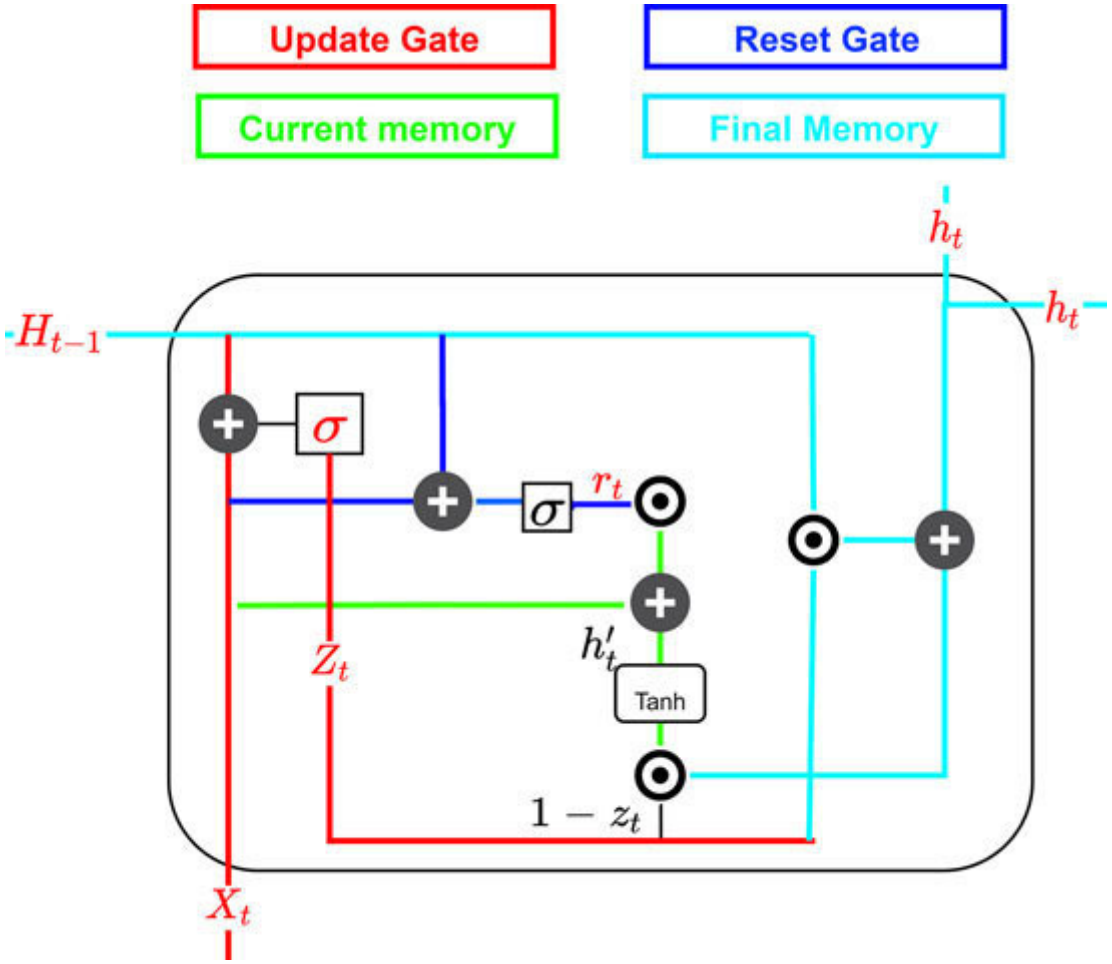


Figure 4.10: *Detailed structure of the GRU unit.*

The four gates functions are as follows:

Update This gate can be given by the following formula:

= +

Here, the input is multiplied by its weight, and the previously hidden tensor that carries the information of the previous *t* - 1 is multiplied by its weight. Then, sigmoid squashes them into a number between 1 and 0. The update gate determines

how much past information to let go of the present time step. This gate helps solve the vanishing gradient problem. If the value of the sigmoid gate is 1, all the information is preserved to solve the vanishing gradient problem.

Reset This gate helps determine how much information must be forgotten from the previous time steps:

$$= +$$

This equation seems to be very similar to the previous one, and the only difference is that the weights are for the reset gate. Next is to use these gates to determine the current memory content and final memory at the end of the output.

Current memory This derives the current memory content using the reset gate value and the current input value. As discussed earlier, the reset gate knows how much information to forget and has a number between 0 and 1. If is zero, the input information contained in the current time step will be completely ignored, and if it is one, the information in the current input will be taken into cell state. The current memory content is calculated as shown here:

Taking the *Hadamard* product of reset gate value and the previous hidden state with its weight

Summing up the value with of

$$= + \Theta$$

Final memory at the current time Final memory is constructed by taking the help of the update gates result and the current memory content. The last memory is formed using the following steps:

Taking *Hadamard* product of the update gate value and

Taking *Hadamard* product of 1 – and the current memory content

Summing up these two values:

$$= \Theta + (1 - \Theta$$

GRU with PyTorch

Now, it's time to implement GRU using PyTorch. The usage is very similar to LSTM and vanilla RNN. The GRU function takes three arguments:

The number of expected features in the input x.

The number of features in the hidden state h.

The number of recurrent layers. For example, setting num_layers=2 would mean stacking two GRUs together to form a stacked GRU, with the second GRU taking in outputs of the first and computing the final results. Default: 1

```
rnn = nn.GRU(10, 20, 2) input = torch.randn(5, 3, 10)
h0 = torch.randn(2, 3, 20)
output, hn = rnn(input, h0)
```

One thing to note here is that the GRU uses only one hidden state to deal with the vanishing gradient problem, whereas the LSTM uses two hidden states. As a result, GRU is a bit faster than LSTM. Let's look at their performance on the movie review dataset. The complete implementation to compare the performance of GRU and LSTM is provided at

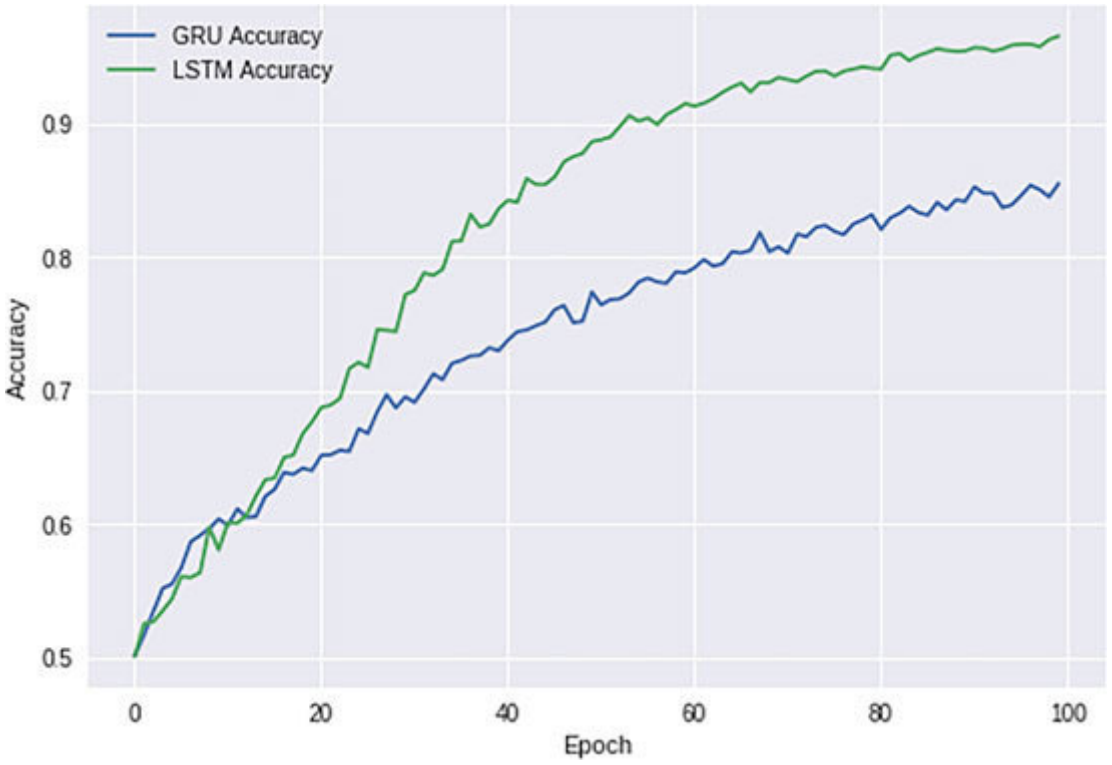


Figure 4.11: *Difference between the accuracy of LSTM and GRU on the text classification task.*

As shown, LSTM produces 95% accuracy, and GRU produces 85% performance. However, this wasn't the case for all datasets; GRU's performance was also found to be superior in some cases.

In the preceding experiment, I am reporting accuracy on the train data only. The entire code has both tests, and validation partitions are available. As part of the exercise, you can try making the following changes in the network to reinforce your learning:

Try with different embeddings to see if any significant difference is noted

Try with different datasets to see whether the trend differs by the change in the dataset (different datasets are provided along with torchtext in

Check the accuracy by iterations on validation and test data

Take a look at the following references:

Learning phrase representations using RNN encoder-decoder for statistical machine translation: *https://arxiv.Org/pdf/1406.1078v3.Pdf*

Empirical evaluation of gated recurrent neural networks on sequence modeling: *https://arxiv.Org/pdf/1412.3555v1.Pdf*

Understanding the Sequence to Sequence Model

The sequence to sequence model is the most researched area in the era of modern NLP. It is a one-size-fits-all kind of algorithm used in all the following tasks:

Machine translation

Summarization

Question answering

Chat-bot

Text simplification

Speech to text

Text to speech

In this recipe, we will start with general sequence to sequence architecture. One by one, we will implement GRU encoder-decoder, batching with encoder-decoder, and batch processing with attention. We will discuss this implementation by tightly integrating algorithms, equations, and code. The

discussion in the upcoming recipe forms the base of modern NLP techniques. To give you an idea of how to do text processing from scratch, I am not using *TorchText* in these topics. The sequence to sequence network contains two parts: an encoder and a decoder. The encoder takes the given input and converts it into fixed-size hidden representation. This hidden representation is converted to output by a decoder. An encoder takes input sequence and passes it through the *Embedding* which converts words into a fixed-size vector. This Embedding Layer is trainable and is trained along with the encoder and decoder.

Before going into details about the algorithm, we must know a few operations to help us understand the sequence to sequence block easily.

Unsqueeze inserts singleton dim at the position given as a parameter. Insert a new axis that will appear at the axis position in the expanded array shape:

```
input = torch.Tensor(2, 4, 3) # input: 2 x 4 x 3
print(input.unsqueeze(0).size())
>>>torch.Size([1, 2, 4, 3])
```

This returns a tensor with all the dimensions of input of size one removed. For example, if the input is of shape: (A×1×B×C×1×D), the out tensor will be of shape: (A×B×C×D). This operation is often used to reshape the tensor to fit the required input size or facilitate matrix operations:

```
x = torch.zeros(2, 1, 2, 1, 2)
y = torch.squeeze(x)
y.size()
>>>torch.Size([2, 2, 2])
```

Used to** **Interchange different axes of the tensor:

```
x = torch.randn(2, 3, 5)
torch.Size([2, 3, 5])
print(x.permute(2, 0, 1).size())
>>>torch.Size([5, 2, 3])
```

Log Soft-max applies logarithm after soft-max

Returns the top k-largest elements of the given input tensor along a given dimension. If dim is not given, the last dimension of the input is chosen. By default, largest is *True,* but the *k* smallest elements are returned if it is A list of (values, indices) is returned, where the indices are those of the elements in the original input tensor:

```
x = torch.arange(1., 10.) #tensor([1., 2., 3., 4., 5.])
```

```
top_value, top_index = torch.topk(x, 3)
print(top_value, top_index)
>>> tensor([9., 8., 7.]) tensor([8, 7, 6])
```

For the sake of simplicity, I am using GRU in this recipe. The sequence to sequence architecture has three essential

components: an encoder, a decoder, and a context vector. The following is a detailed diagram of the sequence to sequence architecture:

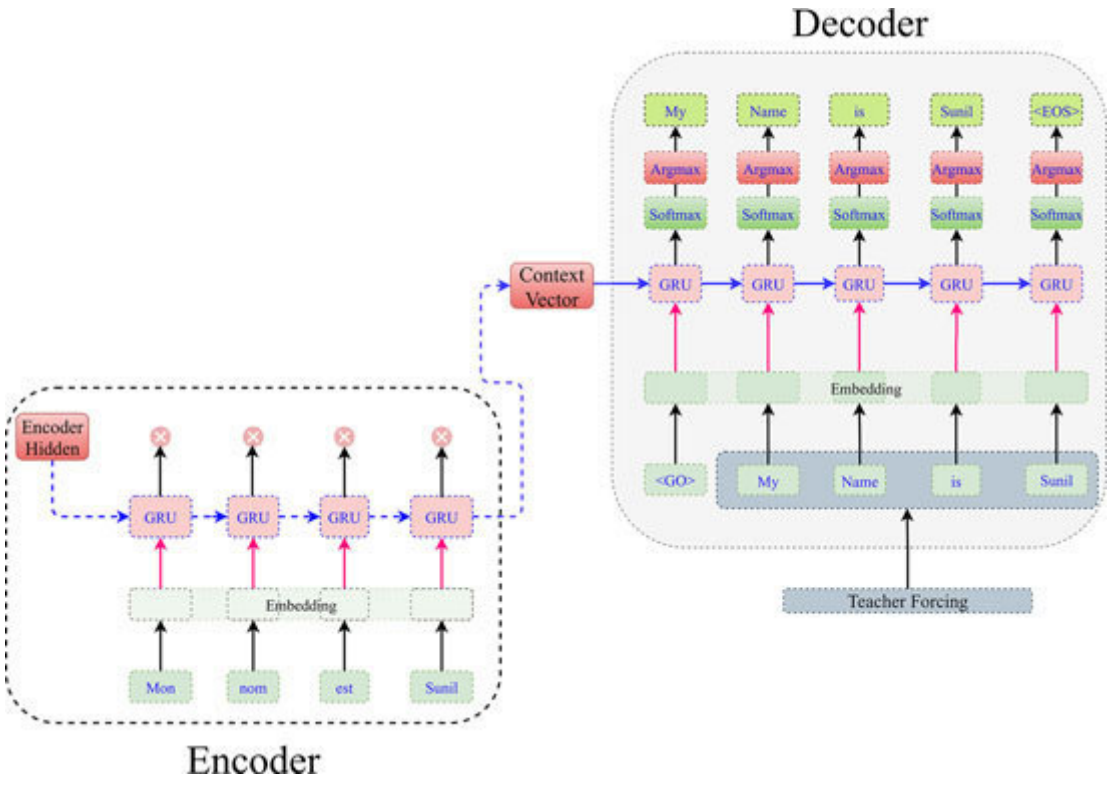


Figure 4.12: *Encoder-decoder architecture while training,*

For the sake of simplicity, we will take two sentences: one in French and another in English. The task is to convert the French sentence/source sentence to the English sentence/target sentence:

French sentence/ source sentence - *mon nom est Sunil*

English sentence/target sentence - *my name is Sunil*

As shown in the preceding diagram, each word is in the form of embedded representation and passed to GRU. GRU also takes a first encoder hidden state as the input. All words of the input sentence are passed sequentially to GRU so that it takes a word and a hidden state of the previous state as input and gives one output and updated hidden state. For now, I will ignore the encoder outputs shown by the symbol.

Encoder and decoder hidden states are represented by the dotted blue arrow throughout. The embedding representation formed from the input is shown in pink color.

After the encoder phase has finished, we have a final encoder hidden state as output. This final encoder hidden state is of value, and it is estimated that as it has seen all the words of the sequence, it will have information regarding the entire sequence. This encoder hidden state is passed to decoder as the first hidden state. This final hidden state is also known as the context vector, which is then passed to the decoder. Usually, the decoder does not know anything initially, but vector provides it with lots of information about the source sequence by a context. The first unit of the decoder is awakened with the or token as input and a context vector. Then, the decoder is trained with teacher forcing. Teacher Forcing is mostly used to train language models, and sequence to sequence task is a kind of language model. Teacher Forcing is a kind of hint in training recurrent neural networks that use the model output from a previous time step as an input to the next time step. For example:

Input sentence for the encoder is a French sentence - *nom est*

The actual target sentence in English - *name is*

In decoding, at first-time step, the token is given as input to GRU along with the context vector. or mark the beginning of decoding, and it will generate first output word in English looking at context vector. Then, GRU will generate

Now, in the second-time be given as input along with the previous time step hidden step. This will give an output as

This generation will continue as long as the token is encountered.

At training time, we know the output sequence for the input sequence, so we provide words of actual output for teacher forcing. At test time, we don't know the output sequence for the input sequence, so the output token generated at time step *t* is given as input to *t+1* step, as shown in the following diagram:

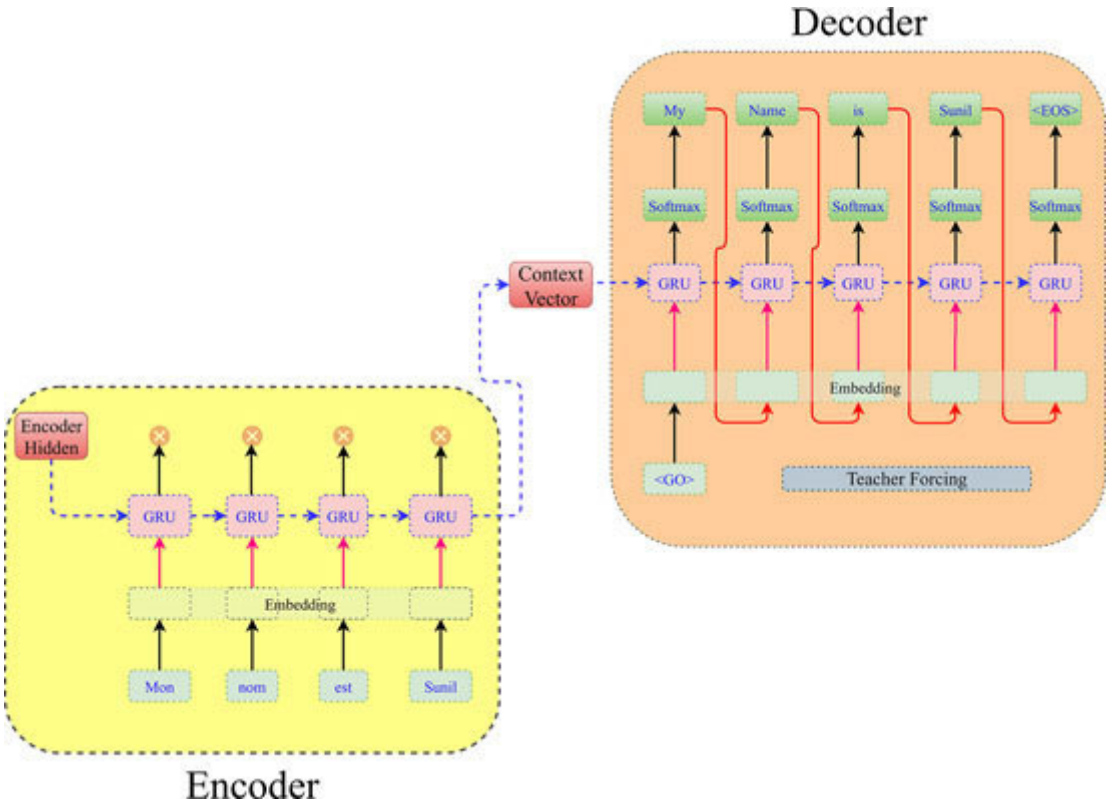


Figure 4.13: *Encoder-Decoder architecture during evaluation/testing.*

Implementing Sequence Encoder/Decoder

The data for this recipe is a set of many thousands of French to English translation pairs and can be downloaded from This dataset is already present in the data directory of Ch4 as To avoid distraction, I will only be converting the essential components in the recipe, and the rest of the code with all required functions to fully execute it can be found at

Encoder

As discussed, the encoder has the following components. Embedding converts our words into dense vectors, and GRU takes these embeddings and gives out hidden and output vectors at each time-step. This hidden and output tensor is returned for any given time step:

```
class EncoderRNN(nn.Module):
def __init__(self, input_size, hidden_size, n_layers=1):
super(EncoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
# embed size = hidden size for simplicity
self.embedding = nn.Embedding(input_size, hidden_size)
self.gru = nn.GRU(hidden_size, hidden_size)
def forward(self, input, hidden):
input = input.unsqueeze(1)
embedded = self.embedding(input) # batch, hidden
output = embedded.permute(1, 0, 2)
output, hidden = self.gru(output, hidden)
return output, hidden
```

Decoder

The decoder has a structure similar to that of an encoder. It has an embedding layer that converts the input to dense vectors, which are then passed to the GRU layer. The first-time step of the decoder receives a context vector as the hidden state vector. A linear transformation is applied that converts the output from GRU. This linear transformation converts the GRU output into sizes equal to the output/target language vocabulary size. A soft-max is applied to this layer, and the values are converted to probabilities. The highest probability for any word indicates that a particular word is being generated as the output by the decoder:

```
class DecoderRNN(nn.Module):
def __init__(self, hidden_size, output_size, n_layers=1):
super(DecoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
#Embeding size = hidden size for simplicity
self.embedding = nn.Embedding(output_size, hidden_size)
self.gru = nn.GRU(hidden_size, hidden_size)
self.out = nn.Linear(hidden_size, output_size)
self.softmax = nn.LogSoftmax()
def forward(self, input, hidden):
output = self.embedding(input) # batch, 1, hidden
output = output.permute(1, 0, 2) # 1, batch, hidden
```

```
output = F.relu(output)
output, hidden = self.gru(output, hidden)
output = self.softmax(self.out(output[0]))
return output, hidden
```

Actual Training

Training includes joining all the bits and pieces we developed earlier, starting with making an object for the encoder and decoder, as follows:

```
encoder = EncoderRNN(input_size, hidden_size)
decoder = DecoderRNN(hidden_size, output_size, n_layers=2)
```

Training has the following components involved sequentially:

Collect all trainable components for encoder and decoder and provide all these components to the optimizer.

Define the loss function.

Use our data loader to get source and target sentences.

Initialize encoder hidden state.

Run through all the tokens in the source sentence and generate the final encoder hidden state.

Pass the final encoder hidden state to the decoder along with the start of sequence token as input.

Run the decoder and allowing it to generate a token using teacher forcing.

When the token is encountered, stop break the decoding loop.

Calculate loss by comparing the generated output to the actual output.

Back-propagate the loss and change the weights.

Optionally track loss to visualize whether the network is converging.

Repeat steps 3 to 10 till all the sentence pairs are processed in each epoch.

Run this training for n epochs and evaluate.

The following diagram shows that the training loss decreases as training progresses:

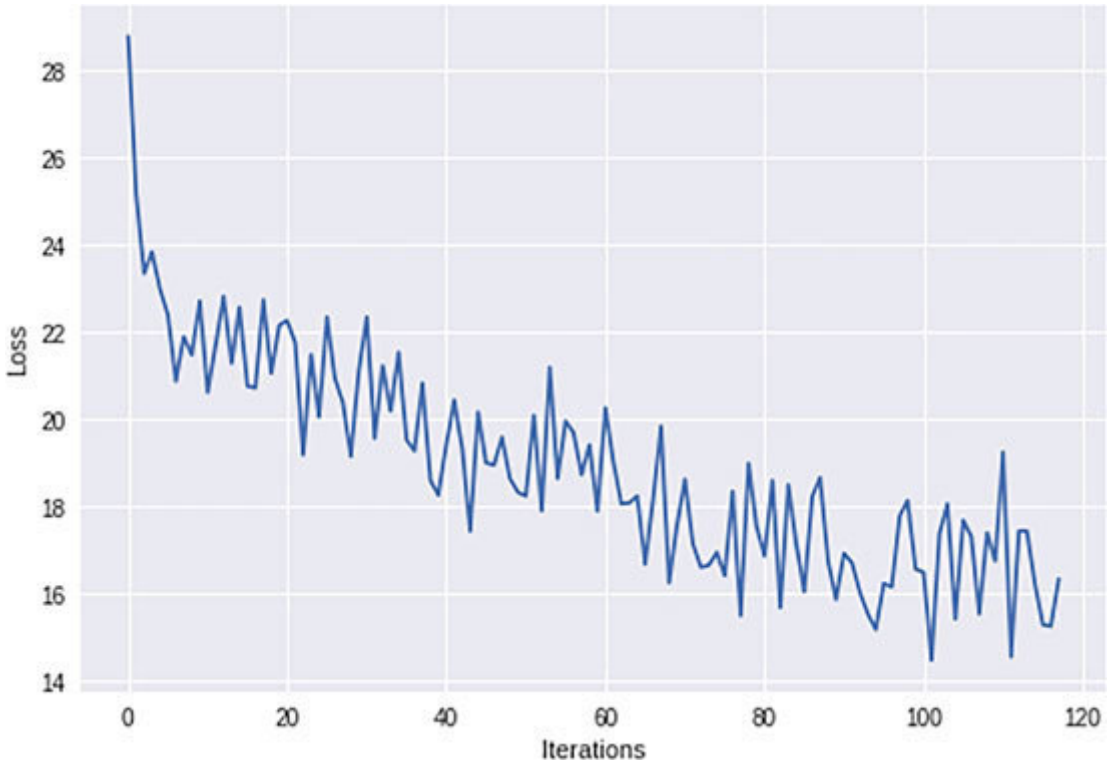


Figure 4.14: *Decrease in error noted when a network is trained for the language translation task.*

Evaluation

The evaluation module follows the same flow as training; it takes the pre-trained encoder and decoder and generates translated text. The main thing here is that teacher forcing with the token generated in the previous time step is given to the next time step. The decoder generates a word at each time step, and the same word is fed as input to the next time step. At test time, we don't know the output sequence for the input sequence, so the output token generated at time step *t* is given as input to the *t+1* step. The following output is generated at the end of evaluation attempt:

attempt: attempt:
attempt: attempt: attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt: attempt:

attempt: attempt: attempt: attempt:

attempt: attempt: attempt: attempt: attempt: attempt:
attempt: attempt: attempt: attempt:

Table 4.2

Try to improve this translation by making a few of the following variations:

With TorchText, use the function to provide the translation dataset. Use this dataset in the previously built model to see whether the model still works and produces a good translation.

Extending one more step in the above pipeline, use pre-trained GloVe or any other embedding to check whether the result improves.

Refer to the following links for more details:

For details regarding unsqueeze, squeeze, logsoftmax, permute, topk: *https://PyTorch.Org/docs/stable/tensors.html*

Professor A new algorithm for training recurrent networks: *https://arxiv.Org/abs/1610.09038*

A learning algorithm for continually running fully recurrent neural networks: *http://citeseerx.Ist.Psu.Edu/viewdoc/download?Doi=10.1.1.52.9724&rep=rep1&type=pdf*

Understanding Batching with Seq2Seq

In the previous recipe, we built a basic model for language translation. One thing to observe is that we have been using CPU for translation in the previous recipe. In the previous recipe, we only took one sentence at a time and translated it. What if we can take many sentences at a time for training? We can decrease the training time. GPU is a powerful device with thousands of computing cores. If we use GPU by translating one sentence at a time, we waste a lot of computing resources by not using them fully. In this recipe, the main task to learn is how to use batches of the source and target sentence for the translation, instead of a single sentence. Another thing to learn is how to use GPU to make our training process faster.

Let's say we want to build a When given a sentence with 26 words, it will convert it to a 52-word sentence. In earlier examples, we used batch *size = 1,* so only one example was processed at a time. It was easy to implement but not efficient, so we must convert it to process *n* sentences together to increase efficiency. This way, we will use roughly *n* processors parallelly and process roughly *n* times faster as well.

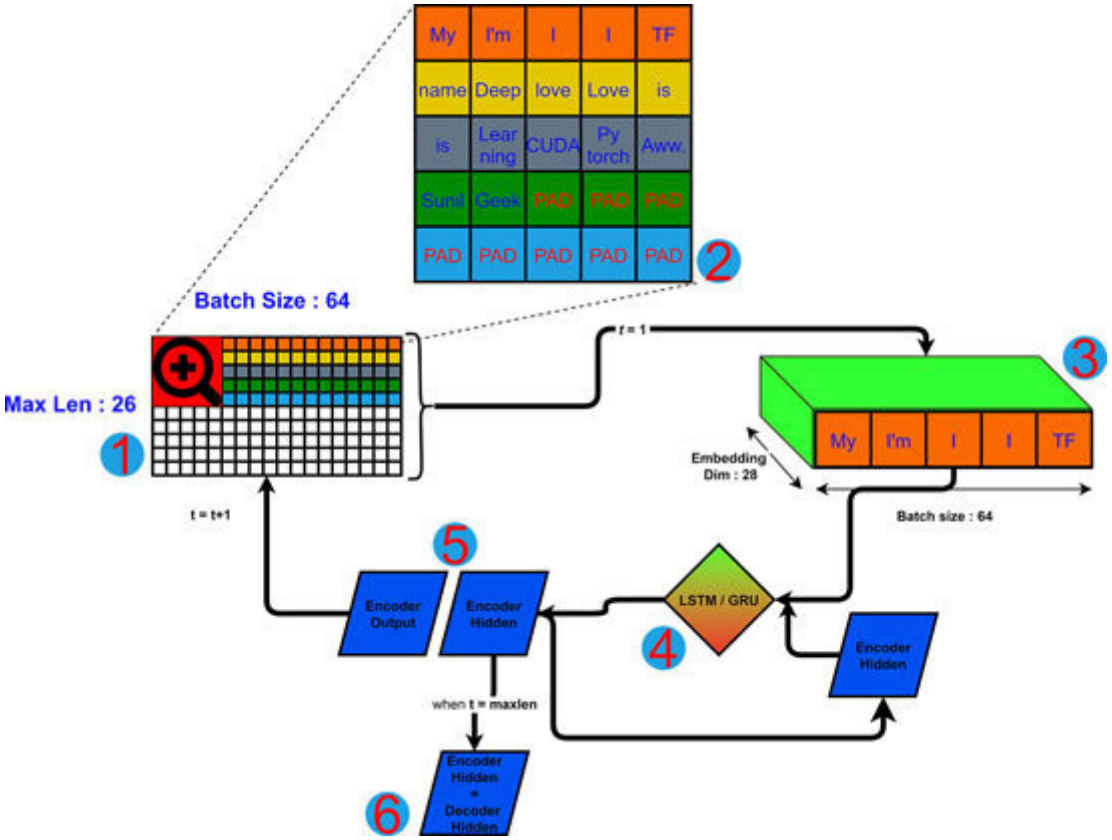


Figure 4.15: *An illustration of how batching works with sequence to sequence - an encoder phase.*

The image illustrates an example where *(1)* shows that 64 sentences with a fixed length of 26 words are processed in a single batch. In five example sentences are shown, such as My name is I'm Deep Learning I love I love TF is Aww. Each sentence is transposed to process them in a batch, and each sentence is padded with *PAD* token to make the length equal to 26, as shown in At each iteration, 64 elements are taken row-wise, and 28-dimensional embedding is calculated for each word, as shown in Then, 64 such elements are processed with LSTM/GRU, as shown in LSTM/GRU results in encoder output and encoder hidden state, which will be used in t (5). Next time, at $t + 1$ iteration, 64 elements each from

one sentence are taken and processed in a similar way. This iteration will be repeated 26 times, and encoder hidden of *t = 26* is passed to the decoder to use it as decoder hidden or context vector:

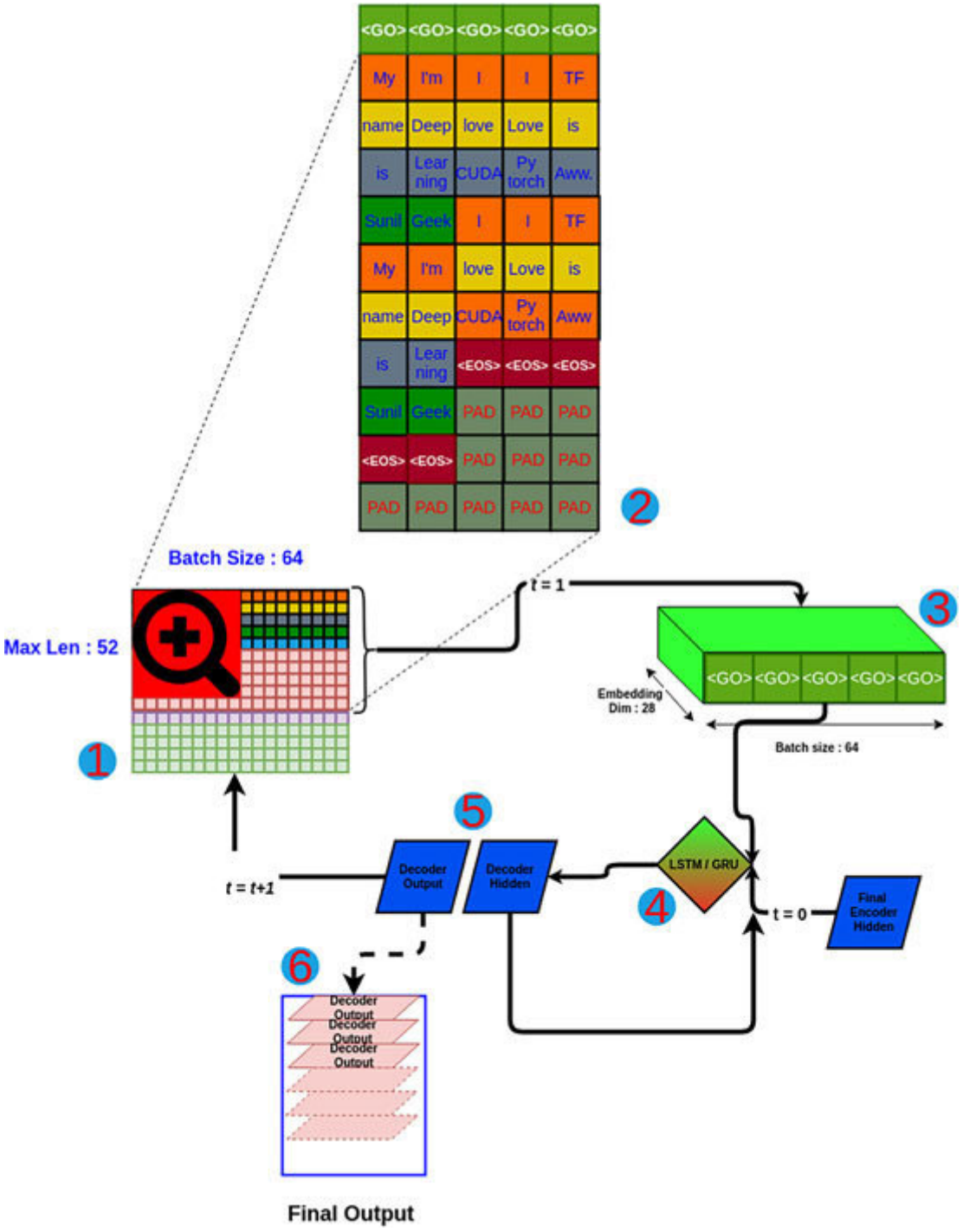


Figure 4.16: *An illustration of how batching works with the sequence to sequence - decoder phase.*

Decoder Phase

Decoding phase is shown in the second figure in the getting ready section of the book:

Sentences of double length, that is, *52 (26*2)* are shown with batch size 64. To make sentences equal, padding (PAD token) is used.

Each time step slice of shape 64 is taken, and 28-dimensional embedding is generated for each word.

Such a slice with embedding of shape [64, 1, 28] is processed with GRU/LSTM.

GRU/LSTM gives two things as output: decoder output, and decoder is hidden.

Decoder outputs are stacked and will be the final output after all time steps = are processed.

At each time step, the decoder's hidden state of the previous time step is used.

While training the decoding phase, the context vector is received as the final encoder hidden state. The decoding process is started with the token for all the sentences of the batch being passed to the embedding layer. Embedding for all tokens is generated, and these embeddings are passed to GRU along with the decoder's hidden state. In the first decoder, the time step, the decode hidden state is the final encoder hidden state, also known as context vector. GRU generates a decoder hidden state and an output. This decoder output is then stacked to the final output array as per the time step. If at time all tokens were passed to the decoder, the output at should have the first word for all 64 target sentences. This way, entire sentences are generated in the batch of 64. In each iteration, the decoder hidden state generated at is used, and entire sentences are generated similarly. Batching can only be done with fixed sizes of sentences. To make all sentences of the same size, padding is done. The end of the sequence is marked by .

Encoder and Decoder with Batching

Let's start with encoder; our encoder uses GRU as RNN units. Encoder unit takes two inputs: input and hidden. Input shape will be a tensor of size [Batch_size, where [64, 26]. For each batch, 64 elements are taken row-wise, and 28-dimensional embedding is calculated. The hidden state shape will be of size [unidirectional, Batch_size, where [1, 64, 28]. Here, we are using trainable PyTorch embeddings. Embed layer will convert each word into a fixed-size vector, so the embed layer will [Batch_size, input_size]_ to [Batch_size, input_size, Embed_size] for each batch. So, the input [64, 1] is given at each time step, and it will be converted to [64, 1, 28]. This says for 64 sentences, one word is taken, and 28 dimensions represent each word:

```
class EncoderRNN(nn.Module):
def __init__(self, input_size, hidden_size, n_layers=1):
super(EncoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
self.embedding = nn.Embedding(input_size, embed_size)
self.gru = nn.GRU(embed_size, hidden_size)
def forward(self, input, batch_size, hidden):
embedded = self.embedding(input).unsqueeze(1) #Input = 64 -
--> #Output [64,1]
```

```
embedded = embedded.view(1, batch_size, embed_size) #Input
= [64, 1] --- > #Output = [1, 64, 28]
output = embedded
output, hidden = self.gru(output, hidden)


return output, hidden #Output 1, 64, 28 #encoder Hidden =
1, 64, 28
ENCODER = EncoderRNN(input_size,hidden_size)
```

Decoder

The decoder unit takes two inputs: input and hidden. Input shape will be a tensor of size [Batch_size, where [64, 52]. Hidden state shape will be of size [unidirectional, Batch_size, where [1, 64, 28]. The last hidden state of encoder will be the first hidden state of the decoder. Here, we are using trainable PyTorch embeddings. The embed layer will convert each word into a fixed-size vector, so it will produce [Batch_size, input_size] to [Batch_size, input_size, Embed_size] for each batch. So, [64, 1] will be converted to [64, 1, 28] (Each batch has 64 sentences, and each word is represented by 28 dimensions). You must decode element by element for the mini-batches. The initial decoder state [batch_size, hidden_layer_dimension] is also fine. You just need to unsqueeze it at dimension 0 to make it [1, batch_size, Note that you do not need to loop over each example in the batch; instead, you can execute the whole batch at a time, but you must loop over all 52 batches of the input of dim [64, 52].

One thing to note in the following code block is that I am applying dropout after the embedding layer. Dropout is the most-used regularization techniques in deep learning architectures. It is applied after embeddings to regularize them. Another thing to note is that dropout is only applied during training, so I am using the following snippet:

```
if training == True:
embedded = self.drop(embedded)
class DecoderRNN(nn.Module):
def __init__(self, hidden_size, output_size, n_layers=1):


super(DecoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
self.embedding = nn.Embedding(output_size, embed_size)
self.gru = nn.GRU(embed_size, hidden_size)
self.out = nn.Linear(hidden_size, output_size)
self.softmax = nn.LogSoftmax(dim=1)
self.drop = nn.Dropout(0.2)
def forward(self, input, batch_size, hidden,training=True):
embedded = self.embedding(input) #Input = 64 ---> #Output
[64,1]
if training == True:
embedded = self.drop(embedded)
embedded = embedded.unsqueeze(1).view(-1,
batch_size,embed_size) # Input = 64, 1, 128 --- > #Output =
52, 64, 128
output = embedded
output = F.relu(output)
output, hidden = self.gru(output, hidden)
output = self.softmax(self.out(output[0]))
return output, hidden #Output 26, 64, 128 #encoder Hidden
= 1, 64, 128
# running decoder
DECODER = DecoderRNN(hidden_size,output_size)
```

The Loss Function for Sequence to Sequence

The loss function for the sequence to sequence architecture is a but different than regular loss calculations. In a batch, all sequences are not of the same length, so we must not calculate the loss for padding (OR PAD) tokens added to input batches. To avoid this, a masked loss is calculated. In addition to PAD, sometimes **End of** is added. Normal NLL loss is calculated, and loss corresponding to PAD is changed to zero by masking. The resultant loss will be equivalent to the average loss derived by dividing the total loss by total non-PAD tokens:

```
class customLoss(nn.Module):
def __init__(self,tag_pad_token = 1):
super(customLoss, self).__init__()
self.tag_pad_token = tag_pad_token
def forward(self,logits, target):
target_flat = target.view(-1)
mask = target_flat>self.tag_pad_token
loss = nn.NLLLoss(reduce=False)(logits,target)
loss = loss*mask.float()
result = loss.sum()/len(target)
return result
```

You can practice with the Jupyter notebook given at You can try to understand different shapes by running the example

codes attached to the encoder and decoder. Change various shapes according to your understanding, and try to see whether the network still runs and gives the desired shape.

A bag of useful tricks for practical neural machine translation - embedding layer initialization and large batch size can be found at *http://www.aclweb.org/anthology/W17-5708*

Translating in Batches with Seq2Seq

In this recipe, we will implement the sequence to sequence network, but we will use batching this time. Batching efficiently utilizes the power of parallel hardware like GPU. In the previous tutorial, I illustrated how batching works with sequence to sequence. In this tutorial, we will use the same encoder and decoder but with some modification in the data pipeline, to achieve our goal. This tutorial will also demonstrate the effect of batch size on learning. For our case, you will see that batch size 32 is computationally 10X more efficient than batch size 2.

Not to reinvent the wheel, I am using pre-processing the code used in the official tutorial of the PyTorch. This code was written to process one sample at a time; in the present recipe, we will be batch processing sentences. This requires very little changes in the code to accommodate padding for source and target sentences. By applying to the pad, the source batch and target batch will be made of equal size. The max sentence size of source and target is taken as reference, and all other sentences are padded to be made equal to that sentence. If you have a closely observed random_batch function in the topics discussed earlier, you will find some addition in the following modified random_batch function for batch processing. In the following script, the highlighted part is changed to find the max length:

```
def pad_seq(seq, max_length):
seq += [0 for i in range(max_length - len(seq))]
return seq
def random_batch(batch_size=3):
input_list = []
target_list = []


# Choose random pairs
for _ in range(batch_size):
pair = random.choice(pairs)
input_list.append(input_lang.indexes_from_sentence(pair[0]))
target_list.append(output_lang.indexes_from_sentence(pair[1]))
# Sort by length
tmp_pairs = sorted(zip(input_list, target_list), key=lambda p:
len(p[0]), reverse=True)
input_seqs, target_seqs = zip(*tmp_pairs)
# For input and target sequences, get array of lengths and
pad with 0s to max length
input_lengths = [len(s) for s in input_seqs]
target_lengths = [len(s) for s in target_seqs]
max_input_target = max(input_lengths+target_lengths)
input_padded = [pad_seq(s, max_input_target) for s in
input_seqs]
target_padded = [pad_seq(s, max_input_target) for s in
target_seqs]
# Create tensor using padded arrays into (batch x seq)
tensors
input_var = torch.LongTensor(input_padded,device = device)
target_var = torch.LongTensor(target_padded, device = device)
return input_var, target_var
```

Implementing Encoder/Decoder Capable of Batch Processing

Let's proceed with the same format as earlier.

Encoder

In encoder and decoder, I have used dropout as an additional operation to add regularize learning. Additionally, one more modification has been made to (device) function. As discussed in the second chapter, to an object helps transfer that object/ data to GPU, and further computation takes place in a defined device. If GPU were not available, computation would take place in CPU without any error. Note that the hidden state weight initialization is done with nn.init.xavier_normal_ here. Xavier / Glorot is the type of weight initialization technique that works well with deep learning. You can learn more about this initialization in the research paper "Understanding the difficulty of training deep feedforward neural networks":

```
class EncoderRNN(nn.Module):
def __init__(self, input_size, hidden_size, n_layers=1):
super(EncoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
self.embedding = nn.Embedding(input_size, hidden_size)
self.gru = nn.GRU(hidden_size,
hidden_size,num_layers=n_layers)
self.drop = nn.Dropout(0.2)
def forward(self, input, batch_size, hidden, training=True):
```

```
embedded = self.embedding(input).unsqueeze(1) #Input = 64, 26 ---> #Output 64, 26, 128
if training == True:
embedded = self.drop(embedded)


embedded = embedded.view(-1, batch_size, self.hidden_size)#Input = 64, 26, 128 --- > #Output = 26, 64, 128
output = embedded
output, hidden = self.gru(output, hidden)
return output, hidden #Output 26, 64, 128 #encoder Hidden = 1, 64, 128
encoder = EncoderRNN(input_size, hidden_size, n_layers=1)
encoder = encoder.to(device)
```

Decoder

This is the usual decoder using GRU as the recurrent units:

```
class DecoderRNN(nn.Module):
def __init__(self, hidden_size, output_size, n_layers=1):
super(DecoderRNN, self).__init__()
self.n_layers = n_layers
self.hidden_size = hidden_size
self.embedding = nn.Embedding(output_size, hidden_size)
self.gru = nn.GRU(hidden_size, hidden_size,num_layers =
n_layers)
self.out = nn.Linear(hidden_size, output_size)
self.softmax = nn.LogSoftmax(dim=1)
self.drop = nn.Dropout(0.2)
def forward(self, input, batch_size, hidden,training=True):
embedded = self.embedding(input) # Input = 1,64, 52 --->
#Output 64, 128
if training == True:
embedded = self.drop(embedded)
embedded = embedded.unsqueeze(1).view(-1,
batch_size,self.hidden_size) # Input = 64, 1, 128 --- >
#Output = 52, 64, 128
output = embedded
output = F.relu(output)
output, hidden = self.gru(output, hidden)
output = self.softmax(self.out(output[0]))
```

```
return output, hidden #Output 26, 64, 128 #encoder Hidden = 1, 64, 128
decoder = DecoderRNN(hidden_size, output_size, n_layers=1)
decoder = decoder.to(device)
```

The Loss Function for Sequence to Sequence

As discussed in the previous recipe, all sequences in a batch are not of the same length. So, we must not calculate the loss for padding (OR PAD) tokens added to the input batches. To avoid this, a masked loss is calculated. In addition to PAD, **End of Sequence** is added sometimes. NLL loss is calculated, and loss corresponding to PAD is changed to zero by masking. The resultant loss will be equivalent to the average loss derived by dividing total loss by the total non-PAD tokens. Now, I will combine all parts of the implementations into one function, and this train_it function takes six inputs:

Encoder object

Decoder object

Batch To help us look at the effect of Batch size on training

The number of random batches used to train the model

To evaluate using random samples while training

To plot the progress

In the Current implementation, I will use a learning rate of 0.001 and an RMSprop optimizer. RMSprop is a kind of optimizer that adapts the learning rate by dividing by the root of the squared gradient. To understand RMSprop in detail, you can check the references implemnttion.

As a rule of thumb, one should use a batch size between 4 and 64. Next, we will use batch size 2. It is clear from the plot of loss vs. iteration that loss is not decreasing, so no learning is taking place. With a smaller batch size, optimum one, it will take more time to get trained. It took total wall time of 1 min 34s.

Added %%time as the magic like at the beginning of the Jupyter notebook cell provides the execution time of the block when the function finishes execution.

```
%%time
encoder = EncoderRNN(input_size, hidden_size, n_layers=1)
decoder = DecoderRNN(hidden_size, output_size, n_layers=1)
encoder = encoder.to(device)
decoder = decoder.to(device)
batch_size = 2
train_it(encoder,decoder,batch_size, 1000, test = False, plot = True)
```

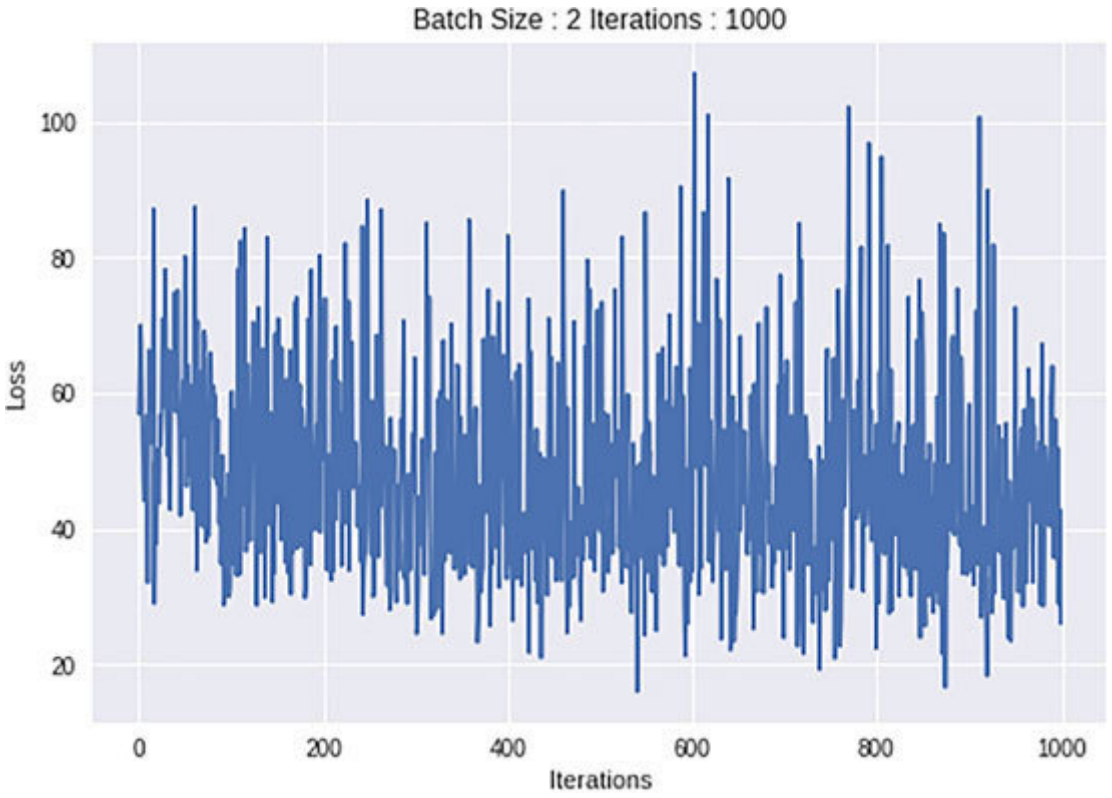


Figure 4.17: *Showing no significant decrease in the loss when training is done keeping the batch size equal to 2.*

Running the same training process with 32 batch size took a total wall time of 2min 21s:

```
%%time
encoder = EncoderRNN(input_size, hidden_size, n_layers=1)
decoder = DecoderRNN(hidden_size, output_size, n_layers=1)
encoder = encoder.to(device)
decoder = decoder.to(device)
batch_size = 32
train_it(encoder,decoder,32, 1000, test = False, plot = True)
```

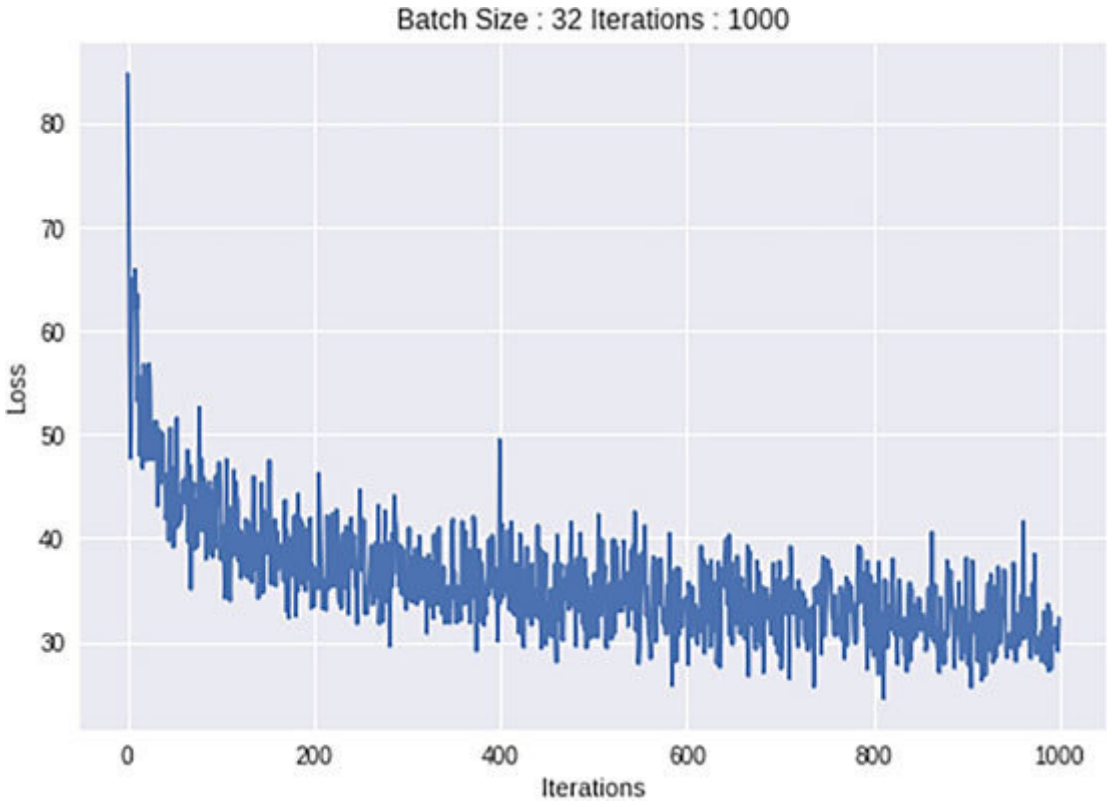


Figure 4.18: *Showing a significant decrease in loss when training is done keeping batch size equal to 32.*

When we run with batch size 2 for 1000 iteration, it processes 2000 samples and takes 1min 34 sec or 94 seconds or 2000/94 = 21.27 samples/sec. When we run with batch size 32 for 1000 iteration, it processes 32000 samples and takes 2 min 21 sec or 94 seconds or 32000/94 = 226.95 samples/sec.** That is a 10X** improvement. The reported loss in the second case is also lower than in the first case. The second case processes more samples, so its loss will be much lower than the first one. Note that the tie reported here is highly dependent on the kind of GPU and CPU you use.

Slowly, you will see that the translation is improving after every 1000 iterations. The entire implementation with supporting functions is given at

There's more to discuss:

I used GRU in the preceding illustrations, and used LSTMs to see if it makes a significant difference in the quality. Note that LSTM requires two states—a cell state and a hidden state—so change the state initialization and input of the LSTM accordingly.

Try to see the effect of batch size on the translation quality.

Take a look at the following references:

How to choose optimum batch size:
https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network

Understanding RMSprop — faster neural network learning:
https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a

Implementing Attention for Language Translation

Understanding the attention mechanism is very important, as it leads to powerful language blocks like The recently-released model by Google research, which claims to beat all previous state-of-the-art language models by a considerable margin, also includes attention variants.

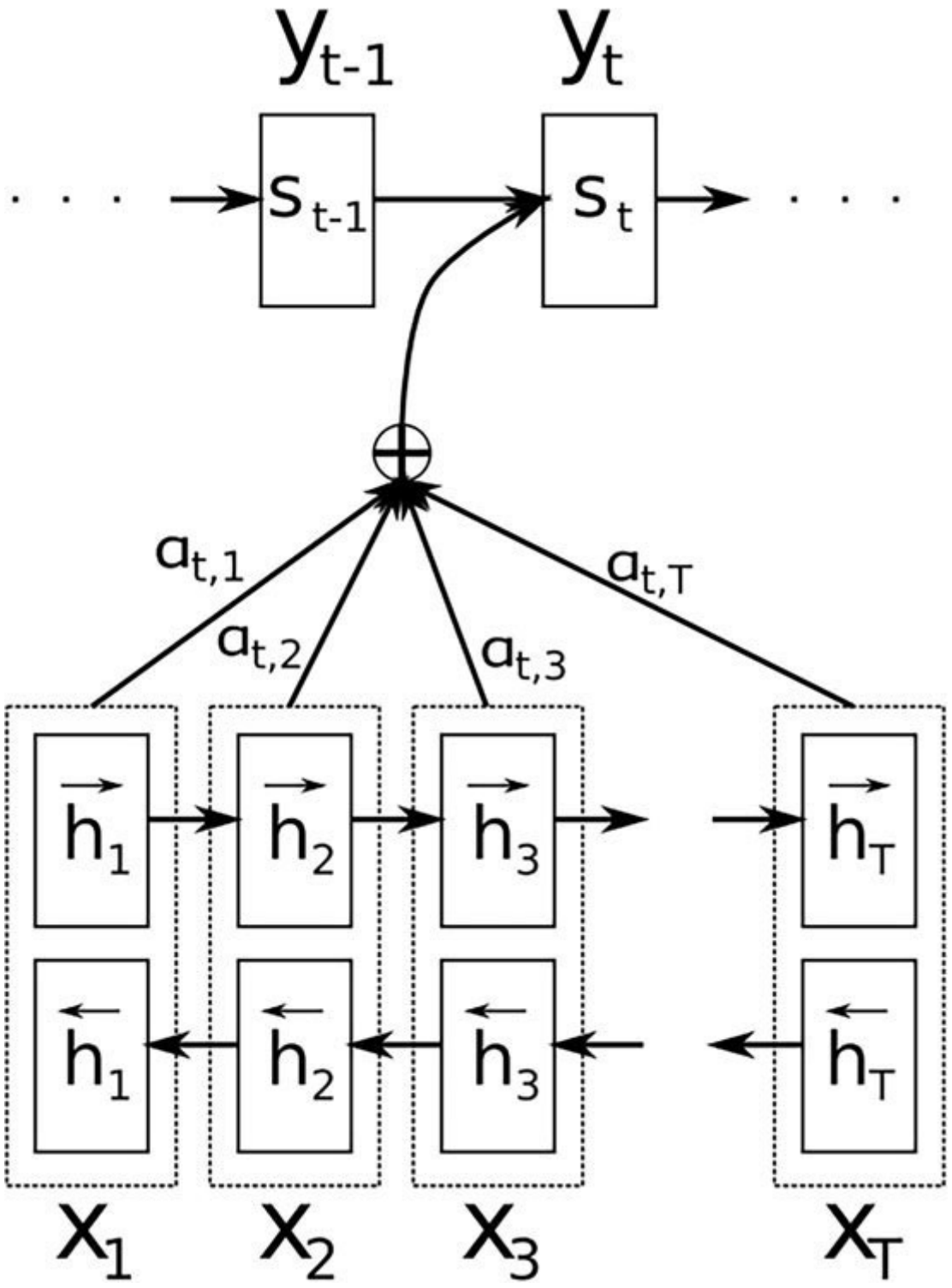


Figure 4.19: *Attention mechanism as depicted in Attention Is All You Need*

This is the famous diagram as depicted in paper, but it does not provide the details required to implement the mechanism. This recipe will help you understand the entire flow and every operation involved in the attention mechanism. In this implementation, we will use sequence-to-sequence models. We will implement the attention mechanism described in the **Neural Machine Translation** paper by jointly learning to *Align* and This model is different from the previous implementations in the following aspects:

We will use the Multi30k dataset, which is considered as a standard dataset when it comes to comparing the performance of different machine translation models.

We will use bi-direction RNN; this is little add-on to the previous RNN layers.

We will use an attention mechanism to translate better.

As we saw in the previous topics, we are only providing the last decoder hidden state (context vector) to the decoder, and the decoder produces a target sentence using this information. The output of the encoder time steps are not used at all. The previous approaches may suffer from the following problem:

The context vector might be unable to remember the entire sequence correctly or may forget information related to the early time steps.

The output vector at each time step with vital information about each time step is not being used at all.

Encoder

Before going ahead, we will understand the bidirectional RNN layers. Up to the previous implementation, we used only RNN (or Unidirectional RNN), which means it runs in a single direction. In this implementation, I am using bidirectional RNN. This means each layer has two RNNs: one running in the forward direction of the sequence, and another running in the backward direction. Bidirectional RNN does not require any extra line of code; it just requires the bidirection=True parameter. After this, we can pass the embedded sentence as we used did in the previous implementations. Here's an example:

example:
example: example: example: example: example: example: example: example: example: example: example:

Table 4.3

Where,

Start of sequence indicator

End of sequence indicator

Mathematically, bidirectional RNN can be represented as follows:

$$O_t, h_t^{\rightarrow} = EncoderRNN^{\rightarrow}(x_t^{\rightarrow}, h_{(t-1)}^{\rightarrow})$$

$$O_t, h_t^{\leftarrow} = EncoderRNN^{\leftarrow}(x_t^{\leftarrow}, h_{(t-1)}^{\leftarrow})$$

Where is input at the current time step, and is the hidden state from the previous time step. For an example of German to English, the first and second input token for forward and backward RNN is given as follows:

Forward

Backward

$x_0^{\leftarrow} =< eos >, x_1^{\leftarrow} = Datenwissenschaftler$

As shown below, it has forward and backward units. Each unit takes one input and generates output *O* and updated hidden state :

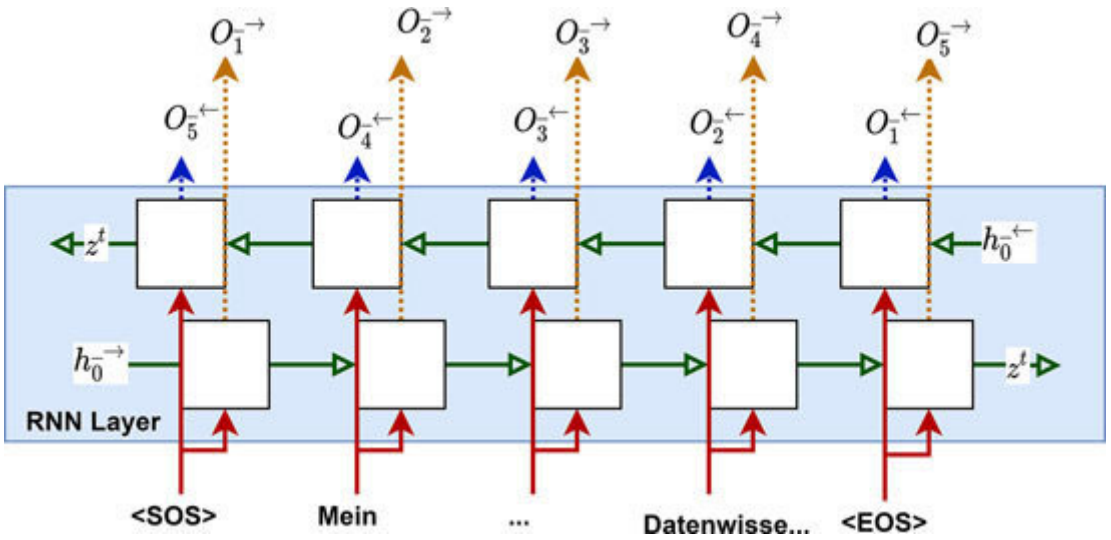


Figure 4.20: *Showing bidirectional RNN: one unit with forward and backward RNN units.*

For understanding, I have shown an RNN layer with bidirectional units in the unrolled state.

Each word of the input in the embedded form is taken in the forward and the backward direction. At each input at time-step, a forward output and a backward output are generated. At the end, the hidden forward state and backward hidden are generated. Hidden state in the forward and backward state is also represented in the form and , respectively.

To keep things simple, we only pass the embedded input to the GRU and leave initialization of the forward state and backward state to the GRU this time. Finally, both the hidden state from forward and backward run are concatenated to get the context vector *Z* for layer, represented by .

The decoder is not bidirectional, and the author only uses the only one hidden state in the original paper. Instead, here I have concatenated both the forward and the backward hidden state and applied activation: , where *C* stands for the final context vector.

Attention Mechanism

The attention layer takes the encoder hidden and encoder output as the output. The attention mechanism has many operations attached, but the main idea is to combine encode outputs and the content vector (final encoder hidden state) to produce the attention weight. This attention weight has information about the weight required for each token in the source language. This attention state represents which source word should be given more weight to generate the next target word. This attention vector after Batch-wise Matrix Multiplication added to the previously generated decoder token and generate the next token at time

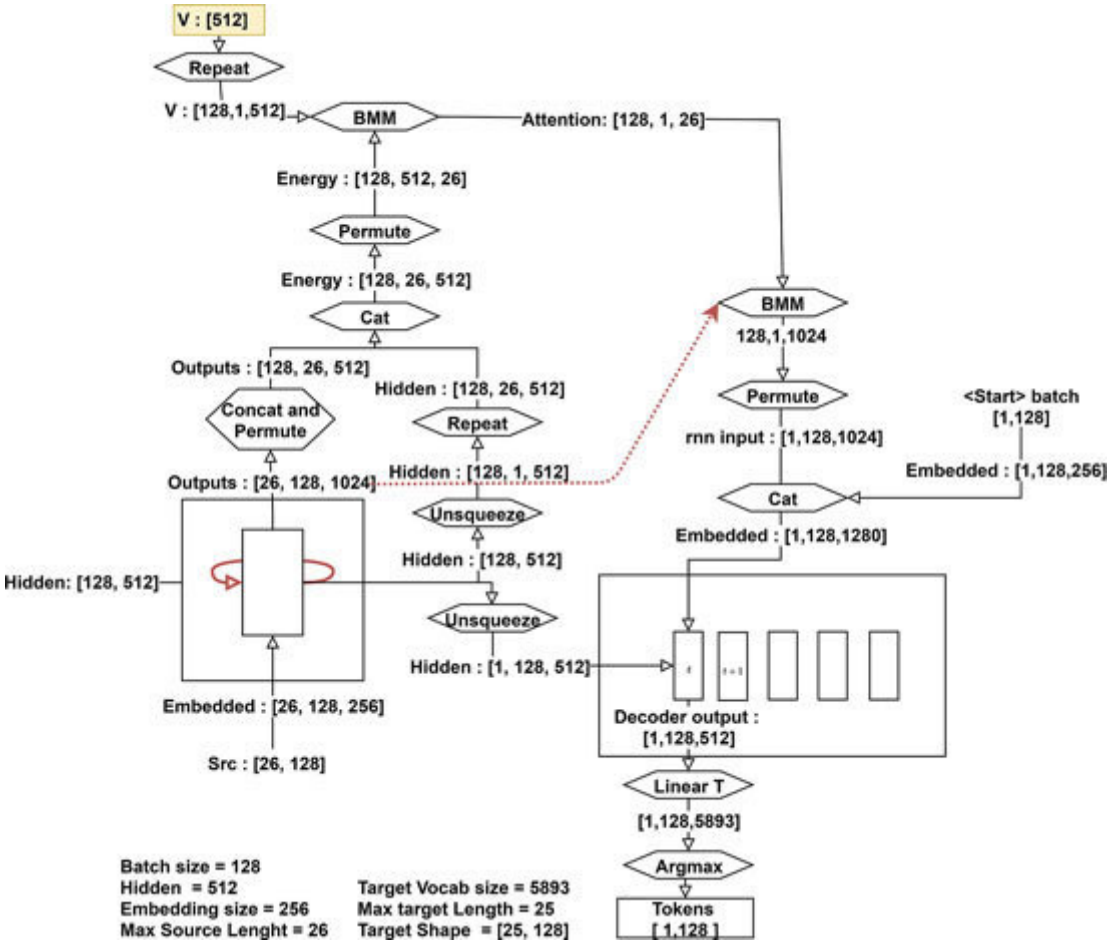


Figure 4.21: *A simplified summary of the attention mechanism. It has three components: encoder, decoder, and attention mechanism.*

Decoder

The decoder contains the attention layer, and the decoder function takes the input, encoder hidden/ context vector and encoder outputs. Suppose we have a batch size of 4. For the first time steps, the indices corresponding to the token are given as This is equivalent to the shape [1, 4]. Let's say we have embedding dimension10, so each index will be represented by a 10-dimensional dense vector, and the resultant shape will be [1, 4, 10] after applying the embedding. This serves as the input to the decoder RNN after addition to attention weight.

The implementation aspect of the following parts of the sequence to the sequence model will be discussed, including:

Dataset

Encoder

Attention mechanism

Decoder

Dataset: Up to the previous iteration, we were using the French to English translation pairs with a few thousand

training samples. In this implementation, to keep the preprocessing part simple, I will use a multi30k dataset. Multi30k is a slightly larger dataset from WMT 2016 multimodal task, also known as In multi30k 29, 000 training, and 1, 014 test samples are provided. The current task is related to the German to English translation. The attention model has many parameters attached, and it is really difficult to train such a model with such a small dataset. Due to this treason, I am using a slightly larger dataset. After this implementation, you will see that the attention model can generate a really meaningful translation.

Encoder: The encoder is similar to our previous implementations and gives two outputs:

Outputs for each time-step will be of shape [src sent len, batch size, hid dim * num directions]

Concatenated hidden states of shape [n layers * num directions, batch size, hid dim]

In the code, [-2,:, :] gives the top layer forward RNN hidden state after the final time-step (i.e., after it has seen the last word in the sentence), and [-1,:, :] gives the top layer backward RNN hidden state after the final time-step (i.e., after it has seen the first word in the sentence).

Attention mechanism: The overall procedure, as shown in the *figure* can be summarized as follows:

Take encoder outputs and encoder hidden.

Repeat encoder hidden to source the sequence length times.

Concatenate both the output after proper permutation.

Apply additional operations like permute and carry out **Batch-wise Matrix Multiplication** with the learnable vector.

After these operations, the generated weight is called Attention weight.

Attention weight undergoes BMM with encoder.

Output: These attention weights are then given to the decoder.

```
class Attention(nn. Module):
def init(self, enc_hid_dim, dec_hid_dim):
super().init()
self.enc_hid_dim = enc_hid_dim

self.dec_hid_dim = dec_hid_dim
self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim,
dec_hid_dim)
self.v = nn.Parameter(torch.rand(dec_hid_dim))
def forward(self, hidden, encoder_outputs):
batch_size = encoder_outputs.shape[1]
```

```
src_len = encoder_outputs.shape[0]
hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)
encoder_outputs = encoder_outputs.permute(1, 0, 2)
energy = torch.relu(self.attn(torch.cat((hidden, encoder_outputs),
dim = 2)))
energy = energy.permute(0, 2, 1)
v = self.v.repeat(batch_size,1).unsqueeze(1)
attention = torch.bmm(v, energy).squeeze(1)
return F.softmax(attention, dim=1)
```

Decoder: The decoder has the following steps to perform:

The attention weight vector is added to the embeddings of the previously generated token in the decoder. If the first token needs to be generated, all embeddings of a token are provided as input. The decoder will output the next token.

As per the teacher forcing training scheme, the generated token is given as input to the next time-step.

The following are a few generated and original sentences:

sentences:
sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences:

sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences:

sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences: sentences:

Table 4.4

When attention mechanisms are applied to the translation task with a strong theoretical background, the quality increases. Yet, the translation quality is up to the mark, and a better network like the transformer is used to get an even better translation. You can find the well commented implementation at

In the preceding example, we tried to train the embedding from scratch, accommodate *Glove* or *FastText* embeddings, and see if it can improve the results. The language translation result is generally judged based on the **Bilingual Evaluation Understudy Score** score. BLEU is a metric for evaluating the generated and the reference sentence, and BLEU score can be easily calculated using the sci-kit library, as follows:

```
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'a', 'Monkey'], ['this', 'is' 'Monkey']]
candidate = ['this', 'is', 'a', 'key']
score = sentence_bleu(reference, candidate)
print(score)
```

This score is the standard metric used to measure the quality of the translation; the higher it is, the more human-like the generated translation is.

You can refer to the following links:

Neural machine translation by jointly learning to align and translate: *https://arxiv.Org/abs/1409.0473*

Effective approaches to attention-based neural machine translation: *https://arxiv.Org/pdf/1508.04025.pdf*

Conclusion

In this chapter, we had our very first experience with models like Vanilla Recurrent Neural Networks, Gated Recurrent Unit, and Long Short-Term Memory. These models are collectively referred to as recurrent networks. We implemented all the forms of RNN using Pytorch, and this chapter also provided the very first practical experience of building embedding, data loader, and text processing before we built the actual model. We also learned how to implement the simple sequence to sequence model by combining two bigger models: encoder and decoder. Then, we gradually built this model so that it processes in batches, and we applied attention logic to it. The attention mechanism is used in all new networks like transformers. We developed our first application using the attention mechanism—a machine translation application. Although the machine translation model is a basic one, it provides a strong hands-on experience on how sequence to sequence model works and helps uplift your implementation skills to the next level.

The next chapter will take you through how to apply CNN in NLP tasks.

CHAPTER 5

Applying CNN in NLP Tasks

Convolution networks are traditionally used for image processing, but it has recently been found that using them for text processing could help us achieve higher accuracy in many tasks. Nowadays, CNN is used in text classification, language translation, question answering, and embeddings generation. State-of-the-art embeddings generation techniques have inbuilt CNN components. In this chapter, we will understand how to use CNN components with PyTorch and look at the application of text and character-based features with a convolution neural network. Later, we will see how to go beyond 30 layers and utilize such a network for text classification. At last, this chapter covers a few methods to train deeper networks with more than ten layers stacked on one another. This recipe will cover the peculiar tricks used in ResNet, Highway networks, and DenseNet to reach deeper.

Structure

In this chapter, we will cover the following recipes:

Understanding CNN

Using word level CNN

Using character level CNN

Training deeper networks

Objective

Understanding convolution operations, understanding the effect of the filter size, and stride width on end prediction

Implementing CNN and testing its ability to generalize on unknown examples

Understanding and applying an advanced function like batch normalization

Understanding word convolution, character convolution, and grouped convolution and reason behind the high efficiency of such networks

This chapter focuses mainly on applying convolutional networks to text.

Pre-requisites

The code for this chapter can be found in the Ch5 folder at GitHub repository This chapter requires the following packages to execute code:

Pandas

Matplotlib

Torch

Numpy

Scikit learn

Numpy

Nltk

Spacy

TensorboardX

Tensorflow

You can install these requirements by installing all the packages listed in requirements.txt simply by issuing pip install -r This chapter uses Ipython Notebook/Jupyter Notebook for easy execution and for connecting thoughts with implementation.

Understanding CNN

Convolutional Neural Networks are popularly known as CNN. Convolve is the mathematical operation, and the concept of convolution operation is not new. The way convolution operations were applied to the images by *Yann LeCun* was a novel approach, and Yann LeCun applied CNN to the MNIST dataset using the LeNet5 architecture. Having achieved success in classifying handwritten numbers, Yann LeCun actively pushed this architecture into the research community.

Convolution is a mathematical function where shape *A* and shape *B* interact to form a shape which is a modification of Convolution neural network is very similar to the feed-forward network, as they are both made up of neurons having learnable weight and bias. CNN can be trained using optimizer and loss function as used for RNN and FFN. In **Feed-forward** the 1-dimensional vector is passed through a series of the layer, each having several perceptions. The perceptrons in the same layer do not interact with each other and work independently with the surrounding layers. Each of these perceptrons is fully connected with the perceptions of the previous and next layer.

Convolution neural networks take 3D inputs like images, which have height, width, and several channels (RGB). A CNN

operates by sharing weights in the same layer, so it can better deal with images. We will see how CNN shares weights in detail. In CIFAR-10, images are only of size 32x32x3, and if we process it by the fully connected network, the input size will be 3072. While this is manageable, image size of 200 x 200 x 3 will mean there are 120,000 features. Handling the weight of size [120, 000, m] is not an easy task, where m is the size of the first hidden layer attached to the input. The following is an illustration of the regular neural network and CNN:

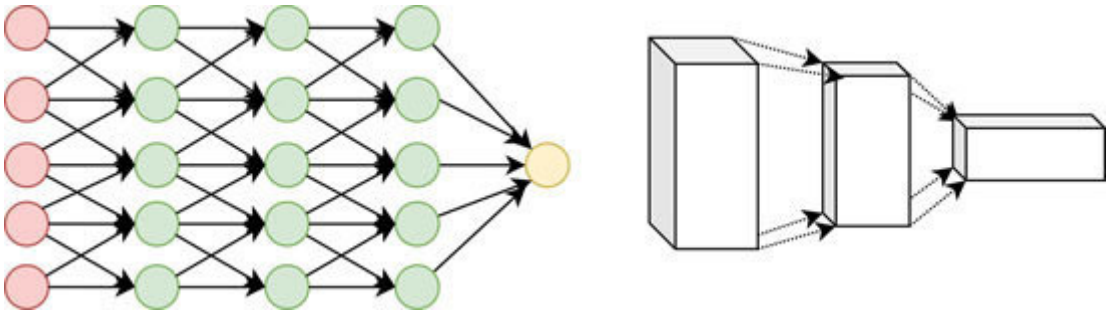


Figure 5.1: *The similarity between the FFN and the CNN*

Both have similar nature; FFN takes 1D input, whereas CNN takes 3D input.

Understanding Convolution Operations

The term convolution is derived from the word convolve, which is a mathematical operation. A CNN consists of many types of layers, including convolution, pooling layer, fully connected layers, and activation layers like Relu.

Convolution Layers

Convolution layers have learnable parameters caller filters. A filter has smaller receptive fields but extends through the entire depth of the input volume. During the forward pass, the filter moves over the width, and the height of the input volume and dot product is calculated between elements of the input shape and the elements of the filter. These filters are learnable, so the network learns to detect essential features. Such filters are activated when a particular pattern is detected. The following diagram contains an input shape and a filter and a resultant product:

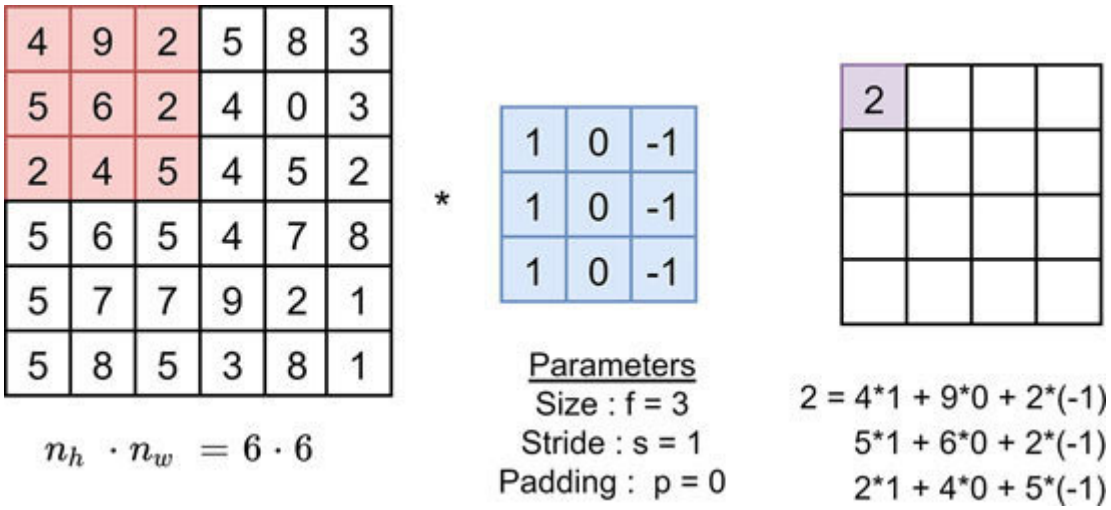


Figure 5.2: *The convolution operation is explained by taking an example of shape and filter.*

The 2D input is of size 66. The 2D filter of size 3 x 3 is convolved over the highlighted part of the 2d input. The resultant value after convolution calculation is inserted into the output shape, as shown. This filter convolves over the entire input shape by jumping one cell at a time(stride).

We can use the following generalized equation to calculate the output shape:

$$\frac{W-F+2P}{S}+1$$

Here in the equation *W* = Input matrix size, *F* =filter size, *P* = Padding, and *S* = Stride For above taken example the parameters are *W* = 6, 3, *P* = 0, and *S* = 1, the resultant shape turns out to be 4:

$$\frac{6-3+2*0}{1}+1=4$$

Let's discuss some of the hyperparameters that can affect the output of the convolution operation.

Padding

Padding is applied to a convolution neural network to control the output shape. If padding equal to 1 is applied to the preceding example, the input volume looks as follows:

The output shape will be 6 – 3 + 2*1/1 + 1 = 6. Hence, if padding equal to 1 is applied to the input shape, the output shape will be of the same size.

Stride

Stride controls how many pixels the filter jumps while sliding on the input shape. If stride = 1, we move one pixel at a time. Stride more than 3 is uncommon in the image-related example. If a stride above 3 is chosen, the filter jumps 3-pixel at a time, leading to a loss of information. The filter position with the various values of stride is shown below. Earlier, we saw that with the input shape of 6, filter size = 3, padding = 0 and stride = 1, the output shape was 4. Let's see what happens if we apply stride = 2:

$$\frac{6-3+2*0}{1}+1=2.5$$

The resultant shape will be of size 2.5, which is not possible, so stride cannot be any value; it has to be applied after making shape calculations.

Pooling layers

Pooling layers are an essential part of convolution neural networks. Pooling layers perform downsampling operation on the input shape. The pooling operations have two inputs: kernel size and stride. Kernel decides the slice of shape where pooling is applied, and stride determines how many pixels a kernel will slide at a time. Generally, two types of pooling operations are applied: *max pool* and *average* Max pool is where the max element in the pooling filter is taken, and average pooling is where the average of all elements in the pooling filter is taken. Optionally, one can apply padding to the input shape before applying pooling operation.

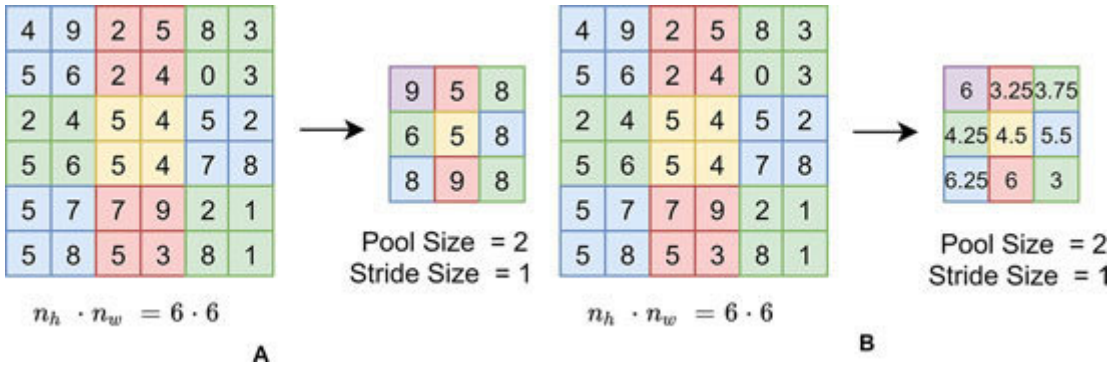


Figure 5.3: *Showing of max and average pool.*

Fully Connected Layers

The main aim of applying convolution and pooling operation is to concentrate the information. After sufficient depth of the convolution and polling is reached, the fully connected layer is applied to classify an image into the desired class. VGG network has all these layers. The Visual Geometry group developed this network at Oxford University. A brief about this theory is covered, and we will understand how to implement convolutional layers using PyTorch as we move ahead.
Convolution layer: PyTorch has a separate convolution layer for 1D, 2D, and 3D inputs.

Convolution 1D

Here, torch.nn. Conv1d takes 16 channels and output 33 channels. The filter kernel size is 3, and the stride is 2. When input shape in a batch of 20 with each having 16 channels with 50-feature is given, the resulting shape will be [20, 33, 24]. The output shape is a little wrinkled than the input one:

```
m = torch.nn.Conv1d(in_channels=16, out_channels=33,
kernel_size=3, stride=2)
input = torch.randn(20, 16, 50)
output = m(input).shape
>>>torch.Size([20, 33, 24])
```

Conv 1D is often used in text processing. Here, the input can be a sentence in the batch of 20, where each sentence has max 16 words, and a 50-dimensional dense vector describes each word. The resultant vector 16, can be given as input to

Convolution 2D

Convolution 2D is often used for images. Here, the Conv2d function defines kernel_size=3 and The image like 3D shape is given as input in batches of 20, with each image of size 50* 50 having 3 channels (RGB):

```
m = torch.nn.Conv2d(in_channels=3, out_channels=33,
kernel_size=3, stride=2)
input = torch.randn(20, 3, 50, 50)
output = m(input)
>>>torch.Size([20, 33, 24, 24])
```

Convolution 2D operation is used for text. Also, let's say, for a given problem, that each sentence can have max word equal to 50, each word can be of max ten characters, and each word can have 64 unique characters. The resultant shape is 10, if such input is processed in a batch of 32, the resultant shape 50, 10, can be given as input to

Pool Layers

Similar to the convolution layer, PyTorch has 1D, 2D, and 3D pooling layers. PyTorch pool takes kernel size as the required parameter and other operational parameters like stride and padding:

```
m = torch.nn.MaxPool1d(3, stride=2)
input = torch.randn(20, 16, 50)
output = m(input)
torch.Size([20, 16, 24])
```

Similarly, 2D and 3D pools are applied. PyTorch has the and AvgPool3d functions to support average pooling.

Rectifier Linear Unit (Relu)

ReLu is the newly invented activation function and produces a state-of-the-art result with faster convergence. The Relu function can be given as. It is very simple, it only allows positive values. Gradient computation is simple, as there are no exponent or division operations that must be differentiated. The simple operation essentially speeds up computation.

Go through the architecture of the **Very deep convolutional networks** for large-scale image recognition and try to code it using PyTorch. We have already seen where 1D and 2D CNN are used, but what about 3D CNN? Try to find out use cases where 3D CNN are used. To hone your CNN implementation skills, we will take a few interesting applications of 2D and 1D CNN in the upcoming chapters.

You can check out the following references:

Gradient-based learning applied to document recognition: *http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf*

Deep learning using **rectified linear units** *https://arxiv.org/pdf/1803.08375.pdf*

Using Word Level CNN

Traditionally, CNN is used for various vision-related applications. In this recipe, we will see how CNN can be applied to the text classification problem. We will use the word level features and pre-trained embedding with CNN for the text classification problem. We will also understand and implement logic as published in *Neural Networks for Sentence* by Jonas Gehring et.al., According to this paper, with pre-trained embeddings, one can achieve excellent results by just using a few layers of CNN. Let's look at this paper in detail and understand how to leverage CNN for text-related tasks.

Before moving to implementation, let's understand the model. The model is shown in the following figure:

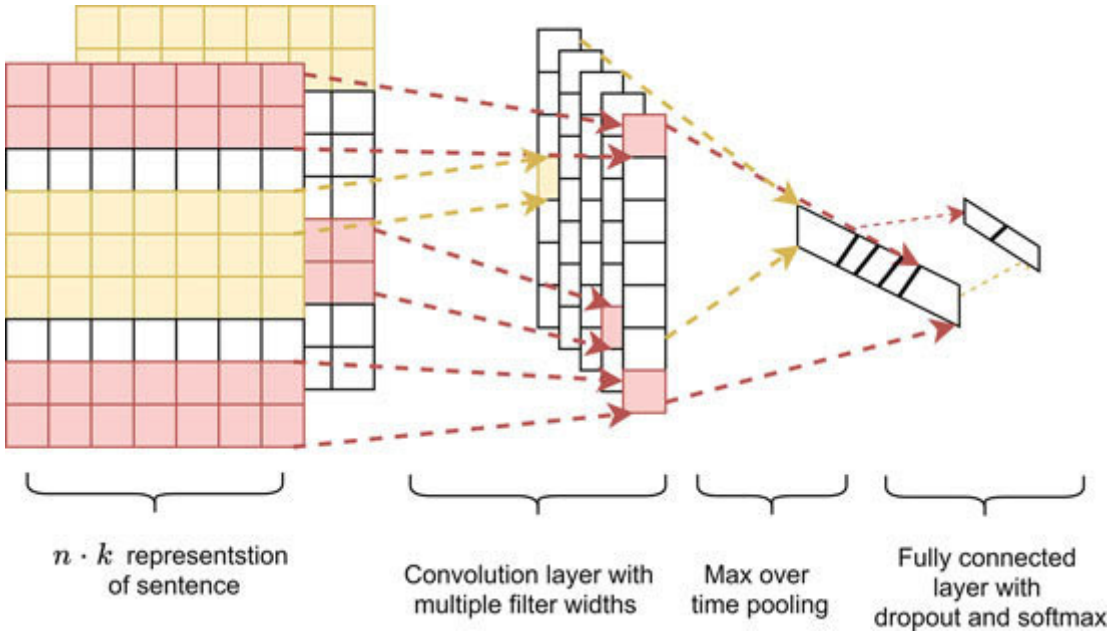


Figure 5.4: *The architecture of the model that takes word level features and performs text classification.*

The figure shows the architecture of the model that takes word-level features and performs text classification. Let's take a sentence with *n* words, each word having *k* dimensional vector; the resultant vector size is *n* * All sentences are expected to be padded to have equal size. This input matrix of size *n* * *k* is then convolved using different filter sizes. In our implementation, we will use filter sizes = [2, 3, 4]. One more thing to observe here is that the stride size is very large. On regular CNN, we hardly go to a stride size of 4-5, but a stride of 100 is used here. In this model, the stride size will always be equal to the size of the embedding By keeping the stride equal to the embeddings, the model learns the feature for each word separately. Mathematically, a filter of height *H* = {2,3,4} is selected with width/stride *k* equal to the dimensions of the embedding vectors. This way, different features are learned by choosing different words. Suppose the input matrix *X* = *n* * If a convolution operation *W* is applied with filter size the derived features can be given as:

$$= (W \bullet + h + b)$$

Here, is the small portion of the input matrix for the sentence over which the convolution operation was applied, and *b* is the bias term. Such operation with different window size/ kernel size is applied and features are collected. Then, the max pool in 1-dimension is applied over the collected

features to identify striking features. After max pooling, all the features are concatenated and the feed-forward layer is applied on top of the previous layers.

CNN is not sequence-based, unlike RNN. In RNN, the previous time step must be completed to compute the next time step, so parallelization is limited. The main advantage of this technique is that computation can be fully parallelized and can better use GPU like devices.

Pre-processing

In this recipe, we will use IMDB, large movie review dataset. It is a dataset for binary sentiment classification containing 25,000 highly polar movie reviews for training and testing. Let's use TorchText to pre-process our data. The pre-processing involves:

Splitting the data into two parts: train, and test

Reading the data using TorchText and applying various pre-processing operations like tokenization, padding, and vocabulary generation

Defining data fields

Generating vocabulary

Making a train and test data iterator

```
# defining data fields
REVIEW = data.Field(sequential=True, preprocessing =
pad_to_equal, use_vocab = True, lower=True,batch_first=True)
LABEL = data.Field(is_target=True,use_vocab = False,
sequential=False, preprocessing =to_categorical)
fields = {'review': ('review', REVIEW), 'label': ('label', LABEL)}
```

```
# constructing tabular dataset
train_data, test_data = data.TabularDataset.splits(
path = '',
train = 'train.json',
test = 'test.json',
format = 'json',
fields = fields)
# constructing vocabulary
REVIEW.build_vocab(train_data, test_data)
LABEL.build_vocab(train_data, test_data)
# making iterator

train_iter, test_iter = data.Iterator.splits(
(train_data, test_data), sort_key=lambda x: len(x.review),
batch_sizes=(16,len(test_data)), device=device,)
```

Embedding

For this experimentation, we will use the GloVe vector of dimension 100 trained on the Wikipedia+Gigaword 5 (6B) dataset. We will use chakin to download GloVe word vectors, and map the vocabulary for our train and test split to the GloVe vector using the following snippet. We will also use this shortcut again in this chapter:

```
vec = vocab.Vectors(name = "glove.6B.100d.txt",cache = "./")
REVIEW.build_vocab(train_data, test_data, max_size=100000,
vectors=vec)
# vocab vecotr mapping
review_vocab = review.vocab
```

This mapping is then passed to the embeddings layer of the network by directly passing it as weights to the PyTorch embedding layer. By setting autograd equal to false, we set this layer as non-trainable:

```
self.embedding = nn.Embedding(embed_num, 100)
self.embedding.weight.data.copy_(review_vocab.vectors)
self.embedding.weight.requires_grad = False
```

Convolution Layers

Embeddings generated in the previous layer for each sentence are passed to the following convolution layer. Generally, when it comes to sentiment analysis, the entire review is passed to the Conv2D with different filter sizes [2, 3, 4] and can be represented as follows:

```
self.conv13 = nn.Conv2d(in_channels = 1, out_channels=8,
kernel_size= 100)
self.conv14 = nn.Conv2d(in_channels = 1, out_channels=8,
kernel_size= 100)
self.conv15 = nn.Conv2d(in_channels = 1, out_channels=8,
kernel_size= 100)
```

The output of the conv2D layer is passed to the maxpoll1D layer, and all the resultant features are concatenated as follows:

```
x1 = self.conv_and_pool(x,self.conv13)
x2 = self.conv_and_pool(x,self.conv14)
x3 = self.conv_and_pool(x,self.conv15)
x = torch.cat((x1, x2, x3), 1)
```

Finally, a fully connected layer, along with dropouts and ReLu, is used to squeeze the features into two output equal to final

classes:

```
x = self.dropout(x)  # (N, len(Ks)*Co)
logit = F.relu(self.fc1(x))  # (N, C)
logit = torch.softmax(logit, dim=1)
```

When I applied the preceding implementation to the IMDB sentiment analysis dataset, it achieved over 95% accuracy on train data and 75% accuracy on the test data. You can go through the code and correlate it with the original research paper. The loss and accuracy of progress throughout the training are given as follows:

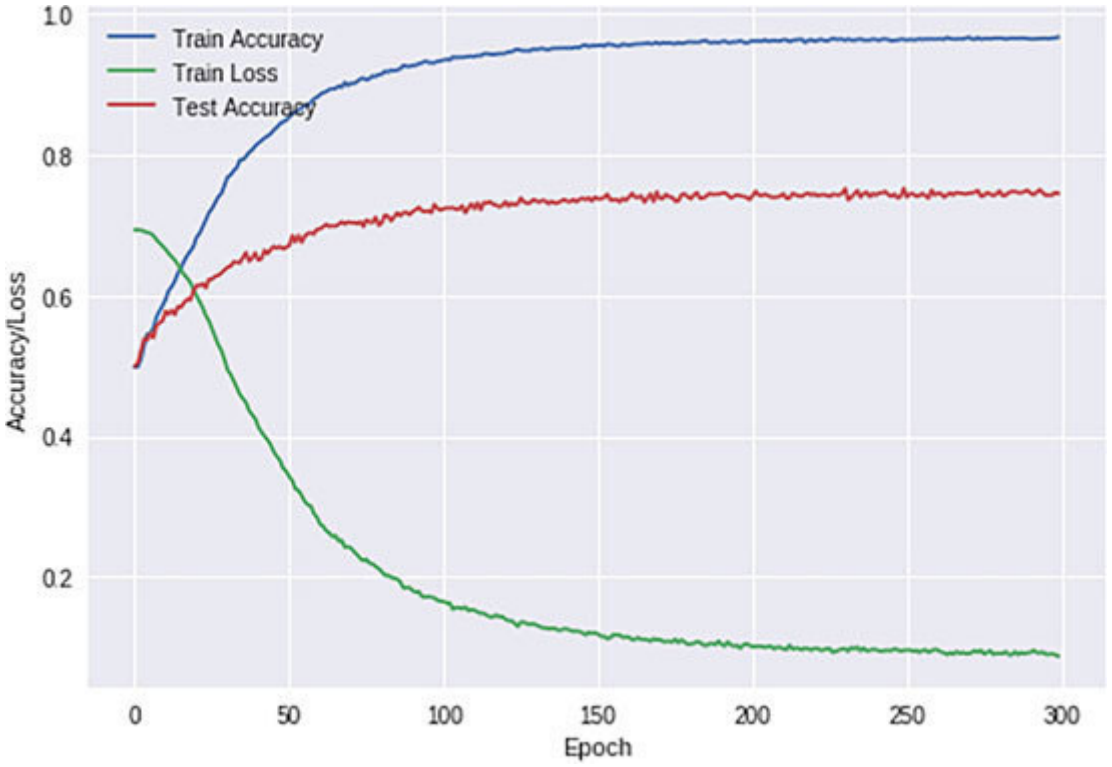


Figure 5.5: *Training progress with iterations.*

The entire code discussed earlier can be found in Ipython notebook as Ch5/using_word_level_cnn_for_text_classsification.ipynb.

The preceding research paper included the state-of-the-art techniques that improve on 4 out of 7 tasks, including sentiment analysis and question answering. For better understanding, you can try with following variations in the network:

Try by training embedding from scratch

With pre-trained embedding, try keeping the embedding trainable (you can keep pre-trained embeddings trainable by adding self.embedding.weight.requires_grad = True to the implementation)

You can refer to the following links:

Natural language processing (almost) from scratch: *http://www.jmlr.org/papers/volume12/collobert11a/collobert11a.pdf*

Large movie review dataset: *http://ai.stanford.edu/~amaas/data/sentiment/*

Using Character Level CNN

Earlier, we explored how to use the word-based features for the text classification. In this recipe, we will use character-based features to classify the text. Character-based features are very powerful and have many advantages over word-based features. The paper we plan to implement in this recipe was published as Character-level Convolutional Networks *for Text Classification* by Xiang Zhang and coworkers.

This network is a deep convolutional network with six convolution layers, followed by dense layers. Each convolution layer is The following schematic diagram illustrates the model:

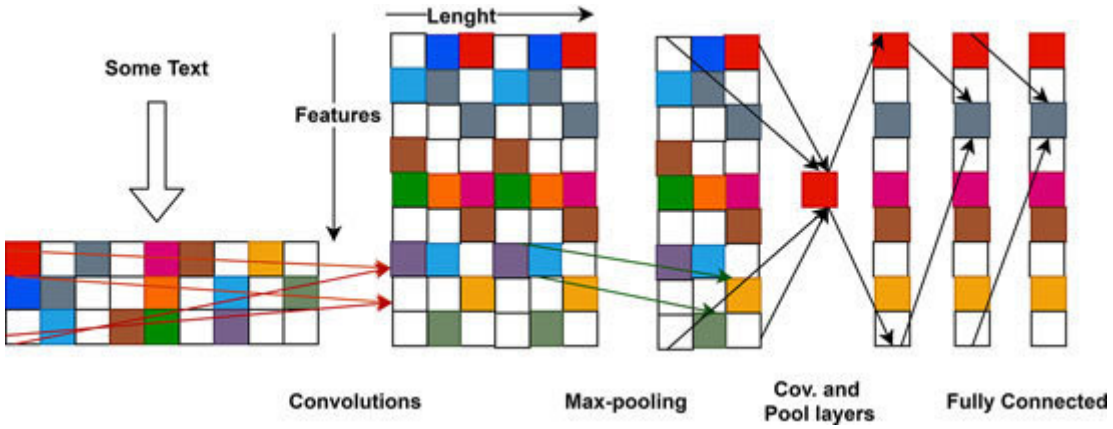


Figure 5.6: *An illustration of the model.*

Each convolution layer is followed by a ReLU convolution layer, and max-pool operation is applied to concentrate

features. In short, each convolution block looks as follows:

```
conv1 = nn.Sequential(
nn.Conv1d(in_channels=self.config.vocab_size,
out_channels=self.config.num_channels, kernel_size=7),
nn.ReLU(),
nn.MaxPool1d(kernel_size=3)
)
```

To construct an individual block, we used the PyTorch function, which helps keep the network tidy and easy to understand. In the larger network, each network sub-block is designed as and all these sub blocks are added to form the entire network. is a container, and the module is executed in the order they are stacked in the constructor. You can also pass the ordered dictionary to the module. An example of both approaches is given as follows:

```
# Constructing Sequential block with Stacking
model = nn.Sequential(
nn.Conv2d(1,20,5),
nn.ReLU(),
nn.Conv2d(20,64,5),
nn.ReLU()
)
# Constructing Sequential block with OrderedDict
model = nn.Sequential(OrderedDict([
('conv1', nn.Conv2d(1,20,5)),
('relu1', nn.ReLU()),
('conv2', nn.Conv2d(20,64,5)),
```

('relu2', nn.ReLU())
]))

The character representation for this module includes fixing character vocabulary like an English text that can have the following characters: **abcdefghijklmnopqrstuvwxyz0123456789, ;** All other characters are ignored. A maximum length of the sentence or document is fixed. In the paper, the max character length was fixed to be 1014. Considering the size of our dataset, the max character length is fixed at 300 in our case. The original model's configuration for the convolution layer with various kernel size is given as follows:

follows:
follows:
follows:
follows:
follows:
follows:
follows:

Table 5.1

On the other hand, in our case, the preceding configuration is slightly changed to converge on the smaller dataset. The configuration of our network as follows:

follows:
follows:
follows:
follows:
follows:
follows:
follows:

Table 5.2

Followed by convolutional blocks, the network has three dense layers, each with Relu activations, and dropout is applied after each dense layer. The model is trained with stochastic gradient descent, and the learning rate will be half after every three epochs.

Understanding Character Representation

To demonstrate character-based text classification, we will take the AgNews dataset for the supervised learning task. *AGNews* is a collection of more than 1 million news articles collected from over 2000 news sources. Ag News corpus can be downloaded from After downloading this corpus, one may require to extract the data from an XML format. To avoid these additional steps, check a cleaner version of the ag news data set in the Ch5/data The dataset is divided into train and test split, referred as Ch5/data/ag_news.test and Ch5/data/ag_news Train are kept in ready to use format.

Character representation: The code snippet generates character representation. It has two main functions. The first is which houses the vocabulary set and defines various limits like the length of data, unique labels, and unique character in the text. The second function is getitem, which constructs the character-based feature matrix:

```
class MyDataset(Dataset):
"""
preparing 2D character array from the text
"""
def __init__(self, data_path, config):
"""
Defining character set
```

```
“””
self.config = config


self.vocabulary =
list(“””abcdefghijklmnopqrstuvwxyz0123456789,;.!?:’\”/\\|_@#$%
^&*~’+-=()[]{}”””)
self.identity_mat = np.identity(len(self.vocabulary))
data = get_pandas_df(data_path)
self.texts = list(data.text)
self.labels = list(data.label)
self.length = len(self.labels)
def __len__(self):
return self.length
def __getitem__(self, index):
raw_text = self.texts[index]
data = np.array([self.identity_mat[self.vocabulary.index(i)] for i in
list(raw_text) if i in self.vocabulary],
dtype=np.float32)
if len(data) >self.config.max_len:
data = data[:self.config.max_len]
elif 0
data = np.concatenate(
(data, np.zeros((self.config.max_len - len(data),
len(self.vocabulary)), dtype=np.float32)))
elif len(data) == 0:
data = np.zeros((self.config.max_len, len(self.vocabulary)),
dtype=np.float32)
label = self.labels[index]
return data, label
```

The character-based feature matrix with 64 unique characters with sentence/document max length equals 300 is shown in the following figure. The position in the matrix where a particular character and its index in sentence/document is marked as 1, else all indices are kept zero:

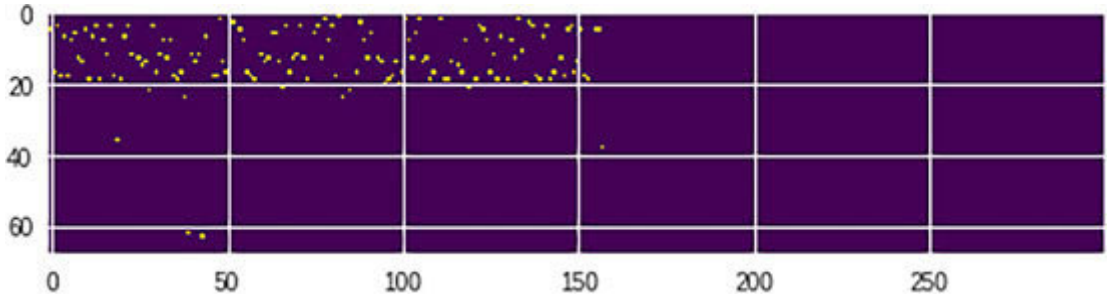


Figure 5.7: *An illustration of how the character is given as features. X is the position of character in the sentence, and y is the location of the character as one hot encoding.*

From the image, we have taken max length of the sentence to be 300. Each character can be one of the 64 different types of predefined characters. If the character is present at the given index in the sentence, the position of the character is marked as 1, and all the others remain at zero. In the figure, the presence of the character in any particular location is shown in Yellow (1).

Network Architecture

A discussed earlier, the network has 6 convolution layers, and each layer is constructed using the module. The output frame length after the last convolutional layer and before any of the fully-connected layers) is. This number multiplied with the frame size at layer 6 produces the input dimension that will be compatible with the first fully-connected layer accepts. Followed by the convolution layer, there are three fully connected layers that are finally converged into several classes:

```
class CharCNN(nn.Module):
def __init__(self, config):
super(CharCNN, self).__init__()
self.config = config
conv1 = nn.Sequential(
nn.Conv1d(in_channels=self.config.vocab_size,
out_channels=self.config.num_channels, kernel_size=7),
nn.ReLU(), nn.MaxPool1d(kernel_size=3))
**Such six convolution block conv1, conv2, conv3, conv4,
conv5, conv6**
conv_output_size = self.config.num_channels *
((self.config.max_len - 96) // 27)
linear1 = nn.Sequential(
nn.Linear(conv_output_size, self.config.linear_size),
.ReLU(),
```

```
nn.Dropout(self.config.dropout_keep)
)
**Such 2 more block linear2, linear3**
self.convolutional_layers =
nn.Sequential(conv1,conv2,conv3,conv4,conv5,conv6).cuda()


self.linear_layers = nn.Sequential(linear1, linear2, linear3).cuda()
def forward(self, embedded_sent):
_sent = embedded_sent.transpose(1,2)#.permute(0,2,1) #
shape=(batch_size,embed_size,max_len)
conv_out = self.convolutional_layers(embedded_sent)
conv_out = conv_out.view(conv_out.shape[0], -1)
linear_output = self.linear_layers(conv_out)
return linear_output
def reduce_lr(self):
for g in self.optimizer.param_groups:
g['lr'] = g['lr'] / 2
print("Reducing Learning Rate to: ", g['lr'])
```

When the network is trained for some epochs, accuracy for train and test set increases gradually, and loss decreases:

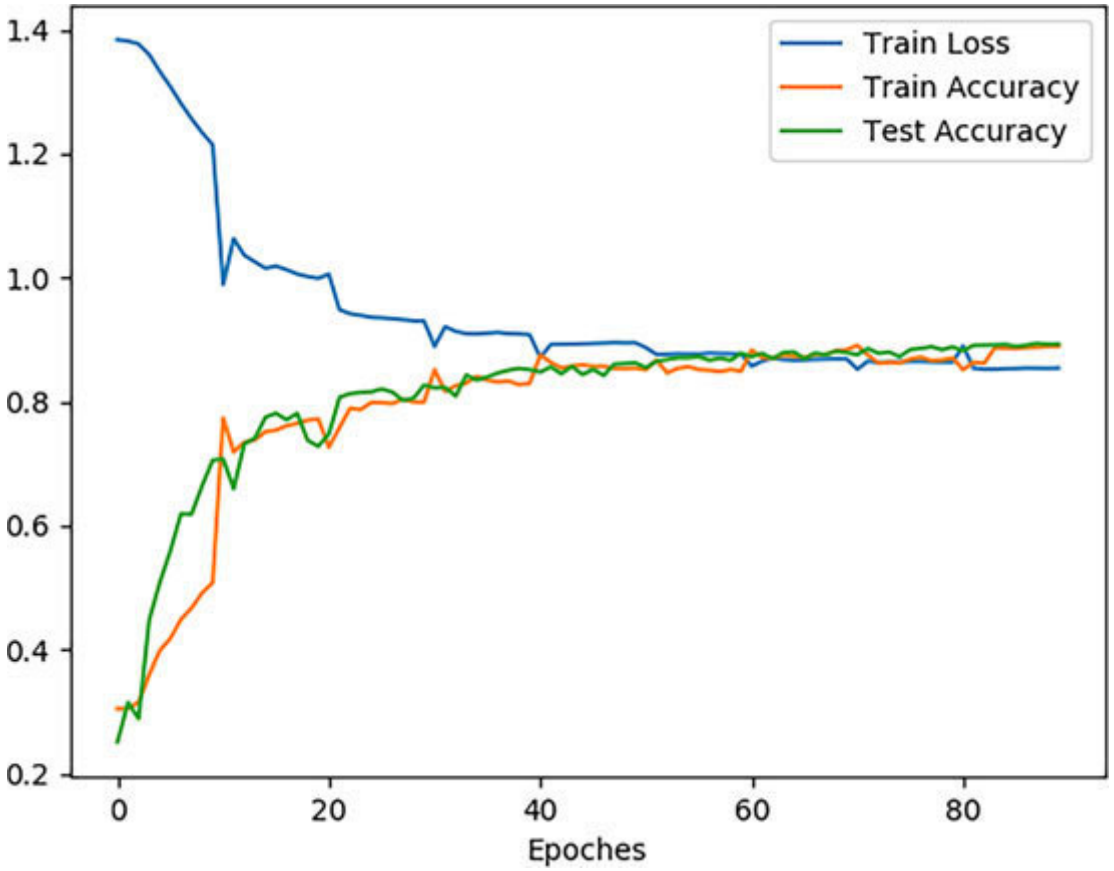


Figure 5.8: *Plotting performance of character-level CNN on text clarification task.*

With the preceding implementation, we finally managed to get over 80% accuracy for the test and training datasets. Entire code with supporting function for text processing, training, and validation can be found in the Jupyter notebook at Ch5/using_character_level_cnn.ipynb.

The preceding network is great, but it is not optimized for our small dataset; maybe we are using excessive layers in their network. Try to cut out some of the layers from this network to decrease the training parameters and see if the network converges. Try to cut as many layers as possible, and

find out the lowest possible configuration that can efficiently work on our data. Usually, this exercise to minimize the network size is done once the network is found to be stable and converging on the given dataset. Lowering the parameters of the network brings the network to a simpler form, helping avoiding high variance and overfitting-related problems.

You can refer to the following links:

Character-level convolutional networks for text classification: *https://arxiv.Org/pdf/1509.01626.Pdf*

Words vs. character n-grams for anti-spam filtering: *http://www.Icsd.Aegean.Gr/lecturers/stamatatos/papers/ijait-spam.Pdf*

Using Very Deep Convolution Network

At the beginning of this chapter, we saw that the use of CNN is catching up in text processing. To help you understand the importance of the deeper architecture, here's another example whereby we will use a very deep convolution neural network for text classification. In this recipe, we will understand and implement the work reported in the research paper *Very Deep Convolutional Networks for Text Classification* by Alexis Conneau and coworkers working with Facebook AI research. This paper claims that with 29 layers deeper network, the model can beat previously reported state-of-the-art techniques.

The deep convolution network goes up to 49 layers deep, and state-of-the-art configuration can be achieved on text classification tasks by going deeper up to 29 convolutional layers. This model is for text classification, and we particularly chose this model for this recipe. There is a stronger reason to select this model—it is organized into blocks; each block repeats and has an optional shortcut connection between the blocks. This model will provide a sense of understanding of how modern networks are going deeper by modifying the traditional architecture. In the next recipe, we will go one step further and understand the various types of changes in the network that promises training beyond 100 layers. The entire network looks as in the following diagram:

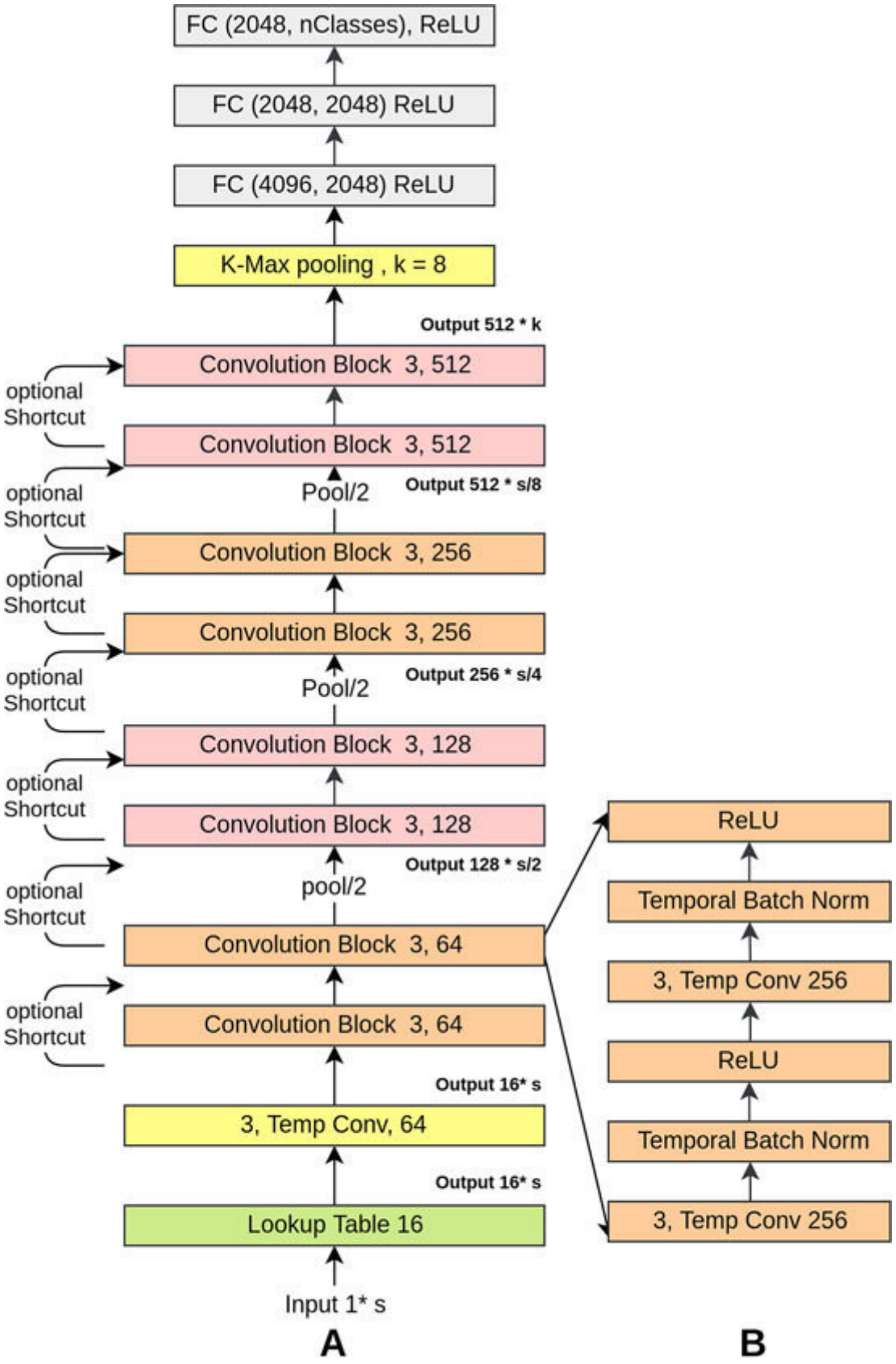

FC (2048, nClasses), ReLU
FC (2048, 2048) ReLU
FC (4096, 2048) ReLU
K-Max pooling , k = 8
Output 512 * k
Convolution Block 3, 512
optional Shortcut
Convolution Block 3, 512
optional Shortcut
Pool/2
Output 512 * s/8
Convolution Block 3, 256
optional Shortcut
Convolution Block 3, 256
optional Shortcut
Pool/2
Output 256 * s/4
Convolution Block 3, 128
optional Shortcut
Convolution Block 3, 128
optional Shortcut
pool/2
Output 128 * s/2
Convolution Block 3, 64
optional Shortcut
Convolution Block 3, 64
optional Shortcut
Output 16* s
3, Temp Conv, 64
Output 16* s
Lookup Table 16
Input 1* s
A
ReLU
Temporal Batch Norm
3, Temp Conv 256
ReLU
Temporal Batch Norm
3, Temp Conv 256
B

Figure 5.9: *Architecture of the very deep convolutional networks for text classification as designed by Alexis Conneau and coworkers.*

The model takes character-based encoding as input. Let's say if our set has 1024 unique characters and we consider max sentence length to be 64; then, the input to the model will be [batch_size, 64, 1024]. This shape is then converted to [batch_size, 1024, 16] by applying embedding to input. Embeddings are shown as a lookup operation in the preceding diagram. A convolution 1D with a filter size of 3 is applied to the output generated by the embedding layer with input dimension=16 and output dimension = 64. The output of a 1D convolution is passed to the convolution block, which has the following layers:

A 1D convolution layer

A batch normalization layer follows each convolution layer

Relu activation is applied to the output of the batch normalization layer

The above-mentioned layers are repeated twice to provide output

If the residual flag is the input given to this block is added to the output (a residual connection)

This convolution block is repeated with different input channels and output channels. Blocks have and 512 input channels and output channels in the entire network. Depending on the depth of the network, different blocks are used in different numbers. Depending on the depth, blocks are repeated in the following pattern: **

** ** ** ** ** ** **

**
**
**
**

Table 5.3

In the next section, we will learn how to design such a network with changing depth according to applied parameters.

Batch normalization: It is a technique to normalize the internal structure of the data for faster training. Min-max normalization can be taken as simple normalization technique *and* can be mathematically given as:

$$x_{new} = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

It normalizes the gen range between 0 and 1. Similarly, batch norm is the normalization of the output in the hidden layer and can be mathematically given as:

Input: Value of *x* over a mini batch: B =

Parameters to be learned γ, β

$$\text{Output: } y_i = BN_{y,\beta}(x_i)$$

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \#min - batchmean$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \quad \#min - batchVariance$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B^2 + \epsilon} \#normalize$$

$$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i) \quad \#shift$$

BatchNorm has the following benefits:

Network trains As the additional layer is added and additional parameters are added to the backpropagation graph, the iteration will be slower. That said, the network will converge faster.

Allows higher learning SGD requires small learning rates but deeper networks face the problem of vanishing gradients with lower learning rates, so they will take a longer time to train such networks with lower learning rates. Batch normalization allows training the network with a higher learning rate.

Makes weight easier to Weight initialization is tricky for deeper networks. Batch normalization makes the network relatively insensitive to initial weights, offering better convergence.

Makes activation function more Sigmoid loses the gradient quickly, and ReLu dies out in the deeper network. As the normalized input is given to the activation after passing through the batch norm, it solves the activation die-out problem.

Simplifies the creation of deeper Over 4 points if taken care, it offers faster and better training of deeper networks.

Provides some Batch norm also adds noise to the network and has some regularization effect.

The Convolution Block

Convolution block is defined with the ConvBlock function. The following network is common for each convolution block shown in the preceding diagram. The convblock function has the following layers:

A batch normalization layer

A convolution 1D layer

An activation function

These layers were repeated twice and followed by residual addition. If the shortcut parameter to the init method of this class is True, the input to this block is added to the output after all the layers. In the PyTorch batch, normalization can be simply used as follows:

```
batchnorm1 = nn.BatchNorm1d(n_filters)
output = batchnorm1(input)
```

nn.BatchNorm1d is applied to 1D input, and likewise, PyTorch has an implementation for 2D and 3D shapes as nn. BatchNorm2d and BatchNorm3d, respectively. BatchNorm is always applied after the activation function:

```
class ConvBlock(nn.Module):
def __init__(self, input_dim=128, n_filters=256, kernel_size=3,
padding=1, stride=1, shortcut=False, downsampling=None):
super(ConvBlock, self).__init__()
self.downsampling = downsampling
self.shortcut = shortcut
self.conv1 = nn.Conv1d(input_dim, n_filters,
kernel_size=kernel_size, padding=padding, stride=stride)
self.batchnorm1 = nn.BatchNorm1d(n_filters)
self.relu1 = nn.ReLU()


self.conv2 = nn.Conv1d(n_filters, n_filters,
kernel_size=kernel_size, padding=padding, stride=stride)
self.batchnorm2 = nn.BatchNorm1d(n_filters)
self.relu2 = nn.ReLU()
def forward(self, input):
residual = input
output = self.conv1(input)
output = self.batchnorm1(output)
output = self.relu1(output)
output = self.conv2(output)
output = self.batchnorm2(output)
if self.shortcut:
if self.downsampling is not None:
residual = self.downsampling(input)
output += residual
output = self.relu2(output
return output
```

Understanding the Network

This network is constructed a little differently than we did until now. To construct the network, an empty list is taken as layers = and all the required layers according to the specified depth are appended to this list. For example, if the network has a depth of 9, 1 ConvBlock, each having input and output size equal to and 512 are added. As the depth increases, the variable number of such blocks is considered, and the network is constructed accordingly. This method is good for the network with variable layers, and the architecture changes with the selection of parameters. At the end, the list layers with all the required layers that need to be included in the network are added to the Similarly, the fully connected layers at the end of the network are constructed:

```
class VDCNN(nn.Module):
def __init__(self, n_classes=2, num_embedding=69,
embedding_dim=16, depth=9, n_fc_neurons=2048,
shortcut=False):
super(VDCNN, self).__init__()

layers = []
fc_layers = []
base_num_features = 64
```

```
self.embed = nn.Embedding(num_embedding, embedding_dim,
padding_idx=0, max_norm=None,
norm_type=2, scale_grad_by_freq=False, sparse=False)
layers.append(nn.Conv1d(embedding_dim, base_num_features,
kernel_size=3, padding=1))

if depth == 9:
num_conv_block = [0, 0, 0, 0]
elif depth == 17:

num_conv_block = [1, 1, 1, 1]
elif depth == 29:
num_conv_block = [4, 4, 1, 1]
elif depth == 49:
num_conv_block = [7, 7, 4, 2]

layers.append(ConvBlock(input_dim=base_num_features,
n_filters=base_num_features, kernel_size=3, padding=1,
shortcut=shortcut))
for _ in range(num_conv_block[0]):
layers.append(ConvBlock(input_dim=base_num_features,
n_filters=base_num_features, kernel_size=3, padding=1,
shortcut=shortcut))
layers.append(nn.MaxPool1d(kernel_size=3, stride=2, padding=1))

ds = nn.Sequential(nn.Conv1d(base_num_features, 2 *
base_num_features, kernel_size=1, stride=1, bias=False),
nn.BatchNorm1d(2 * base_num_features))
layers.append(
ConvBlock(input_dim=base_num_features, n_filters=2 *
base_num_features, kernel_size=3, padding=1,
```

```
shortcut=shortcut, downsampling=ds))
for _ in range(num_conv_block[1]):
layers.append(
ConvBlock(input_dim=2 * base_num_features, n_filters=2 *
base_num_features, kernel_size=3, padding=1,
shortcut=shortcut))
layers.append(nn.MaxPool1d(kernel_size=3, stride=2, padding=1))



ds = nn.Sequential(nn.Conv1d(2 * base_num_features, 4 *
base_num_features, kernel_size=1, stride=1, bias=False),
nn.BatchNorm1d(4 * base_num_features))
layers.append(
ConvBlock(input_dim=2 * base_num_features, n_filters=4 *
base_num_features, kernel_size=3, padding=1,
shortcut=shortcut, downsampling=ds))
for _ in range(num_conv_block[2]):
layers.append(
ConvBlock(input_dim=4 * base_num_features, n_filters=4 *
base_num_features, kernel_size=3, padding=1,
shortcut=shortcut))
layers.append(nn.MaxPool1d(kernel_size=3, stride=2, padding=1))


ds = nn.Sequential(nn.Conv1d(4 * base_num_features, 8 *
base_num_features, kernel_size=1, stride=1, bias=False),
nn.BatchNorm1d(8 * base_num_features))
layers.append(
ConvBlock(input_dim=4 * base_num_features, n_filters=8 *
base_num_features, kernel_size=3, padding=1,
shortcut=shortcut, downsampling=ds))
for _ in range(num_conv_block[3]):
```

```
layers.append(
ConvBlock(input_dim=8 * base_num_features, n_filters=8 *
base_num_features, kernel_size=3, padding=1,
shortcut=shortcut))

layers.append(nn.AdaptiveMaxPool1d(8))
fc_layers.extend([nn.Linear(8 * 8 * base_num_features,
n_fc_neurons), nn.ReLU()])

fc_layers.extend([nn.Linear(n_fc_neurons, int(n_fc_neurons/2)),
nn.ReLU()])
fc_layers.extend([nn.Linear(int(n_fc_neurons/2), n_classes)])

self.layers = nn.Sequential(*layers)
self.fc_layers = nn.Sequential(*fc_layers)
self.__init_weights()

def __init_weights(self):
for m in self.modules():
if isinstance(m, nn.Conv1d):
kaiming_normal_(m.weight, mode='fan_in', nonlinearity='relu')

def forward(self, input):

output = self.embed(input)
output = output.transpose(1, 2)
output = self.layers(output)
output = output.view(output.size(0), -1)
output = self.fc_layers(output)
torch.softmax(output, dim=1)
```

return output

I used the network with nine layers on the IMDB movie review data. The IMDB movie review dataset or the Large Movie Review Dataset has 25,000 movie reviews with binary labels. DeepConv gives stunning results on this dataset. On the train set, t already achieved 99% accuracy, and it achieved 80% accuracy on the test set. The accuracy/loss vs. epoch plot is illustrated here:

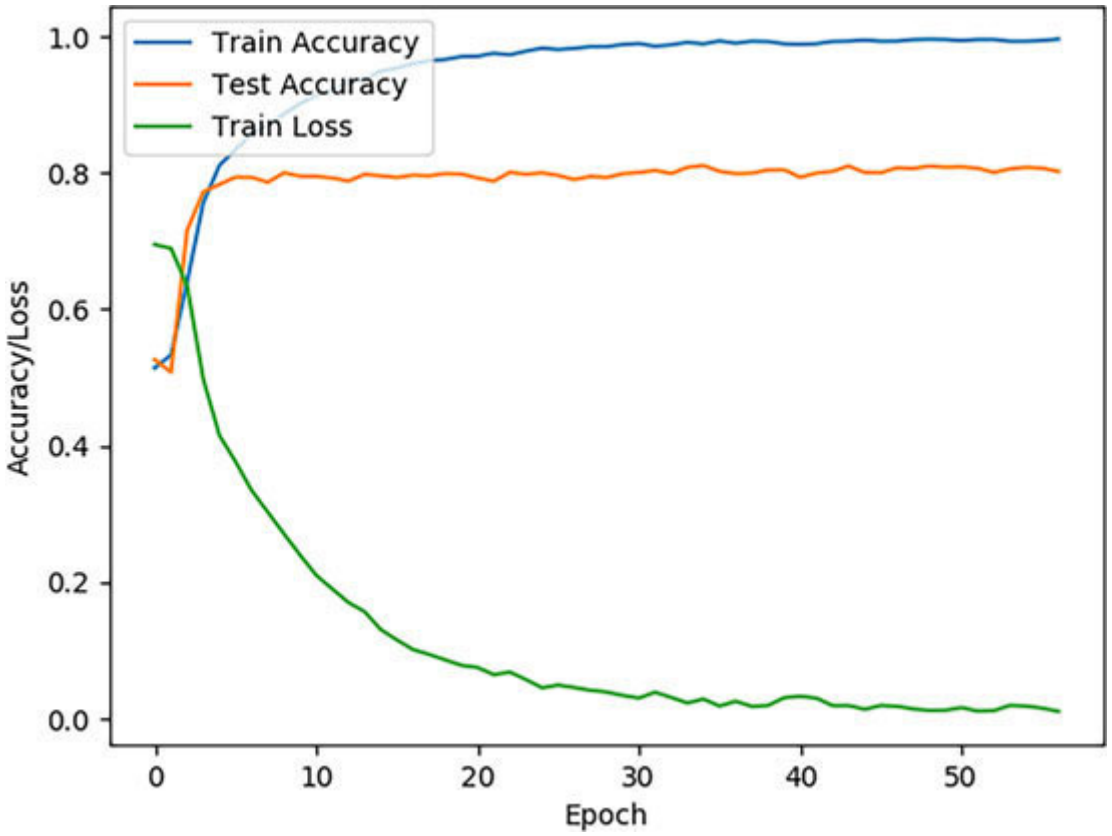


Figure 5.10: *Increase in train and test accuracy and decrease in train loss with epoch when deep convolution network is trained on the IMDB movie review data.*

The implementation of the deepConv for text classification is given at

This network is built such that it can be modified to obtain better convergence. You can try different depths and which one you get the highest results at.

Compare the convergence with the residual layer when it is applied, versus when the residual layer is not applied.

Compare the convergence with batch normalization it is applied versus when batch normalization is not applied. Also compare results when batch normalization is kept before and after the activation function.

Chop out this network by commenting on different blocks and check the minimum configuration required to get the best results on the given data.

You can refer to the following links:

Batch normalization: Accelerating deep network training by reducing internal covariate shift: *https://arxiv.org/abs/1502.03167*

Very deep convolutional networks for text classification: *https://arxiv.org/pdf/1606.01781.pdf*

Learning word vectors for sentiment analysis: *https://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf*

Training Deeper Networks

When it comes to the network design, the direct indication is, the more, deeper, and better it is. A few years ago, the best networks had around 10-12 layers, whereas they are going hundreds of layers deeper nowadays. Stacking hundreds of layers and training such a deeper network on powerful GPU just doesn't work. This type of network requires architecture changes. These architectural changes can be better understood by understanding successful deeper architecture like HighwayNet, DenseNet, and ResNet. In this recipe, we will discuss Highway network in detail and correlate the learning with the architecture of DenseNet and ResNet.

The first question is, why does going deeper by just the stacking layer not work? In a smaller network with less than ten layers, stacking the layers one over another and training such a network might work. Such a model's performance degrades exponentially when more layers are added. The root of performance degradation lies at backpropagation operation. When such a model is trained with back-propagation, the gradient diminishes after each layer. When such a gradient reaches the very first layer, it diminishes such that the weight for this layer cannot be updated. Due to this, learning cannot be imparted to the first layer. This is also referred to as the vanishing gradient problem.

ResNet

We have figured the problem with the deeper network, so we can now understand the measures to solve this problem. The first candidate that tries to solve the vanishing gradient problem and move deeper is ResNet. A ResNet has 34 layers, and each layer is an identical block connected to the previous and the next layer:

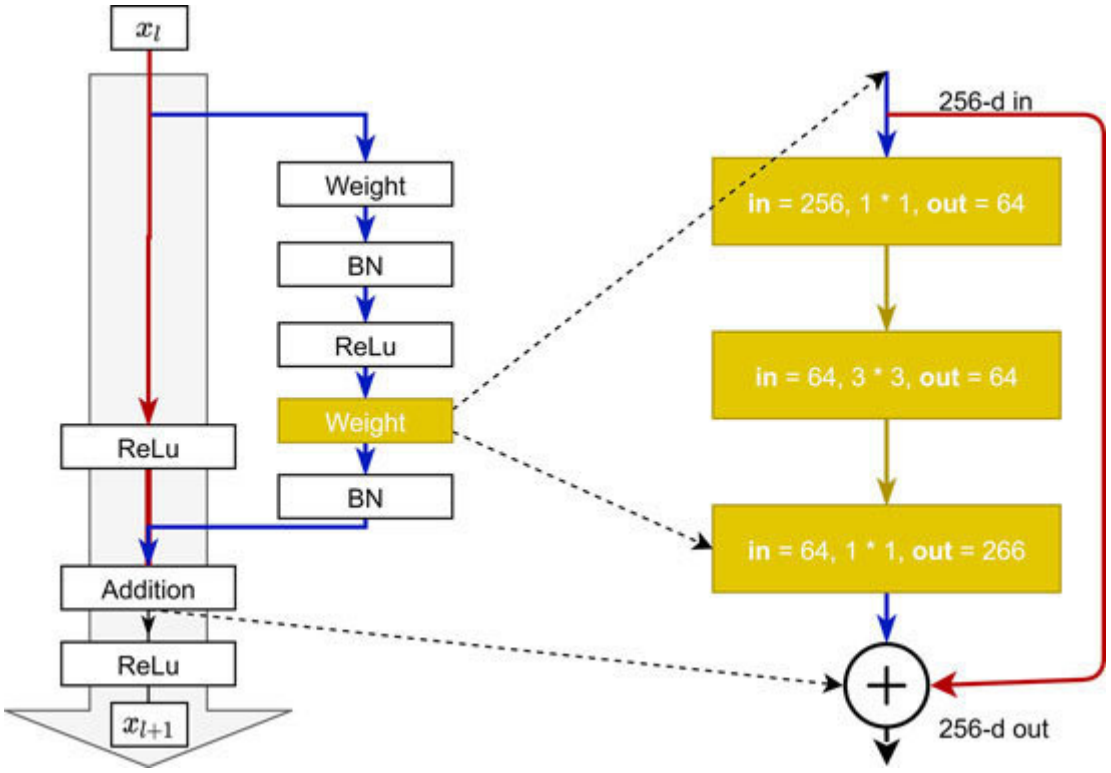


Figure 5.11: *ResNet block*

The branch in blue is where convolution operations are applied, and the branch where only ReLu activation is applied

and merged with the first branch's output.

Residual connections connect each of these. A network without residual block has a stacked-layer, and it follows $y =$ where y is the output, and can be any function involving convolution, dense layers, and non-linearity. In the backpropagation phase, these operations, along with non-linearity, resist the flow of the gradient, leading to the vanishing gradient problem. A network with residual block has $y = +$ where the addition x is a kind of shortcut that allows the backpropagation signals to flow backward without any hindrance.

Highway Network

The highway network was proposed by researchers working at The Swiss AI Lab Istituto DalleMolle di Studisull' Intelligenza Artificiale including Jurgen Schmidhuber, who is an inventor of the Long Short Term Memory (LSTM). Highway network claims to optimize 900 layers deep network with stochastic gradient descent and momentum. It is observed that network parameters optimization with highway network is independent of depth. At the same time, a plain network without a highway suffers greatly with increasing layers. Highway networks take the concept of ResNet one step ahead by introducing learnable parameters into the shortcut. Learning function H with input weight producing output y can be defined as:

$$Y = \bullet$$

Now, we introduce two gates to the preceding function: a transform gate and a carry gate The resultant equation can be defined as:

$$y = \cdot + x \cdot$$

For simplicity in this paper, the author takes an assumption— C = Finally, the layer in the highway network is defined as

follows:

$$y = \bullet + x \bullet (1 -$$

DenseNet

The dense layer takes the thought process of the Highway network one step ahead. The Highway network states that the skip connection from the previous layer to the next layer improves performance. In DenseNet, a layer is connected to all its previous layers by skip connections. In this way, information always has a direct route backward in the back propagation. Here, in the dense network, the layer receives direct input from all its previous layers. Here, ..., refers to the concatenation of the feature-maps produced in layers o, ..., Due to this dense connectivity, the network is known as DenseNet.

We saw three networks with somewhat similar approaches to facilitate deep architectures. Now, we will explore minimal implementation of these networks using PyTorch. For simplicity, we will implement a single block for each network.

Fundamental Block of ResNet

ResNet has two main functions to construct the working ResNet model. One function is named which constructs a block of the 3*3 convolution2D layer, along with batch normalization and Relu as an activation function. The schematic code function is as shown:

```
class ResidualBlock(nn.Module):
def __init__(self, in_channels, out_channels, stride=1,
downsample=None):
super(ResidualBlock, self).__init__()
self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=stride, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(out_channels)
sself.relu = nn.ReLU(inplace=True)
self.conv2 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=stride, padding=1, bias=False)
self.bn2 = nn.BatchNorm2d(out_channels)
self.downsample = downsample
def forward(self, x):
residual = x
out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)
out = self.conv2(out)
out = self.bn2(out)
```

```
if self.downsample:
residual = self.downsample(x)
out += residual
out = self.relu(out)

return out
```

Another function is ResNet, which takes ResidualBlock blocks and connects them into one network, as shown below. ResNet has a make_layer definition that takes these layers and stacks them with one another to residual connection:

```
class ResNet(nn.Module):
def __init__(self, block, layers, num_classes=10):
super(ResNet, self).__init__()
self.in_channels = 16
self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
self.bn = nn.BatchNorm2d(16)
self.relu = nn.ReLU(inplace=True)
self.layer1 = self.make_layer(block, 16, layers[0])
self.layer2 = self.make_layer(block, 32, layers[1], 2)
self.layer3 = self.make_layer(block, 64, layers[2], 2)
self.avg_pool = nn.AvgPool2d(8)
self.fc = nn.Linear(64, num_classes)
def make_layer(self, block, out_channels, blocks, stride=1):**
downsample = None
if (stride != 1) or (self.in_channels != out_channels):
downsample = nn.Sequential(conv3x3(self.in_channels,
out_channels, stride=stride),nn.BatchNorm2d(out_channels))
layers = []
```

layers.append(block(self.in_channels, out_channels, stride, downsample))
self.in_channels = out_channels
for i in range(1, blocks):
layers.append(block(out_channels, out_channels))

return nn.Sequential(*layers)
def forward(self, x):
out = self.conv(x)
out = self.bn(out)
out = self.relu(out)
out = self.layer1(out)
out = self.layer2(out)
out = self.layer3(out)
out = self.avg_pool(out)
out = out.view(out.size(0), -1)
out = self.fc(out)
return out
model = ResNet(ResidualBlock, [2, 2, 2]).to(device)

Fundamental Block of Highway Network

The highway network can be implemented very easily, as shown in the following code block. Each layer will have a highway with learnable parameters can be simply represented as x = gate * nonlinear + (1 - gate) * linear

```
class Highway(nn.Module):
def __init__(self, size, num_layers, f):
super(Highway, self).__init__()
self.num_layers = num_layers
self.nonlinear = nn.ModuleList([nn.Linear(size, size) for _ in
range(num_layers)])
self.linear = nn.ModuleList([nn.Linear(size, size) for _ in
range(num_layers)])
self.gate = nn.ModuleList([nn.Linear(size, size) for _ in
range(num_layers)])
self.f = f
def forward(self, x):
“””
:param x: tensor with shape of [batch_size, size]
:return: tensor with shape of [batch_size, size]
applies σ(x) Θ (f(G(x))) + (1 - σ(x)) Θ (Q(x)) transformation |
G and Q is affine transformation,
f is non-linear transformation, σ(x) is affine transformation
with sigmoid non-linearition and ⊙ is element-wise
multiplication
```

```
“””
for layer in range(self.num_layers):
gate = F.sigmoid(self.gate[layer](x))
nonlinear = self.f(self.nonlinear[layer](x))


linear = self.linear[layer](x)
x = gate * nonlinear + (1 - gate) * linear
return x
```

DenseNet

Dense network implementation is beyond the scope of this book. DenseNet is mainly used for image-related uses cases. To know more about how dense net is implemented, visit

These concepts are very new, and we have not implemented ResNet, highway Network, and DenseNet to keep this book focused. However, we will learn about ELMo embedding that uses highway networks in *Chapter 6, Accelerating NLP with Transfer* You can try developing ResNet and DenseNet in PyTorch; this will greatly help you in reinforcing your convolution network implementation skills. There are many working examples available on GitHub, and these resources can help you when you are stuck.

You can check out the following links:

Deep residual learning for image recognition: *https://arxiv.org/pdf/1512.03385.pdf*

Highway networks: *https://arxiv.org/pdf/1505.00387.pdf*

Densely connected convolutional networks: *https://arxiv.org/pdf/1608.06993.pdf*

Conclusion

This chapter brought some new experiences. You may have known convolutional neural network as the technique to process images and video before starting this chapter, but your perspective might have changed now. In this chapter, we understood the basics of CNN, and we also applied it to tasks like text classifications. We saw that there is no limit to the point where CNN can be used in NLP. We applied CNN at the character and word-level, and we also implemented a very deep convolutional network for text classification. CNN is advanced as compared to the RNN networks. CNN considered to be embarrassingly parallelizable and fully utilize power of parallel processors like GPU. On the other hand, RNN is recurrent, so sequential input is required so it utilize GPU like device to a comparatively lesser extent.

In the next chapter, we will learn about accelerating NLP with transfer.

CHAPTER 6

Accelerating NLP with Transfer Learning

This chapter will help us apply our previous learnings to various NLP tasks. Here, we will cover topics like sentiment analysis, topic modeling, text generator, named entity recognition, language transliteration, and text summarization. For each of the topics, various advanced models with CNN and LSTM components are deployed. We will also understand how to achieve the state-of-the-art results.

Structure

The following recipes will be covered in this chapter:

Introduction

Understanding the transformer

Converting sentence to vector

Getting to know contextual vectors

Training supervised embedding

Understanding and using BERT

Objective

Discuss more recent advancements like a transformer, character-based contextual embedding like ELMo, and sentence vectorization techniques like SkipThought and InferSent.

Coding required fragments of each model.

Forming a complex model that we will discuss in the next chapter.

Pre-requisites

The codes for this chapter can be found in the Ch6 folder at GitHub repository To understand this chapter, you must have some basic knowledge about the following Python packages:

Gensim

NLTK

NumPy

Torch

Matplotlib

SciPy

You can install these requirements by installing all the packages listed in requirements.txt simply by issuing pip install -r

Introduction

This chapter explains an important aspect of the NLP, i.e., using the pre-trained model. The pre-trained model is used just like image processing. It can also be considered a method to perform transfer learning in NLP, just like the same concept exists in image processing. The language task can be anything like classification, language translation, summarization, named entity recognition, and such. Each task starts with textual data relevant to the task. This data is then converted to word level or sentence level vectors using various models like ELMo-Bilm, BERT, SkipThought, Transformer, InferSent, etc. These dense vectors are then processed downward with various layers like LSTM or CNN or dense layers to produce the required output. The following is an illustration of how to use transfer learning:

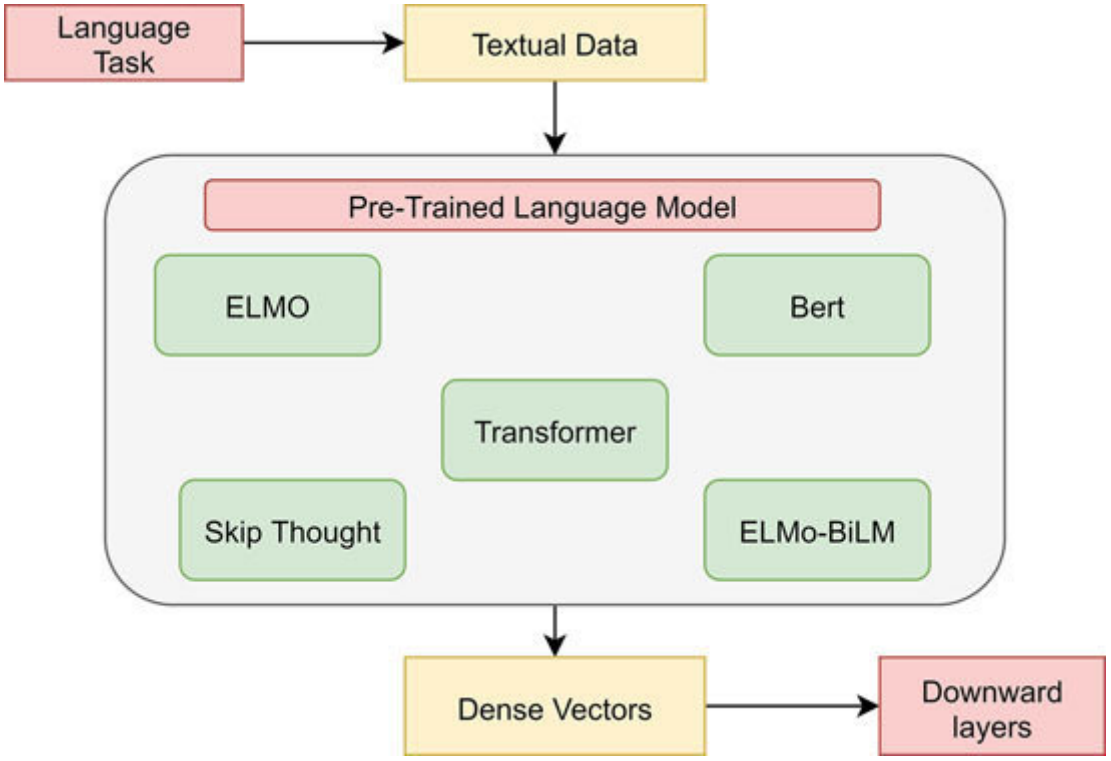


Figure 6.1: *Transfer learning in the NLP.*

In this chapter, all the models have an average parameter size in tens of millions. Training these models from scratch requires enormous computing power, so the training of such models is usually done using multiple GPUs or TPUs. Such models are already trained on very large datasets, and they adapt to the new domain by pre-training on a very small amount of domain-specific data. In this entire chapter, we will use the pre-trained model to get word/sentence vectors. In some cases, we will do pre-training. These embeddings can be used for downward tasks like classification, summarization, or translation.

Understanding the Transformer

In *Chapter 4, Using RNN for* we understood and used the attention mechanism to increase the accuracy of the translation model. That said, the model was based on the recurrent unit and was slow. The transformer uses the concept of attention and provides faster implementation with slight modification. The transformer model was also found to beat state-of-the-art language translation and summarization techniques in many tasks. The transformer was proposed in the paper "Attention is all you need" by the Google brain team.

Before we go ahead, I would like to give you a rough understanding of what the transformer does with the following diagram. The transformer, as is understood by its name, is used to transform one form into another:

Figure 6.2: *The overall idea of the working of transformer.*

Internally, the transformer is an encoder-decoder model. Encoder and decoder are made up of repeating subblocks, as shown in the following diagram:

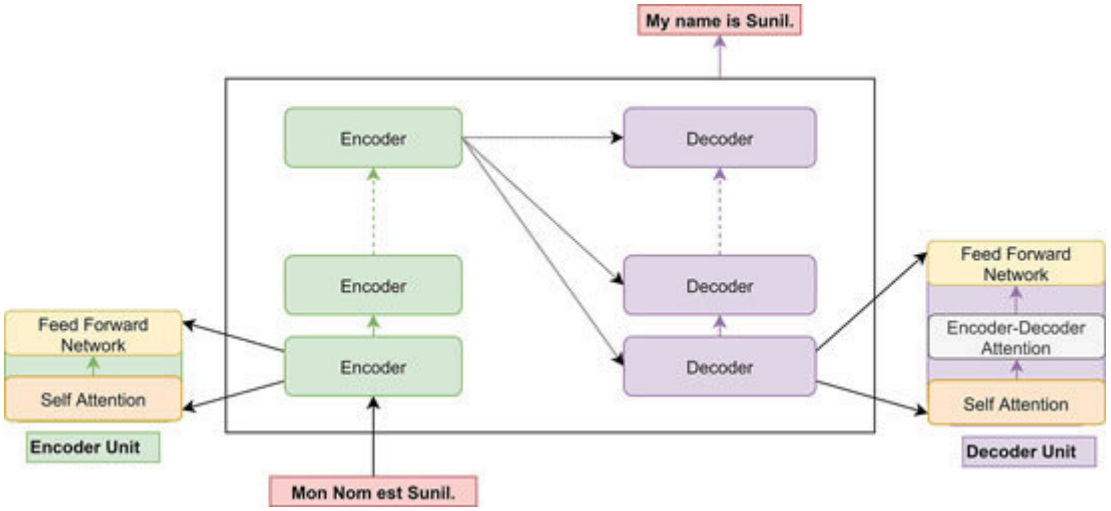


Figure 6.3: *Encoder-decoder arrangement in transformer.*

The encoder is identical in structure with all others. An encoder is made up of two sub-components: a self-attention layer and the feedforward network in it. At the same time, the decoder is made up of three components: a feedforward layer, encoder-decoder attention, and a self-attention layer.

Going Now that we have explored the basic building block of the model, it's time to understand how information flows between different components of the model and is modified by the various operations applied. In the bottom-most encoder, the input of the word after passing through embeddings is provided. The embedded input is only required in the bottom-most encoder. Another encoder block will only take the output of the previous encoder after embedding each of the word flow through both the layers in the encoder, as shown here:

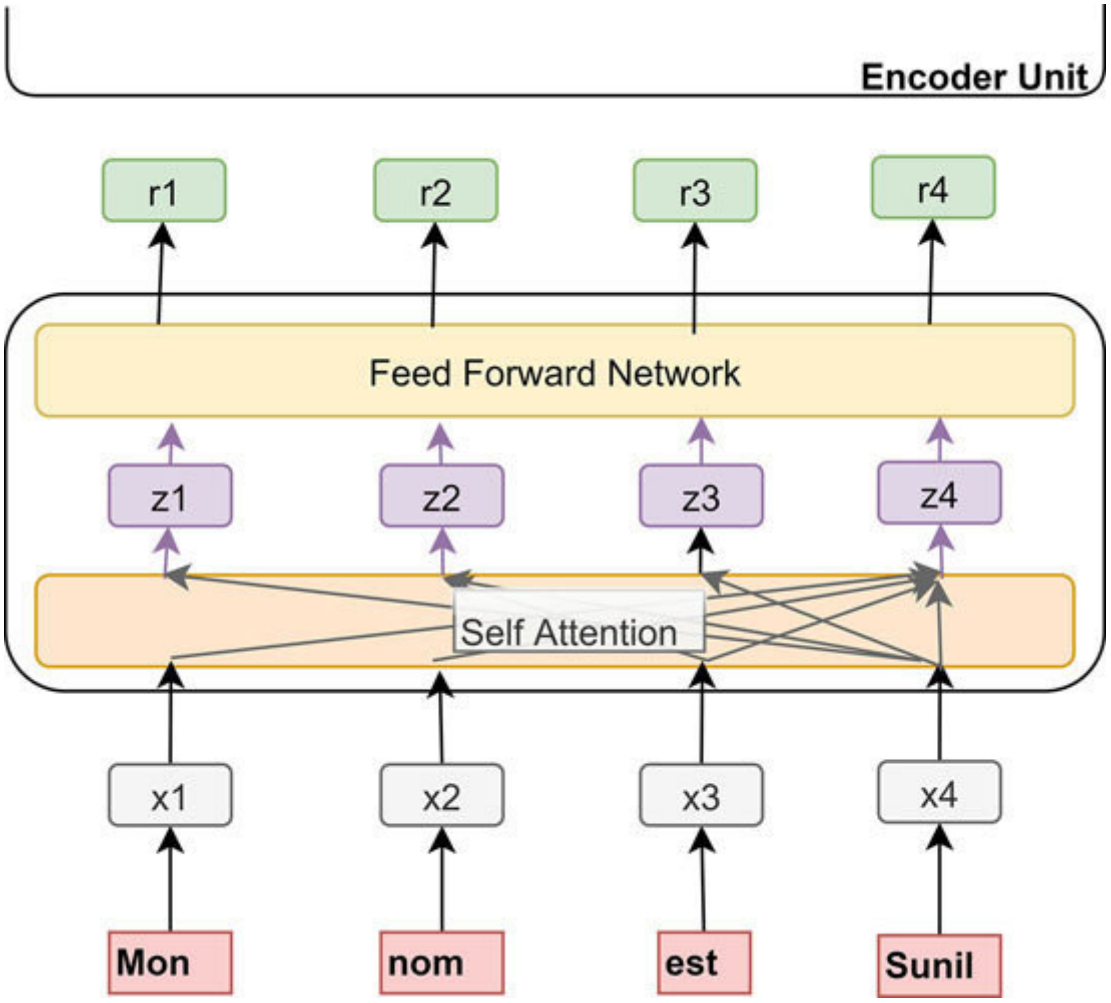


Figure 6.4: *A few details of the layers present in the encoder.*

The key property of the transformer is that all the tokens travel through all the blocks independently. The interaction between different words occurs in the self-attention layer and travels independently through the feedforward network.

Self-attention: Self-attention is the concept to find the relation between words. It is the layer where the network relates two words and tries to find the relationship between them. Let's first understand how self-attention works, and then we will

implement it using matrix operations. The first step is to construct three vectors from the embeddings. So, each word will create the key vector, a query vector, and a value vector. These vectors are created by multiplying embedding with three matrices that are learned during training. Note that the resultant vector after multiplication is smaller in dimension (64) than the original dimension (512). This size is configurable, and the numbers here are purely for demonstration purposes:

Figure 6.5: *Understanding self-attention mechanism.*

The query key and the values vector shown here are used to calculate the attention between two words in the sentence. The attention score will help associate related words and will be calculated as shown below:

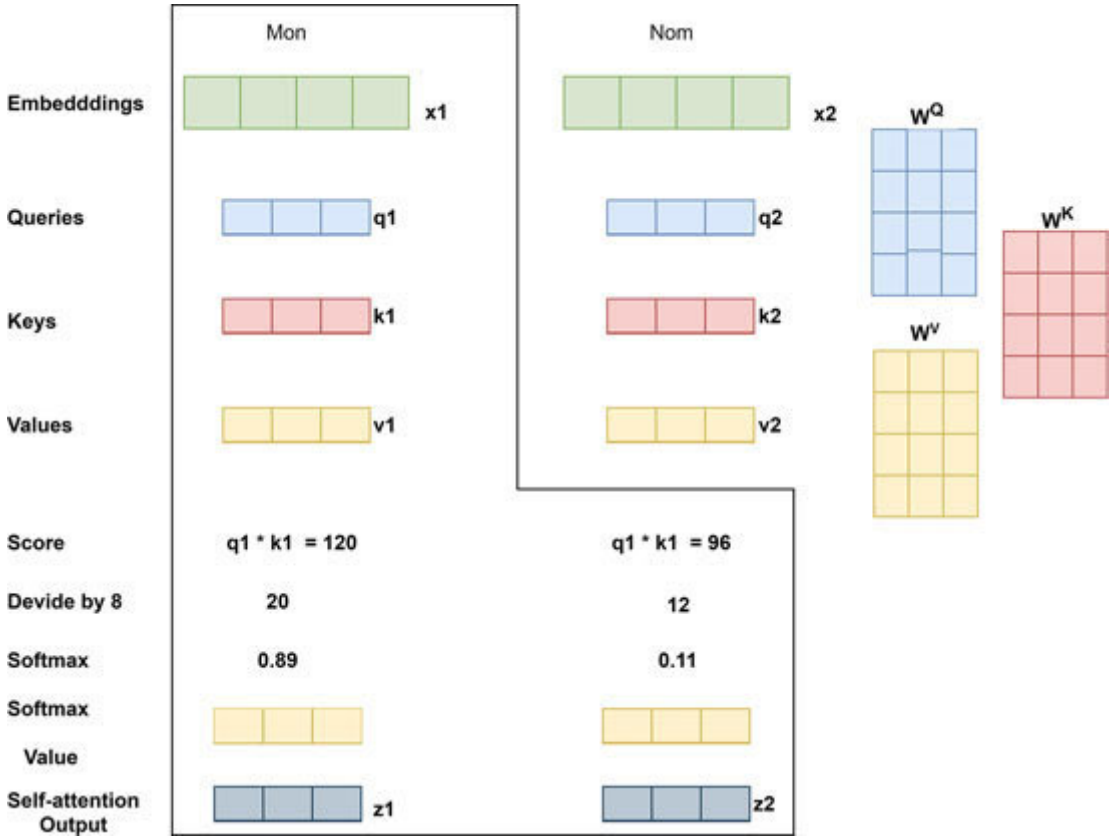


Figure 6.6: *The steps involved in self-attention.*

The attention score is calculated in the followings steps:

Key, query, and value are formed by multiplying inputs to the and matrices, respectively.

Query and key are multiplied for a score.

The score is divided by the square root of key shape. e.g., 8 in our case.

Soft-max of the above score is calculated for normalization purposes.

The calculated soft-max score is then multiplied to the value, and the resultant output is given out for each word from the self-attention layer.

After understanding the process of attention, we will look at how this process will be carried out as matrix operations:

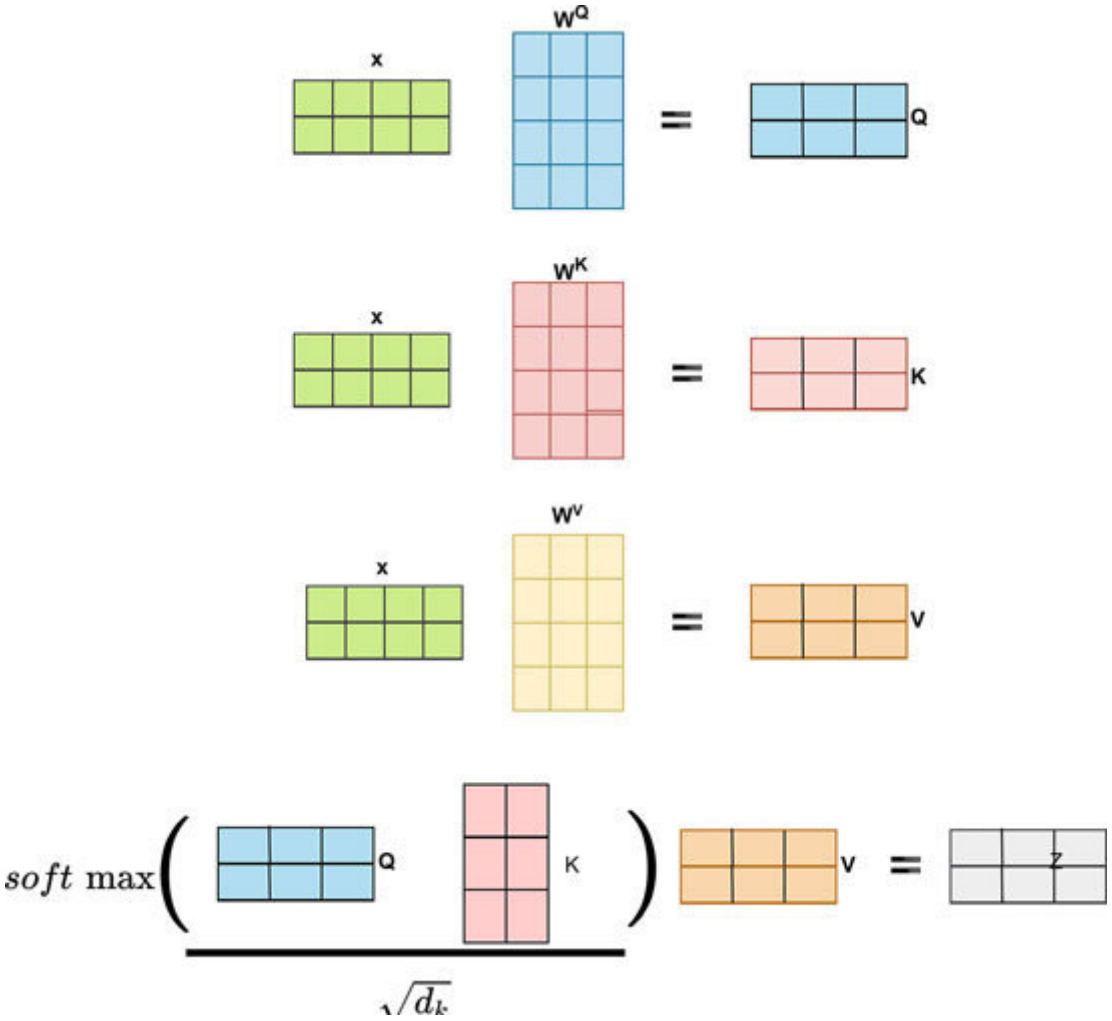


Figure 6.7: *Matrix-level operations to implement self-attention.*

The entire process of self-attention is summarized above with matrix operations. This paper takes the concept of transformer one step ahead and implements a concept caller multi-head attention. Multi-layer attention improves the performance of the attention layer in two ways.

It helps the attention look at multiple words and often helps resolve the sense of the word. For example, if “it” is present in the sentence, which word in the sentence does it refer to?

Multi-headed attention has multiple copies of Query, Key and Value matrices and so, it helps learn in multiple spaces. Each set of these multiple copies is initialized with multiple copies.

The concept of multi-head attention with two heads is shown here:

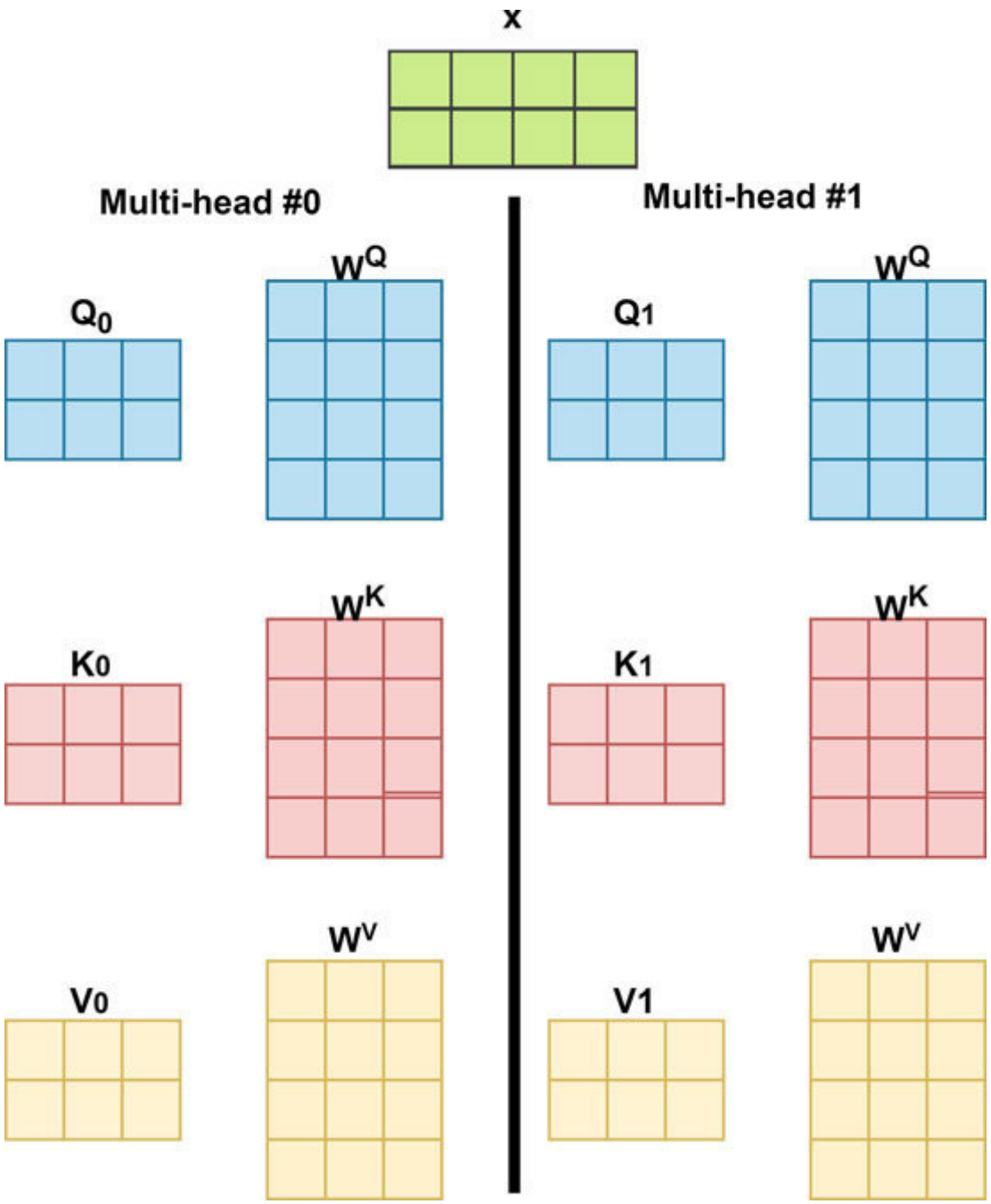


Figure 6.8: *A rough idea of multi-head attention.*

At the end of each multi-head attention, we get corresponding output matrix z, and if we perform the multi-head attention eight times, we get eight z matrices Having eight z matrices is a challenge. The next encoder module is expecting only one z matrix for each word. To solve this, we will multiply

the resultant matrix formed after the multiplication of eight z matrices with another matrix This matrix is learned in the training process, which is as shown here:

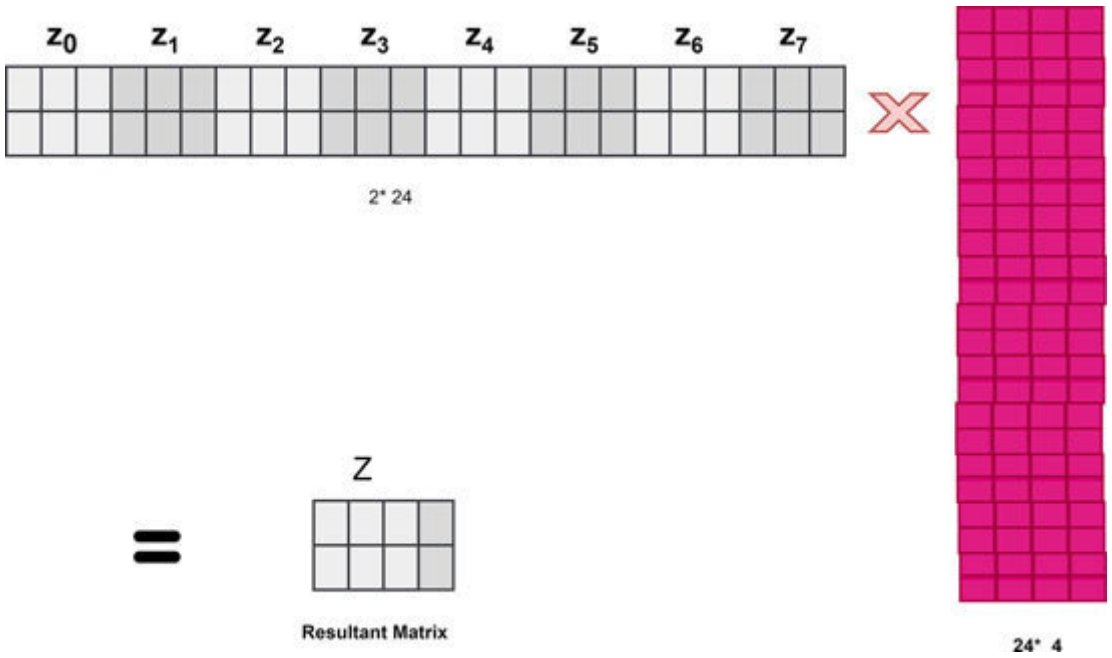


Figure 6.9: *How final output shape from the multi-head attention is made the same as the output from one head by applying addition learnable matrix.*

This is pretty much all about the multi-head attention techniques. All multi-head techniques can be put into a single figure, as illsutrated here:

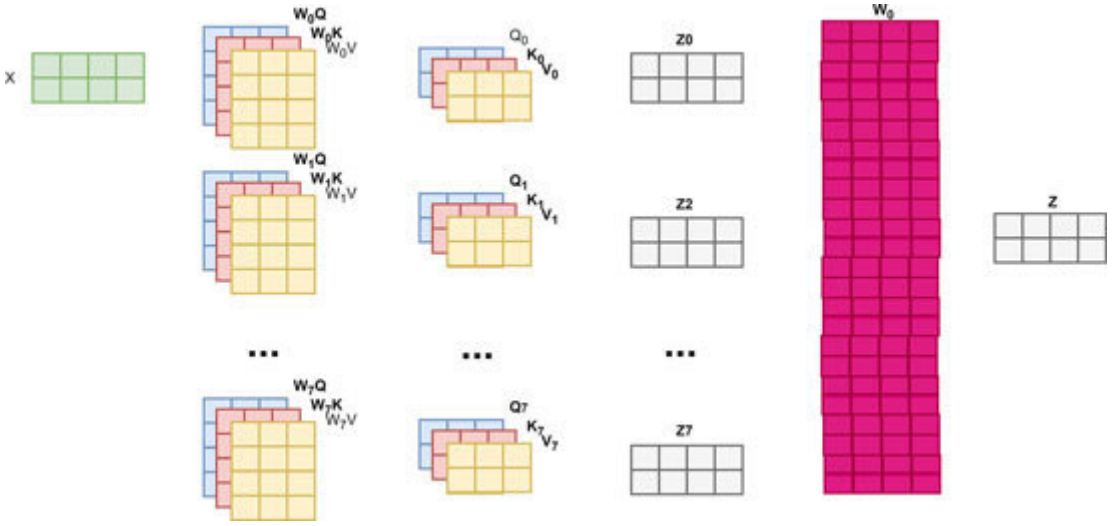


Figure 6.10: *Summarizing multi-head attention.*

This is all about multi-head attention. The innovative approach in the transformer does not stop here, and it has many more things to achieve state-of-the-art results. The next concept is positional encoding.

Positional The order of placement of the word in the sentence is important. So far, we have used various embedding techniques, and we were only taking the vector for the individual word without considering the order of the words. Personal embeddings also consider the order of the word while generating embeddings. Let's say we have an embedding vector size of 4; it is added with a positional encoding vector before injecting it into the self-attention module. This positional encoding vector transforms the embedding vector so that it also represents the order of placement of words in the sentence. The concept of positional embedding is diagrammatically shown as follows:

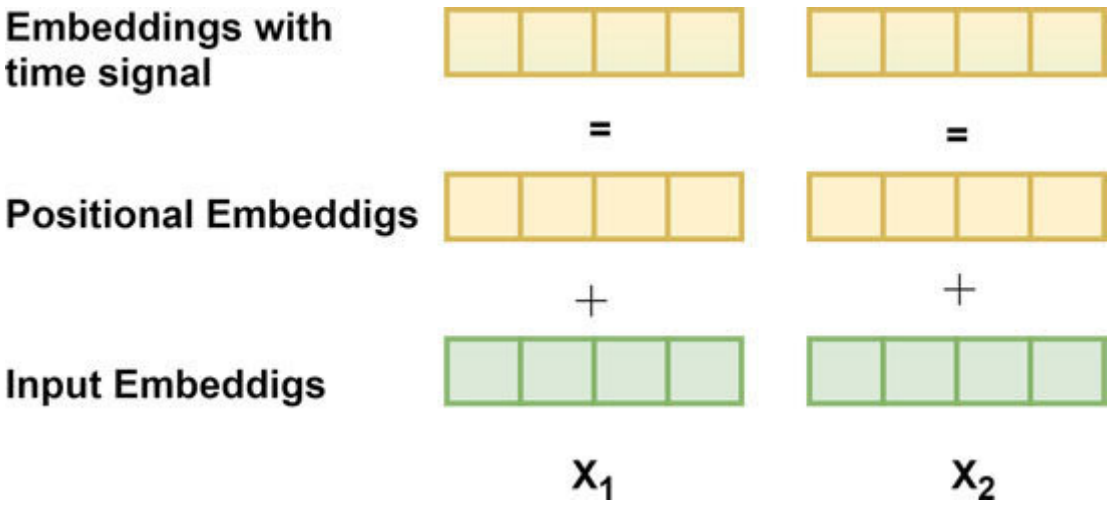


Figure 6.11: *The ideology behind positional encoding.*

Residual The in-detail structure of the encoder or the decoder includes a residual connection. This residual connection function is similar to skip connection discussed in *Chapter 5, Applying CNN In NLP* These residual connections aid better convergence by allowing efficient back-propagation. In addition to residual connection, each encoder or decoder block has "add and normalize layer." The "add and normalize layer" helps add various inputs and helps in normalizing before output. The following is a detailed diagram of encoder and decoder with all essential connections:

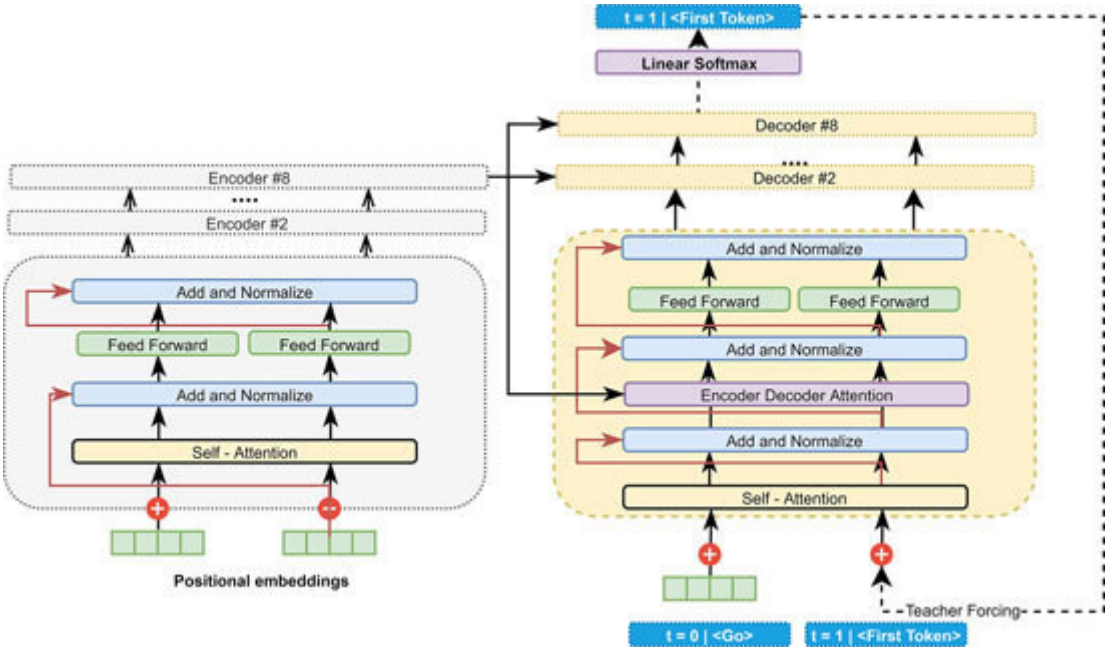


Figure 6.12: *The overall process of translation along with residual connections present in the encoder and decoders.*

We pretty much know everything about encoder. On the decoder side, the final key Query and Value vector are received, and this serves as the initial key for the decoder. At time t = 0, the token is fired, and on the other side, after passing through all the decoders, a linear and soft-max layer provides the output token. Each decoder receives the final encoder state separately. The output at t=1 is fed again to the decoder along with the encoder state to yield the next token at t = 2. It is similar to the teacher forcing mechanism we learned in *Chapter 4, Using RNN for* Similarly, all tokens will be translated into another language.

Source and Target Masking

Coding the entire transformer will be a great experience; in this section, we will only implement a few important concepts of the transformer model. In this section, we will understand how masking and positional encoding works by practically implementing and visualizing them. I will use the following parameters to demonstrate masking and positional encoding in the later sections.

opt = {“d_model”:512, “trg_pad”:1,”src_pad”:1}

Here, d_model is the number of hidden states for the encoder and decoder of the transformer.

Masking has two functions in the transformer network:

In encoder and decoder, to give zero attention output wherever it is padding in the input and target sentences, respectively.

In decoder, to prevent the decoder from cheating by looking (peaking) ahead of the sequences.

Let’s code a dummy source and target sequence to understand these facts. Our dummy source and target

sequence looks as follows, where we have taken source sequence equal to the target sequence. Each sentence is present in each column, and the length of each sentence is made equal by padding = 1:

```
src = torch.tensor([
[2, 3, 4, 5, 6, 7, 8, 9],
[2, 7, 7, 4, 2, 4, 3, 4],
[3, 6, 8, 5, 2, 1, 3, 4],
[4, 7, 9, 6, 3, 1, 7, 1],
[5, 7, 2, 7, 3, 1, 8, 1],

[1, 6, 2, 8, 4, 1, 8, 1],
[1, 1, 1, 1, 5, 1, 9, 1],
[1, 1, 1, 1, 5, 1, 1, 1]])
trg = src
```

The next step is to create the nopeak_mask function that restricts the decoder peaking ahead of the current decoding sequence in the target sequences:

```
defnopeak_mask(size, opt):
np_mask = np.triu(np.ones((1, size, size)),k=1).astype('uint8')
np_mask = Variable(torch.from_numpy(np_mask) == 0)
return np_mask
```

We also need a create_masks function to take the source and target function and apply it to the mask:

```
defcreate_masks(src, trg, opt):
```

```
src_mask = (src != opt["src_pad"]).unsqueeze(-2)
if trg is not None:
trg_mask = (trg != opt["trg_pad"]).unsqueeze(-2)
size = trg.size(1) # get seq_len for matrix
np_mask = nopeak_mask(size, opt)
trg_mask = trg_mask&np_mask
else:
trg_mask = None
return src_mask, trg_mask
src_mask, trg_mask = create_masks(src,trg,opt)
```

Using the plotting function given at the Ch6/understanding_the_transformer.ipynb script, the source masks look like this:

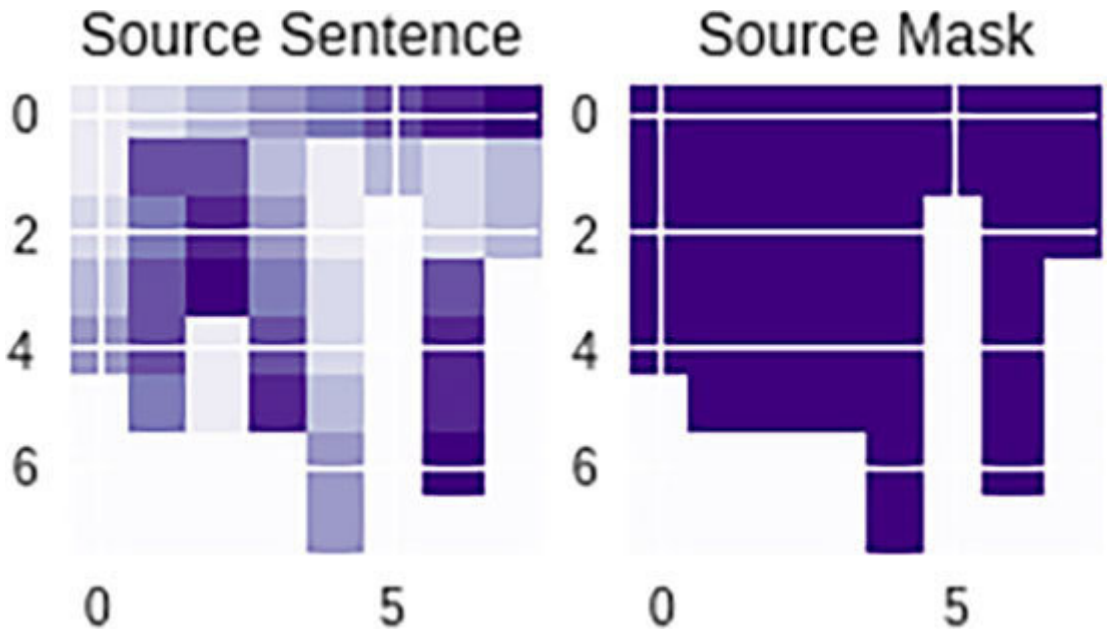


Figure 6.13: *Masking applied to source sentences in the transformer.*

A single source sentence is shown here as the column. Wherever padding is applied as 1, the sentence seems to be truncated. The source masks also have 1 up to the length of each sentence and zero after that. This way the mask is generated. If you look at the target mask, it is a bit tricky to understand:

Figure 6.14: *Making applied to the decoder in the transformer.*

Usually, the decoder's task is to take the encoder output and produce output by teacher forcing the decoder. The mask function here is to allow the decoder to only look at the current sequence and mask all future sequences. Note that wherever padding is applied to the source sequence, the decoder has a mask. After two target masks from top left corner in the third target mask, the fifth sequence is masked because, in the source sentence, the fifth column (sentence) is made up of only two words.

Positional Encoding

As we understood in the previous section, positional encoding is applied to give the encoder and decoder a sense of the position of the word in the sequence. As described in the paper, the positional embedding can be mathematically given as follows:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{1000^{d\frac{zi}{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{1000^{d\frac{2i}{model}}}\right)$$

Here, pos refers to the order in the sentence, and i refers to the position along the embedding vector dimension. Each value in the matrix is then worked out using the preceding equations. The same logic can be implemented as follows:

```
class PositionalEncoder(nn.Module):
def __init__(self, d_model, max_seq_len = 80):
super().__init__()
self.d_model = d_model
# create constant 'pe' matrix with values dependant on
```

```
# pos and i
pe = torch.zeros(max_seq_len, d_model)
for pos in range(max_seq_len):
for i in range(0, d_model, 2):
pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))


pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i +
1))/d_model)))
pe = pe.unsqueeze(0)
self.register_buffer('pe', pe)
def forward(self, x):
# make embeddings relatively larger
x = x * math.sqrt(self.d_model)
#add constant to embedding
seq_len = x.size(1)
x = x + Variable(self.pe[:,:seq_len],requires_grad=False)
return x, self.pe
PE = PositionalEncoder(opt["d_model"])
```

When plotted, positional embeddings will look like this:

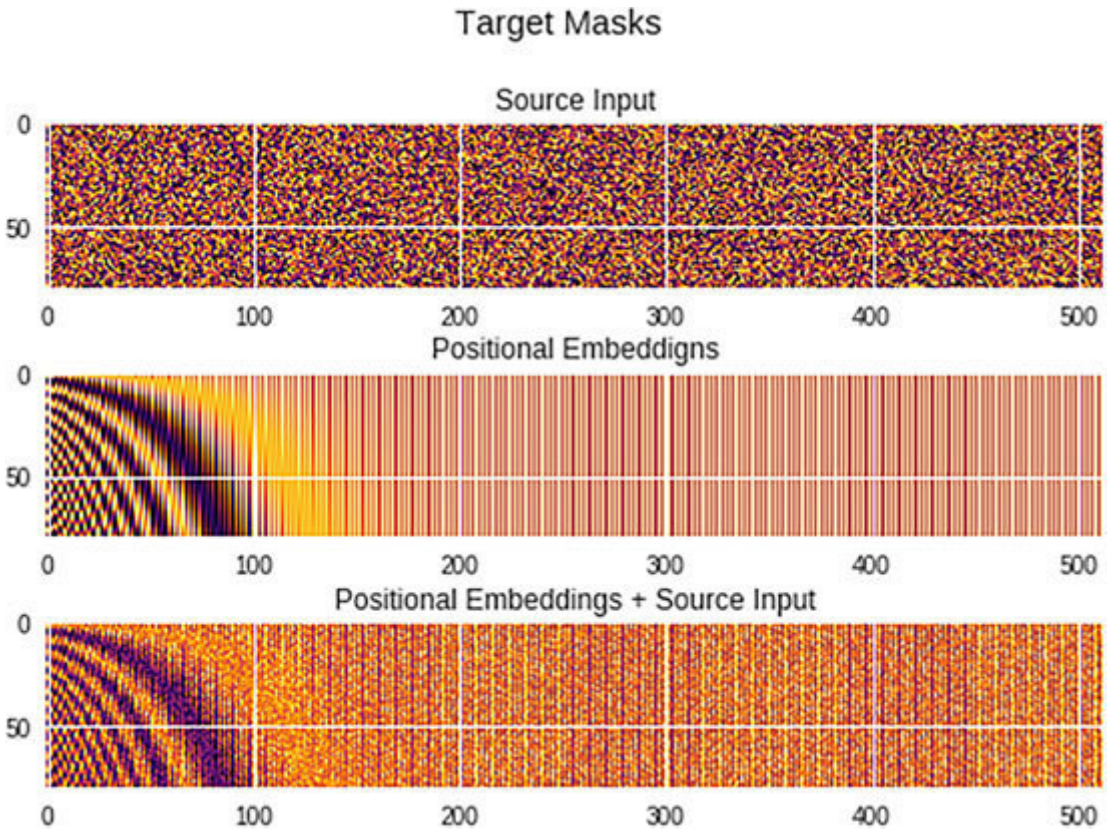


Figure 6.15: *The first is source input, the positional embedding is generated and added to source input to add a sense of position in the source input (third subplot).*

All the implementation discussed here are given in the Ch6/understanding_the_transformer.ipynb Jupyter notebook.

The intent of the transformer was a very important research milestone. The transformer constitutes the core component of many models like BERT, ELMo, and ULMFiT. The implementation of the transformer will help us understand all the transformer-derived models. The step-by-step implementation is out of the scope of this book, but you can always refer to the "The Annotated Transformer" post, where

all the individual parts of the transformer are beautifully explained with the code.

You can refer to the following links:

Attention is all you need: *https://arxiv.org/abs/1706.03762*

PyTorch implementation of opener's fine-tuned transformer: *https://github.Com/huggingface/PyTorch-openai-transformer-lm*

Converting Sentence to Vector

In *Chapter 3, Representing Language* we saw methods like word2vec, Fasttext, and glove, which converts a given token/word into a float vector of n dimensions. However, it isn't enough if we need to compare semantic similarity between two sentences. With word tokenizer, there is no feasible way to find the similarity between two sentences. We require techniques to convert a sentence directly into the vector. One of the techniques to convert the sentence to vector is the skip through technique. SkipThrought was devised by researchers at the University of Toronto. Another technique is doc2vec, which was invented by Thomos Mikolov, the Google researcher. We will see both the techniques in detail in the upcoming sections, along with their working and implementation.

The possible application of converting a sentence to a vector could be the following:

Converting sentence to a vector so that downward tasks like classification, summarization, and language translation can be carried out easily.

Comparing two sentences semantically.

Our goal is to learn vectors from the sentence so that they can be used for semantic similarity or relatedness between sentences. The next question is, why can't we use Word2Vec for this? Can't we apply any technique over word vector to compare two sentences? Let's assume that we have word vectors for the sentence. The one way to get the sentence representation is by finding the centroid for all the words. If so, the distance between two sentences can be found using centroid distance. The same thing can be extrapolated for paragraph or document comparison. This method neglects the characteristics of the language that each word has meaning according to its order of placement and the contextual words. A word's placement in a sentence has significance, and by taking centroid, we are treating all words with equal significance. Let's take an example to understand this. Suppose we have two sentences:

You are going there to earn not learn

You are going there to learn not to earn

Both these sentences carry different meanings, but if centroid based similarity is found, both words will be regarded as similar. This concept was little more enhanced in the paper "From Word Embeddings To Document Distances," published in EMNL 14. According to this paper, if two sentences are given, we take minimum distance from each word of sentence 1 to sentence 2 and add them, as shown below:

Obama speaks to the media in Illinois

The President greets the press in Chicago

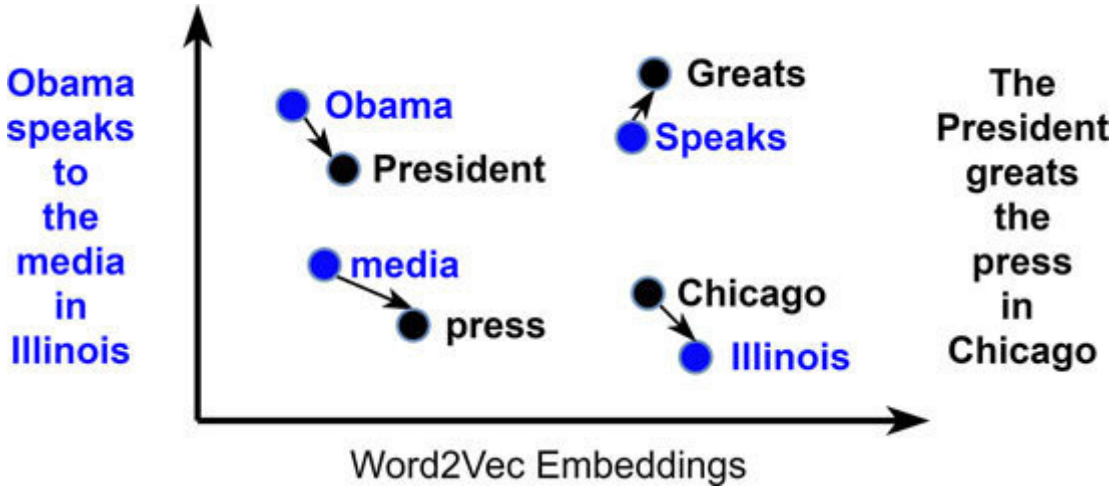


Figure 6.16: *Word mover's distance.*

To generate vector as shown in the diagram 6.15, all stop words were removed, and the word embedding is calculated. The distance between the two documents is the minimum cumulative distance that all words in document 1 need to travel to exactly match document 2.

It approaches in inspired form rules and logic. It is just a preliminary approach, and the major drawback is that it is not considering the order and semantic meaning of the word.

The Paragraph Vector is based on a neural network model. The Paragraph Vector method can represent variable size sentence paragraph and document into a fixed-size vector. It is an unsupervised algorithm that converts any sentence paragraph and document to the fixed-size vector:

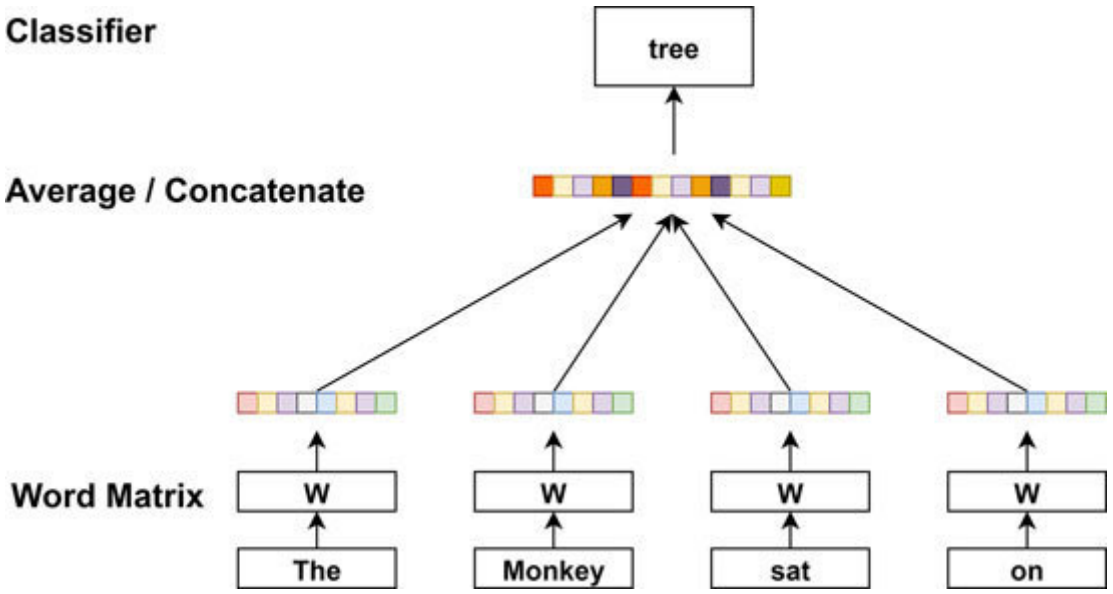


Figure 6.17: *A framework for learning word vectors.*

The context of four words (The, Monkey, sat, on) is used to predict the fifth word (tree). The input words are mapped to a column of the matrix W to predict the output words.

In this method, every word is mapped to a unique vector and represented by a column in a matrix W. The column is indexed by the position of the word in the vocabulary. The concatenation and/or sum of the vector is used as a feature to predict the next word in the sentence. Mathematically, if we have the words, the end goal is to enhance the average log probability of given all other context words:

$$\frac{1}{T}\sum_{t=k}^{t-K}\log p\left(w_t \mid w_{t-k}\cdot w_{t+k}\right)$$

The prediction is made via multi-class classification like softmax. In practice, hierarchical softmax is preferred to softmax for fast training:

$$p(w_t \mid w_{t-k}..w_{t+k}) = \frac{e^{ywt}}{\sum^{ieyi}}$$

The detailed approach, as shown below, has a paragraph vector; every paragraph is mapped to a column in the vector D, and every word is mapped to a column in vector W. The matrices W and D are averaged or concatenated to predict the next word in the sentence. Hidden vector is constructed from the combination of W and D. In this process, the model is compelled to predict the next word using word and sentence level feature. In turn, the model's memory starts remembering information related to the sentence. Due to this reason, this model is known as **Paragraph Vector - Distributed Memory** model The schematic representation of the model is as shown in *figure*

SkipThought SkipThought is an encoder-decoder model to learn the nuance of sentences in an unsupervised way. The paragraph vector is an intra-sentence model, whereas skip thought is an inter-sentence model. If you have gone through the skip-gram approach in *Chapter 3, Representing Language* the same approach is followed here at the sentence level. The SkipThought model has three parts: one encoder and two decoders, as shown in this schematic diagram:

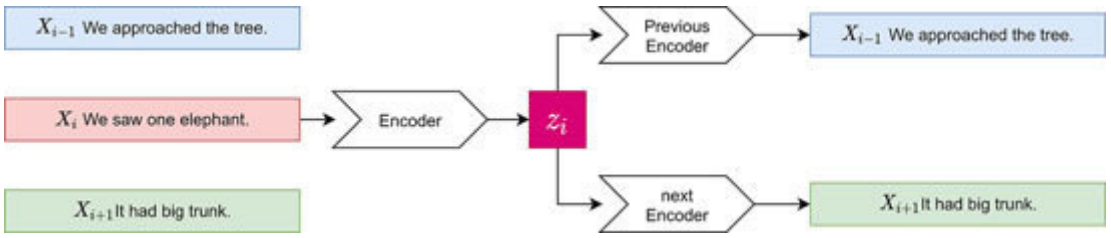


Figure 6.18: *An illustration of how SkipThought works; it has one encoder and two decoders.*

The encoder takes a current sentence, one decoder predicts the previous sentence to the given sentence, and another decoder predicts the next sentence to the given sentence.

The Encoder network: It takes the network and converts it into fixed-length representation using GRU or LSTM. GRU/ LSTM takes each word of the sentence sequentially and, using attention mechanism, generates the fixed-length vector z(i).

Previous decoder network: It takes the fixed vector representation z(i) and generates the previous sentence x(i – 1) to the input sentence x(i).

Next decoder network: It takes the fixed vector representation z(i) and generates the next sentence x(i + 11) to the input sentence x(i).

The assumption behind the SkipThought vector is that the placement of the sentence in the document has a meaning, and this meaning is contextual with respect to the neighboring sentences. If the network is compelled to output context, sentences from the target on the network may start

learning the meaning of the sentence in the fixed-size dense vector.

Sentence to Vector

In the section, we will see how Sentence2Vec can be implemented with Doc2Vec. The next thing we will look at is how to use skip-though vectors.

Sentence to This method is implemented as Doc2Vec in the popular library Gensim. We will see how to use Gensim's implementation of Doc2Vec.

Loading and tagging

```
data = open("Ch6/dataset/small_talk_test.txt").read().splitlines()
tagged_data =
[TaggedDocument(words=word_tokenize(_d.lower()), tags=
[str(i)]) for i, _d in enumerate(data)]
```

The following are the various parameters given to training the model. The dm parameter is used to define which algorithm to use:

dm=1 means 'distributed memory' (PV-DM)

dm =0 means 'distributed bag of words' (PV-DBOW)

We have been introduced to these algorithms in the *Getting Ready...* section of the recipe. The Distributed Memory model preserves the word order in a document, whereas Distributed Bag of words just uses the bag of words approach, which doesn't preserve any word order:

```
max_epochs = 100
vec_size = 20
alpha = 0.025
model = Doc2Vec(size=vec_size,
alpha=alpha,
min_alpha=0.00025,

min_count=1,
dm =1)
model.build_vocab(tagged_data)
for epoch in range(max_epochs):
print('iteration {0}'.format(epoch))
model.train(tagged_data,
total_examples=model.corpus_count,
epochs=model.iter)
# decrease the learning rate
model.alpha -= 0.0002
# fix the learning rate, no decay
model.min_alpha = model.alpha
model.save("d2v.model")
print("Model Saved")
```

The following code snippet performs the following functionality:

Loading the model

Getting a vector for the sentence

Finding similar sentence

```
model= Doc2Vec.load("d2v.model")
#to find the vector of a document which is not in training data
test_data = word_tokenize("Now that you are eighteen".lower())
v1 = model.infer_vector(test_data)
print("V1_infer : ", v1)
# to find most similar doc using sentnce number
similar_doc = model.docvecs.most_similar('1')
print("Simillar Docs : ", similar_doc)
>>> V1_infer : [-0.01016486, .., 0.04624778]
>>>Simillar Docs : [('12', 0.7584813833236694), .., ('1113', 0.7243909835815430)]
```

The working implementation sentence to vector using Doc2Vec is given at

Skip_Thought

The SkipThought vector training is computationally very costly, so we will take the existing model of the SkipThought and see how to use the pre-trained model for the inference. The implemetation for skip_thought is included in the code along with the book in the Ch6 folder. The implementation gives in-depth information about the following topics:

Installation of the SkipThought required Bazel, TensorFlow, NumPy, sci-kit-learn, NLTK, and Gensim

Download pretrained You can download the model trained book corpus using two configurations:

Unidirectional RNN encoder

Bidirectional RNN encoder

Training a Including training a model and running the training script

Expanding the The generated pre-processing script contains only 20,000 words, which are insufficient for many tasks. SkipThought provides the expanding vocabulary function by which one can increase the model vocabulary. It is based on

the “Translation Matrix” based methods, as described in the “Exploiting Similarities Among Languages for Machine Translation” paper.

Evaluating a Evaluation of the model on various datasets is supported. These datasets include SICK-semantic relatedness task, **MSRP** - **Research Paraphrase** paraphrase detection task, MR - movie review sentiment task, CR - customer product review task, SUBJ - subjectivity/objectivity task, MPQA - opinion polarity task, and TREC - question-type classification task. An extensive guide of how to use the SkipThought is already provided at the GitHub repository, so this chapter doesn’t have a working example of skip thought.

If you see the model, the encoder is just fine, but the decoder has been assigned a herculean task. It’s very difficult for humans as well to predict the next or previous sentence, given the current sentence. This is the downside of this model, and because of this reason, only the SkipThought model produced an average result of 77% on various tasks. Besides this, the model is bulky and required a cluster of GPUs for training. The pre-trained model requires more than 32 GB RAM to use this model. The SkipThought model was the first of its kind to produce sentence embedding with considerable performance uplifting. In the upcoming recipes, we will see how other models like Skip thought are used for producing a fixed-sized vector from sentences.

Take a look at the following links:

Supervised from word embeddings to document distances: *http://proceedings.mlr.press/v37/kusnerb15.pdf*

Distributed representations of sentences and documents: *https://cs.stanford.edu/~quoc le/paragraph_vector.pdf*

SkipThought vectors: *https://arxiv.org/pdf/1506.06726.pdf*

Exploiting similarities among languages for machine translation.: *https://arxiv.org/abs/1309.4168*

Getting to Know Contextual Vectors

So far, we have seen many word vector representations, including Word2Vec, Glove, and Fasttext. In all these previously discussed methods, the vector for any given word will be the same for entire documents. The word bank is used for the financial institution, but it can also be the bank of the river. For the technique mentioned above, the meaning of the word bank is the same in both cases. This property of the word bank to have different meaning as per the context is called polysemic. ELMo was proposed in the paper **Deep contextualized word representations** by Matthew E. Peters and coworkers.

ELMo uses the bidirectional language model to generate contextual word representation. The aim is to learn representations that model the syntax, semantics, and polysemy.

The word representations combine all layers of a deep pre-trained neural network. **90M+**

ELMo representations are purely character-based, allowing the network to use morphological clues to form robust representations for out-of-vocabulary tokens unseen in training.

The representation for each word depends on the entire context in which it is used. ELMo model found to be generating different vectors depending on the context of the word, so it deals with **Polysemy** as well.

The architecture has two components:

Token A context-independent token representation is computed with character convolution. Character representation from the CNN layer is taken as input to the LSTM layers.

Bidirectional Language Model This layer follows the conventional concept of predicting token forward given the target token, and the token backward given the target token. For the given sequence of N tokens the forward model computes the probability of token given all the past tokens In a probabilistic way, this can be represented as:

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1 \ldots t_{k-1})$$

A backward language model runs similar to the forward representation but in the reverse direction. A backward model computes the probability of token given all the future tokens ...., In a probabilistic way, this can be represented as:

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1} \ldots t_N)$$

Along with forward and a backward ELMo representation, the context-independent CNN-based representation is generated for each token. The character representation for L layers of the CNN having different filter size is generated, and then it is passed through L layer of forwarding LSTM. For each token K, the LSTM outputs a context-dependent representation where j = 1, ..., L. The top layer LSTM output, is used to predict the next token with a softmax layer. For the backward layer, the CNN represent is calculated as Both layers are combined for the final representation as =

This was just an overview of the internal working of the ELMo model. Finally, it has three outputs: one from forwarding LSTM, one from backward LSTM, and one from the CNN layer. The forward and backward LSTM are context-aware, while the CNN output is content-independent. The schematic representation of the model is as shown below:

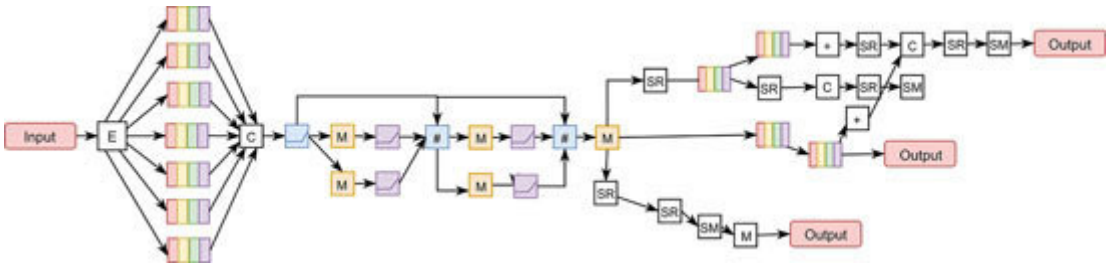


Figure 6.19: *Schematic diagram for the ELMo model. Character-based input is given to the seven layers of the convolution neural network with different filter sizes.*

The output from all the CNN is concatenated into convolution layer C. Then, this information passes through the series of operations like aggregation, concatenate, element-wise

multiplication, linear operators, threading layers, part layers, sequence reverse layers, sequence most layers, replication layer, and finally, flatten layers. Their output will be given out at: two from the LSTM layer and one from the CNN layer. The info-graphic is inspired by

Using the Pre-trained Model

Allen AI has released an official version of the ELMo. By using this API, you can use a pre-trained model to get contextual embeddings of the token in the given sentence.

Installation:

```
!pip installallennlp
```

Quick usage:

```
From allennlp.commands.elmo import ElmoEmbedder
import scipy
elmo = ElmoEmbedder()
Getting Embeddings**
vectors = elmo.embed_sentence(["My", "name", "is", "Sunil"])
vectors.shape
>>> (3, 4, 1024)
```

We have four words in the sentence. As we already know, the ELMo embedding generates three embeddings for each word: two from the LSTM layer and one from the CNN layer. Each of these embeddings has a size of 1024, which is the size of the highest number of convolution filters used in the ELMo model. After checking contextual claim, it's very clear that the

embedding for the word “Apple” is different for both sentences. The difference is clear from the cosine difference between the output generated by LSTM layers. CNN layer is not contextual, so the cosine distance between two instance of “Apple” is the same:

```
def get_similarity(token1, token2, token1_location, token2_location):
vectors = elmo.embed_sentence(token1)
assert(len(vectors) == 3) # one for each layer in the ELMo output

assert(len(vectors[0]) == len(token1)) # the vector elements correspond with the input tokens
vectors2 = elmo.embed_sentence(token2)
print(“=”*50)
print(“Entity 1 : “,token1[token1_location], “ | Entity2 : “, token2[token2_location])
print(“Shape of one of the LSTM vector : “, vectors[2] [token1_location].shape)
print(“=”*50)
print(“cosine distance of 2nd bilstm layer vector”, scipy.spatial.distance.cosine(vectors[2][token1_location], vectors2[2][token1_location]))
print(“cosine distance of 1st bilstm layer vector”, scipy.spatial.distance.cosine(vectors[1][token1_location], vectors2[1][token1_location]))
print(“cosine distance of CNN layer vector”, scipy.spatial.distance.cosine(vectors[0][token1_location], vectors2[0][token1_location]))
return
```

get_similarity(["I", "ate", "an", "Apple", "."], ["I", "have", "an", "iPhone", "made", "by", "Apple", "Inc", "."], 3, 6)

Entity 1 : Apple | Entity2 : Apple Shape of one of the LSTM vector : (1024,)
cosine distance of 2nd bilstm layer vector 0.5723829865455627
cosine distance of 1st bilstm layer vector 0.5360225439071655
cosine distance of CNN layer vector 0.6341522932052612

The embedding for the word "Apple" is different for both sentences. The difference is clear from the cosine difference between output generated by LSTM layers. The CNN layer is not contextual, so the cosine distance between two instance of Apple is the same. The usage of the ELMo-BiLM API is given at Alternatively, ELMo can be used with the Zalandro flair API, a very simple framework for state-of-the-art **Natural Language Processing** The Zalandro flair API is an open-source project that can be accessed at The full Elmo model with seven convolution layers as developed by the inventor. The full model has 92 million parameters. It is generally not required, but for the specific purpose, one may be required to retrain the model. ELMo can be trained on any dataset using the source code given at ELMo-BiLM GitHub repository: To train the ELMo model on your data from scratch, you require three files:

A vocabulary file

A set of training files

A set of held-out files

The vocabulary file has tokens sorted in descending order of the occurrence. This token file must start with three special tokens, followed by other tokens present in the dataset:

.

by other tokens>

The training data must be split into two parts: one for training and one for test/validation. The last thing to do is to change the parameters in the /bin/train_elmo.py file. Here, you can make certain changes like:

Change the convolution layer if you want the smaller network. For example, if you want output size to be 128, keep only the [[1, 32], [2, 32], [3, 64], [4, 128] filters in the /bin/train_elmo.py file.

If you want to run the training on multiple GPUs, change the n_gpus = 3 as desired.

Update the number of tokens n_train_tokens = to a total number of tokens in your vocab files.

You can experiment with other parameters.

After getting these files, you can run the following commands to start the training: python bin/train_elmo.py --train_prefix= to trainingfolder> --vocab_fileto vocab file> --save_dirwhere modelswill be checkpointed>

For deep contextualized word representations, you can refer to

Training Supervised Embedding

We have been many embedding techniques, and the unsupervised way of training embeddings seems to be a normal way of training them on a domain-specific corpus. Then, such learning is passed down to the supervised learning task by providing a dense representation of the word or sentences. As opposed to the previously learned techniques, InferSent is a supervised learning method to learn sentence level embedding. Facebook invented InferSentai with a research team and published it in "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data."_ Conneau et al._ noted that image net trained in a supervised way perform well in the downward tasks. Extending this fact, Conneau et al. trained sentence embedding layer in the supervised manner known as Infersent.

InferSent uses **Stanford Natural Language Inference** data to train the model for Natural Language Inference. NLI task provides two sentences, and the task is to find a relation between two sentences. Sentence 1 can be considered as a premise, and sentence two is considered a hypothesis. The relation can be entailment, contradiction, and neutral. For example, if we have the following two sentences, the relation between them is a contradiction:

A man inspects the uniform of a figure in some East Asian country

The man is sleeping

The intuition is that NLI is a suitable task to learn the semantic relationships between sentences, and by completing this task, a model can also learn the intricacies of the language. There are three parts in the neural network architecture:

Shared encoder that encodes sentence into a fixed-size vector

Three operations are applied to the output of the encoder model:

Concatenation (u, v)

Element-wise product u * v

Absolute element-wise difference |u – v |

Take the previous representation and apply fully connected layers to predict one of the 3 classes.

The generalized architecture is as shown:

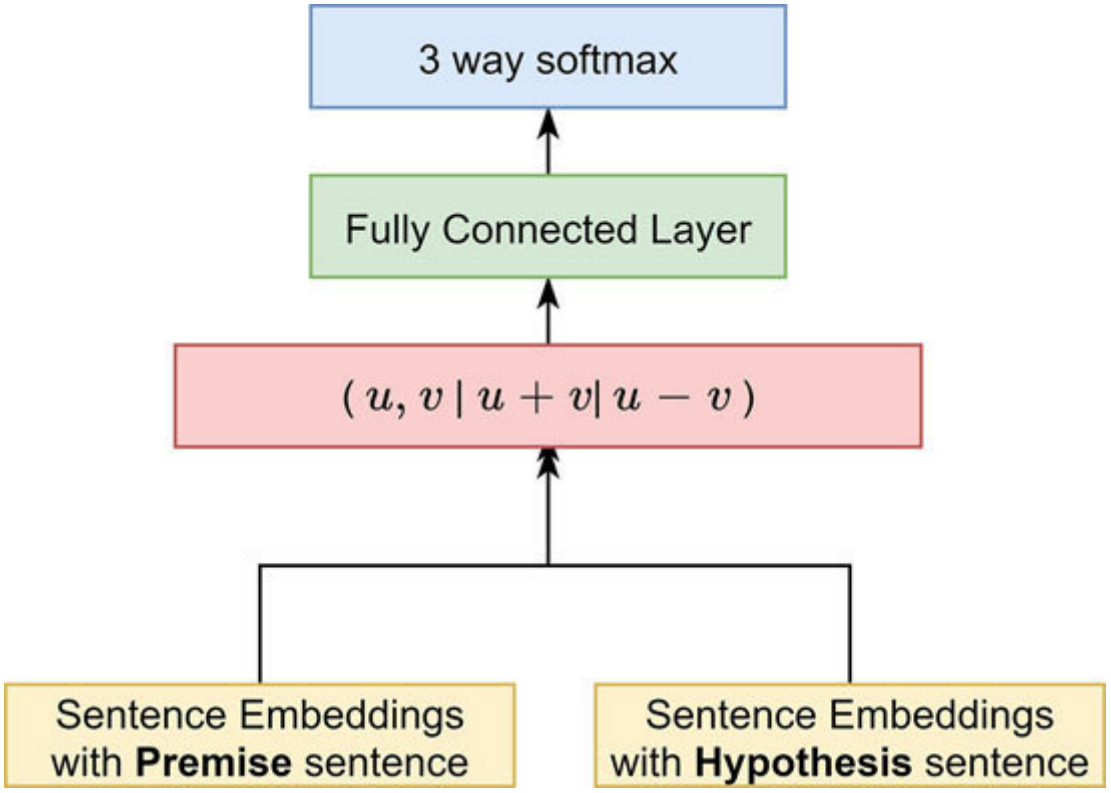


Figure 6.20: *General architecture used to train the infersent model.*

The sentence encoder model can be one of the many listed here:

Standard LSTM

Standard GRU

Concatenation of the last hidden states of forwarding and backward GRU

Bi-directional LSTM with mean polling

Bi-directional LSTM with max polling

Self-attentive network (attention with BiLSTM)

Hierarchical convolution networks

Standard LSTM, standard GRU, and concatenation of last hidden states of forward and backward These approaches take the input of the sentence word by word. The last hidden state of the LSTM or GRU is used as the fixed-size representation for the sentence. In bidirectional GRU, the last hidden layer of the forward and backward GRU is concatenated to form the final sentence representation U and V for each sentence, respectively.

Bi-LSTM with mean/max For the sequence of words, LSTM in the forward and backward direction produces the output vector for each time step. These output for each timestamp are passed to the 1D pooling layer to take the mean or max pooling. These values are then flattened to form U and V vectors. The schematic representation of the entire network is as follows:

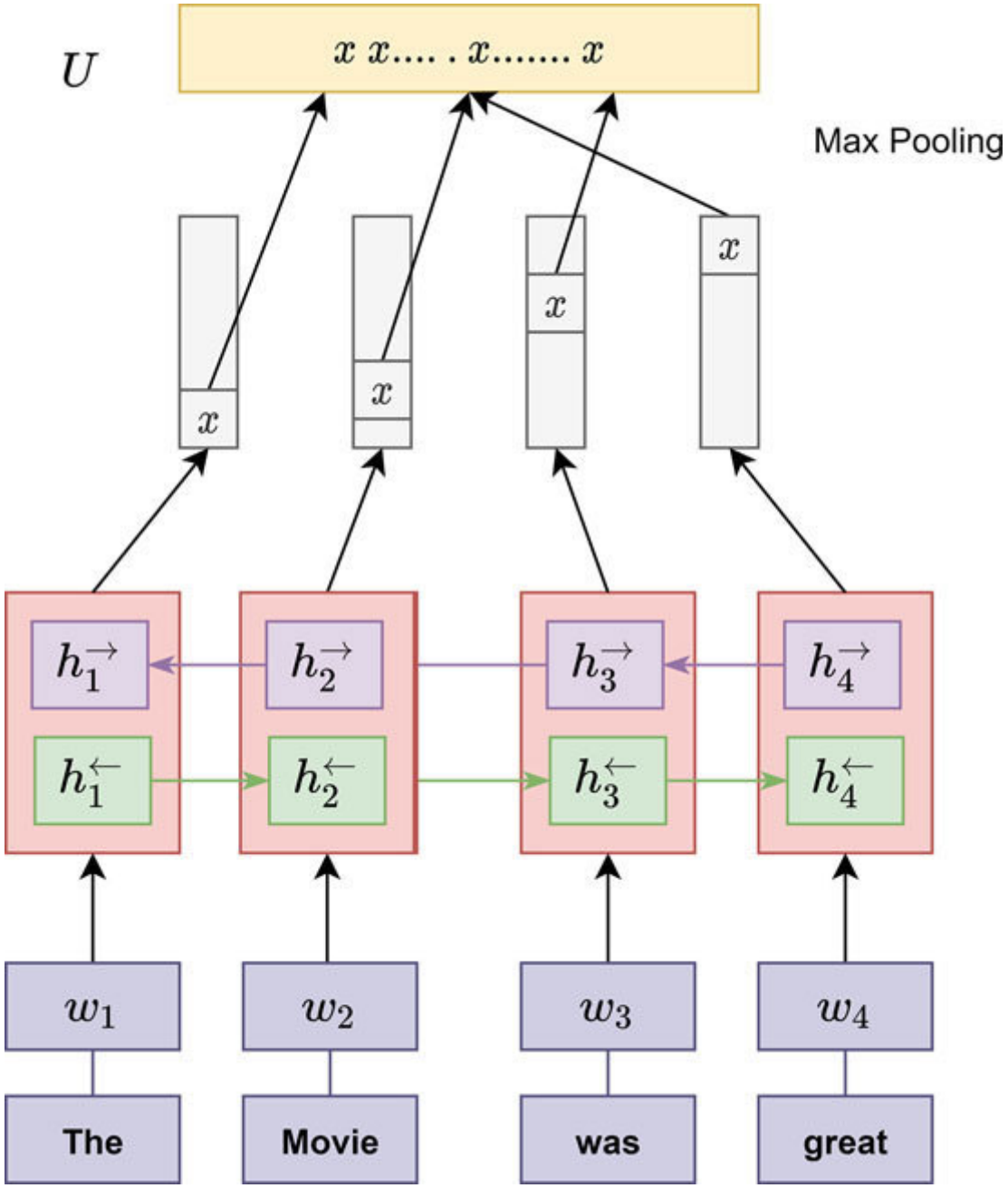


Figure 6.21: *Illustrating how BiLSTM with mean/max pooling can be used to embed a sentence.*

Self-attentive In this model, an attention mechanism is applied in addition to the normal forward-backward RNN model. The attention mechanism is similar to the one we discussed in *Chapter 4, Using RNN for* while making a

language translation module. The following is a schematic representation for the attention mechanism:

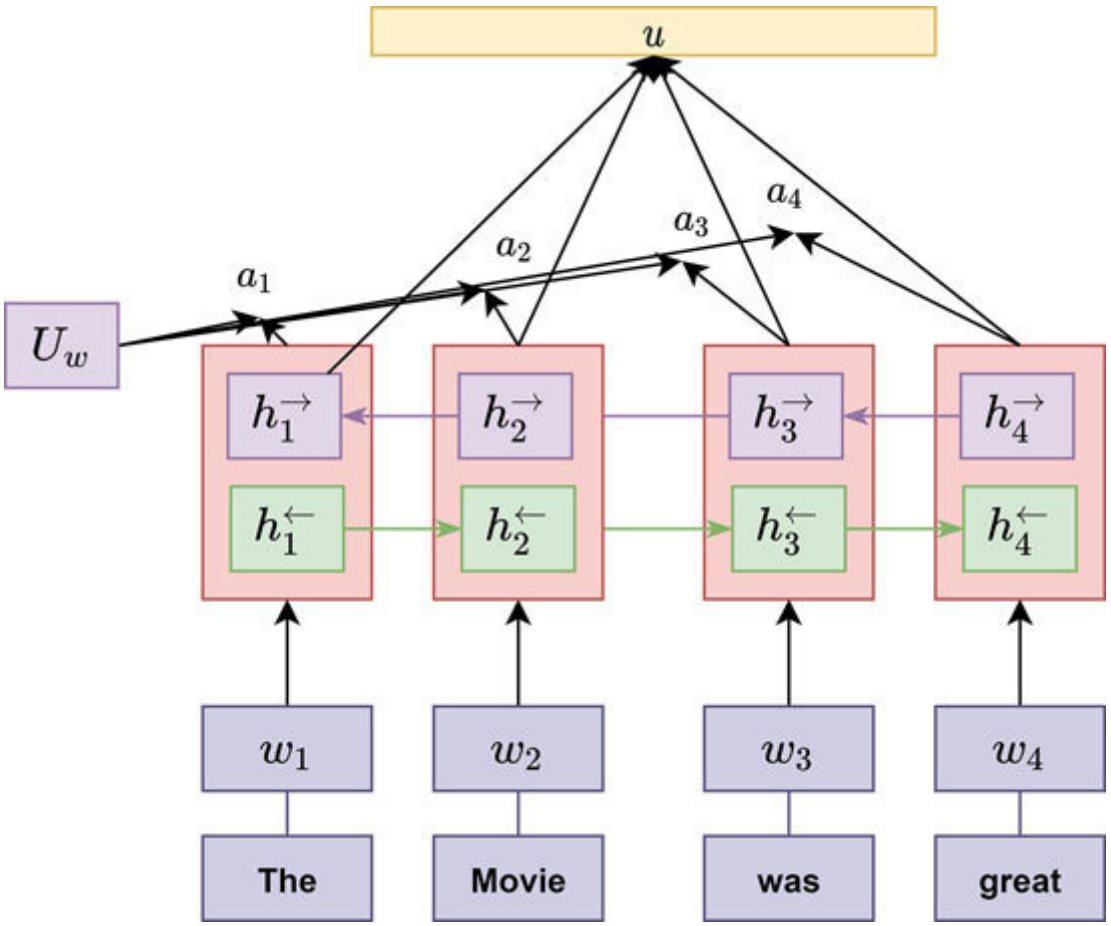


Figure 6.22: *Attention network for sentence representation.*

Hierarchical The hierarchical convolution network comprises four convolution layers. At each layer, max-pooling of the feature maps is done to obtain an intermediate representation. The final sentence embedding is represented by a concatenation of the four max-pooled representations at each layer.

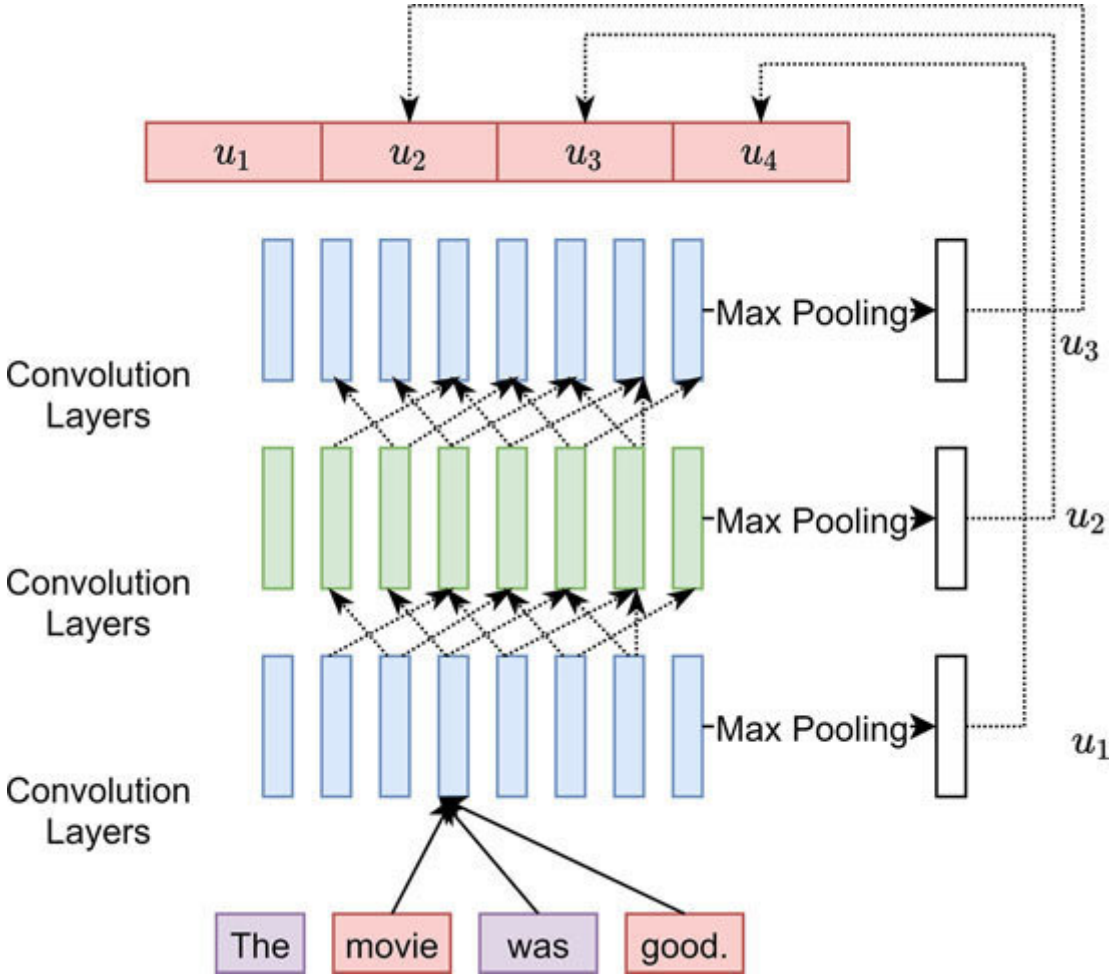


Figure 6.23: *Hierarchical ConvNet Architecture.*

The model was trained using SGD as the optimizer with 0.1 as the learning rate and 0.99 as momentum. At each epoch, the learning rate is divided by five if the accuracy on the validation set decreases. A batch size of 64 was used, and learning was stopped when the learning rate went beyond 1e-5. The fully connected layer had 512 perceptions with all the models. The GloVe vector trained on the Common Crawl 840B with 300-dimensional embeddings was used to convert words to dense vectors to give as input to various sentence encoder models. Of all the models, the BiLSM model with max-pooling achieved the highest accuracy.

Playing with InferSent

InferSent provides semantic representations for English sentences. It is trained on natural language inference data and generalizes well to many different tasks. The Facebook research team provides its implementation at The entire flow from training to getting a vector for the sentence is given here:

Cloning InferSent and adding it to the system path

Downloading the dataset required by InferSent

Downloading the GloVe and FastText vectors

Downloading InferSent pre-trained models

Code snippet for all the above illustrated steps is as given below:

```
# cloning the git repository
!git clone https://github.com/facebookresearch/InferSent.git
sys.path.append("/content/InferSent/")
#Downloading required dataset by InferSent
!bash InferSent/dataset/get_data.bash
# cloaning the git repository
```

```
!git clone https://github.com/facebookresearch/InferSent.git
sys.path.append("/content/InferSent/")
#Downloading required dataset by InferSent
!bash InferSent/dataset/get_data.bash
# Downloading GloVe and FastText vectors
!mkdirInferSent/dataset/GloVe
!curl -Lo InferSent/dataset/GloVe/glove.840B.300d.zip
http://nlp.stanford.edu/data/glove.840B.300d.zip
!unzip InferSent/dataset/GloVe/glove.840B.300d.zip -d
InferSent/dataset/GloVe/

!mkdirInferSent/dataset/fastText
!curl -Lo InferSent/dataset/fastText/crawl-300d-2M.vec.zip
https://dl.fbaipublicfiles.com/fasttext/vectors-english/crawl-300d-
2M-subword.zip
!unzip InferSent/dataset/fastText/crawl-300d-2M.vec.zip -d
InferSent/dataset/fastText/
# Downloading InferSentpretrained models
!mkdir encoder
!curl -Lo encoder/infersent1.pickle
https://dl.fbaipublicfiles.com/infersent/infersent1.pkl
!curl -Lo encoder/infersent2.pickle
https://dl.fbaipublicfiles.com/infersent/infersent2.pkl
```

Fine-tuning:

Loading the pre-trained InferSent model

Providing FastText vectors to the model

Building the Vocab

Fine-tuning the model on a given small corpus

Code snippet for all the above illustrated steps is as given below:

```
from models import InferSent
V = 2
# Loading pretrinedInferSent model
MODEL_PATH = 'encoder/infersent2.pickle'
params_model = {'bsize': 64, 'word_emb_dim': 300,
'enc_lstm_dim': 2048,'pool_type': 'max', 'dpout_model': 0.0,
'version': V}
infersent = InferSent(params_model)
infersent.load_state_dict(torch.load(MODEL_PATH))
# Providing FastText vectors to the model

W2V_PATH = 'InferSent/dataset/fastText/crawl-300d-2M-
subword.vec'
infersent.set_w2v_path(W2V_PATH)
# Building the Vocab
infersent.build_vocab_k_words(K=100000)
# Fine tuning the model on given small corpus
sentences =
open("Ch6/dtaset/dataset_for_infersent.txt").read().splitlines()
[:10000]
embeddings = infersent.encode(sentences, bsize=64,
tokenize=False, verbose=True)
print('nb sentences encoded : {0}'.format(len(embeddings)))
```

Inference : Calculating cosine similarity between two sentences.

```
def cosine(u, v):
return np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
cosine(infersent.encode(['the cat eats.'])[0],
infersent.encode(['the cat drinks.'])[0])
>>> 0.99025655
```

InferSent also provides the importance of each token in the sentence, as shown here:

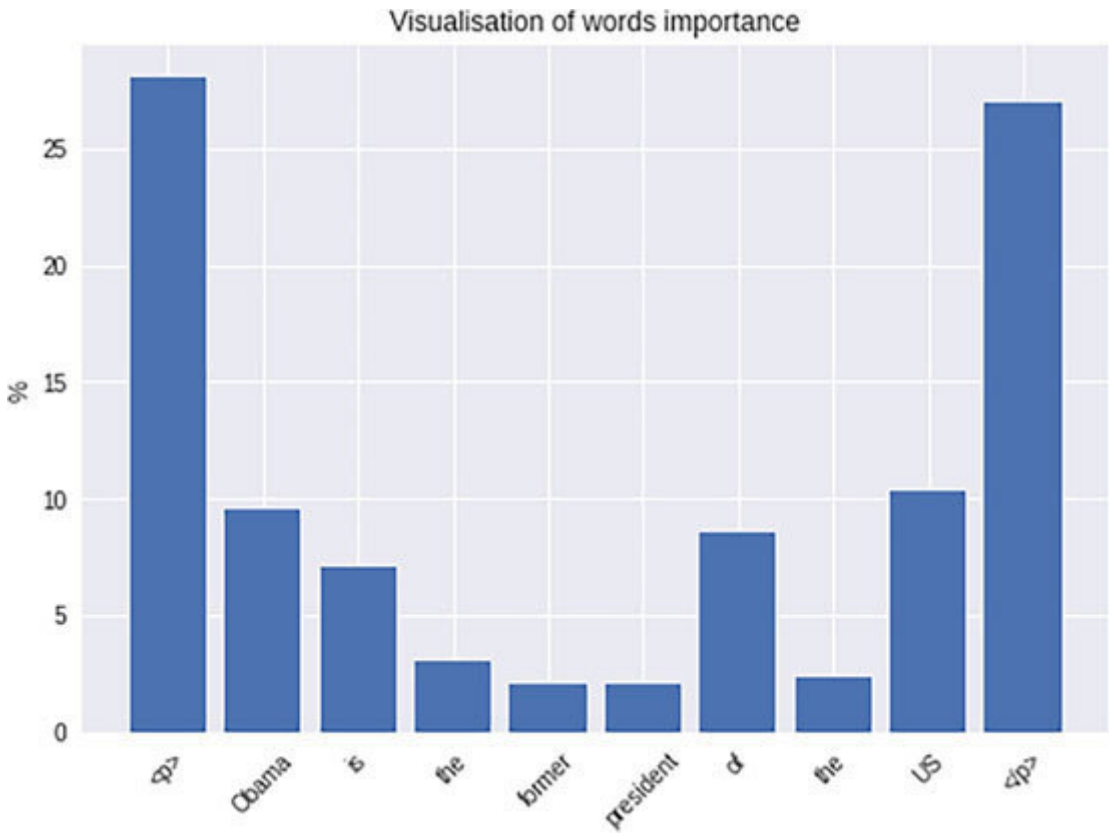


Figure 6.24: *Word importance by plotting vector generated by InferSent.*

Here, the importance of padding is shown higher because we have not completed training a sufficiently large corpus. Once you fine-tune this model on large data, the importance of

padding and stop word will go down, and the importance of the other words will increase. The entire code to reproduce the preceding example is given at

Supervised learning of universal sentence representations from natural language inference data can be found at

Understanding and Using BERT

BERT is short for Bidirectional Encoder Representations from Transformers and is a recently discovered technique for embedding generation by Google researchers. BERT is the state-of-the-art model that provides great results on a wide variety of NLP tasks. It is the key technical innovation applying bi-direction training of the transformer to the language modeling. As we saw in the previous recipe, the ELMo model with the bidirectional model shows better accuracy on language modeling tasks. Extending the concept of ELMo, BERT also uses the bi-direction trained model. In the BERT paper, the researcher used a novel technique, Masked LM (MLM), which allows bidirectional training of the model that was previously impossible.

We have explored the transformer model in detail. The transformer used encoder-decoder architecture with self-attention to produce a prediction for the task. Since BERT's goal is to generate the language model, only the encoder is required. Various models, including BERT, open AI GPT, and ELMo, follow the same logic while training a language model. The task often is to predict the missing word given the context. For example, if the given sentence is "The _____ sat on the mat," the model needs to fill in the blank by predicting the most probable word. ELMo, BERT, and OpenAI

GPT are similar and have little difference in their architecture, as shown here:

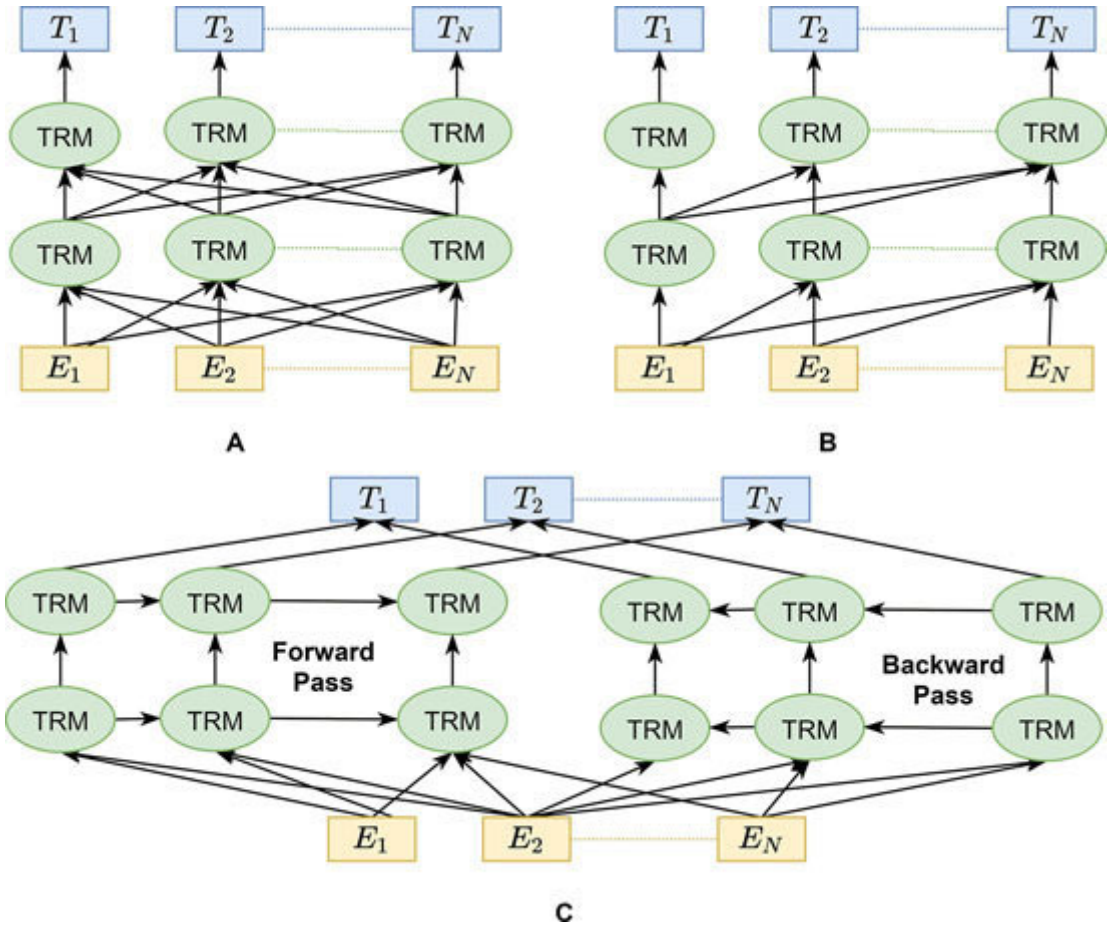


Figure 6.25: *A comparison of architecture between A) BERT, B) ULM-Fit (a model recently proposed by open AI), and C) ELMo-BiLm.*

If all the architecture is similar, why is BERT the most effective? The answer to this question is hidden in the architecture. BERT used word pieces instead of words (e.g., playing -> play + ##ing), and this helps reduce the vocab size.

BERT uses the transformer model as the core component, as we saw in the first recipe of this chapter that the transformer is made by stacking encoder and decoders. Each encoder houses a self-attention and feedforward network. BERT takes input as a combination of positional embedding + sentence embedded and the token embeddings.

To deal with two sentence-related problems like sentence classification, BERT used [sep] as a token to separate two sentences and used sentence embedding. Sentence embeddings are constant for one sentence but different across two. It helps the model know where one sentence ends and the second starts to treat them differently.

To use the model for classification, the author used [CLS] token.

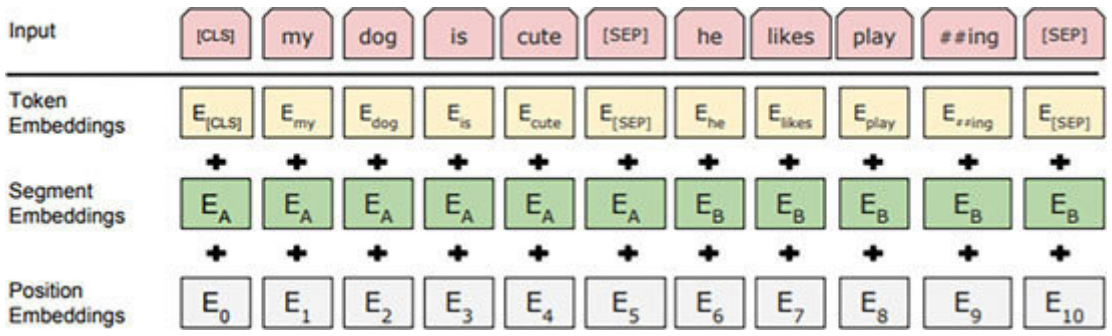


Figure 6.26: *Token, sentence, and positional embeddings being used in the BERT model.*

Source: *https://arxiv.Org/abs/1705.02364*

The model was trained for two models simultaneously: the masked language model and the next sentence prediction

model.

Masked language As is evident from the name, some random words in the sentences are masked, and the model is asked to predict them. It is a major task in this model. In Masked language model paper, they reported randomly masking 10-15% of the input words. The main issue with these techniques is that the model only gets trained for masked tokens. When there is no masked token model, it just avoids the input and provides a random word as the output. This way, the model gets trained for only 10-15% of the input. For better learning, the author replaces a correct word with the dummy words in the rest of the sentences. Inserting random words in the sentences is the strongest form of noise, and the model is compelled to predict masked words after handling the noisy word. This makes the model more robust.

Next sentence prediction In addition to MLM, BERT is trained on the next sentence prediction task by taking question answering or natural language inference tasks. These tasks require knowledge of sense structure. As shown above, the model uses the [sep] token to separate two sentences. In 50% of the cases, the first and second sentences change their order, and in the remaining 50% of the cases, time random sentences are swapped. The model is supposed to predict whether the second sentence is random. Here's an example for the two sentences:

Input = [CLS] the man went to [MASK] store [SEP] he brought a gallon [MASK] milk [SEP]

Label = IsNext
Input = [CLS] the man went to [MASK] store [SEP] penguin [MASK] are fight ##less birds [SEP]
Label = NotNext

Fine-tuning: A sentence beginning with [CLS] token is given as input for classification. The BERT encoder produces a sequence of the hidden state. The task is to convert these output into a single vector, and there are multiple ways to to do this, like applying max pooling and using the attention layer. The author, however, uses a simple approach to take only the first hidden state output. All the input given to the encoder layer interact with each other in the self-attention module, so it is estimated that the first hidden state can be compelled to provide crux of the learning. BERT has two model architectures: one is with a small model and has 12 encoders stacked on one another, and the bigger one has 24 encoders stacked on one another. BERT also has larger feedforward-networks comprising 768 and 1024 hidden units, respectively, and more attention heads 12 and 16, respectively.

Training BERT on having 12-layer to 24-layer Transformer on a large corpus (Wikipedia + BookCorpus) with 1M update steps takes four days on 4-16 Cloud TPUs, and it is fairly expensive. Fine-tuning such a model takes an hour on a single Cloud TPU or a few hours on a GPU. Fine-tuning is fairly inexpensive, but it is required when you are working in specific areas like investment banking and health analytics. So for our purpose, we can use the pre-trained model provided by Google. This model can be used in two ways:

Flair - by Zalando research

BERT - Server client system

Using flair Flair is the open-source project by Zalando research, and it has various embeddings, including BERT and ELMo, provided in the easy-to-use APIs. BERT can be used as follows:

```
fromflair.embeddings import BertEmbeddings
from flair.data import Sentence
# init embedding
embedding = BertEmbeddings()
# create a sentence object
sentence = Sentence('The grass is green .')
# get embeddings
for token in sentence:
print(token)

print(token.embedding)
>>> Token: 1 The
>>> tensor([-0.0323, -0.3904, -1.1946, ..., 0.1305, -0.1365,
-0.4323])
>>> Token: 2 grass
>>> tensor([-0.3973, 0.2652, -0.1337, ..., 0.3715, 0.1097, -1.1625])
```

vector for each word will be of size :

BERT server client This makes more sense as BERT is a bulkier model it's good to run it on the powerful model and

serve it to client APIs from there. BERT-as-service A module on GitHub has excellent server-client support for BERT. BERT with BERT-as-service can be used as:

```
!pip install bert-serving-server # server
!pip install bert-serving-client # client, independent of 'bert-serving-server'
# Downloading and loading models
!wget
https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.zip
!unzip uncased_L-12_H-768_A-12.zip
!bash bert-serving-start -model_dir uncased_L-12_H-768_A-12/ -num_worker=2
# Encodind Sentences
from bert_serving.client import BertClient
bc = BertClient()
bc.encode(['First do it', 'then do it right', 'then do it better'])
```

Executable code for the discussed utilities is provided in the Jupyter Notebook given at

A newly released model by open AI—ULMFiT—was published in the "Universal Language Model Fine-tuning for Text Classification" paper. In state-of-the-art on six text classification tasks, ULMFiT reduces the error by 18-24% on the majority of datasets. ULMFiT is outside the scope of this book, but one must try this model.

For more details, you can refer to the following links:

Universal language model fine-tuning for text classification: *https://arxiv.org/pdf/1801.06146.pdf*

BERT: Pre-training of deep bidirectional transformers for language understanding: *https://arxiv.org/abs/1810.04805*

Conclusion

This chapter covered all the required advanced models to build a state-of-the-art NLP application. Transformers are a basic building block of all models like BERT and Megatron. We also saw that deeper CNN like fully-parallelizable networks are the future of embeddings. Then, we looked at methods to convert the entire sentence to the vector for easy application of NLP on the entire sentence instead of processing it word by word. Then, we learned how to combine CNN and RNN into one network to get contextual embeddings using ELMo. Finally, we explored how to easily use the most-talked about model in the NLP world—BERT. It is the thriving area of research in the domain of NLP, and all major companies are working to produce models to break the current state-of-the-art.

The next chapter will help you learn how to apply deep learning to NLP tasks.

CHAPTER 7

Applying Deep Learning to NLP Tasks

This chapter will cover the practical aspect of NLP by applying NLP to various real-life use cases. This chapter helps apply the concept of CNN, LSTM, and transformer studied in the previous chapter. Topics like topic modeling, text generation, text summarization engine, language translation using a transformer, sentiment analysis, and named entity recognition will be covered in-depth in this chapter. This chapter is equipped with the minimum code required to produce reproducible results along with an in-depth intuitive explanation.

Structure

In this chapter, we will cover the following topics:

Topic modeling

Text generation

Building text summarization engine

Building language translation using a transformer

Advancing sentiment analysis

Building named entity recognition

Objective

Reviewing sentiment analysis

Understanding topic modeling and using various techniques such as LDA, doc2vec

Understanding text generation

Understanding and building a NER model

Building text summarization engine

Building language translation model

Technical Requirements

The code for this chapter is present in the Ch7 folder at GitHub repository This chapter requires the following packages to execute code:

Tqdm

Networkx

Matplotlib

Pandas

Numpy

Torch

Spacy

Nltk

Chakin

Gensim

PyLDAvis

Scikit_learn

TensorboardX

Torchtext

You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r This chapter uses IPython Notebook/ Jupyter Notebook for easy execution and connecting thoughts with implementation.

Topic Modeling

Topic modeling is the method of finding topics from the collection of the document. There are many methods to discover topics from the set of documents, including latent dirichlet allocation, latent semantic analysis, probabilistic latent semantic analysis, and a deep learning-based lda2vector. Latent Dirichlet Allocation or LDA is the most-used method for topic modeling. Some applications of topic modeling are as listed:

To suggest books based on topics/ keywords found in previously purchased books.

To cluster articles based on common topics.

To identify important events in the year by analyzing news dumps; such events can be financial events, and mapping such events to the stock market may provide an additional variable that can help predict future movements better.

To cluster similar images together, creating images similar to the documents.

LDA is the most widely used method and also provides an idea about other methods used in topic modeling. Let's say you have 1000 documents, from which you have chosen a list of 1000 commonly occurring words. Let's say each document has 250 of these commonly occurring words. Commonly occurring words found in documents usually suggests that two documents are discussing the same topic. If you connect each topic to each document, that is about 250 * 1000 = 250,000 associations. These many associations are not acceptable, as one word cannot be related to each document with equal weight. The following is the diagram of the all-topics–to-all-documents connection:

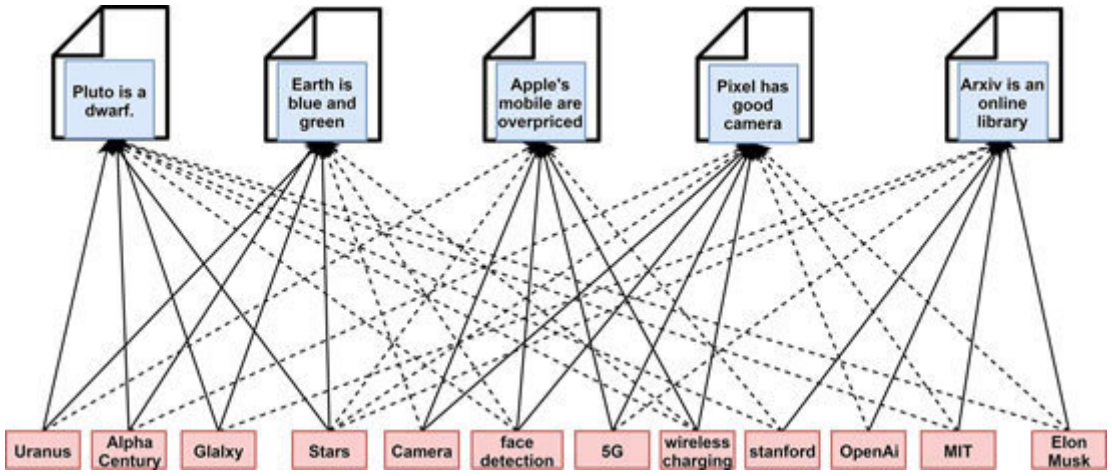


Figure 7.1: *The possible connection of words to the document.*

Solid connections show high connectivity, while dotted connections show less connectivity and weaker connections.

Let's decrease this connection to get fewer important connections. Let's assume that all documents talk about the same thing but don't use the same words. It is something

latent and does not have an absolute word, but it is more like having the same topic. This is done by introducing a latent layer similar to the hidden layer in the neural network. This latent dimension is shown in the following diagram:

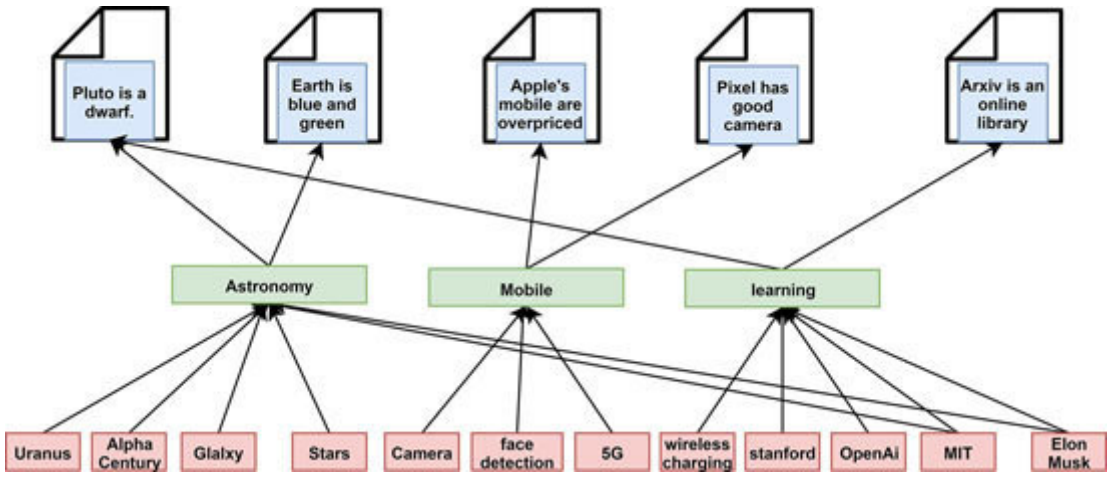


Figure 7.2: *Latent dimension*

Bringing topic as the latent dimension, where the topic serves as the hidden dimension between word and document.

This latent—learning—is the same for all documents. Let's take five latent topics that are connected to all documents and all topics are connected to words. This makes 1000*5 documents to topic connections and 5*100 of the topic to word connections. The total is 10,000 connections. We have reduced the number of connections by 25 fold. This is the core concept of LDA. The latent space finally represents topics. Moving ahead, let's see how these topics are learned.

Before diving into the details, let's fix our notation:

The number of topics a document belongs to (a fixed number); each topic can be given by where $k \subset$ This is predefined by the user.

The vocabulary, each word, can be given by where $\nu \subset$

The number of documents; each document can be given by where $m \subset$

Numbers in each document.

A word document is represented as a one-hot encoded vector of size V.

(capital w): Represents a document (that is a vector of “w” s) of N words.

Corpus, a collection of *M* documents.

z: A topic set of k topics; a topic is a distribution word. For example, it might be planet = (0.7 earth, 0.1 pluto, others).

The following diagram is just for easier understanding and does not represent the actual working of the LSA:

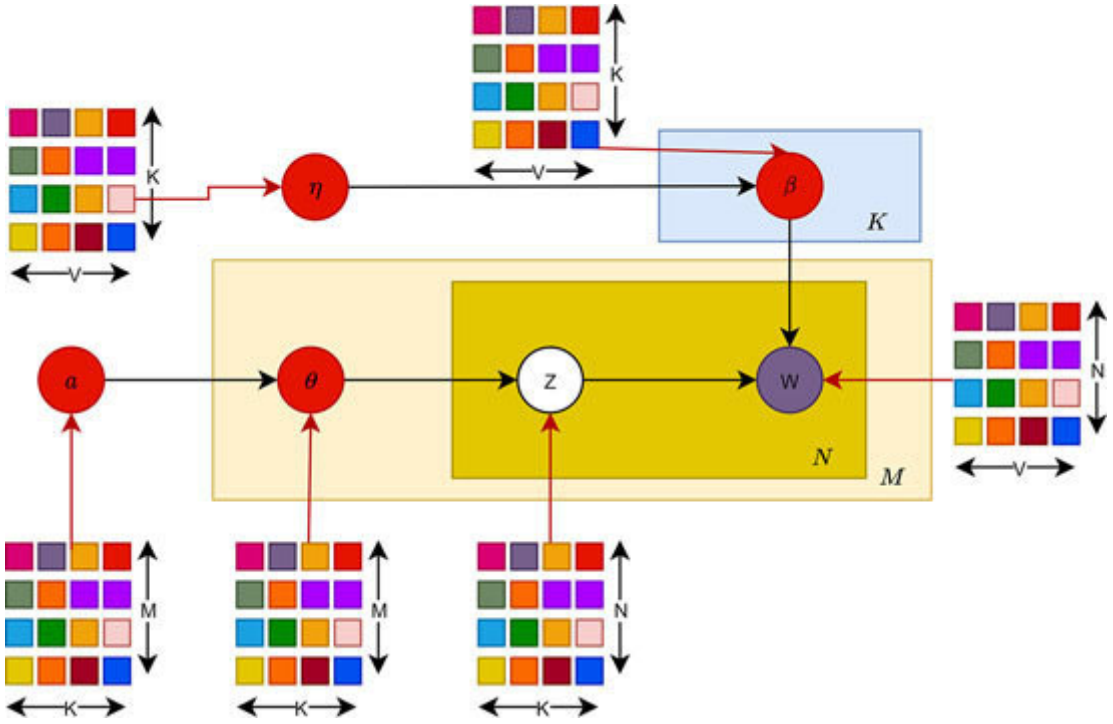


Figure 7.3: *All the matrices involved in the LDA optimization.*

As shown in the preceding diagram, there are certain constants including:

α: Distribution of topics is for all the documents in the corpus looks like, α govern the distribution

θ: A matrix where represents the probability of the document containing the topic.

η: Distribution of words in each topic. η govern this distribution

β: A Matrix where represents the probability of topic containing the word.

Mathematically, what the overall process is about can be represented as follows:

: : : : :

I have a set of *M* documents, with each document having *N* words and each word generated by a single topic from a set of *K* topics. I'm looking for the joint posterior probability of:

θ: A distribution of topics, one for each document.

z: N topics for each document.

β: A distribution of words, one for each topic.

Given,

D: All the data we have (this is, the corpus)

α: A parameter vector for each document (document - topic distribution)

η: A parameter vector for each topic (topic and word distribution)

The only challenging thing remaining is to calculate the posterior probability, and it cannot be easily solved. This problem belongs to the class of optimization-related problems. It requires loss calculation, correcting all the learnable parameters and matrices, and repeatedly relating this until it converges. lda2vec is another related algorithm used for topic modeling. Christopher E Moody proposed this technique in his publication *Dirichlet Topic Models and Word Embeddings to Make* lda2vec is an extension of Word2Vec and LDA to learn word topic and document vectors jointly. It particularly utilizes the skip-gram model, which is also used for training word2vec:

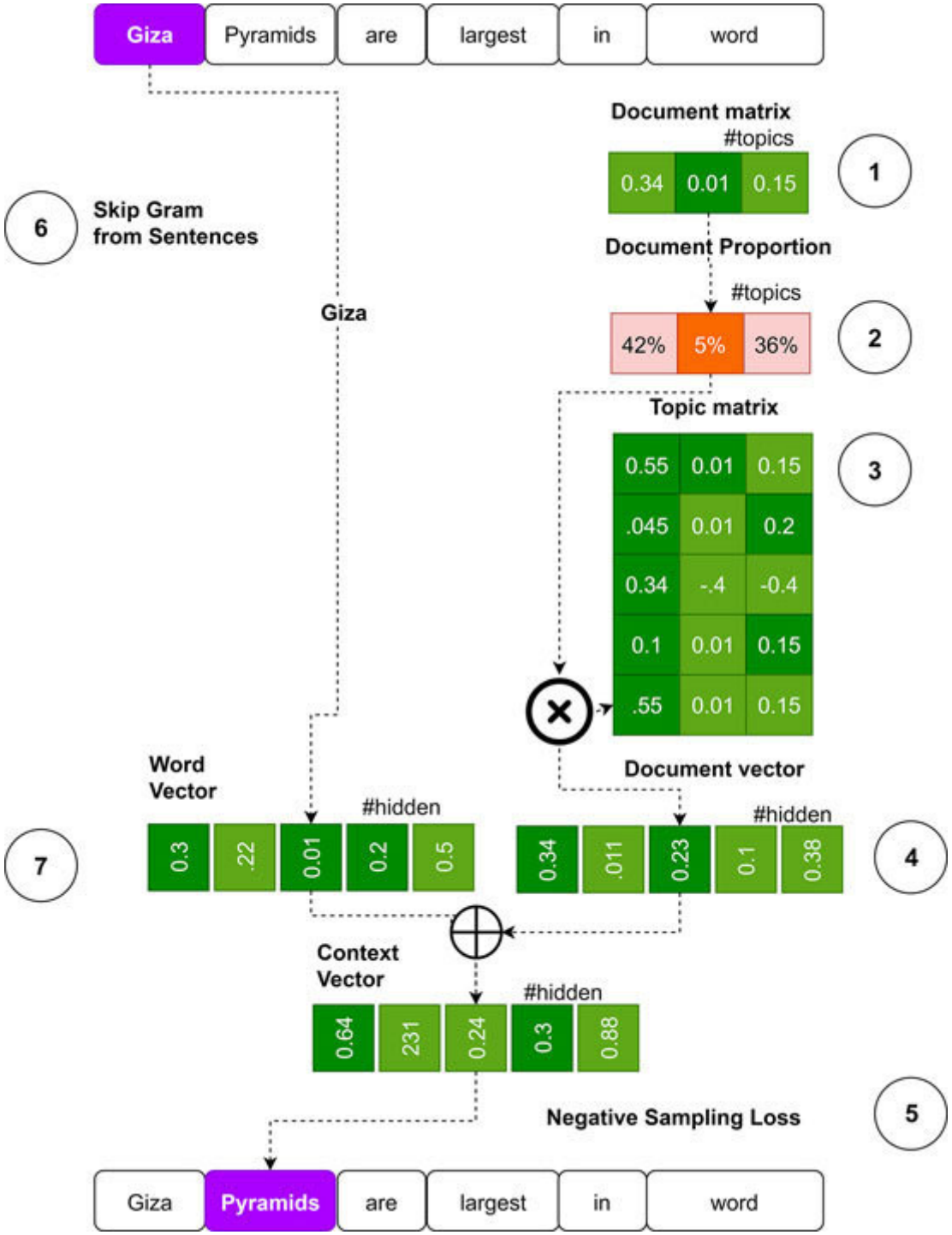


Figure 7.4: *An illustration showing how lda2vec is trained.*

Topic modeling can be done using the following pre-built libraries like Gensim. In the present implementation for demonstration purposes, we will use 20-newsgroup data, a

collection of 20,000 newsgroup documents. 20-newsgroup data has been extensively used in the experimental demonstration related to text classification and text clustering. The dataset has related groups (for example, comp.sys.ibm.pc.hardware / as well as distant classes (such as misc.forsale /

Well, the classes' relation is of lesser concern to us, as we are not using the dataset for text classification. Still, we will use this dataset for extracting topics from the description.

Applying LDA

Data preparation: Data preparation includes reading data from the JSON file at After reading the content into a data frame, the content looks as follows:

follows:
follows:
follows:
follows:
follows:
follows:

Table 7.1

Then, some pre-processing is done, including tokenization of content and removal of email, new line, and quotes.

LDA: Once all the sentences are tokenized into words, they are converted to bi-gram and trigram using This can be done as shown:

Build the bigram and trigram models

```
bigram = gensim.models.Phrases(data_words, min_count=5,
threshold=100) # higher threshold fewer phrases.
trigram = gensim.models.Phrases(bigram[data_words],
threshold=100)

# Faster way to get a sentence clubbed as a trigram/bigram
bigram_mod = gensim.models.phrases.Phraser(bigram)
trigram_mod = gensim.models.phrases.Phraser(trigram)
```

After these words are referenced with IDs because the Gensim LDA model takes IDs of the word as input, Gensim LDA can be applied as shown:

```
lda_model = gensim.models.ldamodel.LdaModel(corpus=corpus,
id2word=id2word, num_topics=20, random_state=100,
update_every=1, chunksize=100, passes=10, alpha='auto',
per_word_topics=True)
```

In the output, you will see that the output for each topic is described by a set of the keywords given, along with keywords describing the topics. For example, topics are represented as 0.099"car" + 0.036"model" + 0.031"option" + 0.027"oil" + 0.027"mike" + ' + '0.026"engine" + 0.024"michael" + 0.021"water" + 0.020"door" + 0.018"internal"'. This means the top 10 keywords that contribute to this topic are car, oil, engine, and so on, and the weight of car on the topic 0 is 0.099.

Visualizing output: A pyLDAvis library can be used for visualization and inference of it is available at https://github.com/bmabey/pyLDAvis and can be installed simply with pip install The visualization function requires trained tokenized and preprocessed corpus, and index to tokens:

```
# Visualize the topics
pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim.prepare(lda_model, corpus, id2word)
```

The interactive visualization looks as follows:

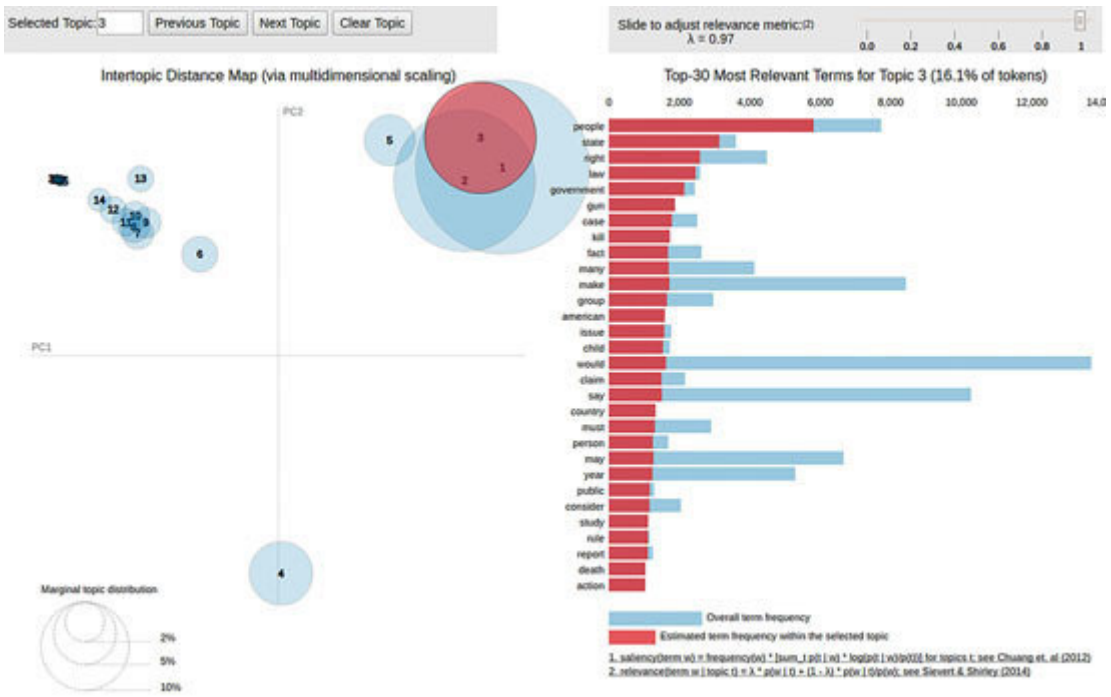


Figure 7.5: *Interactive visualization*

Each bubble on the left-hand side plot represents a topic. The larger the bubble, the more prevalent the topic is. A good

topic model will have fairly big, non-overlapping bubbles scattered throughout the chart instead of being clustered in one quadrant. A model with too many topics will typically have many overlaps, and small-sized bubbles will be clustered in one region of the chart. If you move the cursor over one of the bubbles, the words and bars on the right-hand side will be updated. These words are the salient keywords that form the selected topic.

Evaluating the model: Model perplexity and topic coherence provide convenient measures to judge how good a given topic model is. The topic coherence score has been more helpful. A reference about topic coherence and perplexity is given in the next subsection. Both measures are calculated as given here. The lower the perplexity, the better the topic model.

```
# Compute Perplexity
Perplexity = lda_model.log_perplexity(corpus)
print('Perplexity: ',Perplexity)
# Compute Coherence Score
coherence_model_lda = CoherenceModel(model=lda_model,
texts=data_lemmatized, dictionary=id2word, coherence='c_v')
coherence_lda = coherence_model_lda.get_coherence()
print('Coherence Score: ', coherence_lda)
```

The preceding implementation of the LDA is provided at

An improved LDA algorithm, Mallet's version of LDA, gives better results. LDA mallet is available with Gensim, and it is implemented in the class within Gensim.

You can refer to the following links for more details:

Latent Dirichlet *www.jmlr.org/papers/volume3/blei03a/blei03a.pdf*

Evaluation of topic Topic Coherence:
https://datascienceplus.com/evaluation-of-topic-modeling-topic-coherence/

Text generation

LSTM or any RNN network is generally treated as a predictive model and used for classification. RNN models can also be used as generative models to generate a related new sequence by learning the text. Generative models help study how well the model has learned our data, and they can help augment the data. Generative Adversarial Network-based methods are widely used for data augmentation. In this recipe, we will learn how to develop a generative model using LSTM on different textual data.

Here, we are using RNN as the generative model. Generative models with RNN are similar to the normal classification module, except that the number of classes in generative models is very large. Before going further, we will see how data for such a model is prepared. Here's a crisp explanation of how to prepare data:

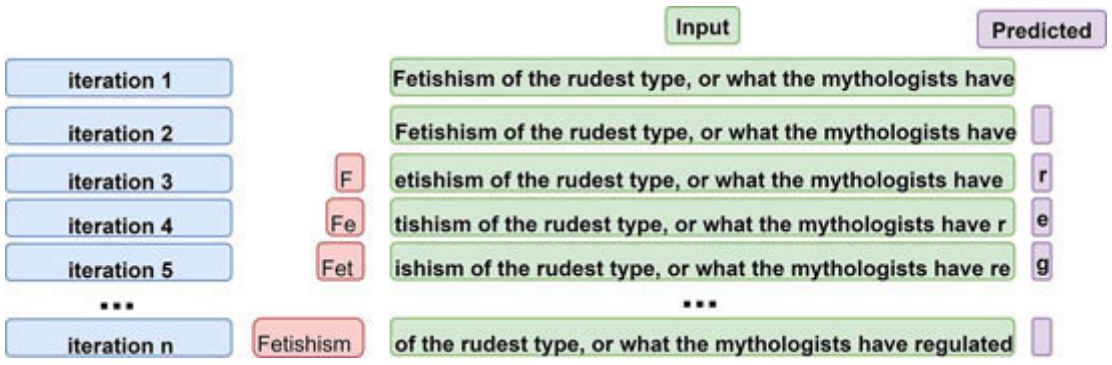


Figure 7.6: *An illustration showing how text generation works.*

The input is indicated in green for each iteration, and the next character (red) serves as the target. This input window then slides, and the next character after windows will be selected as the target. This continues for the entire dataset. In training, the target and predicted characters are then used for the loss calculation. In prediction, the predicted character is appended to the sequence; the very first character is removed from the sequence to make the input size equal, and these steps are repeated for several characters to be generated.

The preceding data preparation considers the character model, and it is similarly applicable to the word-based model. Here, we have taken the character length as 200, so 200 characters will act as the feature, and the next character will be treated as the target. Once this data is prepared, it is inserted into any of the RNN units, such as GRU or LSTM. Once the data is prepared, the next step is to feed it to the RNN unit. The data feeding is done as shown in the following diagram. A total of 200 characters are given to the GRU at once, and this is equal to 200 time steps. After the model is prepared by training it as described earlier, the next generation from such a model follows. In the generation, the following steps are performed to get the next character in the sequence:

Start with the random starting point and take 200 characters from the training/test text.

Provide these characters to the GRU so that the probability distribution for the next character can be generated.

Pick up the next character by sampling.

Add this character to the initial sequence; now we have 201 characters. Take the last 200 characters and repeat steps 2, 3, and 4.

In step 3, as I said that we sample the character from the probability distribution, and we are not taking the character with the highest softmax probability. This sampling technique will insert some uncertainty and let the solution pick a word with less good predictions sometimes. This sampling technique will help minimize repetition. In an implementation, we will see how a non-uniform random sample is drawn based on the prediction probability.

Understanding the Network

In the present implementation, I am using a book from Gutenberg repository –"Gypsy Sorcery and Fortune Telling by Charles Godfrey Leland". You can take any text in replacement for this book. The pre-processing part, along with the entire implementation, is covered in the CH7/character_based_generation.ipynb script. The first part to be discussed here is the network architecture:

```
class CharRNN(nn.Module):
def __init__(self, tokens, n_steps=100, n_hidden=256,
n_layers=2,
drop_prob=0.5):
super().__init__()
self.drop_prob = drop_prob
self.n_layers = n_layers
self.n_hidden = n_hidden
self.chars = tokens
self.dropout = nn.Dropout(drop_prob)
self.lstm = nn.LSTM(len(self.chars), n_hidden, n_layers,
dropout=drop_prob, batch_first=True)
self.fc = nn.Linear(n_hidden, len(self.chars))
self.init_weights()
def forward(self, x, hc):
''' Forward pass through the network '''
x, (h, c) = self.lstm(x, hc)
```

```
x = self.dropout(x)
# Stack up LSTM outputs
x = x.view(x.size()[0]*x.size()[1], self.n_hidden)
x = self.fc(x)

return x, (h, c)
```

The network architecture is very simple; it takes the character encoder to numerical indices as the input. This input is passed to the LSTM cell, which then gives our output shape, hidden, and the cell state. A dropout is applied to the LSTM output, and all the output for each time-step are stacked over one another. Let's take one example, assuming the following variables:

batch size 32

hidden size =256

Sequence length = 100

Considering the preceding input from the above-defined variable, the output from the LSTM will be of size [1, batch size, hidden size] for each time step. This output will be generated for time steps equal to sequence length (100). These 100 outputs are stacked to form the resultant shape [100*batch size, hidden shape]. This shape is then passed on to the fully connected layer, which will transform [100*batch size, hidden shape] in to [100*batch size, number character]. To simplify understanding, let's take the final shape as [32,

100, number of unique characters]. After applying softmax to this shape, it represents the probability of a character out of several unique characters for 100 characters taken as input in a batch of 32. Having understood this, we will move on to the prediction part. The implementation of the prediction function looks like this:

```
def predict(net, char, h=None, top_k=None):
if h is None:
h = net.init_hidden(1)
# character to index
x = np.array([[net.char2int[char]]])

x = one_hot_encode(x, len(net.chars))
inputs = Variable(torch.from_numpy(x), volatile=True)
inputs = inputs.to(device)
h = tuple([Variable(each.data, volatile=True) for each in h])
# forward propagation
out, h = net.forward(inputs, h)
# p = predicted
p = F.softmax(out).data
p = p.to(device)
if top_k is None:
top_ch = np.arange(len(net.chars))
else:
p, top_ch = p.topk(top_k)
top_ch = top_ch.cpu().numpy().squeeze()
p = p.cpu().numpy().squeeze()
char = np.random.choice(top_ch, p=p/p.sum()) #random
choice on the basis of weighted distribution
return net.int2char[char], h
```

This function takes previously trained network as the input, along with other parameters such as char and hidden state. The argument char is a seed—a short sequence required to initiate the generation; it can be something like “the,” “we,” or any other short sequence. After receiving these arguments, the predict function performs the following steps:

Convert character to numerical index by dictionary lookup.

Create new variables for the hidden state; otherwise, it would backdrop through the entire training history.

Parse numerical indexes to the network and get back probability distribution for the new character, as discussed previously in the network architecture.

Choose the top five largest characters, out of which one will be chosen as the final candidate.

Apply random choice based on weighted distribution using Numpy random with the non-uniform distribution. Our predictions come from a certain probability distribution over all the possible characters. We can make the sampled text more reasonable but less variable by only considering some most probable characters. It will prevent the network from giving us completely absurd characters while allowing it to introduce some noise and randomness into the sampled text.

Repeat 1-5 until all max specified length return all the generated characters.

When I allow training the network for 25 epoch, the final generated text looks as given here:

“there when a seal of the places a stall as to the Strunk of all seence, into a command trace the whole the particle that or taken to treat to the work and reperiously trancelly would hus believed to the cu”

It seems like the network has learned to place articles like **the** and but it is making spelling mistakes. Allowing the network to train for longer could yield better results. With a small amount of text, the chance of detecting novelty in the generated text is highly rare. The decrease in training, as well as validation loss, is as given:

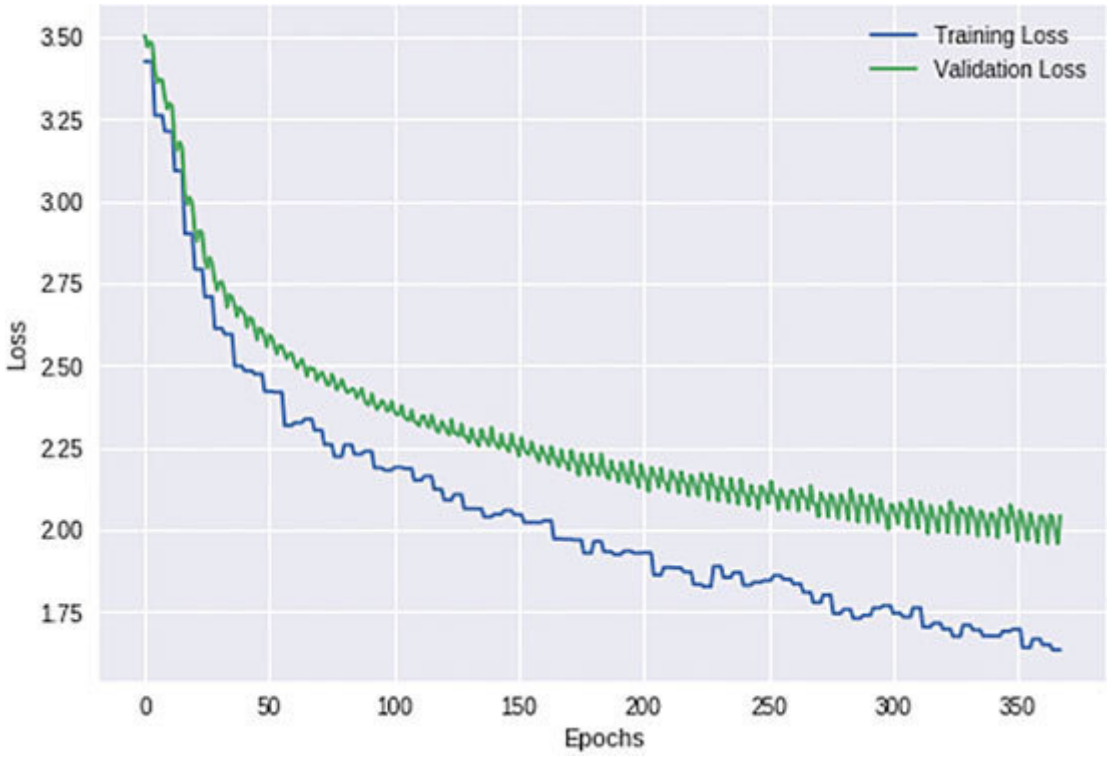


Figure 7.7: *Decrease in training and validation loss while training character-based text generation model.*

There are many wonderful texts available online and not protected under copyright law. One of them is Project Gutenberg. You can easily download any of the books from there and experiment. You can increase the data and see its effect, and you can determine whether it can understand deeper aspects of the text. The text generated by RNN generally seems to be right and has proper grammar. New sentences are hard to find in the generated sentences. The previously discussed method can also generate code if a sufficiently large amount of code is given for training. Try it yourself to see whether the generated text has proper indentation, opening and closing of the bracket, and whether proper variable assignment is generated.

Take a look at the following references for more information:

Toward the controlled generation of text:
https://arxiv.org/abs/1703.00955

Kaggle topic modeling on tweets:
https://www.kaggle.com/errearanhas/topic-modelling-lda-on-elon-tweets

Building Text Summarization Engine

Text summarization is a method to convert a large document into smaller documents while keeping the meaning intact. There are two types of summarization techniques:

Extractive Summarization wherein the most important sentence from the given text is selected so that the extracted sentence represents the same meaning.

Abstractive Abstractive summarization is a generative technique wherein the algorithm first goes through the entire document and then summarizes the text. Here, the algorithm rewrites the entire summary instead of just copying sentences. Abstractive summarization can be done with application algorithms such as LSTM, CNN + LSTM, and other generative models.

Based on the number of documents involved, document classification can be divided into two types:

Single document It involves summarization from a single document, and extractive summarization generally performs better here.

Multi-document It involves preparing a summary from multiple documents. The main problems in summarizing multiple documents are:

Repetition in the summarized text.

Losing local, paragraph-level context.

Needs lots of data for training summarization.

The algorithm needs to remember lots of data, do it requires a bulkier model.

In this recipe, we will understand how extractive summarization works within detailed algorithm implementation. One of the popular algorithms for text summarization is text rank. Text is an extractive and unsupervised technique for text summarization. The page rank algorithm inspires text to rank. Page rank is popularly used in search engine development like Google. Let's see how page rank works. For example, we have five pages with some hyperlinks between them. A hyperlink is a way of navigating between two pages. A page's connectivity is defined as per the number of hyperlinks present between them. For example, let's say we have the following hyperlinks in five pages. The rank of these pages can be defined based on the several links present between them using the Pagerank algorithm. The connectivity between five pages can be represented with probabilities, and this probability can be easily represented with a matrix of 5*5, as shown with all the

probabilities. The page with no connectivity is called a dangling page, and here, --- is the dangling page. The entire process of text rank can be summarized in the following steps. This step is very similar to the page rank algorithm.

Instead of web pages, we use text in text rank.

The hyperlink between two pages as the connectivity measure is replaced by the sentence similarity in the text rank algorithm.

The matrix is constructed with a similarity between all the sentences.

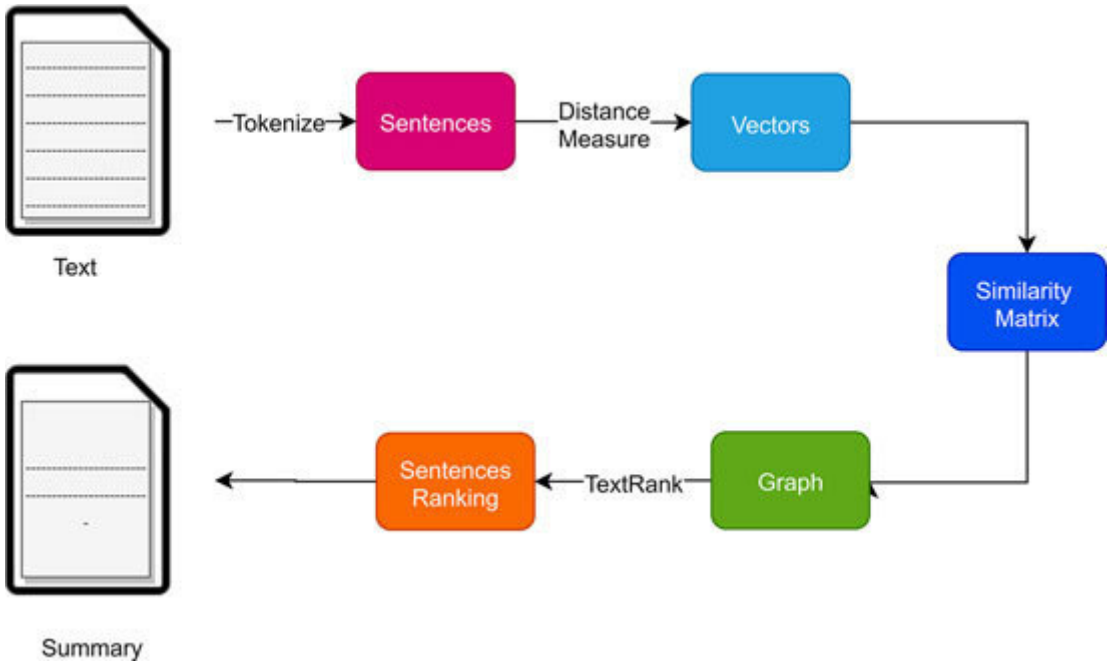


Figure 7.8: *Essential steps in the extractive summarization technique using T.*

In single document summarization, the document is tokenized.

In multiple documents, the first step is to concatenate all the documents, and then we tokenize the text.

Apply distance measure on the sentence after applying embeddings to the tokens. Using this, a similarity matrix is calculated.

A weighted graph is constructed.

Page rank is applied to the constructed graph to form the sentence ranking.

Based on the sentence ranking, the summary is constructed.

Abstractive Text Summarization

The example data is provided at Ch7/data/

Preprocessing and Tokenization is carried out using the NLTK tokenizer. To vectorize the word, we will use Glove Embedding. This includes removing all the non-alphanumeric character and stop words.

Vector representation of All the words in the sentence are vectorized and the mean of all the words is taken to have a vector that represents an entire sentence. It can be done very easily, as shown:

```
sentence_vectors = []
for each_sent in clean_sentences:
if len(each_sent) != 0:
v = sum([word_embeddings.get(word, np.zeros((100,))) for
word in each_sent.split()])/(len(each_sent.split())+0.001)
else:
v = np.zeros((100,))
sentence_vectors.append(v)
```

A variable sentence_vectors will be holding the vector for sentence representation of each vector.

Similarity matrix A cosine similarity will be calculated between all the sentences. For example, if we have 100 sentences, the cosine similarity matrix will be of size (100, 100). It can be simply done as follows:

```
for i in range(len(sentences)):

for j in range(len(sentences)):
if i != j:
sim_mat[i][j] = sentence_vectors[j].reshape(1,100))[0,0]
```

The result is stored in the pre-initialized matrix sim_mat with all zeros.

Applying the PageRank Page rank is applied with the NetworkX package. NetworkX is the collection of algorithms related to the graph and tree data structure. To apply page rank, sim_mat is treated as a weighted graph, and page rank is applied as shown:

```
nx_graph = nx.from_numpy_array(sim_mat)
scores = nx.pagerank(nx_graph)
```

Summary The score is generated for each sentence, and sentences are ranked according to their scores. The sentence with the highest ranks is given as a summary. The code required to reproduce the extractive summarization process is provided as an IPython notebook at

We saw how to use extractive summarization, but it is a very simple technique with many drawbacks. Some of these drawbacks are listed here:

It gives repetitive output.

The output sentence doesn't tell a continuous story; sentences are not related many a time.

It does not produce human-like summarization.

Due to this reason, another technique called **summarization** gained popularity. Abstractive summarization produces the summary by rewriting the content after going through the reference text. It uses the concept of RNN based encoder-decoder and recently coined pointer networks.

Building Language Translation Using a Transformer

The transformer is used in industrial-grade language translation systems like openNMT. Due to its capability to use GPU efficiently, it has replaced the previously known state-of-the-art RNN-based attention network architectures. In *Chapter 4, Using RNN for* we saw how to design a language model for translation using the attention mechanism. In *Chapter* we saw how the mighty transformer works by connecting all the dots. In this recipe, we will understand how to use a transformer for the language translation task. This recipe is focused on how to prepare data and how to provide input to the transformer model.

Before moving ahead, we need to fix some notation. Let's assume that we are trying to train out transformer for German to English translation:

Represents the source language (German)

The language to which the text is converted (English)

The following are the steps involved in preparing and providing input to the transformer model for language translation purposes. The first step is preprocessing, whereby data is brought in to the shape so that the model can take it

easily as input. The entire process is shown in the following diagram:

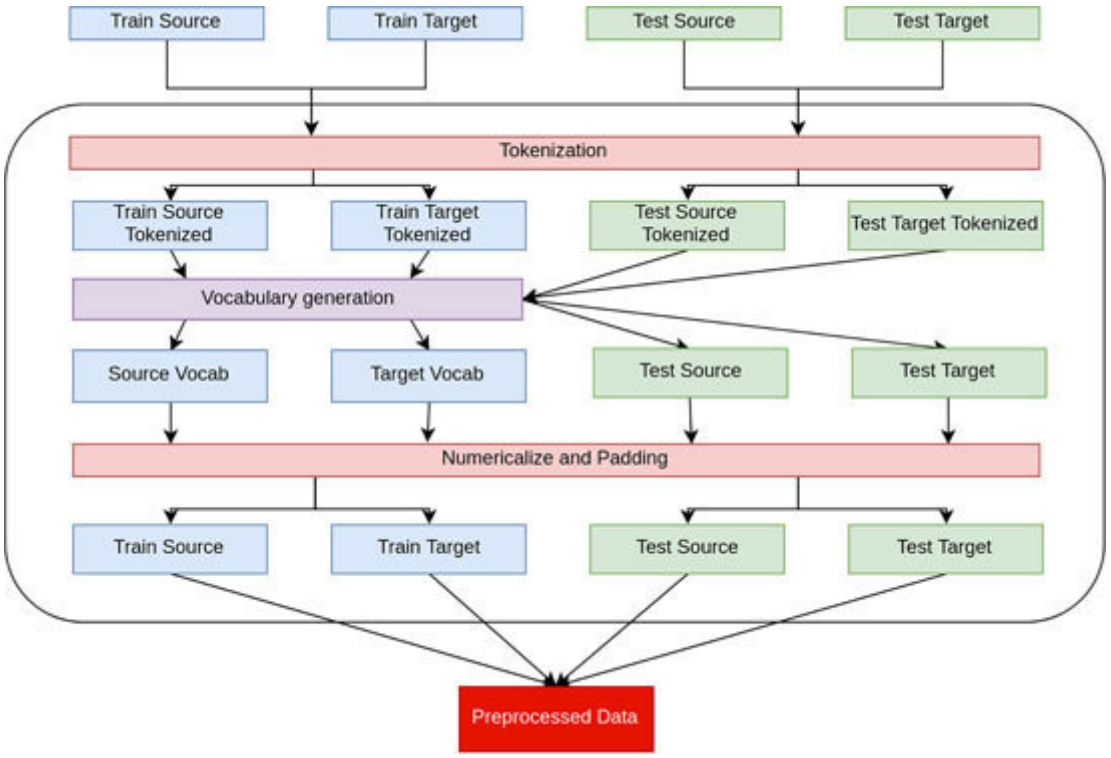


Figure 7.9: *Steps involved in preprocessing*

We start with four partitions of data namely: train source, train target, test source, and test target. Train source and test target have sentences in one language, and train target and test target have corresponding sentences in another language. Sentences are first converted to word tokens. Vocabulary is generated using Train data, and it is applied to the train and test data. All the input is converted into numbers, where each unique number represents a word in the vocabulary. Such a sentence is padded with the beginning of the sequence and end of sequence tags. Then, all four parts are written to the disk as preprocessed data:

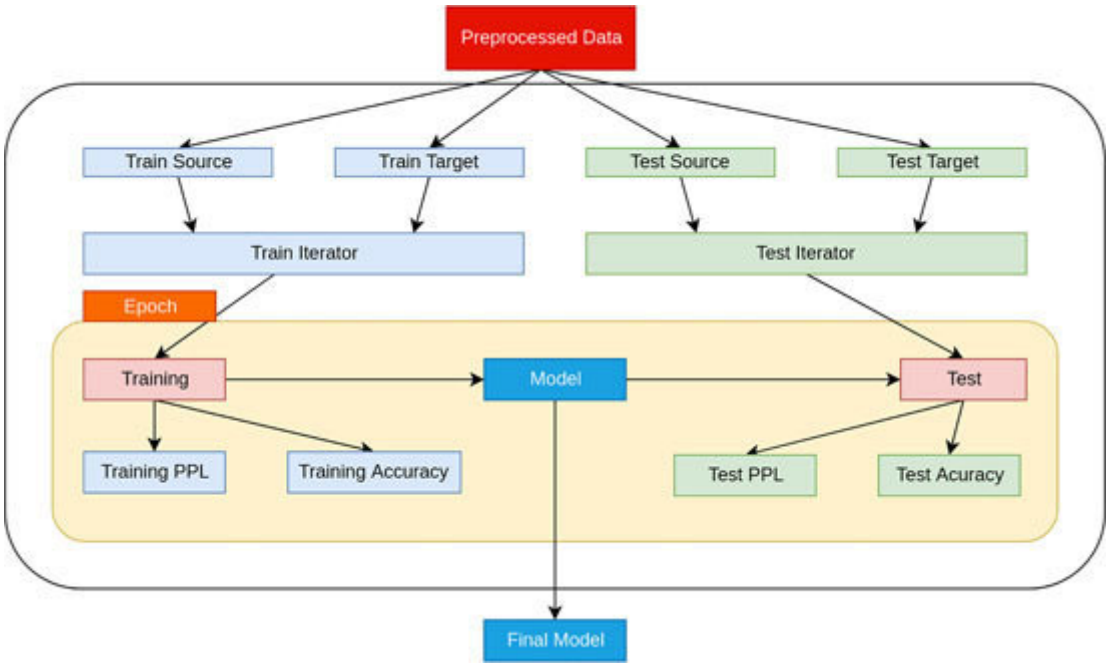


Figure 7.10: *Training a transformer on the preprocessed data.*

The learning process require preprocessed data; two data loaders, one each for train and test data. The batch is designed, which provides an equal number of samples in a batch padded according to the size of the longest sentence in the batch. The data loader also provides the indices for positional embedding. The transformer is trained on the input train source, train target, train source positions, and train target positions. If a sentence in the train source looks like [2, 5, 446, 23, 26, 88, 1, 0, 0, 0, 0, 0, 0], the positional input will be [1, 2, 3, 4, 5, 6, 7, 0, 0, 0, 0, 0, 0]. Positional input is to represent the position of the token in the positional embedding phase of the transformer.

This is another kind of measure generally used in language models. It is a way to capture the degree of ‘uncertainty’ a model has in predicting (assigning probabilities to) some text. It is related to Shannon entropy, so the lower the perplexity, the lower is the entropy and the better is the certainty of the model. Mathematically, let’s say cross-entropy between target distribution p and the predicted distribution q is given by = then, the perplexity (PPL) can be given by cross-entropy over any base b. PPL = This equation is often simplified by taking exponential as the base PPL = In this recipe, will use perplexity (PPL) as a measure to track model performance on the train and test data.

Using a Transformer

Preparing data: Usually, this step includes reading the data from the file and preprocessing it. The output of this step is used in the next step as the transformer model's input. I have provided the entire code with the supporting functions required to train the transformer and translate test sentences at I am using a small amount of data to demonstrate how it is prepared and provided as input to the transformer. In this demonstration, I will translate German to English. The data required for training is present in the Ch7/language_translation/data folder. To help you understand the logic, I will walk you through the code along with an explanation. The following conventions are used to keep logic aligned with code:

< File Name > : < Start Line > – < End line >

Here, Start Line and End Line are line numbers of the code snippet being discussed here. From the given file name, preprocess.py script requires four arguments: train source, train target, validation source, and validation target. Run preproceess.py as given:

python preprocess.py -train_src data/newstest2013.de -train_tgt data/newstest2013.en -valid_src data/newstest2013.de -valid_tgt data/newstest2013.en

I am taking the train and validation data as the same to keep the process simple. Otherwise, train and validation samples should always be different.

The preprocess.py - 84:85 --max_word_seq_len argument defined the maximum length of the sentence. These lines count the beginning and end of the sequence, and the token is added to The default value of the max_word_seq_len was 50; now the maximum value is 52:

opt = parser.parse_args() # opt is instance of argument parser opt.max_token_seq_len = opt.max_word_seq_len + 2 # include the ~~and~~

preprocess.py - The read_instances_from_file function is called to take the train source file, along with max and keep the case as the parameters.read_instances_from_file function performs the following operation on the provided data:

Read the data sentence by sentence

If keep_case is false, convert all sentences to lower case

Tokenize the sentence into words

If the number of words is greater than chop the sentence

Wrap the words of the sentence with Beginning of the sequence and End of the sequence token

Warn the user regarding the number of sentences trimmed; if the number of sentences trimmed is higher in fraction, you can increase the max_token_seq_len parameter and rerun it:

train_src_word_insts = read_instances_from_file(opt.train_src, opt.max_word_seq_len, opt.keep_case)

The same read_instances_from_file function is called upon train target, valid source, and valid target.

preprocess.py - 97:101 and One source sentence in one language corresponds to one target sentence in another language. Ideally, the number of sentences in the source must be the same as the number of sentences in the target. If this is not the case, there must be some problem with the dataset. To check this, some line of code is kept. If the source and target files have an odd number of lines, minimum lines are counted from both subsets. The subset that has more lines than the minimum is truncated to reach the minimum. This check is done for both train and validation data. This is not the correct strategy, as the line might be missing from anywhere and will disturb the entire alignment of the train or validation data:

if len(train_src_word_insts) != len(train_tgt_word_insts):
print('[Warning] The training instance count is not equal.')

```
min_inst_count = min(len(train_src_word_insts))
len(train_tgt_word_insts))
train_src_word_insts = train_src_word_insts[:min_inst_count]
train_tgt_word_insts = train_tgt_word_insts[:min_inst_count]
```

After this function, the train_src_word_insts or train_tgt_word_insts variable will look as shown below. All sentences are passed with ~~and~~ .[[‘~~’, ‘nach’, ‘der’, ‘kanadischen’, ‘krebsgesellschaft’, ‘sind’, ‘die’, ‘studien’, ‘zu’, ‘vitamin’, ‘e’, ‘widersprüchlich.’, ‘’], [‘’, ‘während’, ‘eine’, ‘studie’, ‘die’, ‘verringerung’, ‘des’, ‘risikos’, ‘von’, ‘prostatakrebs’, ‘herausgefunden’, ‘hat, ‘, ‘zeigte’, ‘eine’, ‘andere’, ‘eher’, ‘eine’, ‘erhöhung.’, ‘~~’]]

preprocess.py - 105:106 and Removing both the sentences, one from the source and one from target, if a sentence is empty. This check is done for both source and validation:

```
train_src_word_insts, train_tgt_word_insts = list(zip(*[(s, t) for s, t in zip(train_src_word_insts, train_tgt_word_insts) if s and t]))
```

preprocess.py - This code snippet builds source and target vocabulary. The build_vocab_idx function is called to take previously tokenized sentences and which defines the minimum frequency of the word to be kept in the vocabulary. This function performs the following operations:

Counting all unique words from the given tokenized sentences

Building a dictionary having a word and its frequency in the tokenized sentences

Ignoring all the words from the dictionary with frequency lower than min_word_count

Returning all other words

This function is called on source and target sentences to build separate vocabulary for both.

src_word2idx = build_vocab_idx(train_src_word_insts, opt.min_word_count)
tgt_word2idx = build_vocab_idx(train_tgt_word_insts, opt.min_word_count)

The build_vocab_idx function returns the source and target word to index dictionary like: : tgt_word2idx or src_word2idx : {‘’~~: 2, ‘~~’: 3, ‘’: 0,’’: 1, ‘dass’: 4, ‘haben’: 5, ...}. It is called upon tokenized sentences of source and target sentences. If the user provides the share_vocab argument as True, the function is called with both tokenized sentences of source and target and the vocabulary is shared among both, as shown at preprocess.py - 134: Alternatively, you can pass previously constructed vocabulary to this script by specifying it in the -vocab argument, and it will be loaded by the code snippet preprocess.py - 126:132 by using and torch.load() is

similar to numpy save or load object, which is used to save or load binary objects.

preprocess.py - The convert_instance_to_idx_seq function is called, with train source, valid source and train target, valid target along with source word2idx and target word2idx, respectively. Remember, we are not building any vocabulary for validation subset; the vocabulary built for train data is provided to validation to convert words to indices. After these lines are executed, the train and validation sets have all sentences are numbered and look like this: .[[2, 13, 1, 1, 18, 30, 1, 12, 1, 1, 3], [2, 9, 1, 30, 1, 1, 1, 1, 36, 30, 1, 10, 1, 22, 1, 3].

preprocess.py - All the prepared subsets, along with the user arguments provided while executing the script, are saved using the torch.save() function. This is our preprocessed dataset. The previously prepared preprocessed dataset will be provided to train.py as an argument. Then, train.py will train a transformer model on this data.

Training: A Ch7/language_translation/train.py script takes previously prepared data as the argument, along with other predefined arguments. The script can be executed as shown:

python train.py -data /path/to/preprocessed_data

train.py - Loading the preprocessed data in the data variable and retrieving the max_token_seq_len parameter from

previously-stored argument objects.

train.py - The prepare_dataloaders function is called, and it takes loaded “data,”user-defined arguments as the input. The main purpose of this function is to construct a train and validation data loader. This function performs the following operations:

A function class takes and where src_word2idx and tgt_word2idx are word to index dictionaries, src_insts and tgt_insts are the sources and target tokenized and numericallized sentences as loaded from the prepared data.

A TranslationDataset subclass designed by extending torch.utils.data.Dataset reverses the word2idx dictionary to idx2word and returns it along with the original arguments as parameters to the TranslationDataset class. Here, we are using TorchText as the data loader, and in torch text, all other datasets should be subclasses of All subclasses should override which provides the size of the dataset, and and support integer indexing in the range of 0 to len(self) exclusive.

This TranslationDataset class is embedded under This DataLoader takes and the paired collate function in it has two functions:

It calls collate_fn with src_insts or tgt_insts as the argument.

It considers the batch_size and pad all the sentences as per the sentence with the max size.

The collate_fn function also constructs the array for positional embedding.

It returns src_insts or tgt_insts and the positional embedding array

I have provided an output of and the corresponding positional embedding is in the code as the comment.

The num_workers parameter specifies the number of CPU processes to be used for data preparation.

Similarly, training_data and validation_data are generated.

If you print the length of it will be 4. Each of these data loaders has train source, train target, train source positions, and train target positions. Train source and train source positions will of the same shape [32, 52]. The first dimension in shape represents the batch number, and the second dimension represents the longest sentence size. Similar to the target.training_data and validation_data are returned to the main function.

train.py - The transformer model is instantiated with the required parameters.

train.py - Defining an optimizer class with Adam as an optimizer and different functions to update the learning rate, zero the gradient, and update it is given in This optimizer is the same as the one we used in an earlier chapter, but it is defined differently.

train.py - Calling a train function with training, validation data, transformer model object, optimizer, and arguments provided while executing the script. Train function defined at. train.py - It performs the following operations:

Defining log files to report/write epoch loss and accuracy.

At epoch, the following steps are performed:

Training the transformer model for an epoch and giving out train loss and training accuracy.

Using the preceding trained model on the validation data and giving out validation loss and validation accuracy.

Checkpoint the model as a model.ckpt file.

We ran the code on 100 German to English translation dataset. It is a very small dataset, but the model was able to increase accuracy from 0 to 55. It is still nowhere near the

so-called good, but the model is converging, and that's a good sign.

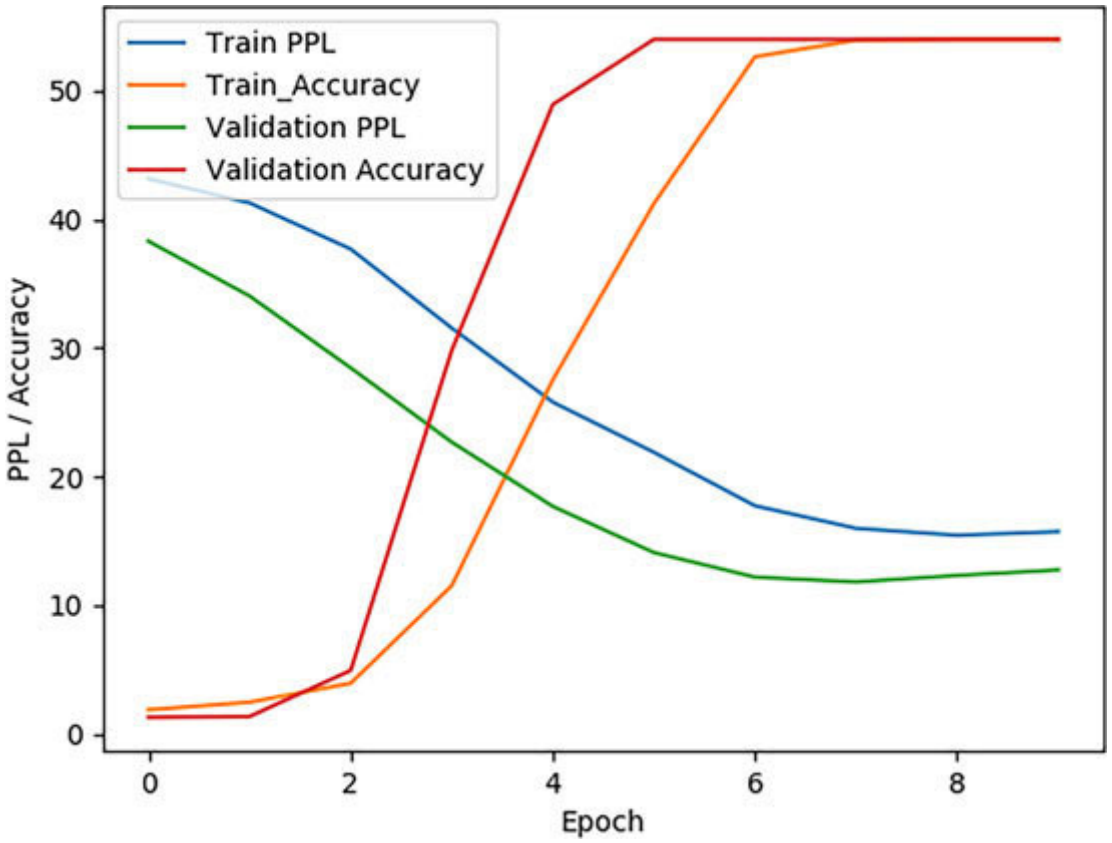


Figure 7.11: *The PPL and accuracy curve v/s epoch.*

PPL is the standard metric used in NLP for language modeling. The train and validation matrices look the same as the train data was used for validation purposes. The model was trained with these parameters:

(batch_size=64, cuda=True, d_inner_hid=2048, d_k=64, d_model=512, d_v=64, d_word_vec=512, data='preprocess/preprocessed.data', dropout=0.1, embs_share_weight=False, epoch=10, label_smoothing=True, log=None, max_token_seq_len=52, n_head=8, n_layers=6,

n_warmup_steps=4000, no_cuda=False, proj_share_weight=True, save_mode='best', save_model='trained', src_vocab_size=41, tgt_vocab_size=44)

The model built after training can be used to translate any German text to English:

python translate.py -model trained.chkpt -vocab data//home/sunil.patel/Desktop/transformer-master/preprocess/preprocessed.data -src data/newstest2013.de -no_cuda

I have provided another translation related dataset at This dataset has train, validation, and test partitions. Run the preceding implementation on this corpus and tweak the model parameters to achieve higher test accuracy. There are many implementations available for transformers, and one of them is PyTorch implementation of OpenAI's finetuned transformer. The pre-trained weight for this transformer modular is already provided, which means you can achieve better performance with little data to fine-tune. Do experiment with this model. Compare the accuracy of other models trained from scratch and the model with trained weight.

The following resource provides function by function in a detailed explanation of how the transformer was developed:

The Annotated Transformer:
http://nlp.seas.harvard.edu/2018/04/03/attention.html

Advancing Sentiment Analysis

We saw various recipes in the earlier chapters where sentiment analysis was applied to various datasets. In *Chapter 4, Using RNN for* we saw how attention can use information from each timestamp and combine it with the final hidden state to get state-of-the-art results. This recipe is about using the attention mechanism for the text classification problem. In this chapter, we will construct a network with and without attention mechanism, and we will compare both to see which one performs better.

The recurrent network with attention can be diagrammatically represented as follows:

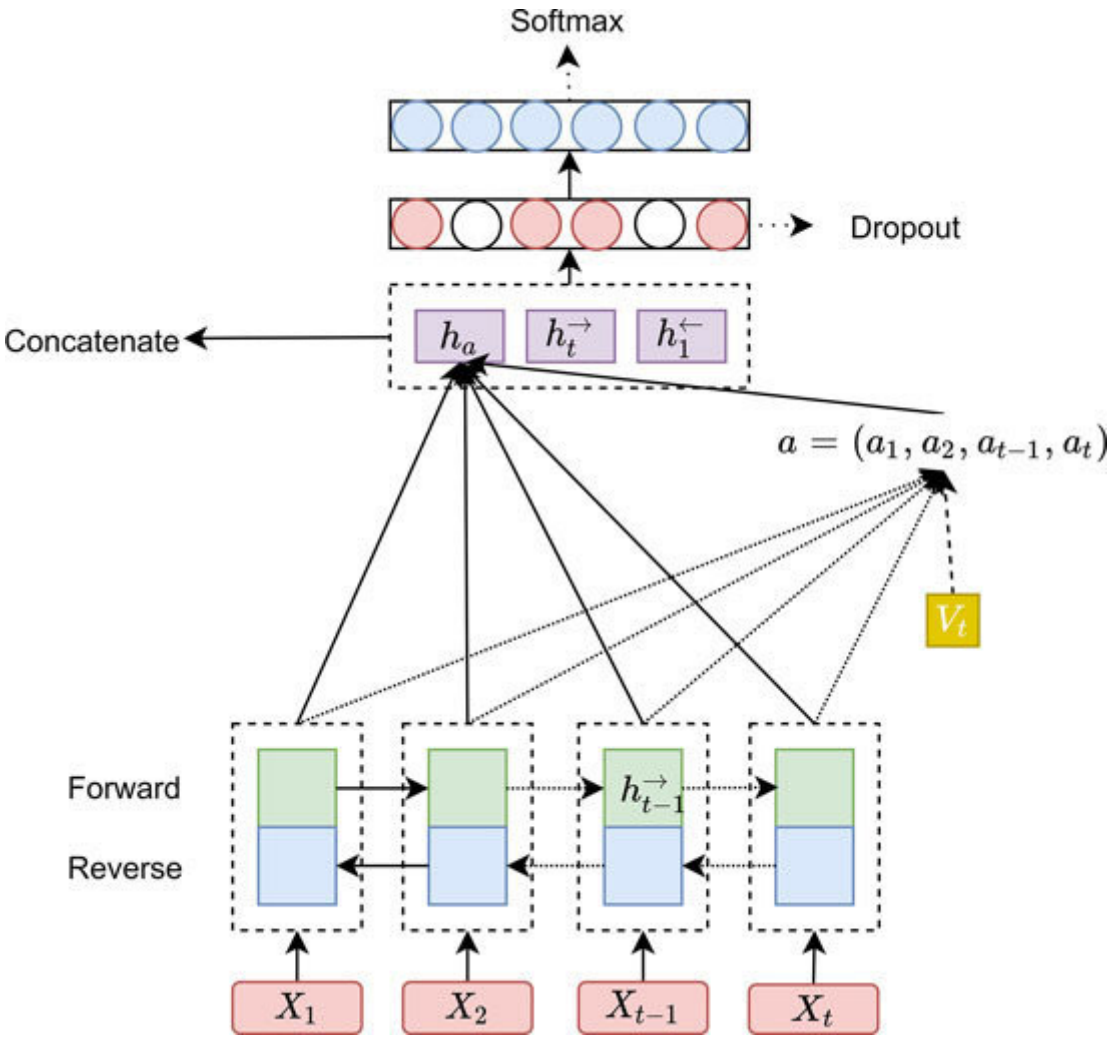


Figure 7.12: *Attention-based bidirectional RNN structure.*

As shown in the figure, the network is fed with a word at a different time step. The **Recursive Neural Network** has both forward and reverse directions, and RNN can be any unit like vanilla RNN, GRU, or LSTM. Hidden states are shown by ..., The direction of the arrow over a hidden state shows its direction. shows a hidden state moving forward along the sequence, and shows a hidden state moving backward with respect to the sequence. A hidden state for any token, is a concatenation of forward and backward state =

In the RNN without the attention mechanism, the and vectors are concatenated to form the final representation. has the crux of the entire sequence in the forward direction, while has the crux of the entire sequence in the backward direction. In this method, the importance of a particular token is not considered, an all tokens have equal weight. In RNN with attention, varying weight is considered for the input tokens in the sequence. The attention mechanism does this. In short, by name, it implies that different attention is given to different sequences. In the attention mechanism, the weighted sum of the output feature is calculated after calculating the importance of each token (weight). can be calculated using the **Batch-wise Matrix Multiplication** function of PyTorch. We have seen details of how BMM works in *Chapter 4, Using RNN for* While implementing this network, we will see how to use BMM.

Understanding Attention Mechanism

Understanding the The main part of this recipe is to design the model, which takes the help of attention mechanism for text classification. In this model, the implementation here is very similar to the attention mechanism implemented in *Chapter 4, Using RNN for* The forward function takes the input sentence in shape [input size, batch size]. Trained embedding's lookup is applied to this representation to convert it to the shape of [input size, batch size, embedding's size]. This input is passed on to the LSTM cell, along with the hidden and cell state. LSTM produces output for all the time steps, along with hidden and cell states. LSTM output, along with the final state, is given to attention_net function carries out the batch-wise matrix multiplication between LSTM output and the hidden state to produce attention Attention weights are then used to produce an updated hidden state, and a linear transformation is applied to the updated hidden state to convert it into output classes. A softmax transformation is applied after that to normalize the output:

```
class RNNAttentionModel(torch.nn.Module):
def __init__(self,config_object,vocab_size, weights):
super(RNNAttentionModel, self).__init__()
self.batch_size = config_object.batch_size
self.class_num = config_object.class_num
self.hidden_size = config_object.hidden_size
```

```
self.vocab_size = vocab_size
self.embed_size = config_object.embed_size
self.device = device
self.word_embeddings = nn.Embedding(self.vocab_size,
self.embed_size)
self.word_embeddings.weight.data.copy_(weights)
self.word_embeddings.weight.requires_grad = True
self.lstm = nn.LSTM(self.embed_size, self.hidden_size)
self.label = nn.Linear(self.hidden_size, self.class_num)
def attention_net(self, lstm_output, final_state):
hidden = final_state.squeeze(0)
attn_weights = torch.bmm(lstm_output,
hidden.unsqueeze(2)).squeeze(2)
soft_attn_weights = F.softmax(attn_weights, 1)
new_hidden_state = torch.bmm(lstm_output.transpose(1, 2),
soft_attn_weights.unsqueeze(2)).squeeze(2)
return new_hidden_state
def forward(self, input_sentences):
input = self.word_embeddings(input_sentences)
input = input.permute(1, 0, 2)
h_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size)).to(self.device)
c_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size)).to(self.device)
output, (final_hidden_state, final_cell_state) = self.lstm(input,
(h_0, c_0))


output = output.permute(1, 0, 2)
attn_output = self.attention_net(output, final_hidden_state)
logits = self.label(attn_output)
return torch.softmax(logits, dim=1)
```

In addition to this network, I have to build a network without attention mechanism while keeping everything the same, including the data pipeline input, optimizer, and loss function. I have compared the performance of the network with attention to the network without it.

Ensuring In the earlier chapters, we made many comparisons and ignored the fundamental aspect of reproducibility. In networks, the weight is initialized randomly, and it has a great effect on convergence. To compare two networks, we must initialize the weights of the two networks in a reproducible manner. PyTorch has a mechanism to facilitate reproducible results by fixing seed and setting the engine as deterministic. It can be done as follows:

```
# fixing seeds and device
torch.manual_seed(0)
np.random.seed(0)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

Examining The accuracy comparison of the network with and without attention is given below. The RNN with attention logic reached up to 90% accuracy, whereas the RNN without attention logic reaches about 60% accuracy in the same number of iterations:

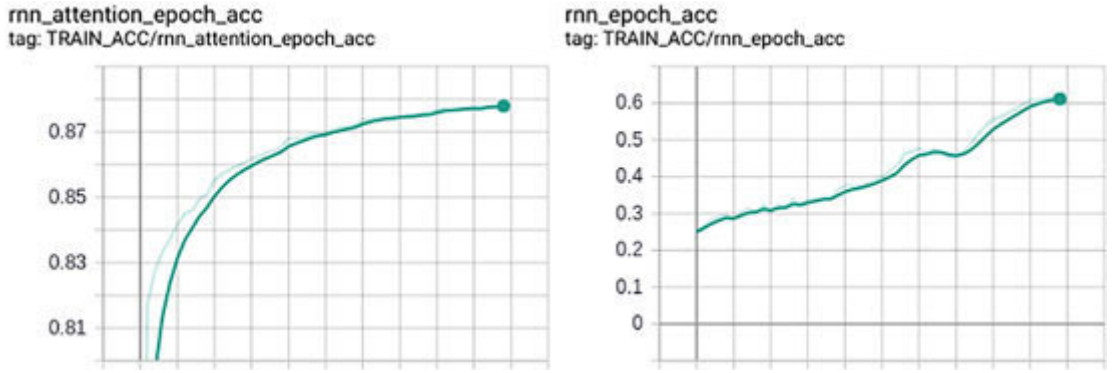


Figure 7.13: *Comparing the increase in accuracy for a model with and without attention.*

The loss of networks with and without attention logic is as follows: the network with attention mechanism achieved minimum loss below 0.05, whereas the network without attention mechanism achieved minimum loss of 0.13 in the same number of iterations:

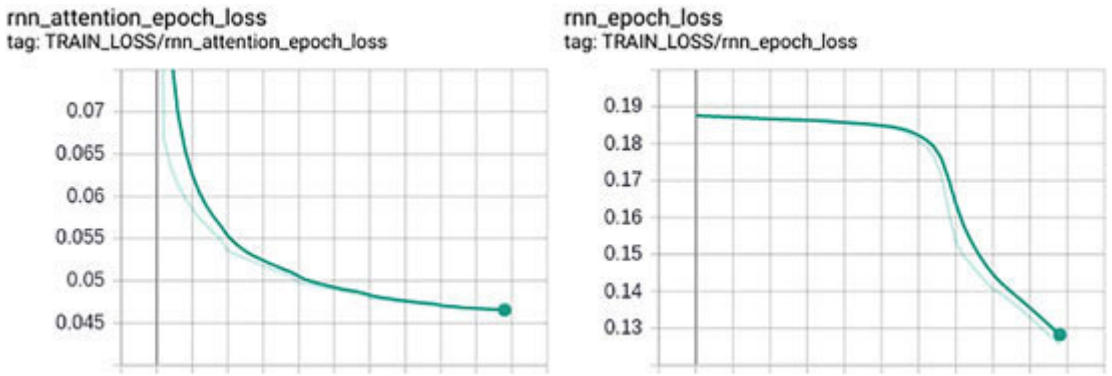


Figure 7.14: *Comparing decrease in loss for a model with and without attention.*

A similar trend has been observed in the case of test accuracy. The RNN with attention logic reached up to 87%

accuracy, whereas the RNN without attention logic reaches around 65% accuracy in the same number of iterations:

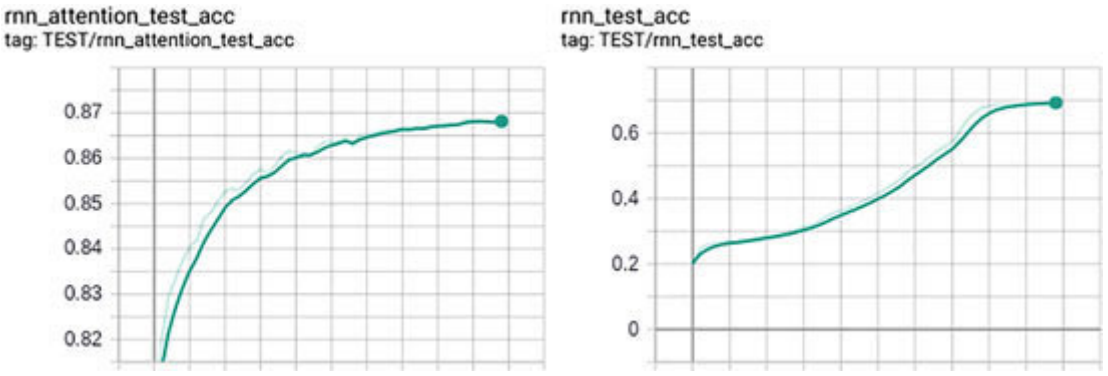


Figure 7.15: *Comparing the increase in test accuracy for a model with and without attention.*

The preceding experiments provide clear intuition that attention mechanism provides better and faster convergence as compared to a network architecture without it. A well-illustrated code block with all the required functions to reproduce these results is given at

Various variants of the attention mechanism are available, and two of them are as listed here:

Proposed by Dzmitry Bahdanau et al. in their paper *Neural Machine Translation by Jointly Learning to Align and*

Proposed by Minh-Thang Luong et al. in their paper *Effective Approaches to Attention-based Neural Machine*

Check out these papers, try to implement the logic in the preceding code, and then compare the efficiency of different attention mechanisms on this task. I used Dzmitry Bahdanau's implementation in the preceding code.

Take a look at the following link:

Text classification research with attention-based recurrent neural networks:
http://univagora.ro/jour/index.php/ijccc/article/download/3142/pdf

Building Named Entity Recognition

Named entity recognition or named entity resolution is a similar concept known as NER in short. NER tags the subpart of the sentences with a certain class. This subpart can be one word or a combination of many words occurring together. NER is the hot topic in the field of NLP and has several practical use cases. Some of them are given here:

Writing an efficient search engine by extracting key terms from the text.

Suggesting reading content based on the entity mentioned in the literature, and similarly, suggesting a product based on its description.

Keeping an eye on the market by parsing feeds from Twitter.

In this section, we will see some of the practical means to implement the NER task. Many datasets are available to experiment with NER, and some of them are listed below. In this chapter, I will use the CONLL2003 datasheet to demonstrate convergence. CONLL2003 was released by Conference on Computational Natural Language Learning. The CONLL2003 dataset has nine unique entities. It has four

entity types tagged: **Location Organization** and **Miscellaneous** All these types of the entity have a beginning (B-TYPE) and intermediate tag (I-TYPE) constituting eight unique entity classes. Additionally, a word that does not belong to any of the categories assigned to class All these combined constitute nine classes. There are tab-separated columns, and the respective columns are: word token, parts of speech of the given token, syntactic chunk tag, and entity type. The pre-processed CONLL2003 data is kept at The dataset looks as follows:

follows:
follows:
follows:
follows:
follows:
follows:
follows:

The NER task can be carried out by taking a word and the character level features:

Word level Each word token is embedded with an n-dimensional Glove vector. To predict class for the given token, we need to have a context. Context means the surrounding words; I have two words after and before the target word as context here, so there will be five words in each input. Each word can have 100-dimensional Glove Embedding. If a batch

of 32 words is taken, the resultant shape of the input will be [32, 5, 100]. The target will be one hot encoded vectors [0, 1, 0, 0, 0, 0, 0, 0, 0], and the final batch of the target will have the shape: [32, 9]:

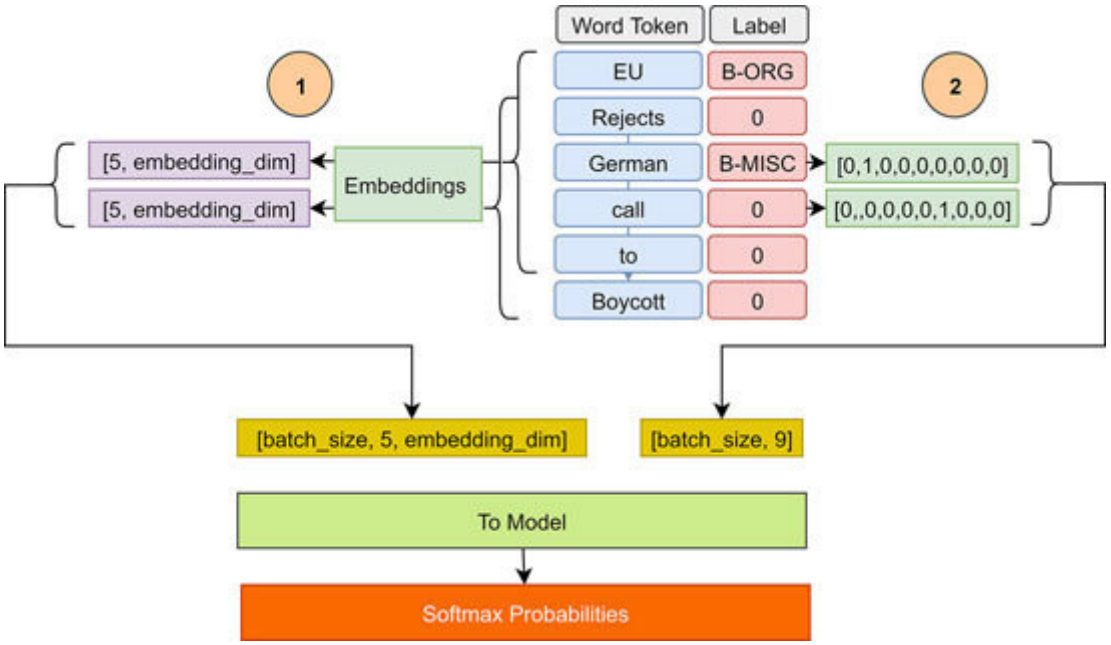


Figure 7.16: *An illustration showing how word-based feature is generated.*

The preceding figure shows the following:

The features are generated taking context window as 2.

Labels are converted into one hot embedding.

This input representation will now be processed with the LSTM network. The detailed implementation is covered in the next section of the recipe. Similarly, we could use convolution

networks here. There will be a change in the representation. Let's take that the max size of our word as 10, and our vocabulary will have a maximum of 69 characters. Each word can be represented as the matrix of [10, 69] in one hot encoded form. This is for one word if we consider the window of 2 words before, and the input size will be [5, 10, 69] after including target word. Processing such input in the batch of 32 will give the final size as [32, 5, 10, 69], which will be the input to the convolutional layers.

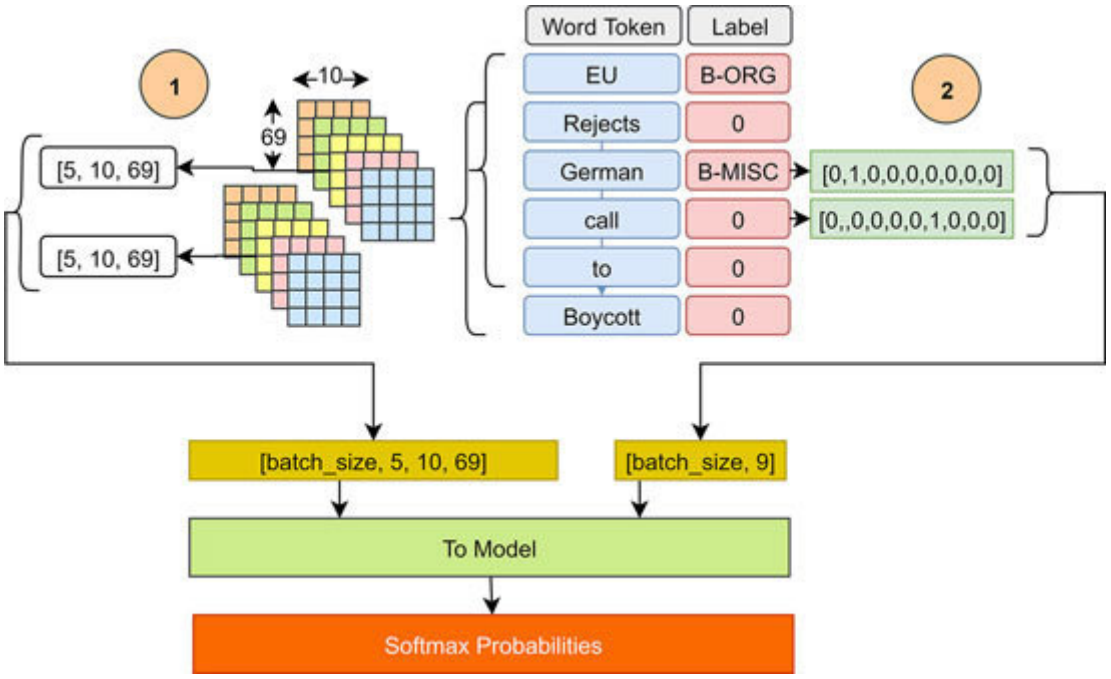


Figure 7.17: *An illustration of how character-based feature is generated.*

The preceding figure shows the following things:

The features are generated taking context window as 2

The labels are converted into one hot embedding

Word-level NER

NER using word-level features: I used the RNN with attention here to get the state-of-the-art model. This model was explained in the previous recipe—advancing sentiment analysis. This model will take the input representation, as discussed in the previous section, and give a probability for each class. Data loaders are essential to complete the task in constant memory. The main thing to observe is how the data loader is implemented. I have used a custom data loader and not the Torchtext data loader. The data loader is designed to provide input embedded with Glove 100-dimensional embedding. So, we are not using any embedding layer in the network and are directly loading Glove Vector:

```
class RNNAttentionModel(torch.nn.Module):
def __init__(self,batch_size, class_num, hidden_size,
embed_size):
super(RNNAttentionModel, self).__init__()
self.batch_size = batch_size
self.class_num = class_num
self.hidden_size = hidden_size
self.embed_size = embed_size
# self.word_embeddings.weights = nn.Parameter(weights,
requires_grad=False)
self.lstm = nn.LSTM(self.embed_size, self.hidden_size)
self.label = nn.Linear(self.hidden_size, self.class_num)
```

```
def attention_net(self, lstm_output, final_state):
hidden = final_state.squeeze(0)
attn_weights = torch.bmm(lstm_output,
hidden.unsqueeze(2)).squeeze(2)** soft_attn_weights =
F.softmax(attn_weights, 1)
new_hidden_state = torch.bmm(lstm_output.transpose(1, 2)
soft_attn_weights.unsqueeze(2)).squeeze(2)
return new_hidden_state
def forward(self, input_sentences):
input = input_sentences.permute(1, 0, 2)
if self.batch_size is None:
h_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size).type(torch.FloatTensor)).to(device)
c_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size).type(torch.FloatTensor)).to(device)
else:
h_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size).type(torch.FloatTensor)).to(device)
c_0 = Variable(torch.zeros(1, self.batch_size,
self.hidden_size).type(torch.FloatTensor)).to(device)
output, (final_hidden_state, final_cell_state) = self.lstm(input,
(h_0, c_0)) # final_hidden_state.size() = (1, batch_size,
hidden_size)
output = output.permute(1, 0, 2) # output.size() =
(batch_size, num_seq, hidden_size)
attn_output = self.attention_net(output, final_hidden_state)
logits = self.label(attn_output)

return torch.softmax(logits, dim=1)
```

The following is the performance on the train data. The accuracy reaches up to about 87%, and loss decreases considerably:

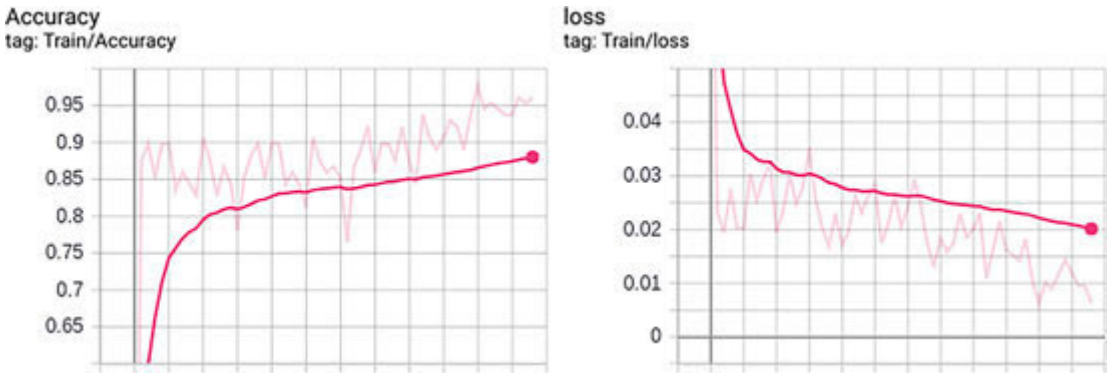


Figure 7.18: *Decrease in the loss and increase in accuracy on train data when model trains NER task taking word-level feature.*

The performance of the model on the test data is also notable; the accuracy reaches 87% in this case as well. This means the implementation generalizes well on unseen data:

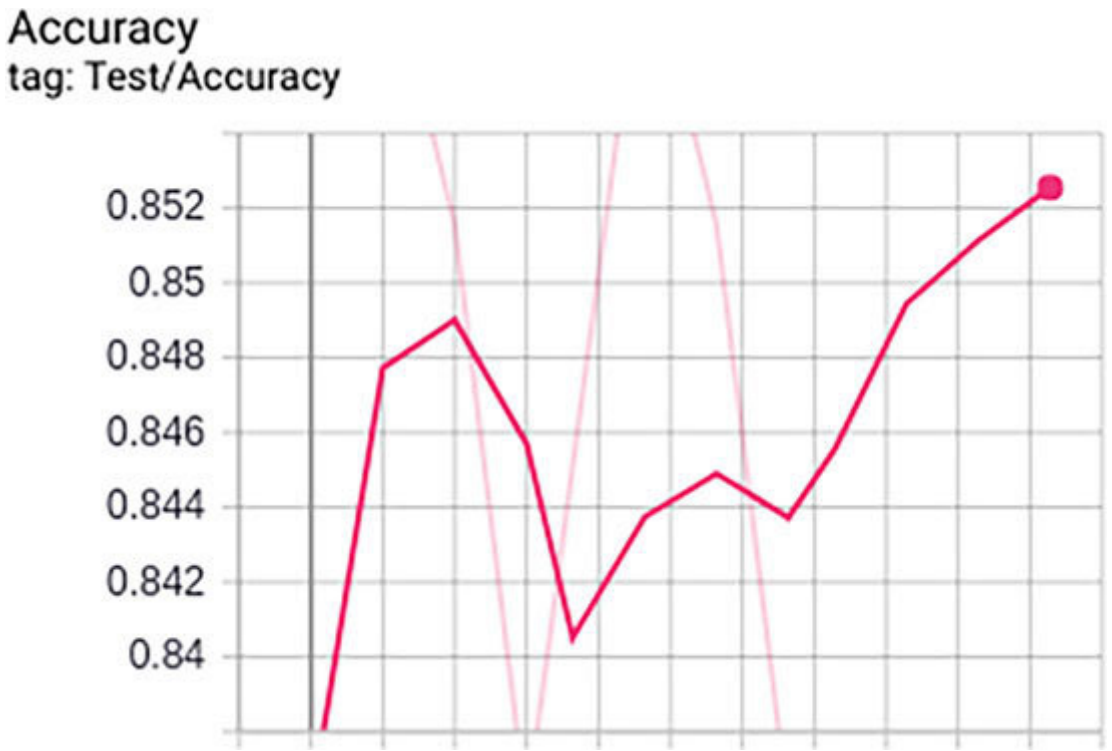


Figure 7.19: *Increase in accuracy on test data when model trains NER task taking word-level feature.*

The entire implementation with pre-processing and data loaders is provided at

Character-level NER

The following model has CNN components to process our character-based representation. The model accepts [128, 5, 10, 68], dimensional input wherein 128 is the batch size, 5 is target plus context words, 10 is the max character in the word, and 68 is several unique characters considered. In another term, if we compare the input to the image, 128 is the batch size, 5 is similar to channels in the image of dim 10*68. The input shape passes through a series of convolution operations, and the number of channels increases from 5 to 40. The resultant tensor is passed to a linear layer to convert it to output probabilities after the application of the softmax layer:

```
class CNNmodel(torch.nn.Module):
def __init__(self,batch_size, class_num):
super(CNNmodel, self).__init__()
self.batch_size = batch_size
self.class_num = class_num
self.conv1 = nn.Conv2d(in_channels=5, out_channels=10,
kernel_size=3, stride=1)
self.conv2 = nn.Conv2d(in_channels=10, out_channels=20,
kernel_size=3, stride=1)
self.conv3 = nn.Conv2d(in_channels=20, out_channels=40,
kernel_size=3, stride=1)
```

```
self.linear1 = nn.Linear(in_features=40*4*62,
out_features=self.class_num)
def forward(self, input):
conv1_out = self.conv1(input)
conv2_out = self.conv2(conv1_out)
conv3_out = self.conv3(conv2_out)

linear1_out = self.linear1(conv3_out.view(self.batch_size,-1))
return torch.softmax(linear1_out, dim=1)
```

The following is the performance on the train data. The accuracy reaches about 85%, and the loss decreases considerably:

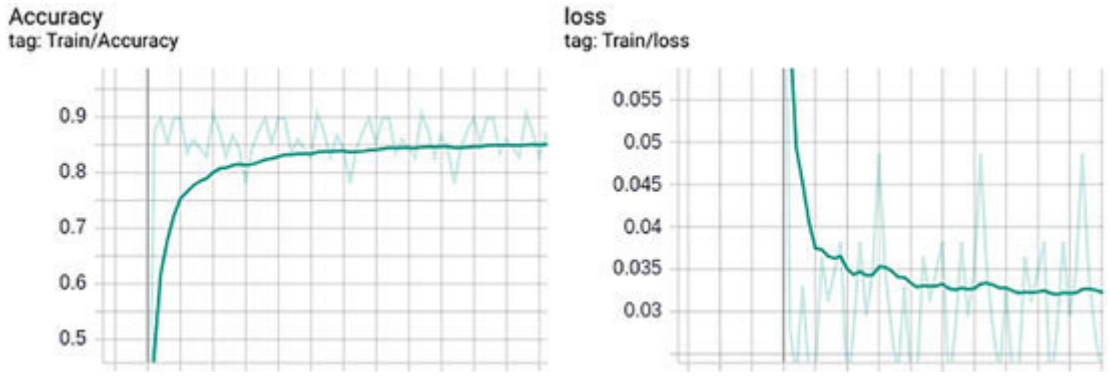


Figure 7.20: *Decrease in loss and increase in accuracy on train data when model trains NER task taking character-level feature.*

The model's performance on the test data is also notable; the accuracy reaches 85%. This means our implementation generalizes well on unseen data as well:

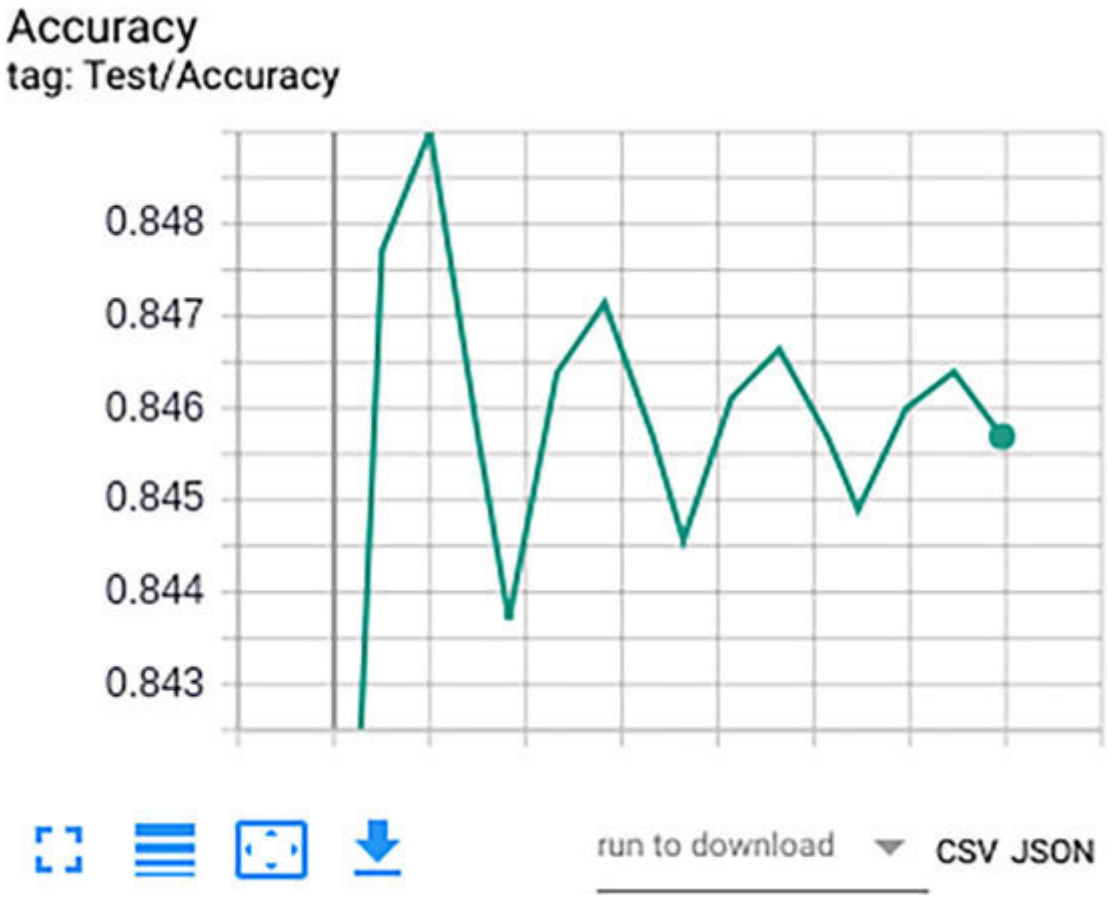


Figure 7.21: *Increase in accuracy on test data when model trains NER task taking character-level feature.*

The entire implementation with pre-processing and data loaders is provided at

Compare the model with and without attention mechanism.

Design an ensemble taking both word and character-level features and see whether it works better.

Many other datasets are available for the NER task, and some of them are given as follows. Use this dataset and keep

learnable embedding, as it would be harder to find trained embeddings for German and Portuguese. See if your classifier converges on these datasets:

W-NUT2017

GermEval2014

Europeana Newspapers

HAREM

A Survey on Recent Advances in Named Entity Recognition from Deep Learning models: *https://www.aclweb.org/anthology/C18-1182*

Conclusion

We implemented more on NLP applications in this chapter, and we covered 70% of the NLP applications that developers work with. We started with topic modeling and visualization. Topic modeling is an important area of research, and it has a lot yet to be discovered. The next big thing is summarization. We saw how to practically perform extractive summarization. You can go ahead and write a sequence to sequence the network for abstractive summarization. We saw line by line how a transformer works, what input it takes, and how to implement it easily. We also saw applications like sentiment analysis and named entity recognition, which is a form of classification task at the core. The most important thing was to know how transformers are implemented line by line. This chapter will surely help you going forward. After completing this chapter, you can start building your kernels in the Kaggle competition by participating in the preliminary competition.

The next chapter will take you through the application of complex architectures in NLP.

CHAPTER 8

Application of Complex Architectures in NLP

This chapter is perhaps the most practical and conceptually advanced one in the entire book. It covers practical implementations that are among the most cited research in 2017 and 2018. Topics covered in the chapter with their importance are as summarized as follows; *SentencePiece* covers the essential techniques to decrease the effective vocabulary size and augment the text data for text-related applications. We will experience the power of automatic Ml with the **Random Multi-Model** After this, we will discuss how to get multiple models by training the network just once, using a snapshot ensemble technique. This chapter covers the *Siamese* network, which is used in *Apple's Face ID* and can be used for text comparison as well. We will also cover two amazing topics: CTC loss and image captioning. CTC loss is yet another kind of loss function used in speech-related applications. Image captioning explores how to generate a caption for the image; in this recipe, we will see how to use CNN and RNN layers in a single network.

Structure

The following topics will be covered in this chapter:

Understanding SentencePiece

Understanding random multi-model

Ensembling by taking a snapshot

Getting to know Siamese networks

Application of RCNN

Understanding CTC loss

Captioning image

Objective

Understanding sentence piece

Understanding random multi-model

How to the ensemble by taking a snapshot

Getting to know Siamese network

Understanding CTC loss

Technical Requirements

The code for this chapter can be found in the Ch8 folder at GitHub repository To understand this chapter, you must have basic knowledge about the following Python packages:

Torchtext

Nltk

Matplotlib

Torchvisio

Torch

Tqdm

Pandas

Numpy

Pillow

RMDL

Sentencepiece

Scikit_learn

TensorboardX

You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r

Understanding SentencePiece

SentencePiece is the neural network-based tokenization and de-tokenization technique often used in neural network-based language generation tasks where vocabulary size limit is an important factor. SentencePiece allows us to make an end-to-end system that does not depend on language-dependent pre-processing and post-processing. It uses the **bit pair encoding** and the **unigram language** SentencePiece is similar to **subword-nmt** and but it is a little advanced. The bit pair encoding algorithm used in the Sentnecepiece is slightly different from the original bit pair encoding algorithm. In this recipe, we will understand how bit pair encoding and unigram language model help design SentencePiece.

There are some great features of the SentencePiece algorithm, and some of them are given as follows:

A number of unique tokens are In ordinary tokenization, the final vocabulary is infinite, and the final vocabulary is made limited by putting threshold. This leads to loss of information; especially rare word results in a token. Otherwise, standard tokenization techniques have infinite tokens. With sentence piece, the final token size is always pre-decided, and the algorithm performs accordingly while training.

Training from Subword techniques like **subword-nmt** and **WordPiece** require pre-tokenized sentences. Sentence piece works without pre-tokenized sentences and allows an algorithm to be used on languages like Chinese and Japanese, where no explicit rule for tokenization exists.

Whitespace is treated In the standard tokenization techniques, the word is tokenized into following three tokens [“Hello”, “world”, This method is not reversible, as detokenization does not yield the original token. Tokenized != detokenize “world”, SentencePiece treats the strings as the sequence of Unicode, and the white space also considered to be one of the characters in the Unicode. In SentencePiece, the white space is replaced with the”_ “character, which is U+2581 in the Unicode. According to sentence piece, the word will be treated as "Hello world."→"Hello_world."→["Hello][_wor][ld][.]. Since the white space is preserved, we can detokenize the text without any ambiguity, as shown here:

```
detokenized = ''.join(["Hello","_wor","ld",["."]]).replace('_',' ')
```

Similarly, the Chinese language can be tokenized as follows; remember the Chinese do not have explicit spaces between words.

Subword regularization: This is the technique to virtually augment the train data by subword sampling. It helps enhance the robustness of the natural machine translation model. The salient features of the sentencing piece are listed here:

Purely No intermediate pre-processing techniques are required.

Language There is no need for language-specific pre-processing; it can be applied to any language.

Tokenization and detokenization are obtained in a deterministic way as long as the same model is used.

Vocabulary ID Sentence Piece manages vocabulary and can directly generate the vocabulary ID sequence from a raw sentence.

The sentence piece takes about 6 MB of memory and can process 50k sentences.

To demonstrate the use of SentencePiece, we will use the official sentence piece Python API. The official SentencePiece can be installed as shown here.

On Ubuntu, the build tools can be installed with

sudo apt-get install cmake build-essential pkg-config libgoogle-perftools-dev

Installation from source requires the building tool

sudo apt-get install cmake

After building and installing sentence piece:

```
% cd /path/to/sentencepiece
% mkdir build
% cd build
% cmake ..
% make -j $(nproc)
% sudo make install
% sudoldconfig -v
```

Alternatively, you can install it as a Python package:

```
pip install sentencepiece
```

Bit Pair It is a simple data compression technique works by replacing the most occurred pair of byte by a single unused byte. This algorithm is adapted for word segmentation. Here, for the task on word segmentation, character pairs are replaced instead of the byte pair. To give you an idea of how bit pair encoding works, I have given a simple example as follows: Let's say we have a word Here, the character pair *mm* occurs most often, so taking an assumption Z = the new sequence will be Now the character *mb* is repeated more often, so it is replaced by R = The new sequence will be We can stop here based on the hyperparameters provided. Now, the original word, when segmented, will be ["mm", "mb", "d", "mm", "mb", The model is trained with this tokenization and $Z = R =$ *mb* serves as the key-value pairs in the model.

When this model is called, all the given text will be replaced by key-value pairs. Unicode has 17 planes, and each plane has 65, 536 Unicode characters. This gives us a total of 1, 114, 112 characters. It is highly unlikely that one corpus will have all these characters, so we can use all the remaining characters to replace the existing character paints and represent them as key-value pairs. Sentence Piece can be easily integrated with your existing pipeline using its Python API. Sentence piece can be simply imported as:

```
import sentencepiece as spm
```

SentencePiece can be trained by providing any text file. Upon training, it yields the model and vocab file. In the following sentence, I am using a book *Stories of Great Inventors* by *Hattie E. Macomber* as the input. After training, m.model and m.vocab will be written to the disk:

```
spm.SentencePieceTrainer.train('--input=---.txt --model_prefix=m --vocab_size=2000')
```

You can load the m.model as follows:

```
sp = spm.SentencePieceProcessor()
sp.load('m.model')
```

Encoding sentence to pieces and ids:

```
print("As Pieces : ",sp.encode_as_pieces('My name is Sunil, and I like to Learn.'))
```

print("As Ids : ",sp.encode_as_ids('My name is Sunil, and I like to Learn.'))

Decoding original sentence from pieces and ids:

print("Joining Pieces : ",sp.decode_pieces(['_My', '_name', '_is', '_', 'J', 'an', 'ki', ',', '_and', '_I', '_like', '_to', '_teach', '.']))
print("Joining by Ids : ",sp.decode_ids([494, 396, 31, 201, 306, 229, 3, 10, 6, 75, 8, 612, 25, 89, 30, 4]))

Getting vocab size (This will help initialize embeddings before passing data to CNN and RNN).

print("Vocab Size : ", sp.get_piece_size())

Getting piece by id and id by piece can help in the decoding phase of translation or summarization:

print("Getting Piece by id : ", sp.id_to_piece(209))
print("Getting Id from Piece : ",

Treating an unknown token. By default, SentencePiece assigns id = 0 for unknown token:

print("Getting id for unknown word : ",sp.piece_to_id('__UNKNOWN__'))

Sampling n-best segmentation for the given input. When the model is trained with type --model_type=unigram, you can perform the sampling for augmentation and make more robust models. A sampling of any word can be performed as follows:

```
for n in range(10):
print(sp.sample_encode_as_pieces('Good Morning', -1, 0.1))
>>> ['_Good', '_M', 'or', 'n', 'ing']
>>> ['_Good', '_', 'M', 'or', 'n', 'i', 'ng']
>>> ['_', 'G', 'o', 'o', 'd', '_M', 'or', 'n', 'ing']
>>> ['_Good', '_', 'M', 'or', 'n', 'ing']

>>> ['_', 'G', 'o', 'o', 'd', '_M', 'or', 'n', 'ing']
>>> ['_Good', '_M', 'o', 'r', 'n', 'ing']
>>> ['_Good', '_M', 'or', 'n', 'i', 'ng']
>>> ['_Good', '_M', 'o', 'r', 'n', 'i', 'ng']
>>> ['_G', 'o', 'o', 'd', '_', 'M', 'o', 'r', 'n', 'i', 'n', 'g']
>>> ['_Good', '_M', 'o', 'r', 'n', 'in', 'g']
```

Similarly, id for each piece will be generated by the following syntax:

```
for n in range(10):
print(sp.sample_encode_as_ids('Good Morning', -1, 0.1))
```

You can experiment with SentencePiece by a running script at

Integrating the SentencePiece with the existing pipeline is very easy. In PyTorch, you can define an additional function in the

class that defines the model. The dummy implementation will look like this:

```
class dummy_with_sentecencepiece(torch.nn.Module):
def __ini__(self, sp, embed_size):
super(dummy_with_sentecencepiece, self).__init__()
self.sp = sp
self.embed = torch.nn.Embeddings(input_size =
self.sp.get_piece_size(), output_size = embed_size)
self.other_layer_1 = torch.nn.LSTM(...)
self.other_layer_2 = torch.nn.Conv2D(...)
def forward(self, input_sent):
pieces_batch = self.apply_sentencepiece(input_sent)
padded_batch = self.padding(self, batch)
embedded_sent = self.embed (padded_batch)
# applying other layers on to embedded output

def apply_sentencepiece(self, input_sentences)
"""Getting pieces ids"""
return self.sp.encode_as_ids(input_sentence)
def padding(self, batch):
"pad_to_make_all_pieces_equal_and_return"
return batch_with_padding
```

The dummy_with_sentecencepiece class will have four methods. The init method will take various parameters necessary to initialize layers, which can be embedded, convolution, and RNN layers. The embedding layer will take SentencePiece vocabulary size as the input size and the embed size as the output size. When an input is given to the apply_sentencepiece method, it returns pieces for the

given sentences. Such pieces are then padded to make the size of all equal to a sentence with the highest pieces. Now, the padded batch with ids is transferred to the embedding layer, and the output of the embedding layer is passed on to another layer for specific tasks.

You can check out the following links:

Neural machine translation of rare words with subword units: *https://www.aclweb.org/anthology/P16-1162*

Subword regularization: Improving neural network translation models with multiple subword candidates: *https://arxiv.org/abs/1804.10959*

SentencePiece: A simple and language-independent subword tokenizer and detokenizer for neural text processing: *https://arxiv.org/abs/1808.06226v1*

Understanding Random Multi-Model

Random multi-model deep learning or RMDL are the techniques inspired by Auto ML, Auto-ml, or Automatic Machine Learning. It is a group of techniques focused on how to apply machine learning end to end on any practical task. There are many tools available, and some of them are given here:

A Bayesian hyperparameter optimization applied to WEKA.

A Bayesian hyperparameter optimization applied to scikit-learn.

A Python library capable of automatically creating and optimizing pipelines with the help of genetic programming.

A grammar-based framework utilizing genetic programming for optimizing scikit-learn.

Auto An open-source Python package for neural network architecture tuning.

Google has also come up with the Auto-ml offering in its cloud. Here are a few advantages of using the auto-ml tool in development:

It provides a relative idea about parameters that can provide better results.

It eliminates the chances of a model being falsely declared non-convergent after a few manual experiments.

When a few parameters are known, the user can fine-tune manually.

It allows the developer to focus on more productive tasks.

It seems that auto-ml can solve everything related to tuning the model after developing it. It's true, but there are also certain shortcomings of the auto-ml techniques. Auto-ml generally relies on checking the effect of model architecture and parameters by training each model, and it requires immense computing resources. Auto-ml is often not applied to bigger models with millions of parameters. Considering the current speed of computing, auto-ml is only viable for smaller models with a few hundred thousand parameters. RMDL is also a kind of model inspired by Auto-ml approaches.

RMDL solves the problem of finding the optimum deep neural network architecture by simultaneously improving the accuracy and robustness of deep learning architecture. RMDL is an ensemble consisting of three kinds of network architecture:

Feed Forward Network

Convolution Neural Network

Recurrent neural network

The overall model looks like this:

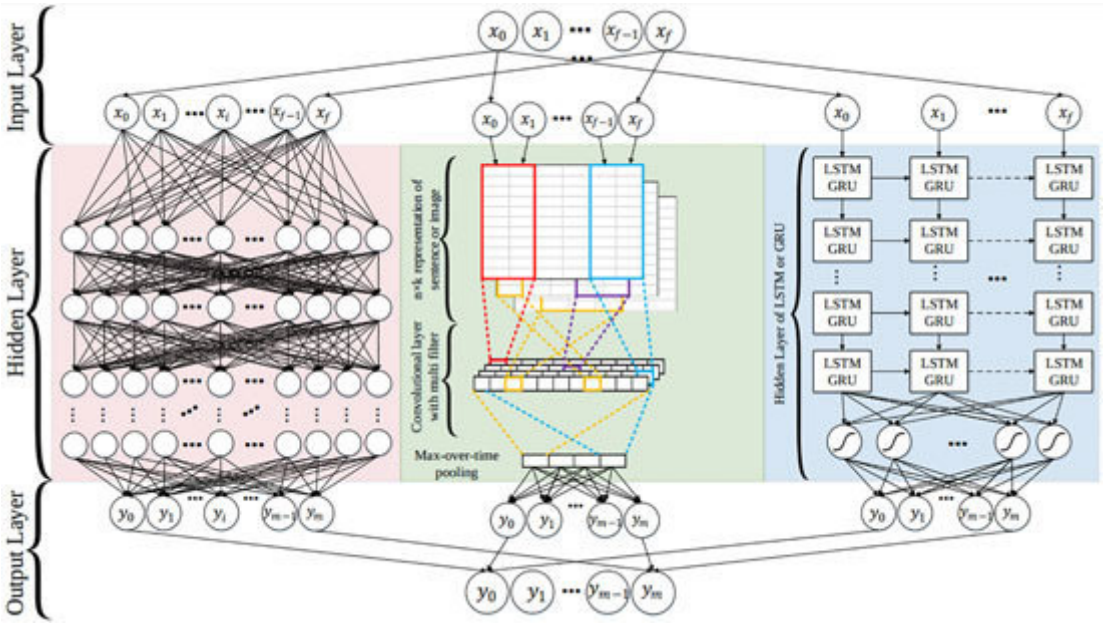


Figure 8.1: *RMDL schematic architecture.*

Source: *https://github.com/kk7nc/RMDL*

Parameters, like several layers in FFN, CNN, and RNN, are changed randomly, a defined number of random configurations are tested on the data, and the best model is given back. With three essential components, RMDL can work with all types of data like text, images, video, and fully structured data. In total, nine deep learning models are generated: three each from DNN, FFN, and CNN. All of them

are unique owing to random creation. The following equation can explain the in-detail function of the ensemble:

$$M(yi1, yi2, \ldots, yin) = \left[\frac{1}{2} + \frac{\Sigma_{j=1^n} y_{ij} - \frac{1}{2}}{n} \right]$$

Where *M* is the number of random models, and is the prediction of model j on the data point These outputs from all the models are collected . The Soft-max is applied to the output of all the models, and then the prediction is considered by majority vote.

$$\hat{y}^{*}_{i,j} = Argmax[Softmax(\hat{y}_{i,j})]$$

This model is trained with SGD, Adam, RMSProp, Adagrad, and Adamax as optimizers. The interesting thing here is that all models trained again with random optimizer to ensure that model should be tested in all possible conditions, which might help in convergence.

Creating Flexible Networks

We must first understand how variation in the model architecture is made by changing the parameters of the layers. PyTorch has excellent support to build the model, which gets deeper and shallower only by changing certain parameters. While building such an ensemble model, it is advisable to stack all the layers in to list and then use nn. Sequential method is used to connect all layers to form the model.nn. Sequential can be used flexibly, as shown:

```
layers.append(layer_1)
layers.append(layer_2)
self.layers = nn.Sequential(*layers)
```

A flexible model with dense Let's say you want to build a dense network with variable layers and perceptron and dropout in between. You can implement such a model as follows:

```
perceptron_in_layers = [200, 100, 50, 25]
dropout = 0.2
activation = torch.nn.ReLU()
layers = []
num_layers = len(perceptron_in_layers)
for i in range(0,num_layers-1):
```

```
layers.append(torch.nn.Linear(in_features =
perceptron_in_layers[i], out_features =
perceptron_in_layers[i+1]))
layers.append(activation)
layers.append(torch.nn.Dropout(dropout))
layers = nn.Sequential(*layers)
```

One more thing to learn here is that it's always better to declare what the model is being used for. We were not following this convention till time, but it is required. Some layers like dropout and batch-norm function differently when a model is used for train and when a model used for test. If you use you will see the following output, which shows parameters for the model:

```
layers.train()
Sequential(
(0): Linear(in_features=200, out_features=100, bias=True)
(1): ReLU()
(2): Dropout(p=0.2)
(3): Linear(in_features=100, out_features=50, bias=True)
(4): ReLU()
(5): Dropout(p=0.2)
(6): Linear(in_features=50, out_features=25, bias=True)
(7): ReLU()
(8): Dropout(p=0.2)
)
```

If you use the following output will be shown. It lets the model know that the weight need not be updated, and only

forward pass must be done without accumulating parameters for the backward pass. If the model is declared for evaluation, it affects layers like batch normalization and dropout. These layers behave differently during training and evaluation.

A flexible model with RNN This was about the Feedforward model. Now, let's see which parameters are available in the recurrent network if one wants to develop a flexible architecture. GRU or LSTM or Vanilla RNN has common parameters that can be declared to change the network architecture. For example, with LSTM, one can change various parameters, as follows:

The number of features of the input will generally be equal to the size of the embedding.

Hidden state size for any RNN unit.

RNN can be stacked in the layers, and it looks as follows. More layers are required for more complex data.

If bidirectional is true, RNN runs in both directions of the sequence.

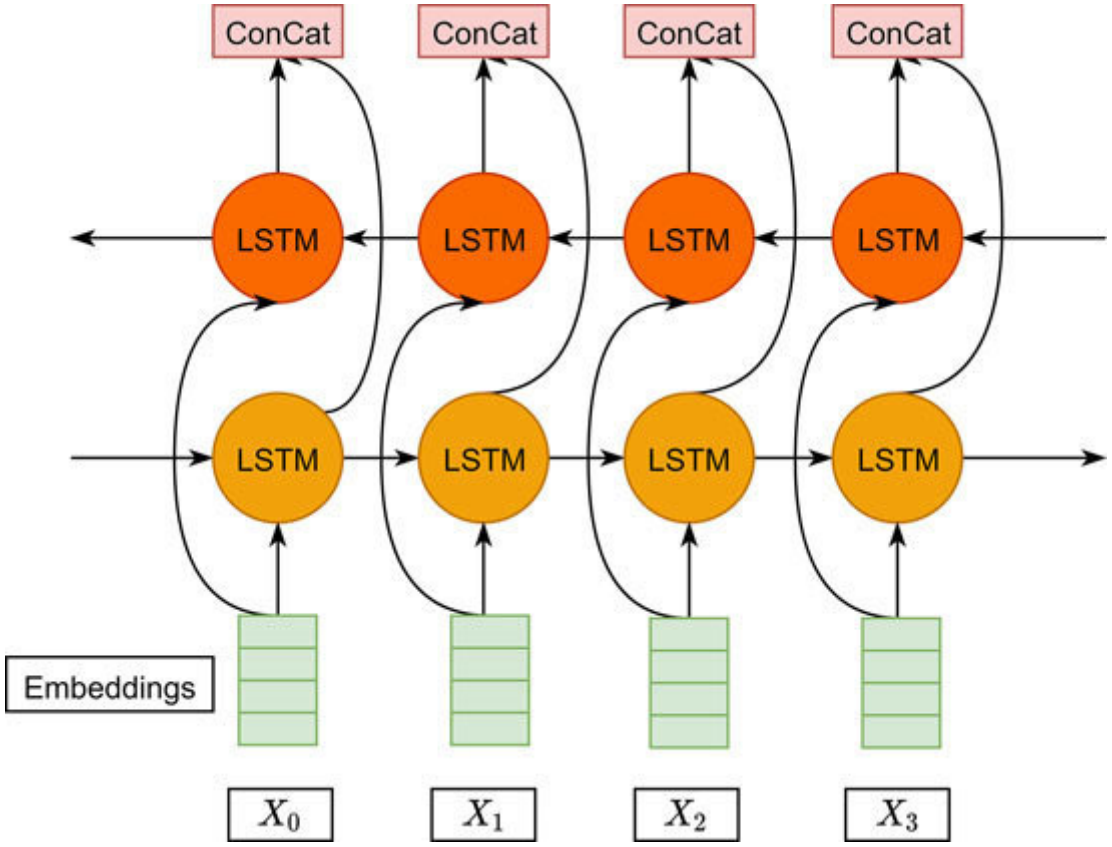


Figure 8.2: *An illustration showing how bidirectional LSTM works and how the final output is provided.*

The final output at each time step will be the concatenation of both the forward and backward output from forward and backward run. (Implementation-wise, RNN runs in only one direction, but the sequence is reversed and given to RNN, and the output so produced is called the reverse direction output.)

Various network architectures can be generated randomly using these options. In addition, various other additives like attention mechanism.nn can be applied to RNN. Sequential (*layers) can be used to stack LSTM layers too.

Using RMDL

Random multi-model deep learning for classification is the framework for testing various network topology and with varying hyper-parameters on the given data to get the best model with the highest accuracy. The implementation of RMDL is provided with the git repository: RMDL can be simply installed with pip install

```
from keras.datasets import mnist
import numpy as np
from RMDL import RMDL_Image as RMDL
```

Applying RMDL on Reuter Data

Reuters text classification is the benchmark data set for multi-label and multi-class classification. The dataset has 90 classes, 7769 training documents, and 3019 testing documents. More about this dataset is available from: *https://archive.ics.uci.edu/ml/datasets/reuters-21578+text+categorization+collection*

Loading and pre-processing Reuters corpus is present in the NLTK package; if not present, it can be downloaded with Once downloaded, train and test docs can be loaded as follows:

```
documents = reuters.fileids()
train_docs_id = list(filter(lambda doc: doc.startswith("train"), documents))
test_docs_id = list(filter(lambda doc: doc.startswith("test"), documents))
```

After separating x and y for the train, labels with y are binarized using

```
X_train = [(reuters.raw(doc_id)) for doc_id in train_docs_id]
X_test = [(reuters.raw(doc_id)) for doc_id in test_docs_id]
mlb = MultiLabelBinarizer()
```

```
y_train = mlb.fit_transform([reuters.categories(doc_id)
for doc_id in train_docs_id])
y_test = mlb.transform([reuters.categories(doc_id)
for doc_id in test_docs_id])
```

Training the Data prepared in the previous step is given to the RMDL module, as shown in the following code block. A model constraint is provided, which specifies how many layers a network can have for Feedforward RNN and CNN subnetwork. The following model network specifies constraints with variable. The Text_Classification function takes Y for train and test, batch size, model constraints, and embeddings file as input:

```
batch_size = 100
sparse_categorical = 0
Random_Deep = [3, 3, 3] ## DNN--RNN-CNN
RMDL.Text_Classification(X_train, y_train, X_test, y_test,
batch_size=batch_size, sparse_categorical=True,
random_deep=Random_Deep, epochs=n_epochs,
GloVe_dir="../embedidngs/glove.6B/")
```

After this, RMDL starts network searching, and teh accuracy for each network will be shown in the verbose output. At the end, the best model will be returned. The code related to techniques that can be used in PyTorch to create an RMDL model is given at The usage of the RMDL package and the Reuter data is given at

RMDL can be applied to a variety of datasets in the text and image domain. For example, have used only a few arguments, including and In practice, RMDL can take more than 20 such arguments and provide fine-grain control over the experiments to be performed. For more information related to the parameters RMDL can take, you can refer to https://github.com/kk7nc/RMDL and some of the examples provided by the author to run RMDL on a variety of datasets at

For more details, refer to the following:

Random Multimodel Deep Learning (RMDL) for classification: https://arxiv.org/abs/1805.01890

Ensembling by Taking a Snapshot

Ensemble techniques are famously used to combine many weak classifiers and form a stronger one. Ensemble methods are traditionally used to produce state-of-the-art results in the famous competition like ImageNet. There are three types of ensembling techniques, namely:

Bagging

Boosting

Stacking

Let's understand them all one by one:

Bagging is also known as **Bootstrap**

It is a parallel ensemble wherein each model is built independently using a subset of the data.

The output of all classifiers is weighted equally.

Bagging decreases variance, not bias.

Suitable for a complex model with high variance low bias.

Random forest is one of the classifiers where multiple weak trees are formed, and the decision is made based on a combination of the vote.

A different model is constructed using a different subset of the data.

The next model takes the subset wherein the previous model performed poorly and tries to perform better on the given subset.

Using a weighted voting final classifier combines multiple classifiers.

This decreases variance, not bias.

Boosting is good for a model with low variance and high bias.

Gradient boosting is an example of a tree-based boosting algorithm.

Stacking:

It is normally used in competitions, commonly those hosted by *Kaggle* and *Coda* labs.

Using multiple models of a different kind and averaging the output produced by all algorithms on the single dataset to get higher accuracy of prediction.

In this chapter, we will discuss one of the techniques that fall into the category of bagging. This technique is unique and can be implemented with very minimal changes in the existing pipeline, so I have selected it to be included in this book. This technique was proposed by *Guo Huang* and coworker in their research paper *Ensembles: Train 1, get M for* It is an innovative technology that takes advantage of non-convex surfaces formed by features.

With an increase in the features, the number of local minima or maxima increases exponentially. There is no sure-shot way to find global minima or maxima. Often, the optimizers are found to get stuck in the local minima and produce a model with high variance. To understand this, let's visualize feature space once. Take a look at the following screenshot:

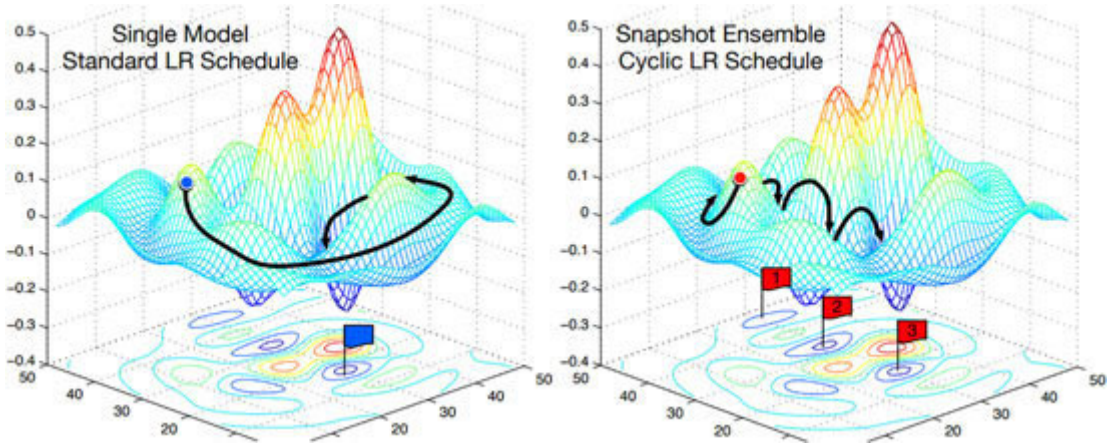


Figure 8.3: *(Left) How the standard learning rate converges the model by providing one model. (Right) How snapshot ensemble provides a different mode for each minima using cyclic learning rate.*

The diagram on the left shows the typical energy landscape with only three features. In machine learning, we try to decrease the loss of the model, and the point where the loss is minimum is known as global minima, and the model should ideally converge into the global minima. As there is no sure shot method to find the global minima, the task is difficult. At the same time, the local minima are the place minimal as compared to close vicinity. In addition, the number of local minima exponentially increases with an increase in the features. The stochastic gradient descent optimization technique often gets stuck in the local minima and produces a poor result. The snapshot ensembles technique solves this problem mindfully. Snapshot ensembles exploit the behavior of convergence of the learning rate to get better models. When the learning rate is high, the gradient overshoots and escapes from the minima, and when the

learning rate is low, it converges into the local minima. With this paper, the author proposes to have *M* parallel models in one training shot. The training epoch *T* is divided into *M* cycles. Each cycle starts with a higher learning rate and monotonously decreases to ensure convergence in the local minima. *M* models from the *M* cycles are collected and used for making the final decision, which is the average of all the models. The monotonously decreasing function is given as: . Here, is the new learning rate, is the previous learning rate, *T* is the total number of epochs, and *M* is the total number of learning rate oscillation cycles.

In *Chapter 5, Applying CNN In NLP* we covered a recipe *Word Level CNN For Text* To keep the implementation simple and easy to understand, we are incorporating snapshot ensemble implementation of word-level CNN for text classification. We will use the same large movie review dataset. With the same loss function and optimizer, we will incorporate three additional mechanisms:

A function to decrease loss monotonously.

Measures to record the snapshot at the end of each cycle.

Measures to use given a snapshot for the prediction.

The Learning Rate Modifier

The optimizer uses a cyclic annealing schedule to quickly lower the learning rate and converge the model in the nearest local minima. While in training, the learning rate is decreased, as shown here:

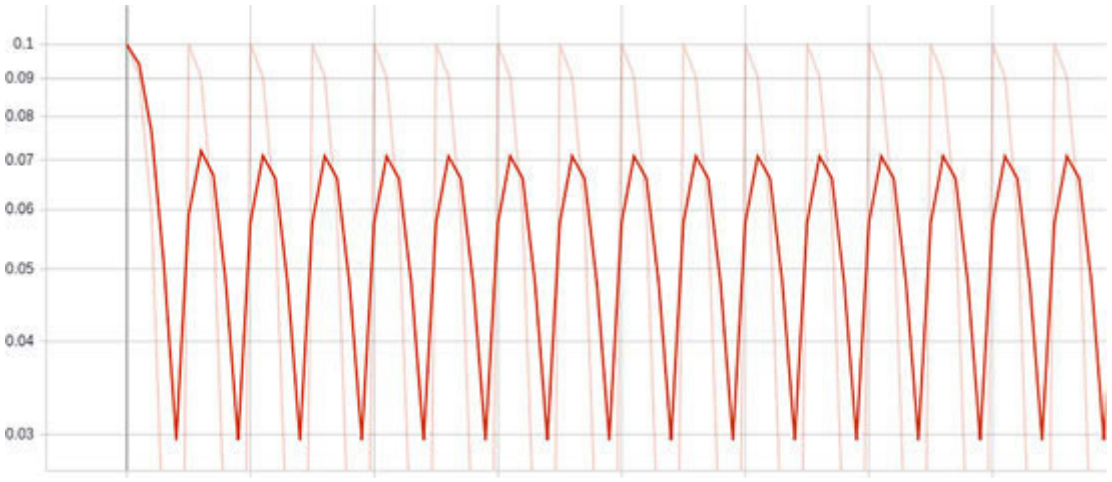


Figure 8.4: *Cyclic changes in the learning rate.*

In each cycle, the learning rate starts with some high value, and then it monotonically decreases to converge the learning in local minima, which is provided as one of the snapshot models. The X-axis shows the cycles, and the Y-axis shows the learning rate. Each cycle starts with a higher learning rate and quickly decreases to converge in the local minima. This functionality is implemented with Python definition

```
def proposed_lr(initial_lr, iteration, epoch_per_cycle):
return initial_lr * (math.cos(math.pi * iteration /
epoch_per_cycle) + 1) / 2
```

Recording Snapshots

There is no change in the model, but there is a change in the training schedule. The total epoch and the number of the cycle are defined. The epochs are equally divided into each cycle by dividing total epochs with several cycles. An initial learning rate is fixed. Here, we have fixed 300 as a total epoch, and training will be carried out for 60 cycles. Each cycle will have 300/60 = 5 epochs. In each cycle, the loss is allowed to decrease quickly using the proposed_lr function. A total of 60 model snapshots are collected, one for each cycle taking the snapshots = [] as the model accumulator. Here, each snapshot is the weight for each model. In PyTorch, the weight of an individual model can be accessed by calling the models' state_dict() function. Similarly, we will get the weight of the model using

```
epochs = 300
cycles = 60
snapshots = []
_lr_list, _loss_list = [], []
count = 0
initial_lr = 0.1
epochs_per_cycle = epochs // cycles
writer = SummaryWriter()
total_iterations = 0
for i in range(cycles):
```

```
lr = initial_lr
for j in tqdm(range(epochs_per_cycle)):
_epoch_loss = 0
lr = proposed_lr(lr, j, epochs_per_cycle)
optimizer.state_dict()["param_groups"][0]["lr"] = lr


for batch in train_iter:
feature, target = batch.review, batch.label
optimizer.zero_grad()
predictions = cnn(feature.to(device))
loss = criterion(predictions.type(torch.FloatTensor),
target.type(torch.FloatTensor))
loss.backward()
optimizer.step()
_epoch_loss = _epoch_loss + loss.item()
acc = binary_accuracy(predictions.type(torch.FloatTensor),
target.type(torch.FloatTensor))
writer.add_scalar('epoch_loss',_epoch_loss, total_iterations)
writer.add_scalar('learning_rate',lr, total_iterations)
total_iterations = total_iterations +1
snapshots.append(cnn.state_dict())
```

Now we have 60 snapshots, and we can get predictions using all of them.

Predicting Using Snapshots

Prediction using the snapshots accumulated earlier is implemented in the test_snapshot_model function. This function takes the following parameters:

Model: An original PyTorch model.

weights: All the snapshots with different weights.

num_last_model: the number of last models to be used for the prediction.

test_iter: Test data iterator.

model_param: Parameters as required by the model while loading it.

Below given is the function that implement above given steps:

```
def test_snapshot_model(Model, weights, num_last_model,
test_iter, model_param):
# parsing model parameters
embed_num = model_param["embed_num"]
embed_dim = model_param["embed_dim"]
```

```
class_num = model_param["class_num"]
kernel_num = model_param["kernel_num"]
kernel_sizes = model_param["kernel_sizes"]
dropout = model_param["dropout"]
static = model_param["static"]
stride = model_param["stride"]
# Fetching number of last models to be used
index = len(weights) - num_last_model
weights = weights[index:]
# initializing all the models with weight of the snapshot

model_list = [Model(embed_num, embed_dim, class_num,
kernel_num, kernel_sizes, dropout, static, stride) for _ in
weights]
# initializing all the models with weight of the snapshot
for model, weight in zip(model_list, weights):
model.load_state_dict(weight)
model.to(device)
# Predicting from all models and averaging the predictions
for batch in test_iter:
feature, target = batch.review, batch.label
optimizer.zero_grad()
predictions = cnn(feature.to(device))
output_list = [cnn(feature.to(device)).detach().numpy() for
model in model_list]
output_list = torch.Tensor(np.array(output_list))
output = torch.mean(output_list, 0).squeeze()
test_loss = criterion(output.float(), target.float()).data[0]
acc = binary_accuracy(predictions.type(torch.FloatTensor),
target.type(torch.FloatTensor))
metrices = {"Accuracy":acc.item()*100,"Test_Loss" :
test_loss.item()}
```

```
return metrices
```

The function can be evoked as follows:

```
model_param = {
“embed_num” : embed_num,
“embed_dim” : embed_dim,
“class_num” : class_num,
“kernel_num” : kernel_num,
“kernel_sizes” : kernel_sizes,
“dropout” : dropout,
“static” : static,

“stride” : stride
}
metrices = test_snapshot_model(CNN_Text,snapshots,
10,test_iter, model_param)
print(metrices)
```

Using a snapshot ensemble, the accuracy was found to be 75.8%, and the minimum binary cross-entropy loss was found to be 1.35. The following diagram illustrates the change in the loss as the cycle progresses:

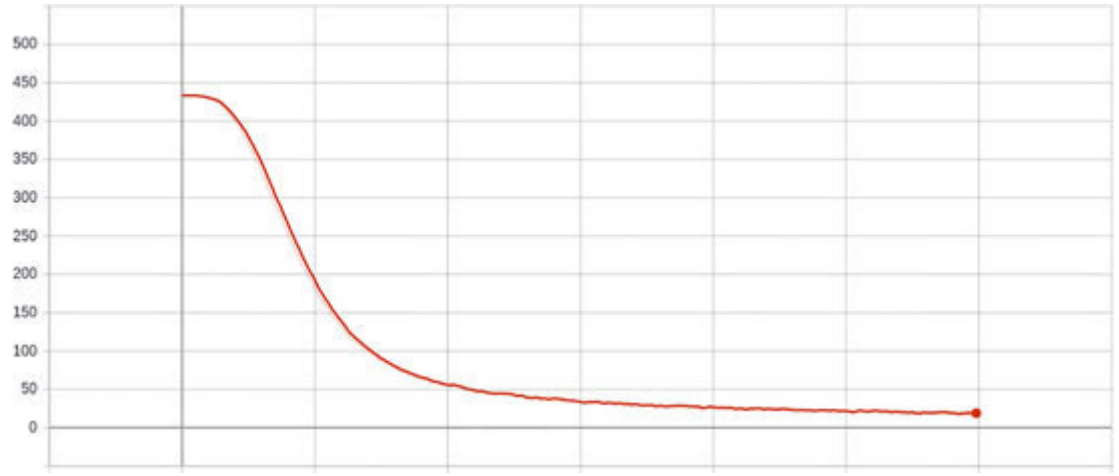


Figure 8.5: *Decrease in the learning rate over various epoch with multiple learning rate cycles, Snapshot ensemble is generally applied to models with millions of parameters.*

I have applied it to a smaller model for illustration purposes. When is applied to a bigger model, it causes fluctuation in the loss of the learning rate changes.

The entire training code with supporting function is given at:

Ch8/using_word_level_ann_with_snapshot.ipynb

Compare it with the model without a snapshot ensemble applied and determine the difference.

Plot a curve where you consider the increasing number of the snapshot for testing and note the change in loss and accuracy on test data.

Use a snapshot ensemble method for a complex model like deep convolution network. Snapshot ensembles: train 1, get m for free: https://arxiv.Org/pdf/1704.00109.pdf

Getting to Know Siamese Networks

Siamese networks are gaining popularity in daily usage, and these networks have wide applications. A Siamese network is an architecture that can be used to rain a model to compare two things. These networks are presently used in the following applications:

Signature verification

Apple photo ID

Comparing text with paraphrasing

Comparing two texts to detect, neutralize, contradict, and enlighten sentences in the SNLI dataset.

DeepFace: Facial recognition system created by research uses a Siamese network

In the upcoming sections, we will learn how to neatly implement the Siamese network with PyTorch and apply it to text comparison-related tasks.

Siamese network architecture has two sister networks connected by the common stem, as shown in the following figure:

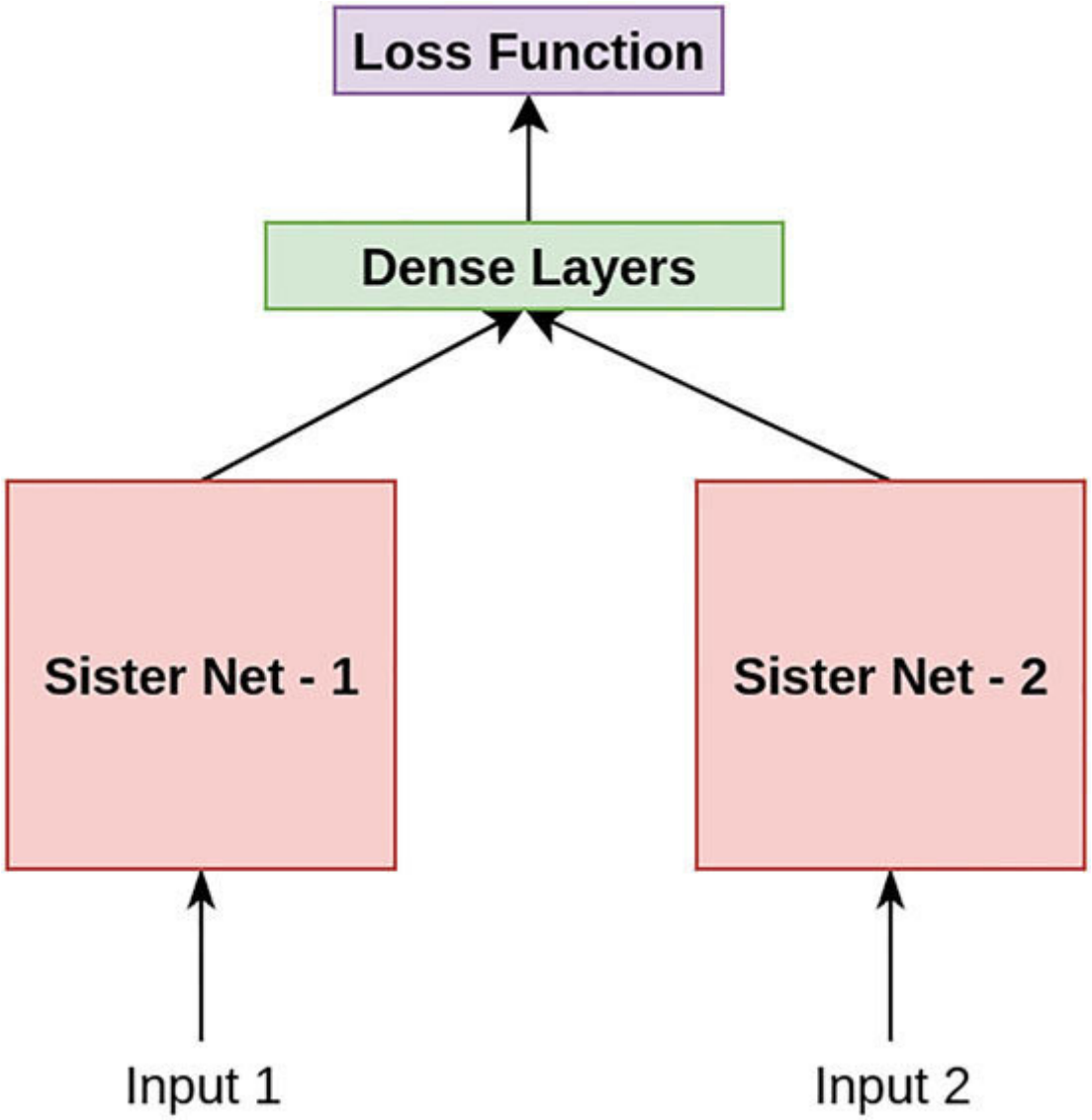


Figure 8.6: *A schematic structure of the Siamese network.*

It always has two sister networks connected to a common stem. Some of the custom loss functions used to train this kind of networks are mentioned in the description.

It is important to note that the two arms of the network must have similar architecture, and they must share the weights. The Siamese network can have various types of layers in the two arms. For example:

Dense layers to process numerical data.

Convolution layer to compare two images.

Recurrent layers to compare two sentences.

A combination of convolution and recurrent layers to compare two signals. These signals can be anything like video or audio streams.

Usually, the Siamese network is used to calculate the binary classification, so it can be trained using the binary cross-entropy loss function. Another cross-function used in the famous paper of style transfer is triplet loss. Contrastive loss can be used as one of the alternatives in the Siamese networks. The conservative loss can be defined as follows:

$$YD_w^2 + (1-Y)max(0, m - D_w)^2$$

$$D_w = \sqrt{\{G_w(X_1) - G_w(X_2)\}^2}$$

Here, is the Euclidean distance between the input of two arms, *ma* is the margin, and *X1* and *X2* are the labels. *Y* is

the label, and it can be either 0 or 1. If the input is from the same class, the value of y is otherwise it's 0. Margin is something similar to the threshold. If given two dissimilar inputs, their distance should be lesser than the margin; else, loss will be calculated. Similarly, if two inputs are similar, the distance should be greater than the margin, or else loss will be incurred.

The contrastive divergence loss can be implemented as follows:

```
loss_contrastive = torch.mean((1-
label)*torch.pow(euclidean_distance, 2) +
(label)*torch.pow(torch.clamp(self.margin-euclidean_distance,
min=0.0), 2))
```

Triplet loss can be used with the Siamese network as one of the alternatives.

Dataset Description

To demonstrate the effectiveness of the Siamese network in comparing two texts, we will use a small dataset present at This dataset was acquired from the Google dataset search, and it is a text similarity dataset available under Database content license Some of the rows from the dataset are given as follows. The dataset is about comparing similar ticker description A stock ticker is a report of the price for certain securities, updated continuously throughout the trading session by the various stock exchanges. The same_security column is used as the label. The goal of our Siamese network to take text x and y and predict whether they are similar.

similar. similar.
similar.
similar.
similar.

similar.
similar.
similar.

Table 8.1

Loading and Pre-processing Data

The Torchtext subclass data. Iterator.splits are used for loading the data, and glove 300-dimensional glove embedding is used as the pre-trained embedding. A similar code snippet of torchtext will be used to get train and test iterators:

```
# defining data fields
TEXT1 = data.Field(sequential=True, preprocessing=tokenize,
use_vocab = True,batch_first=True)
LABEL = data.Field(is_target=True,use_vocab = False,
sequential=False, preprocessing = to_categorical)
fields = [(None, None), ('description_x', TEXT1),('description_y',
TEXT1), (None, None),(None, None), ('same_security', LABEL)]
# constructing tabular dataset
train_data, test_data = data.TabularDataset.splits(
path = 'data/text_simillarity',
train = 'train.csv',
test = 'test.csv',
format = 'csv',
skip_header=True,
fields = fields)
train_iter, test_iter = data.Iterator.splits((train_data, test_data),
sort_key=lambda x: len(x.description_x),batch_sizes=
(config.batch_size,config.batch_size), device=device)
```

Constructing a Sister Network

Here, we have taken LSTM units in the sister network for text processing. Each sister network is taken as an input shape of [batch_size, After the application of embeddings, this shape changes to [batch_size, input_length, The output of the embeddings is given to the LSTM unit. The hidden shape of the LSTM is passed to the dense layer to generate any arbitrary output size. In our case, the sister network outputs [batch_size, Both the sister networks generate such output:

```
class Piller(nn.Module):
def __init__(self, config : Config, vocab_size):
super(Piller, self).__init__()
self.config = config
self.embed = nn.Embedding(vocab_size,
embedding_dim=config.embed_dim)
self.lstm1 = nn.LSTM(config.embed_dim, config.hidden_size,
batch_first=True)
.dense = nn.Linear(self.config.input_size *
self.config.embed_dim,self.config.piller_out_class)
self.init_hidden()
def forward(self,input):
embed_out = self.embed(input)
lstm_out, (self.ho, self.co) = self.lstm1(embed_out, (self.ho,
self.co))
```

```
dense_out =
self.dense(lstm_out.contiguous().view(self.config.batch_size, -1))
return torch.softmax(dense_out, 1)
def init_hidden(self):


bidiractional_state = (1 if self.config.bidirectional==False else 2)
self.ho =
Variable(torch.Tensor(np.random.rand(self.config.n_layers *
bidiractional_state, self.config.batch_size,
self.config.hidden_size)))
self.co =
Variable(torch.Tensor(np.random.rand(self.config.n_layers *
bidiractional_state, self.config.batch_size,
self.config.hidden_size)))
```

The Stem

The stem is the network where both the sister networks converge, and eventually, a fully connected layer is applied before making a comparison by classification:

```
class Stem(nn.Module):
def __init__(self, config):
super(Stem, self).__init__()
self.config = config
self.dense1 = nn.Linear(config.piller_out_class*2,
config.piller_out_class)
self.dense2 = nn.Linear(config.piller_out_class,
config.num_class)
def forward(self, input):
stem_dense1 = self.dense1(input)
stem_dense2 = self.dense2(stem_dense1)
return stem_dense2
```

This network was trained using *Mean Squared Error* as the loss function and SGD as the optimizer. The decrease in training loss and increase in training accuracy as observed with Tensorboard are given here:

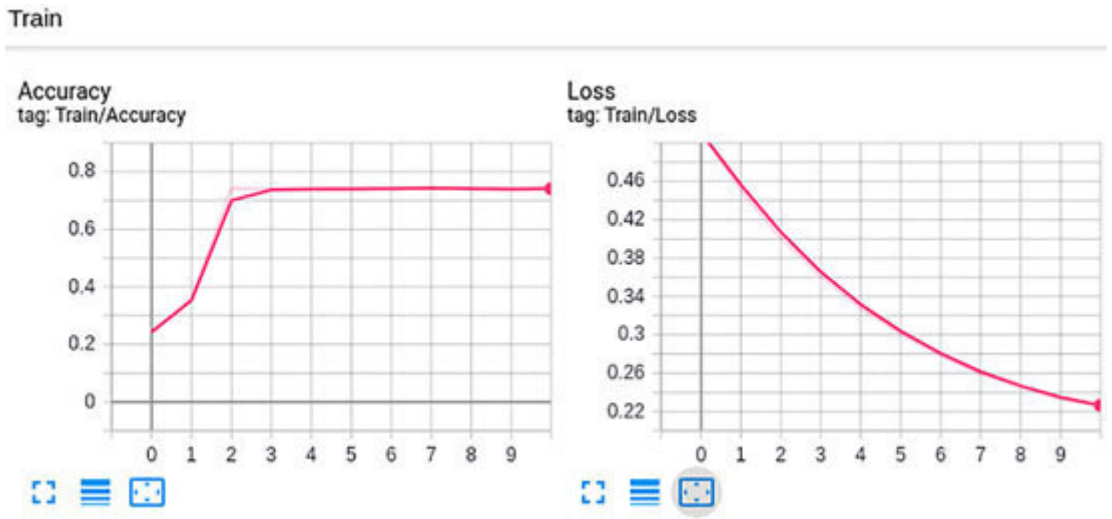


Figure 8.7: *Convergence of the Siamese architecture on the text comparison-related task.*

The shown result *i* on the train data, but the code has commented block to test the accuracy of the test data as well; check it yourself. Siamese network implementation, as discussed earlier, is provided at

Alternatively, the **Stanford Natural Language Inference** dataset can be used. SNLI has 570k paired manually curated sentences with the labels entailment, contradiction, and neutral. The **Natural Language Inference** task is also known as **Textual Entailment** The SNLI dataset U-net is another revolutionary architecture; it was used for biomedical image segmentation. This model is unique as it has convolution layers connected by a skip connection between two arms. Each arm has four convolution layers that gradually compress the given image and then decompress it to bring it to the original shape but with segmentation maps.

Refer to the following links for more details:

Signature verification using a time-delay neural network: *http://papers.nips.cc/paper/769-signature-verification-using-a-siamese-time-delay-neural-network.pdf*

Dimensionality reduction by learning an invariant mapping: *http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf*

U-Net: Convolutional networks for biomedical image segmentation: *https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/*

Application of RCNN

RCNN can have two meanings: **Recurrent Convolutional Neural Network** and **Regional Convolutional Neural** In this chapter, we are referring to Recurrent Convolutional Neural Network, which is also known as **Recurrent Convolutional Network** Two-dimensional CNN helps give state-of-the-art performance in image recognition-related tasks and are very good at preserving spatial information. One problem with CNN is that they are stateless, so the prediction on the current frame has no relationship with the previous frame. On the other hand, RNN is good at understanding the temporal sequence and has demonstrated state-of-the-art results in the speech recognition task. If one wants to process a video stream with a temporal relationship between a two-time-frame, we require a combination of these two. Here, temporal relationship means there exists a connection between a current slice of image and the neighboring images. To process data with a temporal relationship, one would require a network to have both convolution and recurrent components. In 2015, RCN was introduced by *et* to learn video representation.

In this section, we will understand how a convolution unit with recurrent components is developed. The concept is relatively simple to implement, and we take a simple GRU

unit. The following equations can mathematically represent a GRU unit:

$$= + + = + +$$

$$= + (1 - + +$$

Here, is the update gate at time is the reset gate, and is the updated hidden state at time When a convolution operation is applied to the above gate, dot products at various places in the above equation are replaced by convolution operations:

$$= + +$$

$$= + +$$

$$= + - + +$$

Here, * represent the convolution operation. The effective implementation of convolutional GRU is given here:

```
reset_gate = nn.Conv2d(input_size + hidden_size, hidden_size, kernel_size, padding=padding)
update_gate = nn.Conv2d(input_size + hidden_size, hidden_size, kernel_size, padding=padding)
out_gate = nn.Conv2d(input_size + hidden_size, hidden_size, kernel_size, padding=padding)
```

Each time, the current input_ and prev_state are passed to the update, reset gate, and output gate. The new state value is calculated from the values of these states. The model outputs the new state value:

```
stacked_inputs = torch.cat([input_, prev_state], dim=1)
update = F.sigmoid(self.update_gate(stacked_inputs))
reset = F.sigmoid(self.reset_gate(stacked_inputs))
out_inputs = F.tanh(self.out_gate(torch.cat([input_, prev_state * reset], dim=1)))
new_state = prev_state * (1 - update) + out_inputs * update
```

In the next section, we will see how to utilize these layers to model the data with temporal nature.

Preparing the Dataset

In real-life applications, RCNN architecture is applied to the video stream processing as done by Nitish Srivastava et al., in a research paper named *Learning of Video Representations using* This seems amazing, but it requires a great amount of computing power to process video. The previous model for video representation was trained on *Nvidia Titan GPU* for 300. To learn the concept easily, we require less data that allows experimentation. For this purpose, we will first prepare the dataset and then apply CNN and RCNN on it. The dataset consists of a frame and a box that moved to and fro in the frame. The task is to predict the next position of the box, as shown in yellow. This box will be moving to and fro in the shown frame:

Why Is It Difficult?

Based on the current position of the box, it can have two positions: in the right of the box and in the left of the box, as shown in the following diagram. CNN considers the current frame without any previous information about the direction in which the box was moving. In contrast, the RCNN predicts the next position of the box by taking input from many previous frames:

Figure 8.8: *Showing probable prediction looking at previous two frames.*

How Can It Be Solved?

CNN, combined with the temporal memory, can solve this problem. This combination is commonly known as Recurrent-CNN architecture:

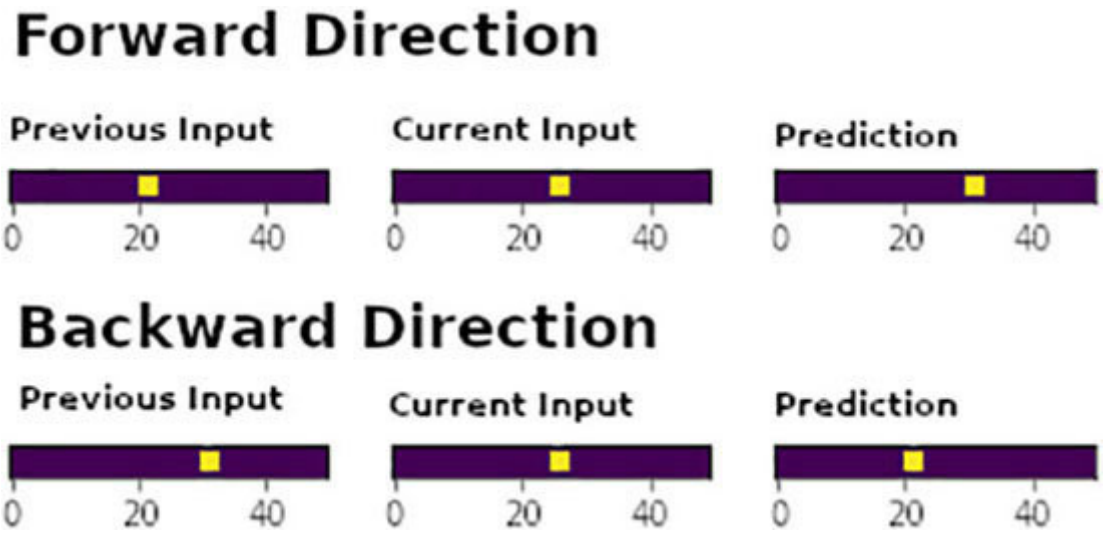


Figure 8.9: *Showing how RCNN can help to mitigate problem by introducing context.*

You can see the final output of the data as a GIF file format at To get an idea about how the synthetic dataset looks, you can look at the interactive *ipython* notebook at

Predicting Using CNN

Let's see what happens when CNN is used for predicting the position of the box given the current position. Our CNN model looks like the following:

```
class CNN(nn.Module):
def __init__(self):
super(CNN, self).__init__()
self.conv1D = nn.Conv1d(in_channels=5,out_channels=2,
kernel_size =3)
self.dense1 = nn.Linear(in_features=2*48,out_features=10)
self.dense2 = nn.Linear(in_features=10,out_features=1)
def forward(self, input):
conv_out = self.conv1D(input)
conv_out_reshape = conv_out.view(-1,48*2)
dense1_out = self.dense1(conv_out_reshape)
dense2_out = self.dense2(dense1_out)
return dense2_out
```

It has one Conv1D layer, followed by dense layers. The input to this network will be a 2D array with the position of the box in the frame. The expected output will be the next position of the box in the frame. The entire experiment, with pre-processing, model construction, and loss calculation, is given at The final loss with this model was 17.48. The main problem with CNN is that it is not made for keeping

temporal or sequential information. So, the model is unable to predict the direction in which the box is moving. As a result, error occurs in the prediction.

Predicting Using RCNN

A similar experiment was carried out using RCNN. The layers in the model are arranged as discussed in the previous section:

```
class ConvGRUCell(nn.Module):
"""
Generate a convolutional GRU cell
"""
def __init__(self, input_size, hidden_size, kernel_size):
super().__init__()
padding = kernel_size // 2
self.input_size = input_size
self.hidden_size = hidden_size
self.reset_gate = nn.Conv2d(input_size + hidden_size,
hidden_size, kernel_size, padding=padding)
self.update_gate = nn.Conv2d(input_size + hidden_size,
hidden_size, kernel_size, padding=padding)
self.out_gate = nn.Conv2d(input_size + hidden_size,
hidden_size, kernel_size, padding=padding)
init.orthogonal(self.reset_gate.weight)
.orthogonal(self.update_gate.weight)
init.orthogonal(self.out_gate.weight)
init.constant(self.reset_gate.bias, 0.)
init.constant(self.update_gate.bias, 0.)
init.constant(self.out_gate.bias, 0.)
```

```
def forward(self, input_, prev_state):
# get batch and spatial sizes
batch_size = input_.data.size()[0]
spatial_size = input_.data.size()[2:]
# generate empty prev_state, if None is provided

if prev_state is None:
state_size = [batch_size, self.hidden_size] + list(spatial_size)
if torch.cuda.is_available():
prev_state = Variable(torch.zeros(state_size)).cuda()
else:
prev_state = Variable(torch.zeros(state_size))
# data size is [batch, channel, height, width]
stacked_inputs = torch.cat([input_, prev_state], dim=1)
update = F.sigmoid(self.update_gate(stacked_inputs))
reset = F.sigmoid(self.reset_gate(stacked_inputs))
out_inputs = F.tanh(self.out_gate(torch.cat([input_, prev_state *
reset], dim=1)))
new_state = prev_state * (1 - update) + out_inputs * update
return new_state
```

This model takes the previous frame with box location and current frame with box location as input and provides the next box location. The ConvGRUCell class only provides the new state, and everything else is transformed in the ConvGRU The ConvGRU class takes this new state, applies a few dense layers, and provides the final output, that is, the location of the yellow box. Applying RCNN to the preceding data reduces the total loss to 9.63. You can experiment with the Ipython notebook at Ch8/RCNN/testing_rcnn_using_synthetic_data.ipynb to get more solutions to this problem.

A research paper Recurrent convolutional neural networks for text classification wherein RCNN applied to the text classification, and state of the art result is achieved. CNN is good at understanding character-level similarity and is used for state-of-the-art models like RCNN could be the next bigger domain to be explored to get state-of-the-art results.

Take a look at the following links for more information:

Exploring the limits of language modeling:
https://arxiv.org/pdf/1602.02410.pdf

Recurrent convolutional neural networks for text classification:
https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9745

Unsupervised learning of video representations using LSTMS:
https://arxiv.org/abs/1502.04681

Understanding CTC Loss

CTC loss or the connectionist temporal classification is an important milestone in the area of speech recognition and **optical character recognition** CTC is a loss function specifically designed to help discretize components from the sequence. Jurgen Schmidhuber (the inventor of the LSTM) and coworkers originally proposed CTC loss.

The Simplest Choice

We will take an example to demonstrate the necessity of CTC loss. Let's consider handwriting recognition as the task. The easiest choice would be to apply RNN and to the equal size slices taken from the image and apply CNN to the pixel-level features. To train such a network, one would require the loss function to be such that it outputs the character score for each element of the written word and represents it in the form of the matrix. Such a loss function should be differentiable to allow gradient-based optimizers to train such networks. Such loss function should be able to the decode contained text. The generalized pipeline is as shown here:

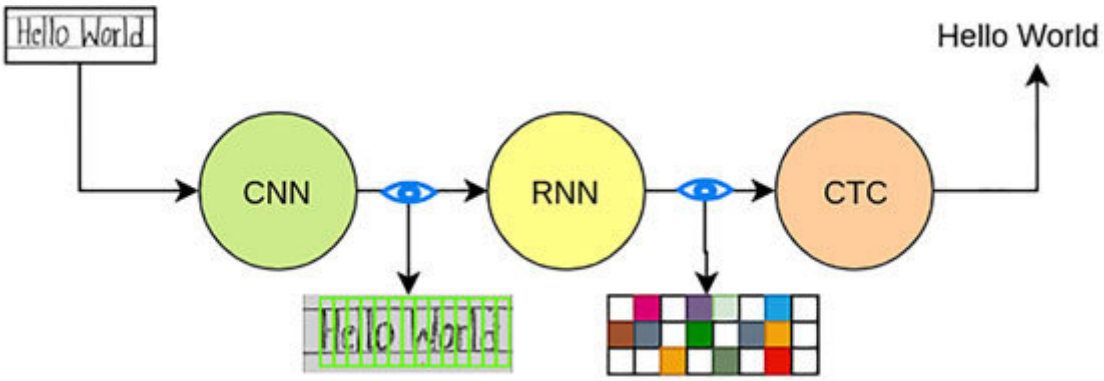


Figure 8.10: *The placement and importance of CTC loss in the pipeline.*

Importance of the CTC loss The OCR and speech recognition tasks will always be performed as supervised learning. If we

don't use CTC loss, we would require two things to apply the supervised learning paradigm:

A horizontal location of each image

The corresponding character for each slice must be tagged

However, annotating data like this is extremely time-consuming. Plus, after data is annotated like this, one needs to deal with merging character that is spanning in multiple slices. For example, *W* is wider than *a* in the preceding example. According to slicing, the translation should be Taking another example of the word man, translation performed in the annotated way would yield CTC loss solves both the problems as:

We only need to input the CTC loss to the image and the output text as label, and the rest will be taken care of by CTC loss.

No further processing of text, like merging, is needed.

How Does CTC Work?

We only feed the output of the previous layers' neural network layers to the CTC loss, along with the ground truth labels. CTC loss figures out how to map the text to characters and provide non-repetitive output. We will see how CTC loss works. The one issue was with encoding repeated characters, as we discussed earlier. A sign space character is used to designate the space between the characters in the word. According to the out translation for the word, **man** will be changed to ***--mmm-a-nn*** and will be reduced to man by combining the duplicate characters. This character encoding has side effects. Let's say we have the word **woo!** It will be converted to We will see how this will be taken care of.

Loss Calculation

For the word I have shown a simple matrix to trace one character at each time step. The following figure illustrates the tracing of character with different probabilities of that character being present. The probability for each combination is calculated, and in the following diagram, the max probability comes out to be ---, which corresponds to --- as the output sequence. All parameters were randomly initialized at the beginning of the training. The loss of output --- with the ground truth man is calculated. This loss is simply the negative logarithm of the probability. It is backpropagated, and the gradients are backpropagated for parameter updates.

Understanding Decoding

When we train the model as described earlier, we can use it for inference on an unknown image. As shown earlier, the algorithm can have a single path, and that's the one with the highest probability associated. Still, to find out such path, the algorithm needs to traverse through all the permutation combinations and check through the best there can be -- the number of paths. For the preceding example, the equation says there are -- paths. In practice, the sequence will not be much a simpler one, and the combination explodes and become computationally expensive as the length increases.

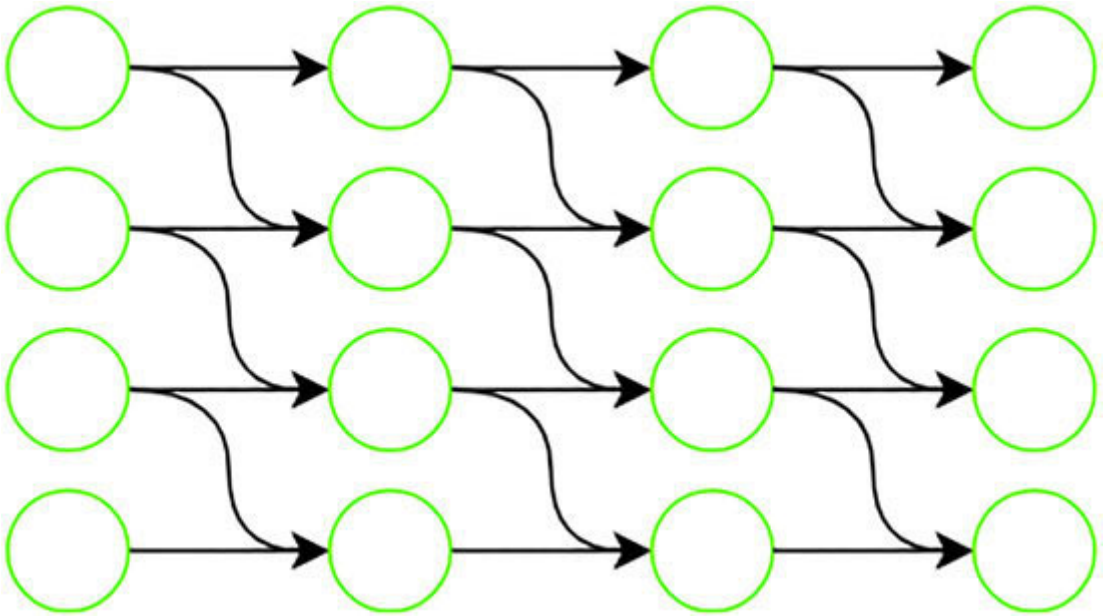


Figure 8.11: *Schematic diagram showing how decoding could take place. The circles represent each character and arrows represent their tendency to select another character.*

We can apply a dynamic algorithm to this problem to simplify the calculation. We will see how dynamic programming solves the problem in the next section in finding the best path. Another way to solve this problem in the least possible computational cost is to choose the best **path decoding** The best path decoding algorithm selects the highest probability at each timestep and moves to the next time step. This way, it selects the path with all characters having the highest probability for the given time step. After this, the algorithm combines duplicates and removes the character of the space to provide the final output. By property, CTC loss is conditionally independent. This is, in fact, the shortcoming of the CTC loss algorithm. It assumes each output to be conditionally independent of the previous outputs. It is bad for many sequences to sequence problems like voice recognition and optical character recognition.

CTC loss calculation requires an external module, like and The detailed use of both these module is discussed under the following headings:

Installation

Usage

Even with solid understanding of CTC loss, its implementation is very difficult. The algorithm has several measures to deal with edge cases, and faster implementation should be written

in low-level programming languages. Due to this, we will simply use a PyTorch function to demonstrate the use of CTC loss. PyTorch implementation for the CTC loss is provided as a package

Installation

CTC-warp is compatible with torch version 0.4.
WARP_CTC_PATH and should be set to the location of a built WarpCTC This defaults so you can build WarpCTC from within a new warp-ctc clone, as follows:

```
cd warp-ctc
mkdir build; cd build
cmake ..
make
# Now install the bindings:
cd PyTorch_binding python setup.py install
```

Usage

CTC loss can be used as in the example:

```
import torch
from warpctc_PyTorch import CTCLoss
ctc_loss = CTCLoss()
```

Taking two sequences:

```
probs = torch.FloatTensor([[[0.1, 0.6, 0.1, 0.1, 0.1], [0.1, 0.1, 0.6, 0.1, 0.1]]]).transpose(0, 1).contiguous()
```

Creating dummy labels and storing the length of the label and the required probability to be received from the CTC module:

```
label_sizes = torch.IntTensor([2])
probs_sizes = torch.IntTensor([2])
```

Calculating CTC loss:

```
probs.requires_grad_(True) # tells autograd to compute gradients for probs
cost = ctc_loss(probs, labels, probs_sizes, label_sizes)
cost.backward()
```

As shown above, CTC loss can be easily incorporated into an existing pipeline.

Another Baidu package named is compatible with *python* This package can be installed using pip install

Take a look at the following links for more information:

Connectionist temporal Labeling unsegmented sequence data with recurrent neural networks:
https://www.cs.toronto.edu/~graves/icml_2006.pdf

Sequence Modeling with *https://distill.pub/2017/ctc/*

Captioning Image

Image captioning is the process of describing what is happening in an image. An example of the image and its captions is given as follows:

Figure 8.12: *An example image.*
Source: *https://www.pexels.com/photo/girl-holding-balloons-1140713/*

This image can be captioned as:

A girl holding balloons.

A white girl holding three balloons.

A white girl holding balloons at the lake side.

Since CNN is not good at keeping temporal information, the image captioning task can be divided into two models; image-based that takes features from the image, and a language model that takes the feature from the previous model and generates the description, like the language translation task.

We have been using RNN and CNN separately in many tasks, namely, classification, translation, and embedding generation. In this chapter, we will use CNN to input the image, and the learned information will be passed down to the LSTM. Here, RNN acts as the generative model and will help generate appropriate descriptions for the image. We will train our machine in a supervised manner. Here, CNN is used as the encoder, and RNN is used as the decoder. The following schematic diagram illustrates how the task will be accomplished. It is the simplest model with a few CNN layers, followed by linear/dense layers. The output of the dense layer is passed to the RNN units. The RNN unit is fed with the start of sequence token, and it generates the next word. The generated word at time step *t* is fed to RNN at *t+1* time-step, and a new word is generated. This continues until the **End of sequence** token is reached. Take a look at the following figure:

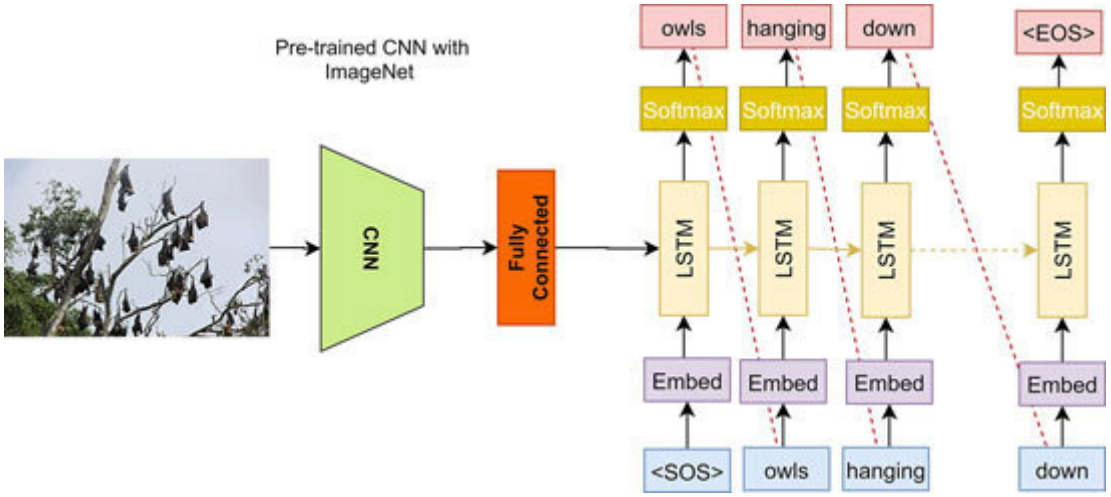


Figure 8.13: *A model architecture for image captioning.*

This seems to be simple isn't, it? It is very simple to make the image captioning model; but the difficult part is dealing with training data. To train this task, we will use the 13 GB MS-COCO data. By getting to know the data-size, you must have realized that this model requires a high-end machine with GPU to train. Due to the data size, one cannot train this model on the Google lab. I have trained the model on my PC that has 32 GB RAM and *Nvidia 1080 Ti* with 11GB VRAM attached. You can use AWS or Google Cloud. Coding and converging this model is the next level of experience and will surely boost your confidence in building a model with PyTorch.

The preceding model is a basic one for the task. Many researchers have come up with advanced models by modifying this basic model to optimize the output and convergence.

Downloading the Data

To demonstrate the concept of image captioning, we will use the *Flickr8k* dataset released by Flickr. This dataset has one image and five captions describing it in different ways. You can download this dataset from As an alternative academic, torrent can be used to download the dataset for non-commercial purposes. The Flickr8k dataset can be downloaded from academic torrents by clicking on this link: *http://academictorrents.com/details/9dea07ba660a722ae1008c4c8afdd303b6f6e53b*

Implementation

Image Image augmentation is often used for better generalization. Image augmentation means increasing images by applying edits and increasing the training data. Here, we will augment the images using torchvision. Transform function. We will apply effects like Random crop, Random Horizontal flip, and normalizing image, as follows:

```
transform_train = transforms.Compose([
transforms.RandomCrop(224),
transforms.RandomHorizontalFlip(),
transforms.ToTensor(),
transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
transform_test = transforms.Compose([
transforms.RandomCrop(224),
transforms.ToTensor(),
transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
```

Encoder Module

As discussed in the schematic diagram of the model architecture for image captioning, the encoder is made up of convolution layers. The encoder takes an image and converts it to the image context vector. Generally, a pre-trained model is used to convert an image into a context vector. This trained model can be any network like and A ResNet model is loaded, and the last layer of such a pre-trained network is removed so that it gives an n-dimensional vector for any image. This n-dimensional vector has information related to the images and is later consumed by the decoder module:

```
class EncoderCNN(nn.Module):
def __init__(self, embed_size):
"""Load the pretrained ResNet-50 and replace top fc layer."""
super(EncoderCNN, self).__init__()
resnet = models.resnet50(pretrained=True)
modules = list(resnet.children())[:-1] # delete the last fc layer.
self.resnet = nn.Sequential(*modules)
self.linear = nn.Linear(resnet.fc.in_features, embed_size)
self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)
def forward(self, images):
"""Extract feature vectors from input images."""
with torch.no_grad():
features = self.resnet(images)
features = features.reshape(features.size(0), -1)
```

```
features = self.bn(self.linear(features))
return features
```

Decoder Module

The decoder module is simple and is similar to the decoder module we used for language translation in *Chapter 4, Using RNN for* It has one LSTM layer, followed by a linear transformation. The generation takes place using teacher forcing:

```
class DecoderRNN(nn.Module):
def __init__(self, embed_size, hidden_size, vocab_size,
num_layers, max_seq_length=20):
"""Set the hyper-parameters and build the layers."""
super(DecoderRNN, self).__init__()
self.embed = nn.Embedding(vocab_size, embed_size)
self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,
batch_first=True)
self.linear = nn.Linear(hidden_size, vocab_size)
self.max_seg_length = max_seq_length
def forward(self, features, captions, lengths):
"""Decode image feature vectors and generates captions."""
embeddings = self.embed(captions)
embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
packed = pack_padded_sequence(embeddings, lengths,
batch_first=True)
hiddens, _ = self.lstm(packed)
outputs = self.linear(hiddens[0])
return outputs
```

```
def sample(self, features, states=None):
"""Generate captions for given image features using greedy search."""
sampled_ids = []

inputs = features.unsqueeze(1)
for i in range(self.max_seg_length):
hiddens, states = self.lstm(inputs, states) # hiddens: (batch_size, 1, hidden_size)
outputs = self.linear(hiddens.squeeze(1)) # outputs: (batch_size, vocab_size)
_, predicted = outputs.max(1) # predicted: (batch_size)
sampled_ids.append(predicted)
inputs = self.embed(predicted) # inputs: (batch_size, embed_size)
inputs = inputs.unsqueeze(1) # inputs: (batch_size, 1, embed_size)
sampled_ids = torch.stack(sampled_ids, 1) # sampled_ids: (batch_size, max_seq_length)
return sampled_ids
```

Appropriate loss function and We are using a cross-entropy loss function. Ideally, I need to take care of the padding in batch by not calculating the loss for pad tokens, but I want to keep this implementation simple, so I'm using nn. CrossEntropyLoss() from

```
criterion = nn.CrossEntropyLoss()
```

We are using Adam optimizer with a learning rate of 0.0001. Training has the following steps:

Sorting is applied to captions and images according to the caption length.

A PyTorch pack_padded_sequence function helps pack variable-length caption to a max length of any of the captions.

The image is passed to the encoder and getting an image vector/context vector.

The decoder module takes these features and generates the caption word by word.

Loss calculation and backpropagation take place.

Below given is the function that implement above given steps:

```
for epoch in range(epochs):
for i, (images,captions,lengths) in enumerate(train_dataloader):
images = images.to(device)
captions = captions.to(device)
images,captions,lengths=sorting(images,captions,lengths
targets =
pack_padded_sequence(captions,lengths,batch_first=True)[0]
features = encoder(images)
outputs = decoder(features,captions,lengths)
```

Below given results where the caption is generated by taking an image. The captions generated are very accurate:

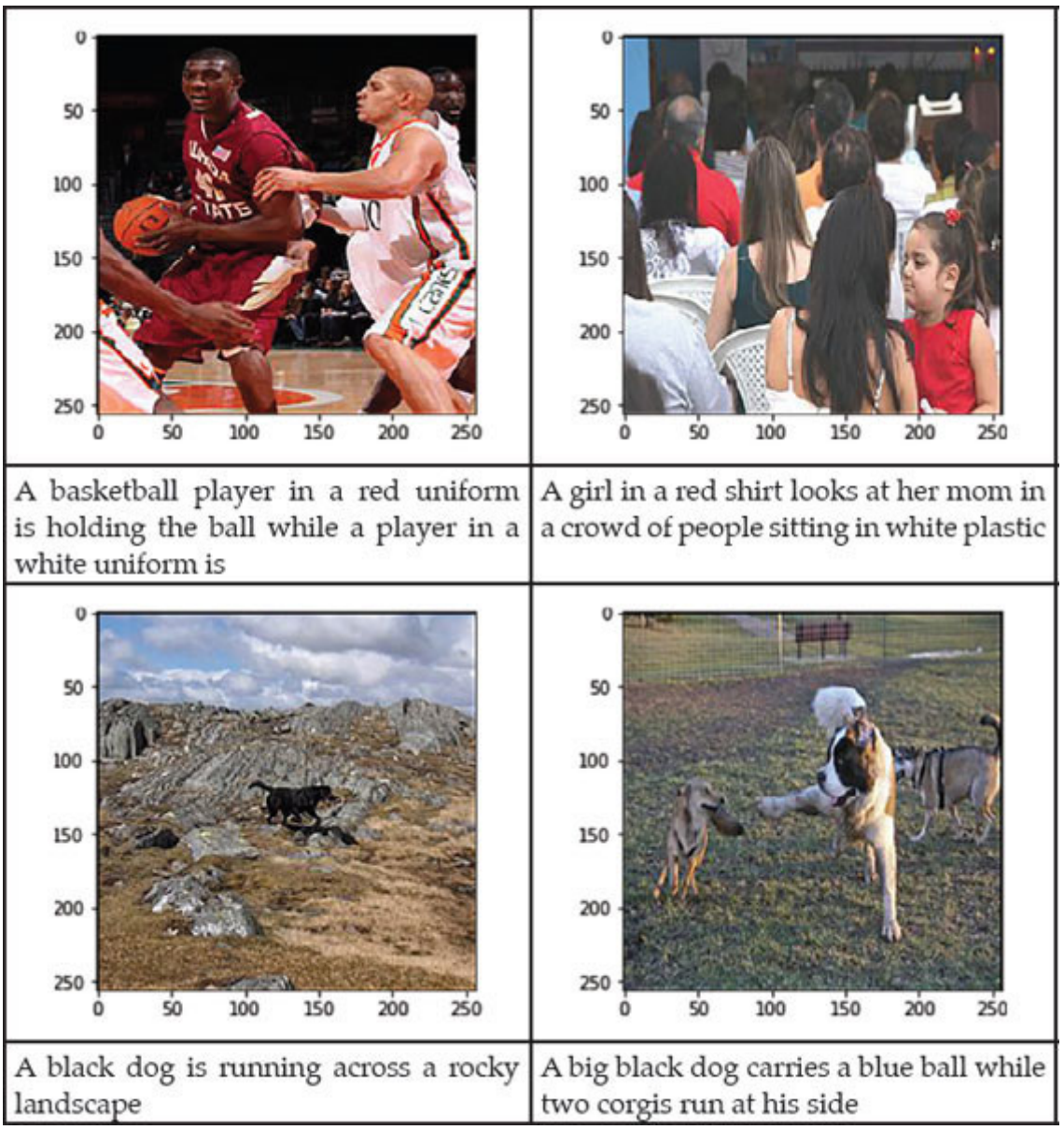


Figure 8.14

The entire code with proper comments is given at You can run the code and experiment by providing unknown images to see whether accurate captions are generated.

Beam Search

We have used LSTM-based decoders in language translation. We know the translation was not great and need measures to improve it. One of the problems with our technique lies in the decoding phase. We have always taken the word generated with the highest probability at each time step and added to the sequence. This method of taking the best at each time step is known as a greedy approach. At first, it may seem that this technique can provide the best generation, but it doesn't if you select senescent level score by multiplying the probabilities of each token. The name suggests greedy, which means a method that seeks immediate reward. In contrast, beam search considers rewards over some time. By taking argmax at each time step over the probabilities of tokens, we were seeking immediate rewards. In beam search, the concept is to trace the path that gives maximum rewards and not just selects the token with maximum probability each time. Beam search can be performed using the following function, which takes data in the form of 2D array and which is the number of different paths that require high probability. Below given is the function that implement above given steps:

```
def beam_search_decoder(data, k):
path =[[list(), 1.0]]
for row in range(len(data)):
```

```
selected_path = []
index_score = []
for i in path:
sequence, score = i
for j in range(len(data[row])):

index_score.append([sequence+[j],score * -log(data[row][j])]) #
log to check overflow or underflow
selected_path.append(index_score)
sorted_index_score = sorted(index_score, key= lambda tup :
tup[1])
path = sorted_index_score[:k]
return path
```

For demonstration purposes, we will take one [10, 10] Numpy random array as an array for the vocabulary of 10 words and 10-time steps. Each row in this data represents the probability of the prediction of each word:

```
data = np.random.rand(10,10)
data = array(data)
# decode sequence
result = beam_search_decoder(data, 3)
```

The result will be something like this for *k* =

```
[[7, 7, 3, 4, 1, 1, 1, 6, 3, 7], 1.2111146290953402e-12]

[[7, 7, 7, 4, 1, 1, 1, 6, 3, 7], 1.2455253954806299e-12]
```

[[1, 7, 3, 4, 1, 1, 1, 6, 3, 7], 1.4476142406986034e-12]

Each result has a sequence indicating the location of the tokens and the resultant score. The sequences so generated are sorted in descending order of score. If such a path is plotted using Matplotlib, three of the paths are somewhat different and give varying sequences. This deviation will increase when we have a sufficiently large vocabulary size and provide different sentences similar in meaning but written differently.

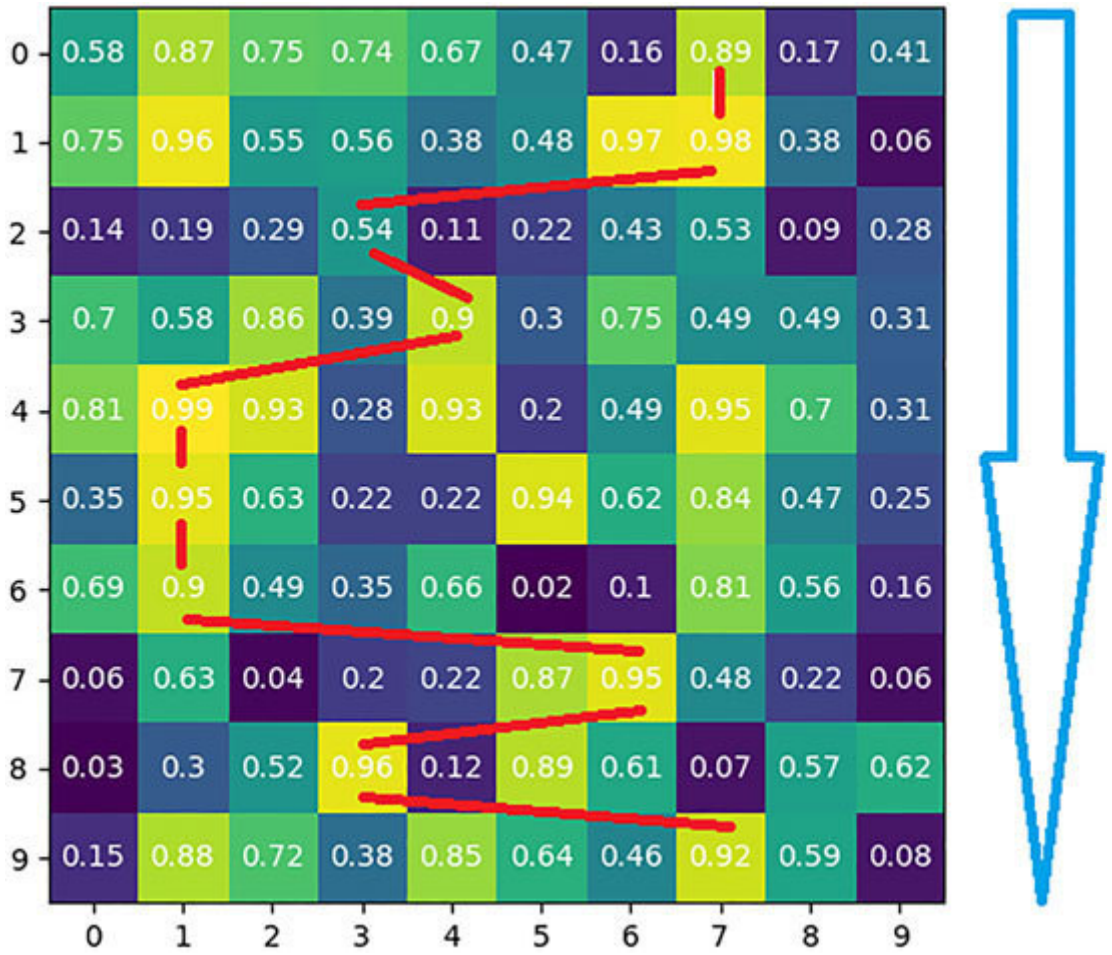


Figure 8.15: *Beam search illustrated; it takes different paths that jointly maximize the probability.*

Possible path with maximum probability:

[[7, 7, 3, 4, 1, 1, 1, 6, 3, 7], 1.21111462909534o2e-12]

[[7, 7, 7, 4, 1, 1, 1, 6, 3, 7], 1.24552539548o6299e-12]

[[1, 7, 3, 4, 1, 1, 1, 6, 3, 7], 1.44761424069860342-12]

It takes different paths that jointly maximize the probability. Path tracing for the first path is displayed in the preceding diagram.

Variants

If you don't have enough computing power at your disposal, you can try to train the preceding model on smaller datasets. You can use the following ones:

The name suggests greedy, which means a method that seeks immediate reward, and, in contrast, beam search considers rewards over time. By taking argmax at each time step over probabilities of tokens, we were seeking immediate rewards. The greed search can be demonstrated by the following given an example. Pascal sentence dataset:
http://vision.cs.uiuc.edu/pascal-sentences/

Flickr 30k image caption corpus:
http://shannon.cs.illinois.edu/DenotationGraph/

Bringing semantics into focus using visual abstraction learning the visual interpretation of sentences:
https://vision.ece.vt.edu/clipart/

For other sources, you can refer to
http://www.cs.toronto.edu/~fidler/slides/2017/CSC2539/Kaustav_slides.pdf by *Kaustav* A student at the *University of*

The preceding model is good enough to provide an insight into all the components required for the image captioning model. Here onward, you can implement any of the following models. These models are easy to implement and converge in a relatively shorter duration to provide better captions:

Rich image captioning in the *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/ImageCaptionInWild-1.pdf*

A neural image caption generator: *https://arxiv.org/pdf/1411.4555.pdf*

Conclusion

With this chapter, we moved a little ahead with complex networks. We covered techniques like sentence piece, which is the token encoding technique used with new networks like Bert and Transformers. It greatly simplifies and compresses the vocabulary size. We also saw techniques like random multimodal learning and snapshot ensemble. Snapshot ensemble is a technique that provides *n* models per training run. After training, such models can be used parallely for better accuracy. Siamese networks are versatile and can be used for comparing images, text, or videos, depending on the component layers. Lastly, we saw a hybrid where CNNs are used recurrently and have unique properties. Until now, we have been using known loss functions like entropy and mean squared loss in all the NLP tasks.

The next chapter will walk you through *Generative*

CHAPTER 9

Understanding Generative Networks

This chapter covers brilliant inventions like the Generative Adversarial Network (GAN), by Ian Goodfellow. Ian Goodfellow currently works for Apple Inc and is a director of machine learning in the Special Projects Group. This chapter will cover the theoretical and practical aspects of the GAN, starting with unsupervised pretraining and how it can help gain good accuracy with less supervised data. This chapter will cover how to apply GAN on MNIST data and generate real-like data. GAN also helps in data augmentation. After gaining practical knowledge of the GAN, we will look at all its theoretical aspects. The last recipe in this chapter will cover a variation of GAN, i.e., Conditional GAN or CGAN. CGAN are widely used for additional data generation. GAN can be used to generate super-resolution images.

Structure

This chapter will cover the following recipes:

Understanding unsupervised pretraining

The GAN architecture

Implementing GAN for MNIST

Understanding the theory behind GAN

Generating image from the description

Objective

Understanding the fascinating world of Ian Goodfellow.

An understanding basic concepts of Generative Adversarial Network such as Nash Equilibrium, KL-Divergence, KL-Divergence, JS-Divergence, and KullbackLeibler Divergence.

Tips and tricks to solve the problem of an unstable gradient in GAN.

Understanding and coding different types of GAN like VariationalAutoencoder and learning the application of GAN in generating images from text.

Technical Requirements

Codes for this chapter are in the Ch9, Understanding Generative Networksat GitHub repository To understand this chapter, you must have basic knowledge about the following Python packages:

Torch

TorchVision

Matplotlib

NumPy

TensorBoardX

You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r

Understanding Unsupervised Pretraining

Today, networks are trained with a huge amount of data in a supervised manner to achieve state-of-the-art result. For example, the image net dataset has 1M images for 1000 classes, manually labeled by humans. Labeling such big datasets can take a significant amount of time, which cannot be the optimal strategy when for the long-term goals. Imagine the amount of manual effort required to create the dataset with 1M classes and labeling each video frame, each video with 100 thousand frames. This is the reason we would require some techniques to decrease manual effort. In this recipe, we will explore the basic building blocks of the GAN to pre-train the network so that it can produce state-of-the-art results with smaller supervised corpus.

GAN's main goal is to train the encoder in an unsupervised manner so that it can be used in supervised pretraining. The aim of achieving a state-of-the-art result is very similar to the one we have seen previously by using embeddings. The difference being the variation of the concept behind the training. Typically, GAN has two blocks: a Generator and a discriminator. Let me give you an example to provide a better picture of the GAN. The GAN works by game theory, specifically, a min-max game.

Assume that a fake wine shop is competing with a branded one, and every time it goes for customs clearance, the wine inspector identifies it to be fake and rejects it. The wine shop gets more revenue if the wine made is so close to real that it cannot be identified by the inspector and goes to market. The inspector gets an incentive if they can identify real wine correctly and reject the fake wine. With time, the fake wine shop becomes much more experienced and makes close-to-real-wine samples. Meanwhile, the inspector also gains experienced about how to identify fake samples. Now, the fake winery must make wine as close as possible to the real one, and the inspector's job is to gather enough experience to ensure that no fake sample passes the test. To generate samples as close to the real one as possible, the wine requires all the ingredients in the correct quantities. The inspector's job is to give continuous feedback by rejecting the wrong and accepting the correct samples. In GAN, the generator is equivalent to the fake wine shop, and the discriminator is equivalent to the inspector. In the upcoming section, we will explore the overall architecture of the GAN.

GAN Components

Let's understand the various components of GAN.

The Generator

The generator's function is to generate an image sample using a random vector with a normal or uniform distribution. With z as the input, we create an image *x* using a generator Mathematically, this can be given as *x* = G(z). The X can be anything; it can be an actress's face, a coffee mug, or any digit of the MNIST dataset. The generator initially generated random noise, but over time, it gets feedback from the discriminator and generates what is required. Here, the generator is the magical function made up of the deep neural network, as shown in the following image:

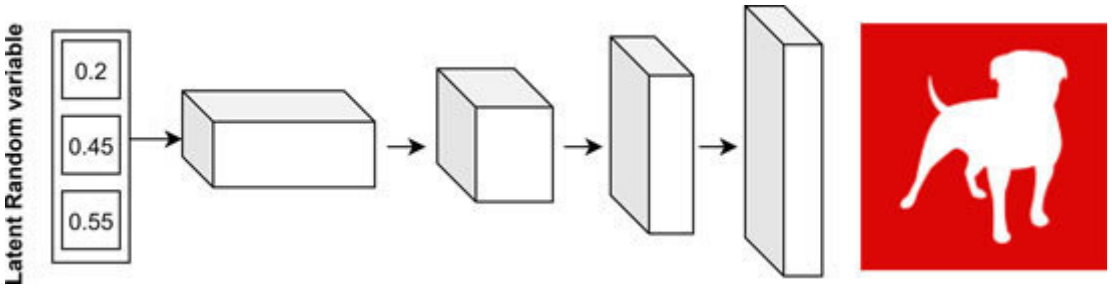


Figure 9.1: *The generator network accepts any noisy input (latent random variable) and provides an image at the output.*

The generator can be made up of the feedforward network, CNN, or a combination of both. Having a generator network with RNN unit is rare, and research is being conducted for applying GAN to the language/speech data.

The Discriminator

The discriminator is a similar network, but it shrinks the image size gradually and categorizes it into any of the two classes. The discriminator's role is to classify the image as fake or real. It may look like this:

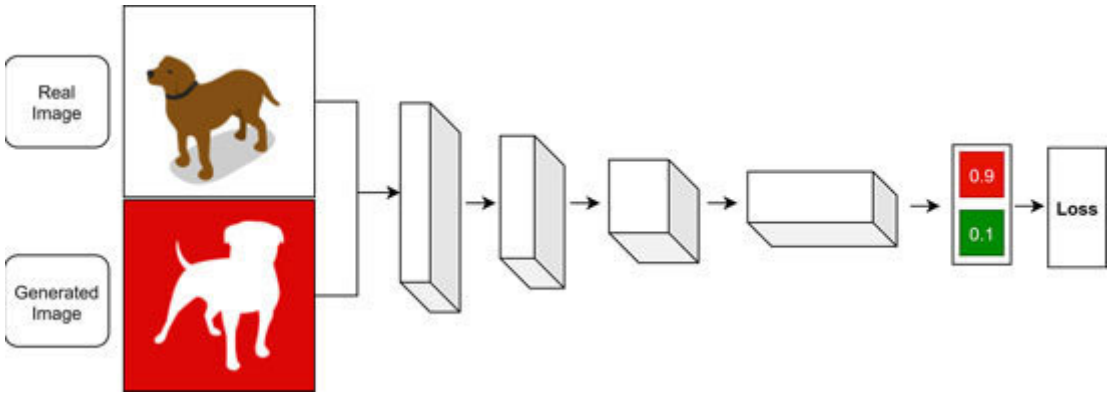


Figure 9.2: *Discriminator network having CNN layers.*

Discriminator takes either real or fake images and identifies them. The discriminator *D* takes an image *x* (real/ fake) and outputs a probability indicating whether the image is real or fake, respectively. Mathematically, this can be shown as: $\hat{y}$ = Ideally, the probability of the image generated by the generator provided to the discriminator should be toward zero, and the probability of the real image provided to the discriminator should be 1.

The GAN Architecture

The overall architecture has a generator and a discriminator. Additionally, the architecture has a mechanism to provide real or fake images to the discriminator and a custom loss function. We will look at the custom loss function in the upcoming recipes. The schematic diagram of the network looks as follows:

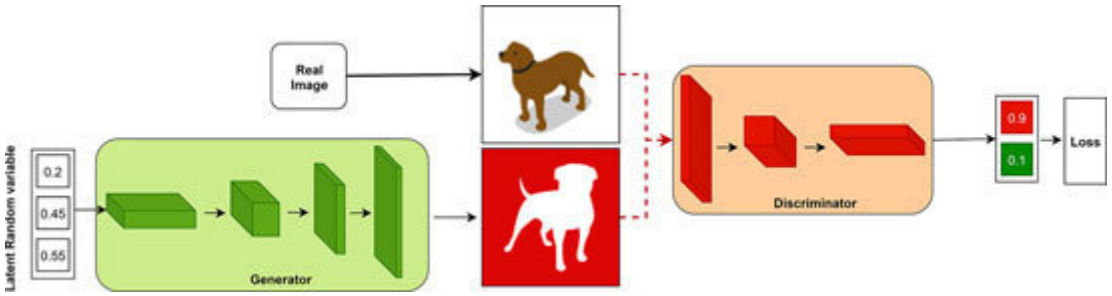


Figure 9.3: *Showing GAN architecture comprises of the generator and discriminator.*

The generator takes a random vector or latent random variable and generates an image (this image is referred to as fake - as it is generated by generator). Real images and fake images are given to the discriminator, and the discriminator's job is to identify them correctly. The generator is trained to generate images so that the discriminator can be cheated.

The Loss Function

GAN is trained for binary classification, and we usually use the binary cross-entropy loss very often. For the binary classification, the generalization of the cross-entropy can be given as: $p\ log(q) + p\ log(1 - q)$.

Similarly, for the discriminator, the loss is given as:

$$J(D) = \underbrace{\log(D(x))}_{\text{Recognisereallmagebetter}} + \underbrace{\log(1 - D(G(z)))}_{\text{Recognisefakelmagebetter}}$$

For the generator, the loss is given as:

$$J(G) = \underbrace{\log(1 - D(G(x)))}_{\text{Optimize G that can fool the discriminator the mos}}$$

The overall architecture with the flow of loss to the generator or discriminator can be given as follows:

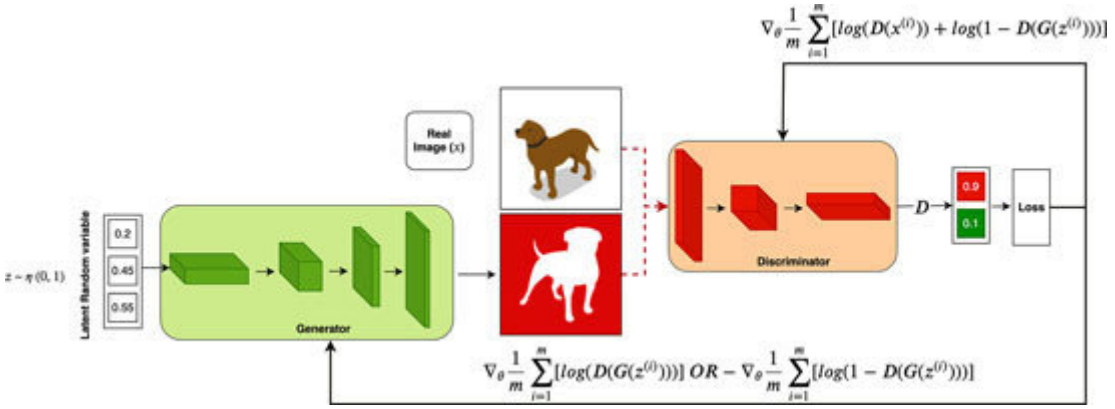


Figure 9.4: *Loss back-propagation to the discriminator and generator after a forward pass.*

In the next recipe, we will see how to train a very simple GAN model on the MNIST data.

From the network itself, it is very clear that the generators have more work than the discriminator. It is always easier to distinguish the real and fake images in the early stages of training by the discriminator. As a result, the discriminator is easily trained, and the discriminator loss quickly decreases to zero. Consequently, the gradient quickly vanishes at discriminator and makes the generator's training impossible. Also, the image generated by the generator cannot pass the discriminator, and the loss of the generator gradually increases. This phenomenon is known as the mode collapse. We will see how advance architecture is taking care of mode collapse. Due to this, it is very hard to train the GAN. Also, GANs are very sensitive to hyperparameters. While we implement the RNN by tracing the generator loss, we will see how mode collapse occurs.

Take a look at Generative Adversarial Nets:

Implementing GAN for MNIST

In the previous recipe, we saw various building blocks to construct the GAN architecture. In this recipe, we will take the MNIST dataset and design the GAN with a generator and discriminator. We will also understand how to implement the loss function and provide fake and true images to the discriminator.

You might probably know about the MNIST dataset; it is a dataset for handwritten digits from 0-9 with 60,000 training and 10, 000 test samples. Usually, GAN has convolutional layers in the architecture, but we will use only fully-connected layers in the present demonstration for simplicity.

The following steps illustrate how to generate MNIST using GAN:

The Each MNIST image is of size 28*28, as we will use fully-connected layers so that we flatten these images into shape 784.

The The generator looks like as given in the next code block. The generator takes a random vector of size 100 and has three fully-connected layers; each subsequently dilates the input shape and output shape of 784 so that it can form an

image of size (28*28). It is the image generated by the generator.

The It takes an image of size 784 and gradually shrinks it to 1 by passing it through 3 fully-connected layers. If the output probability is toward one, the input image is classified as true, else it is classified as fake.

Training The overall training process has the following steps, along with the code:

Generating random vector of size 100:

```
noise = Variable(torch.randn(images.size(0), 100).cuda())
```

Generating fake images by passing them through a generator and generating labels with all zeros for these images:

```
fake_images = generator(noise)
fake_labels = Variable(torch.zeros(images.size(0)).cuda())
```

Training discriminator using fake images along with labels and real images along with labels. After training, the discriminator provides which is a summation of the loss generated by real images and fake images. This function also provides real_score and The former indicates the prediction of the discriminator for images with original labels; ideally, this output should be near 1. On the other hand, the latter indicates the prediction of the discriminator for images with

fake labels; ideally, this output should be near 0. Monitoring real_score and fake_score provides a good idea about the convergence of the discriminator:

```
deftrain_discriminator(discriminator, images, real_labels,
fake_images, fake_labels):
discriminator.zero_grad()
outputs = discriminator(images)
real_loss = criterion(outputs, real_labels)
real_score = outputs

outputs = discriminator(fake_images)
fake_loss = criterion(outputs, fake_labels)
fake_score = outputs
d_loss = real_loss + fake_loss
d_loss.backward()
d_optimizer.step()
returnd_loss, real_score, fake_score
```

The generator generates Then-after, fake images. The label for these images is all ones (the discriminator should treat all images as real images for the loss to be 0). These fake images are passed on to the discriminator, and the output generated by the discriminator and the original labels are used for the loss calculation. Based on the images generated and provided to the discriminator and label predicted by the discriminator, the generator loss g_loss is calculated:

```
noise = Variable(torch.randn(images.size(0), 100).cuda())
fake_images = generator(noise)
outputs = discriminator(fake_images)
```

```
g_loss = train_generator(generator, outputs, real_labels)
```

The overall training process is summarized as follows:

```
for epoch in range(num_epochs):
for n, images in enumerate(train_loader(batch_size=100)):
images = Variable(images.cuda())

real_labels = Variable(torch.ones(images.size(0)).cuda())
# Sample from generator
noise = Variable(torch.randn(images.size(0), 100).cuda())
fake_images = generator(noise)
fake_labels = Variable(torch.zeros(images.size(0)).cuda())
# Train the discriminator
d_loss, real_score, fake_score = train_discriminator
(discriminator, images, real_labels, fake_images, fake_labels)
# Sample again from the generator and get output from
discriminator
noise = Variable(torch.randn(images.size(0), 100).cuda())
fake_images = generator(noise)
outputs = discriminator(fake_images)
# Train the generator
g_loss = train_generator(generator, outputs, real_labels)
```

After training for a few iterations, the following sample is generated by the generator. Further training could provide better output.

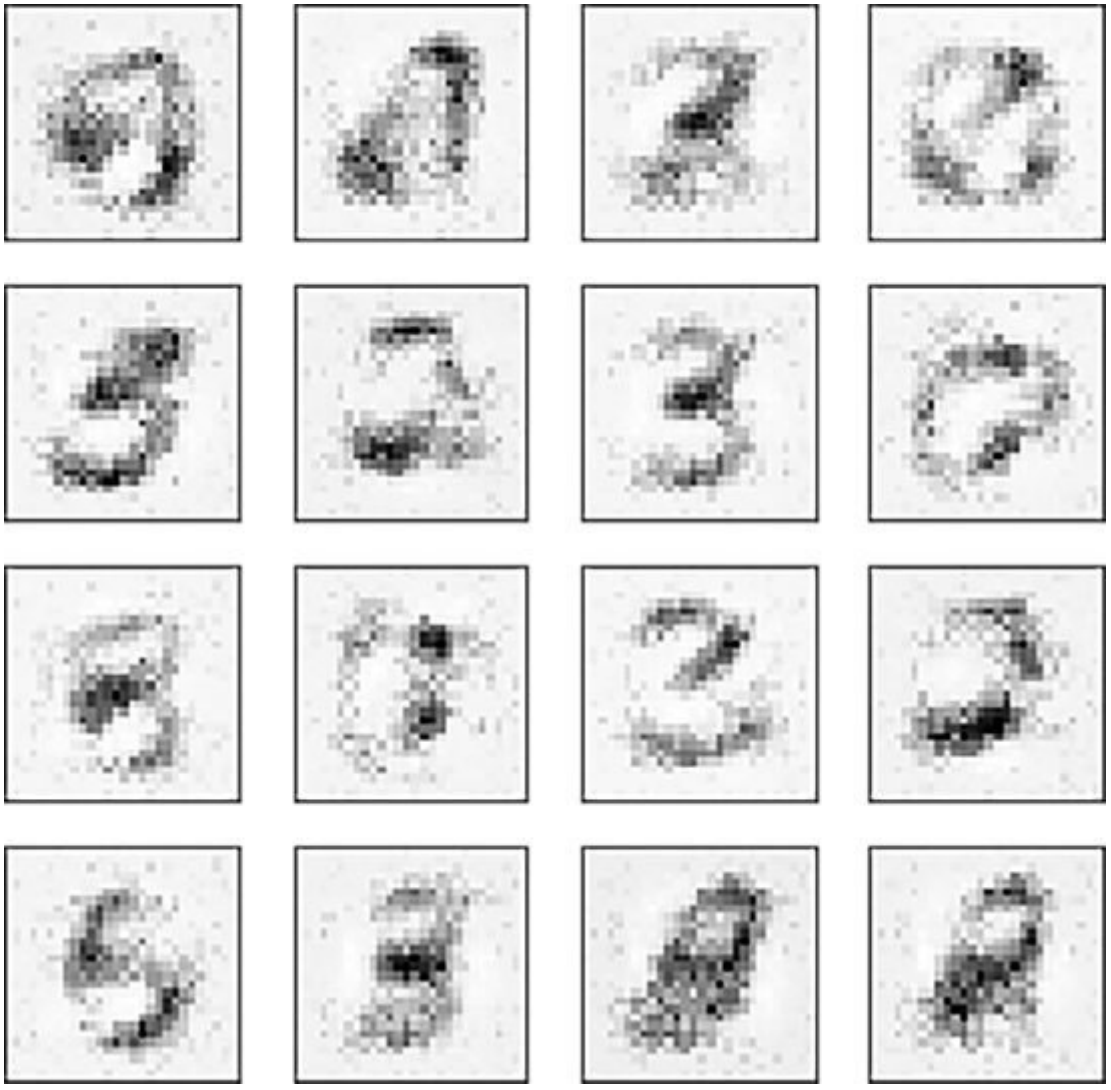


Figure 9.5: *Sample generated by the generator on the MNIST data.*

The overall implementation with supporting functions is given at

GAN is a new technique, but the interest of developers las led to many varieties of the GAN evolving within a short span. These varieties of GAN architecture are developed after modifying the basic version for particular cases. Here are a few such variations of the GAN architecture, along with their applications:

Transfering images from one domain to the other domain

Given pictures of a celebrity, it suggests the merchandise

Creating super-resolution images from low-resolution ones

Text to The image is synthesized according to given text description (we will discuss this in detail in the next recipe)

Music GANs are used for music generation

The approach of basic GAN is modified a little for any of the above-mentioned applications. The following list mentions some of the ways in which modifications can be made in the GAN architecture to achieve specific goals. Implementing GAN variation is super easy, and converging them is an art. Some of the papers that you may implement are given here:

Comparing GAN techniques for image creation and modification: *https://arxiv.org/pdf/1803.09093.pdf*

Image-to-image translation with conditional adversarial networks: *https://arxiv.org/pdf/1611.07004.pdf*

High-resolution image synthesis and semantic manipulation with conditional GANs: *https://arxiv.org/pdf/1711.11585.pdf*

Pixel-level domain transfer: https://arxiv.org/abs/1603.07442

Unpaired image-to-image translation using cycle-consistent adversarial networks: https://arxiv.org/pdf/1703.10593.pdf

The Understanding Theory behind GAN

We have already explored the basic components of the GAN, along with its implementation. GANs are based on the game theory and are very unstable. To train GAN effectively and converge such networks, you must know some of the basic concepts that form the base of the GAN.

As mentioned earlier, the GANs are unstable, so there exist many challenges in training them. Some of these challenges are listed here:

The model accuracy constantly oscillates, destabilizes but never converges.

Mode The generator collapses and produces only limited varieties of samples. As aresult, a false increase in the performance is noted, but generalization never occurs. Real-life data are multi-labels (multi-model). For example, MNIST data has ten modes (0 to 9). Two different GANs generate the following output. The upper row with one GAN model, where the generator yield all the classes whereas in the second row, is synthesizing only a few classes. It is called a model collapse, where the model shows a low error, but the generalization is compromised. This type of partial collapse is common in GAN models:

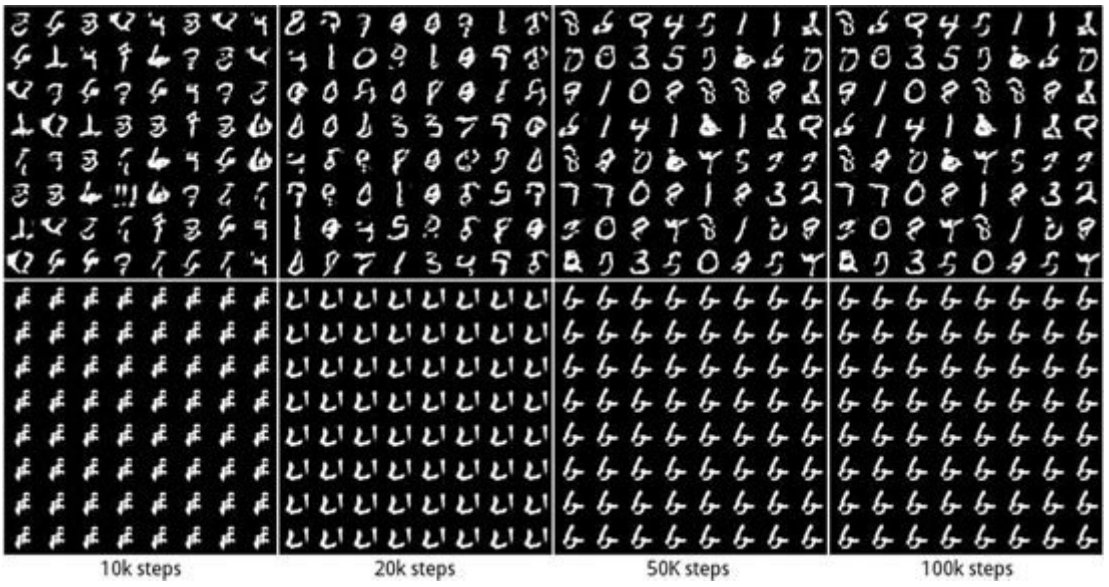


Figure 9.6: *Diagram showing the MNIST sample generated by two different GAN models.*

The upper row generated by the GAN model is without mode collapse. Another GAN model generates the lower row and it has mode collapse.

Diminished As we saw earlier, the sample generated by the generator is given to the discriminator, and the error is backpropagated. When the discriminator gets too successful, it always identifies the generated sample as wrong, but as the discriminator is right, the large error does not propagate back to the generator. The generator gradient vanishes and learns nothing, and the imbalance between the generator and discriminator causes overfitting.

Hyper-Parameter GANs are highly sensitive to hyperparameters, and when selected wrongly, such a network

never converges. To effectively deal with this instability, it is advisable to learn some of the principles that affect convergence in the GANs, and we will do just that in the next section.

The problem of instability can be understood and elevated with the help of the following topics:

Nash GAN is based on the game theory—zero-sum non-cooperative game. It means if one wins, the other must lose. The generator and discriminator want to minimize each other's actions to maximize their own. This is called the minimax game. To achieve a balance between the generator and discriminator is called Nash equilibrium:

$$\underbrace{\min_{G}}\underbrace{\max_{D}}V(D,G) = \underbrace{\log(D(x))}_{\text{Recognisereallmagebetter}} + \underbrace{\log(1-D(G(z)))}_{\text{Recognisefake Imagebetter}}$$

Both opponents are trying to undermine each other, and the Nash equilibrium occurs when one player will not change its action regardless of the other player. To understand the preceding equation, let's take a simple equation when players A and B control the values of x and y; the equation is given as follows:

$$\underbrace{\min_{G}}\underbrace{\max_{D}}V(D,G) = xy$$

Where, player A wants to minimize and player x wants to maximize the value of xy. In this case, the Nash Equilibrium

is when x=y=0, as it is the state when the opponent's action does not change.

Feature As mentioned earlier, the concept of the GAN is based on the minimax game, where two opponents try to defeat each other. In this game, the discriminator often gets too powerful and wins over the generator. To defeat the discriminator, the generator becomes too greedy. It only produces a few classes of the image very realistically so that the discriminator fails to recognize the difference, and it loses. This is the primary reason behind mode collapse. Mode collapse is caused because the generator's goal is to produce just realistic images and win over the discriminator (just to get a class right). To help with this, the cost function for the generator is changed to additionally minimize the statistical difference between a feature of generated and real images. Often, we measure L2 difference between the mean of the feature of generated images and real images. This difference is calculated in the discriminator just before a few layers of classification. Mathematically, this is represented as: – where is the feature vector extracted from the layer just before the classification layer in the discriminator. Dummy code for the feature matching GAN is given here:

```
import torch
from torch import nn
# we keep the discriminator as simple as possible
class Discriminator(nn.Module):
def __init__(self, input_size, num_features):
super().__init__()
```

```
self.features = nn.Linear(input_size, num_features)
self.classifier = nn.Sequential(
nn.Linear(num_features, 2),
nn.Sigmoid()
)
def forward(self, x):

# we return both outputs and the features
feature = self.features(x)
class_ = self.classifier(f)
return class_, feature
# criterion
feature_matching_criterion = nn.MSELoss()
fake_samples = G(noise) # generated fake data from
generator
real_samples = ... # real data
fake_prediction, fake_features = D(fake_samples)
real_prediction, real_features = D(real_samples)
# now, calculating the new objective
loss = feature_matching_criterion(fake_features, real_features)
loss.backward()
```

As with all the other networks, this does not solve the problem but stabilizes training.

When mode collapse occurs, the generator synthesizes all images that are similar in nature. To detect the model collapse and penalize the generator, the similarity between the images generated by the generator is determined. When the model collapse starts, all the image generated by the discriminator will be similar, so the discriminator can use this

score to penalize the generator if it is cheating by mode collapsing.

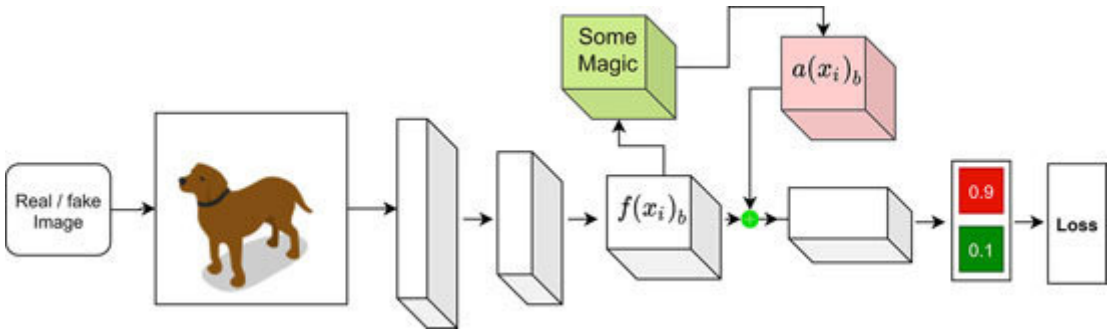


Figure 9.7: *Showing the location in the discriminator where some magic is applied to calculate the intra-batch similarity This intra-batch similarity is added as an extra feature and passed to the next layers. It helps the discriminator determine whether mode collapse has occurred.*

Minibatch discrimination is applied in the discriminator. Let's say it denotes the vector and size for input in the some of the layers. Then, this is multiplied to the tensor giving measure Then, distance is calculated between and all the other images in the batch, and the negative exponent is applied as for batch *B* is calculated as = – ∈ . Such a difference is calculated between and all other images as: = . This feature is concatenated with the real output by layer, and the rest of the discriminator works as it is. It is how the extra feature created within the discriminator allows it to identify mode collapse better. Minibatch Discrimination works better with the feature discrimination.

One-sided label Deep learning algorithm tends to observe confidence and looks at some of the key features to conclude. In GAN, this may also happen when the discriminator turns out to be dependent on some of the features, and the generator may generate only those features. This can harm the performance. To check this, we penalize the discriminator when the prediction of real images goes beyond 0.9 by setting our target level at 0.9 instead of 1.0:

```
# Use 0.9 instead of 1.0.
real_label = [[0, 0, 0, 0.9, 0, 0, 0, 0, 0, 0]] # Image with label "3"}
# predict_real_image is the logits calculated by
# the discriminator for real images.
discriminator_real_loss = tf.nn. BCELoss(labels=p, logits=predict_real_image)
```

Historical This intuition comes from the way optimizer ADADelta works— take the mean of the last n batches and then update instead of updating the weight for the current batch. Sometimes, the model oscillates around a point. According to historic averages, it can act as a damping force to converge the model. For past epoch iteration, the average of eight is taken as l2, and the cost is calculated as .

New GAN architecture often has a label attached to it, which is supposed to stabilize the training. Until now, we were giving the random vector as the input to generate the desired output. With Conditional GAN (CGAN), along with the random vector, the generator is also provided with the label

of the output to be generated. We will discuss these new techniques in detail in the next recipe.

Some of the other methods to stabilize GAN are mentioned here, along with the research papers: Avoid Sparse Gradients: ReLu, MaxPool: The stability of the GAN suffers if you have sparse gradients. Use LeakyReLu in Generator and Discriminator. For Down-sampling, use Average pooling, Conv2d + Stride. For Upsampling use PixelShuffle, ConvTranspose2D + stride. Refer to the following papers to learn how these techniques are used:

Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network: *https://arxiv.org/abs/1609.05158*

Add noise to inputs, decay over time: Add some artificial noise to discriminator:

Instance noise: A trick for stabilizing GAN training: *https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/*

Improved techniques for training

Generating an Image from the Description

In the previous recipe, we generated MNIST digit by injecting random noise in the generator. That was a very primitive architecture with a generator and discriminator. A human can infer the image generated by the GAN. The output of such a network is not controllable, as we don't have any mapping between the random vector and the generated output. So, the concept of conditional GAN was proposed. In this recipe, we will learn how the architecture of the conditional GAN is developed, taking inspiration from the thought process. This chapter will also help you understand the process of inventing better architectures.

Conditional GAN means we can make the network conditional by applying the new condition to the generator and discriminator, and the network will generate the same to decrease the loss. This way, we have control over what a network should generate. The condition can be a text description or a class label. In this recipe, we will discuss the GAN-CLS algorithm. The network diagram for the GAN-CLS is as follows:

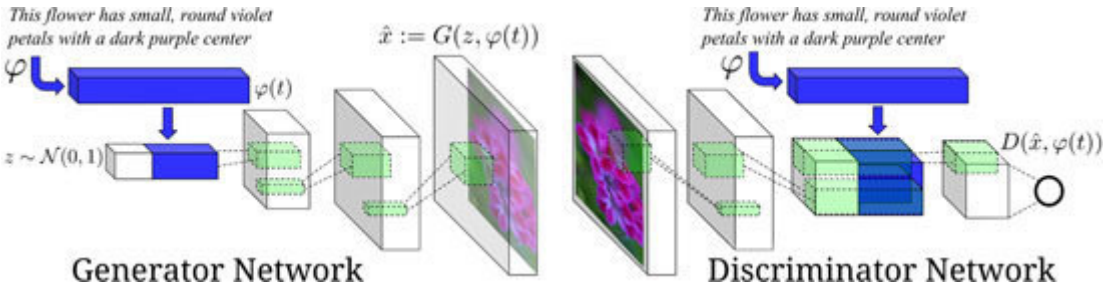


Figure 9.8: *GAN-CLS schematic diagram.*

The network is based on the **Deep Convolution Generative Adversarial Network** The additional portion to condition this network based on text is added to it. The text features are encoded by the hybrid character level convolution neural network. Such encoded text features are added to the generator input and discriminator output. Along with the random vector (latent random variable), the text embedding variable is attached. This will help the generator to generate the image according to the description. The generated image, referred to as fake, is passed to the discriminator, which should recognize it as fake.

If the discriminator recognizes the generated image as fake, the loss passes back to the generator, and the generator learns to generate better.

If the discriminator recognizes it as the real image, the generator is doing well.

This is how the generator is being trained.

The discriminator is also trained to recognize real and fake images, and to do so, the generator is given three types of samples. The type of sample and the expected prediction from the discriminator are mentioned here:

here:
here: here: here: here: here: here: here: here: here: here:
here: here: here: here: here: here: here: here: here: here:
here: here: here: here: here: here: here: here: here: here: here: here: here: here: here: here:

Table 9.1

A Binary cross-entropy loss is calculated between the expected and predicted labels backpropagated to the generator and discriminator. A code snippet for all the vital components of the GAN-CLS is given in the next section, along with a detailed explanation.

In GAN-CLS, the overall training paradigm is the same as the one we used to train GAN for MNIST. The only difference is in the discriminator and the loss function. In the next section, we will understand GAN-CLS with the help of the following sub-topics:

Understanding Such textual features *t* are appended to the random noise *z* to be added to the generator network. It

ensures that the network is preconditioned on what we want to generate. As given in the network, the self.projection variable is the small sequential block. The network architecture of the generator in GAN-CLS using PyTorch is given here. The generator is mainly made up of convolution layers with upsampling and batch normalization operations. The generator for the GAN-CLS is no different than a typical generator in the GAN, except the input where the embedding vector of the text is added to the random vector. The self.projection variable has the embedding vector, and variable z has a random value in the following code:

```
class generator(nn.Module):
def __init__(self):
super(generator, self).__init__()

self.image_size = 64
self.num_channels = 3
self.noise_dim = 100
self.embed_dim = 1024
self.projected_embed_dim = 128
self.latent_dim = self.noise_dim + self.projected_embed_dim
self.ngf = 64
self.projection = nn.Sequential(
nn.Linear(in_features=self.embed_dim,
out_features=self.projected_embed_dim),
nn.BatchNorm1d(num_features=self.projected_embed_dim),
nn.LeakyReLU(negative_slope=0.2, inplace=True)
)
self.netG = nn.Sequential(
nn.ConvTranspose2d(self.latent_dim, self.ngf * 8, 4, 1, 0,
bias=False),
```

```
nn.BatchNorm2d(self.ngf * 8),
nn.ReLU(True),
# state size. (ngf*8) x 4 x 4
nn.ConvTranspose2d(self.ngf * 8, self.ngf * 4, 4, 2, 1,
bias=False),
nn.BatchNorm2d(self.ngf * 4),
nn.ReLU(True),
# state size. (ngf*4) x 8 x 8
nn.ConvTranspose2d(self.ngf * 4, self.ngf * 2, 4, 2, 1,
bias=False),
nn.BatchNorm2d(self.ngf * 2),


nn.ReLU(True),
# state size. (ngf*2) x 16 x 16
nn.ConvTranspose2d(self.ngf * 2,self.ngf, 4, 2, 1, bias=False),
nn.BatchNorm2d(self.ngf),
nn.ReLU(True),
# state size. (ngf) x 32 x 32
nn.ConvTranspose2d(self.ngf, self.num_channels, 4, 2, 1,
bias=False),
nn.Tanh()
# state size. (num_channels) x 64 x 64
)
def forward(self, embed_vector, z):
projected_embed =
self.projection(embed_vector).unsqueeze(2).unsqueeze(3)
latent_vector = torch.cat([projected_embed, z], 1)
output = self.netG(latent_vector)
return output
```

Understanding Textual features are also appended to the final output generated by convolutional layers of the discriminator,

and then a final convolutional transformation is applied to predict it as fake or real. The network architecture of discriminator in GAN-CLS using PyTorch is given here. The convolution, batch normalization, and pooling operations are present in the discriminator as well. Just like the generator, the sentence embedding vector is given in the discriminator so that it can verify that the image is real/fake with respect to the description:

```
class discriminator(nn.Module):
def __init__(self):
super(discriminator, self).__init__()
self.image_size = 64
self.num_channels = 3
self.embed_dim = 1024
self.projected_embed_dim = 128
self.ndf = 64
self.B_dim = 128
self.C_dim = 16
self.netD_1 = nn.Sequential(
# input is (nc) x 64 x 64
nn.Conv2d(self.num_channels, self.ndf, 4, 2, 1, bias=False),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf) x 32 x 32
nn.Conv2d(self.ndf, self.ndf * 2, 4, 2, 1, bias=False),
nn.BatchNorm2d(self.ndf * 2),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf*2) x 16 x 16
nn.Conv2d(self.ndf * 2, self.ndf * 4, 4, 2, 1, bias=False),
nn.BatchNorm2d(self.ndf * 4),
nn.LeakyReLU(0.2, inplace=True),
```

```
# state size. (ndf*4) x 8 x 8
nn.Conv2d(self.ndf * 4, self.ndf * 8, 4, 2, 1, bias=False),
nn.BatchNorm2d(self.ndf * 8),
nn.LeakyReLU(0.2, inplace=True),)


self.projector = Concat_embed(self.embed_dim,
self.projected_embed_dim)
self.netD_2 = nn.Sequential(
# state size. (ndf*8) x 4 x 4
nn.Conv2d(self.ndf * 8 + self.projected_embed_dim, 1, 4, 1, 0,
bias=False),
nn.Sigmoid()
)
def forward(self, inp, embed):
x_intermediate = self.netD_1(inp)
x = self.projector(x_intermediate, embed)
x = self.netD_2(x)
return x.view(-1, 1).squeeze(1), x_intermediate
```

The generator It is calculated on the basis of the sample generated by the generator and identified as fake by the discriminator. The generator gets no loss if the sample it generated is flagged as real by the discriminator; else, the generator will be panelized:

```
class generator_loss(torch.nn.Module):
def __init__(self):
super(generator_loss, self).__init__()
self.estimator = nn.BCELoss()
def forward(self, fake):
batch_size = fake.size()[0]
```

```
self.labels = Variable(torch.FloatTensor(batch_size).cuda().fill_(1))
return self.estimator(fake, self.labels)
```

The discriminator The discriminator calculates three types of losses:

Real image with Right text - A

Real image with Wrong text - B

Fake image (generated) with Right text - B

These three images are combined in a different proposition and given back to the discriminator. Here, we are using Binary Cross Entropy as the loss function. In the following implementation, these losses are combined as d_loss = A + 0.5*B +C:

```
class discriminator_loss(torch.nn.Module):
def __init__(self):
super(discriminator_loss, self).__init__()
self.estimator = nn.BCELoss()
def forward(self, real, wrong, fake):
batch_size = real.size()[0]
self.real_labels =
Variable(torch.FloatTensor(batch_size).cuda().fill_(1))
self.fake_labels =
Variable(torch.FloatTensor(batch_size).cuda().fill_(0))
```

return self.estimator(real, self.real_labels) + 0.5 * (self.estimator(wrong, self.fake_labels) + self.estimator(fake, self.fake_labels))

The overall implementation of generative adversarial text-to-image synthesis paper is provided at Understanding the given description can make it easier for you to understand the GitHub implementation.

The preceding implementation can be applied to the birds, flowers dataset. Also, LeakyReLu activation is used with a purpose in this implementation. Leaky ReLU stabilizes GAN; you can replace this activation and try any other activation function as a part of the experiment.

Take a look at the following links:

Generative adversarial text to image synthesis: *https://arxiv.org/pdf/1605.05396.pdf*

Generate the corresponding image from text description using modified GAN-CLS algorithm: *https://arxiv.org/pdf/1806.11302.pdf*

Conclusion

Generative networks are pioneering the next wave in AI and deep learning. Although there is very little to talk about GAN and text, it's certainly evolving at a very fast pace. In this chapter, we understood the basic theory of the GAN and implemented a very small model with the MNIST dataset. Also, we understood the instability associated with the GAN due to its minimax optimization nature and possible workarounds to converge such networks. Later, we saw how to generate an image based on the description, which is also an example of how convolutional and recurrent networks are utilized in combination to accomplish a task.

In the next chapter, we will cover the techniques of speech processing.

CHAPTER 10

Techniques of Speech Processing

In this chapter, we will explore an important and new area where deep learning is used extensively—speech processing. Speech is also related to Natural Language Processing, as it has a temporal domain. In speech, everything is relative to the text, and it depends on the context. As we face a problem with processing text with different languages, these problems reach the next level with speech processing. Speech faces challenges with respect to the accent and language. Despite these challenges, the market is flooded with personal assistants like Siri, Alexa, Google Home, and Cortana. These devices are getting better each day with newer techniques. This chapter will introduce you to the units of sound, techniques used to read sound, and feature extraction. This chapter will use the Urban sounds dataset, wherein various sounds we hear in the day to day life are classified. Then, we will understand how DeepSpeech and DeepVoice are developed and used for speech to text and text to speech applications.

Structure

The following topics will be covered in this chapter:

Learning about Docker

Getting to know Phonemes

Training a small network

Understanding speech to text

Understanding text to speech

Objective

Learning the advanced techniques of speech processing

Understanding how audio signals are captured and stored

Spoken Digit Recognition were end to end model is discussed in details

Getting to know advance frameworks like DeepSpeech and DeepVoice

Technical Requirements

The code for this chapter can be found in the Ch10 folder at GitHub repository To understand this chapter, you require basic knowledge of the following Python packages:

Librosa

NumPy

Pandas

Matplotlib

Tqdm

Ipython

Scikit_learn

In this chapter, we will use Docker containers, and preliminary information about Docker containers is given here. You can install these requirements by installing all the packages listed in requirements.txt by simply issuing pip install -r

Learning about Docker

Docker is a computer software that works by operating-system-level virtualization. It runs a different container, and each container is isolated from others existing in the system. Each container can be a different operating system with different packages, and each Docker container serves as an isolated system with its network and security. The container is created from images of a different kind. An image can be created by anyone and can be used by anyone if it is open-sourced. Images are often made by combining and modifying standard images. An image may have custom hardware support like GPU or may have custom functionality, including the operating system packaged into one system. Nvidia Docker is another variant of Docker that has Nvidia GPU support. Going forward, we will use Nvidia Docker to have GPU support and everything will be built into one package. Use the following commands to install Nvidia Docker:

curl -s -L https://nvidia.github.io/nvidia-Docker/gpgkey | sudo apt-key add - curl -s -L https://nvidia.github.io/nvidia-Docker/ubuntu16.04/amd64/nvidia-Docker.list | sudo tee /etc/apt/sources.list.d/nvidia-Docker.list
sudo apt-get update
sudo apt-get install -y nvidia-Docker
sudopkill -SIGHUP Dockerd # Restart Docker Engine

sudonvidia-Docker run --rmnvidia/cudanvidia-smi # finally run nvidia-smi in the same container

Alternatively, you can install Nvidia-Docker by following the guidelines at Fetching Nvidia-Docker container:

sudonvidia-Docker pull PyTorch/PyTorch

After a successful pull, you can see all the Docker images present in your system as:

sudoDocker images

The Docker container can then be run with the following commands:

Sudo nvidia-Docker run -it --rm -v subdirectory>:path> -p::

Mounts a local subsystem to a location inside the Docker container. One can pass files to Docker containers, but it will unnecessarily make the container bulkier.

Runs the Docker in an interactive mode.

Removes the Docker after you exit from it. It avoids dangling Dockers and prevents them from using memory.

Binds a system port to Docker container port. It is particularly useful for running Jupyter notebooks inside the Docker.

After this step, you can directly run PyTorch with GPU support.

Getting to Know Phonemes

Sound is an important part of our life—a means of communication, a means of medical help, and plants can also perceive it. All animals perceive sounds through their ears and other auditory organs. Sound travels through the air by making oscillations short of vibrations. This vibration can only be perceived through the ears or can be captured into a machine-readable format by a device called mic. Sound so captured can be stored in a variety of file formats like WAV, MIDI, or MP3. There are many such open sources, and proprietary file format exists to store sound in machine-readable form.

In the infant stage, our brain perceives sound and the neurons in the brain are well-trained to decode information through these signals. You may optionally go through this research paper, wherein a scientist from the Massachusetts Institute of Technology rewires the neurons and visual projections routed to the auditory pathway. Still, the experimental animal was able to see. This experiment tells that our neuron is learning, and similarly, we can make artificial neural networks to learn such information. In today's world, speech analytics is actively used in several areas:

Indexing and recommending music according to the genre and similar content

Similarity search on the audio files - Shazam (An Apple subsidiary) is a wonderful example for the same

Speed processing and generating artificial voice

For surveillance purposes

In the upcoming sections, we will see how to practically implement and use text to speech and speech to text in project pipelines. Before getting into the practical stuff, let's see how to extract features from speech signals to insert them into the pipeline later. Audio has three-dimensional representation: time, amplitude, and frequency:

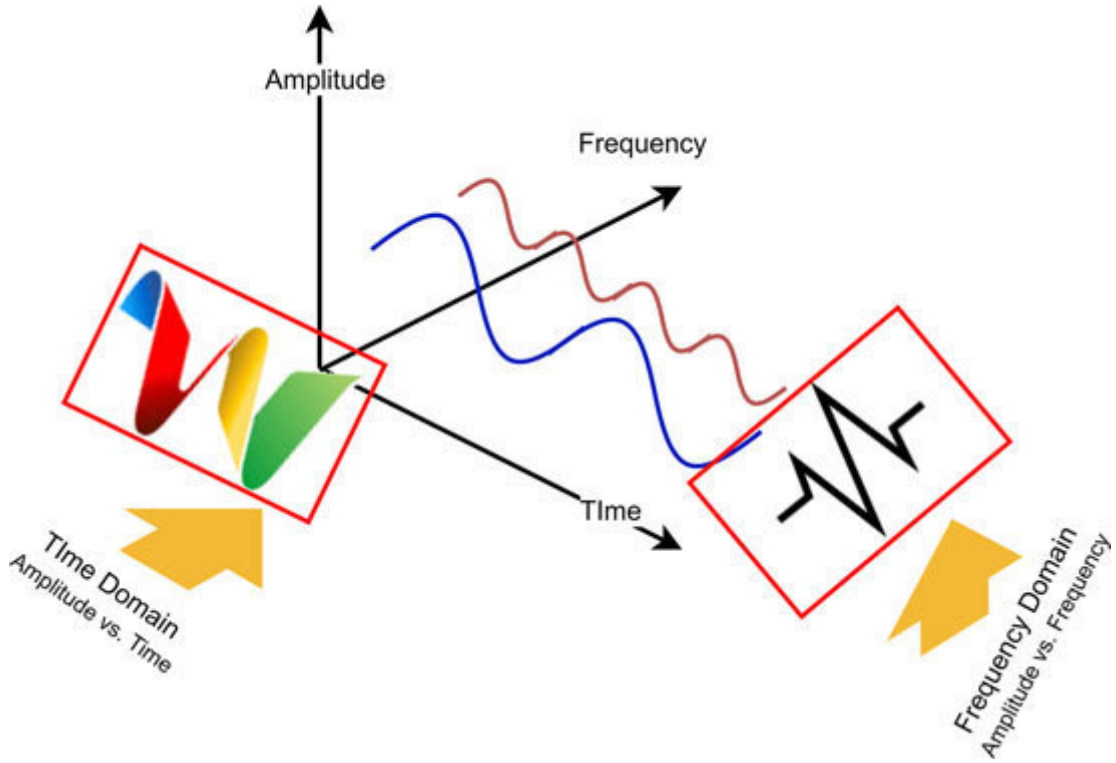


Figure 10.1: *Three - components of the sound wave.*

The amplitude represents the power in the sound wave. High amplitude sound will be loud, and low amplitude sound will be quiet. Low-frequency sound might be a low rumble, while a high-frequency sound might be more like a sizzle. We have a lot of options to read and manipulate speech data. We will use librosa for analyzing and extracting the audio features.
For playing audio in the Ipython notebook, we will use which can be installed using pip install librosa. Alternatively, if you can install librosa to anaconda using conda install -c conda-forge You can install pyAudio as pip installPyAudio.

This section will cover the most basic aspects of speech processing, including loading audio, playing audio, visualizing signals, and feature extraction.

Loading an Audio File

The audio file can be loaded with the load function. After loading a file, it is decoded into 1-dimensional time series. The sampling rate is 22KHz by default:

```
importlibrosa
audio_path = 'audio-path'
time_series, sampling_rate = librosa.load(audio_path)
print(type(time_series), type(sampling_rate))
```

The sampling rate can be changed as follows:

```
librosa.load(audio_path, sampling_rate = 44100)
```

Sampling can be disabled by specifying the parameter sr = In this case, the entire signal will be decoded to output and will take long to be processed using a machine learning pipeline:

```
librosa.load(audio_path, sampling_rate = none)
```

Playing an Audio File

Throughout this chapter, we will often conduct our experiments using the Ipython notebook. So, we will use the pyAudio package to play audio from the Ipython notebook directly:

```
importIPython.display as ipd
ipd.Audio(audio_path)
```

It will open another window to play sound directly, and it has a simple interface with a few buttons.

Visualizing the Signals

To display or visualize the audio signal, you can use the function that supports visualization in the form of wave plot, spectrogram, and colormap. The spectrogram is important to plot, as it shows the amplitude and frequency of the audio at a given time. Amplitude and frequency are important features of the audio signal. The waveform is used to plot amplitude versus time, wherein the is amplitude and the X-axis is time. The following spectrogram and waveform are for an example audio file:

```
plt.figure(figsize=(14, 5))
librosa.display.waveplot(time_series, sampling_rate)
```

A wavelet plot looks like as given below:

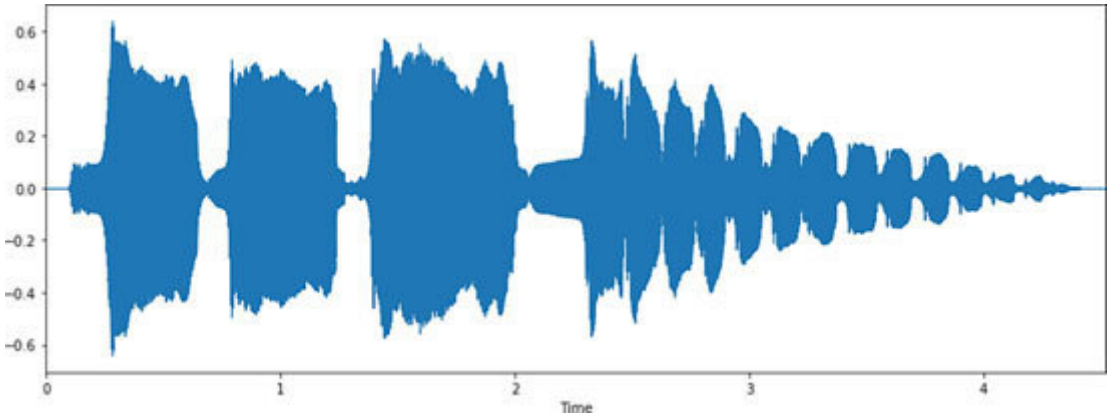


Figure 10.2: *Wavelet*

A spectrogram is the graphical representation of the spectrum of the frequency of sounds. Here, you can easily see the change in frequency with respect to time:

```
time_series_shift = librosa.stft(time_series)
Xdb = librosa.amplitude_to_db(abs(time_series_shift))
plt.figure(figsize=(14, 5))
librosa.display.specshow(Xdb, sr=sampling_rate, x_axis='time',
y_axis='hz')

plt.colorbar()
```

The spectrogram is as follows:

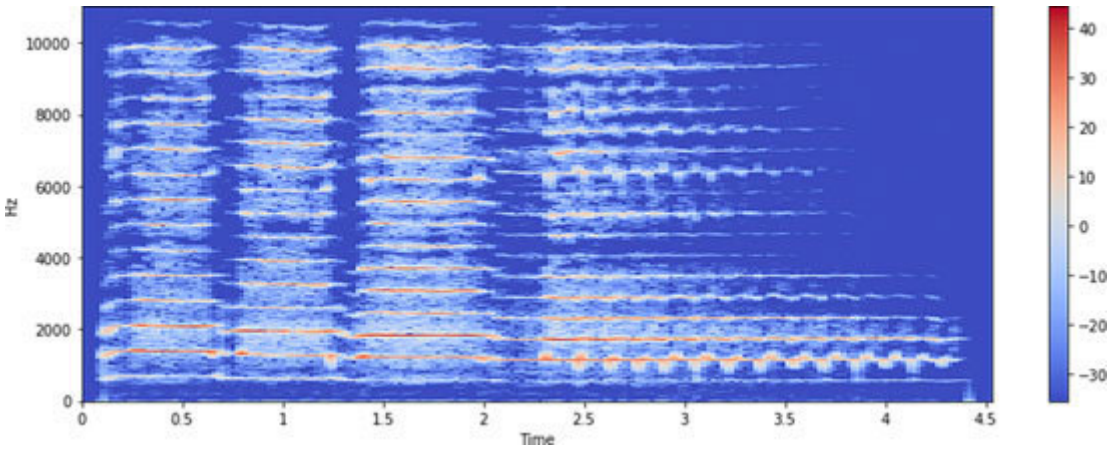


Figure 10.3: *Spectrogram*

Here, STFT is the short-term Fourier transform that converts signals such that we can know the amplitude of a given frequency at a given time.

Feature Extraction

Various feature extraction techniques are applied to audio data before pushing it into a machine learning pipeline. Some of these techniques are as listed:

MFCC — Mel-Frequency Cepstral Coefficients

Spectral Centroid

Spectral Rolloff

MFCC — Mel-Frequency Cepstral Coefficients

MFCC is the most important feature while working with audio data. MFCC of the signal is a small set of feature that concisely describes the overall shape of the spectrum. You can calculate MFCC using librosa, as follows:

```
mfccs = librosa.feature.mfcc(time_series, sampling_rate)
print("MFCC Shape :",mfccs.shape)
#Displaying the MFCCs:
librosa.display.specshow(mfccs, sr=sampling_rate, x_axis='time')
```

The first dimension represents the number of MFCC, and the second represents the number of such frames available:

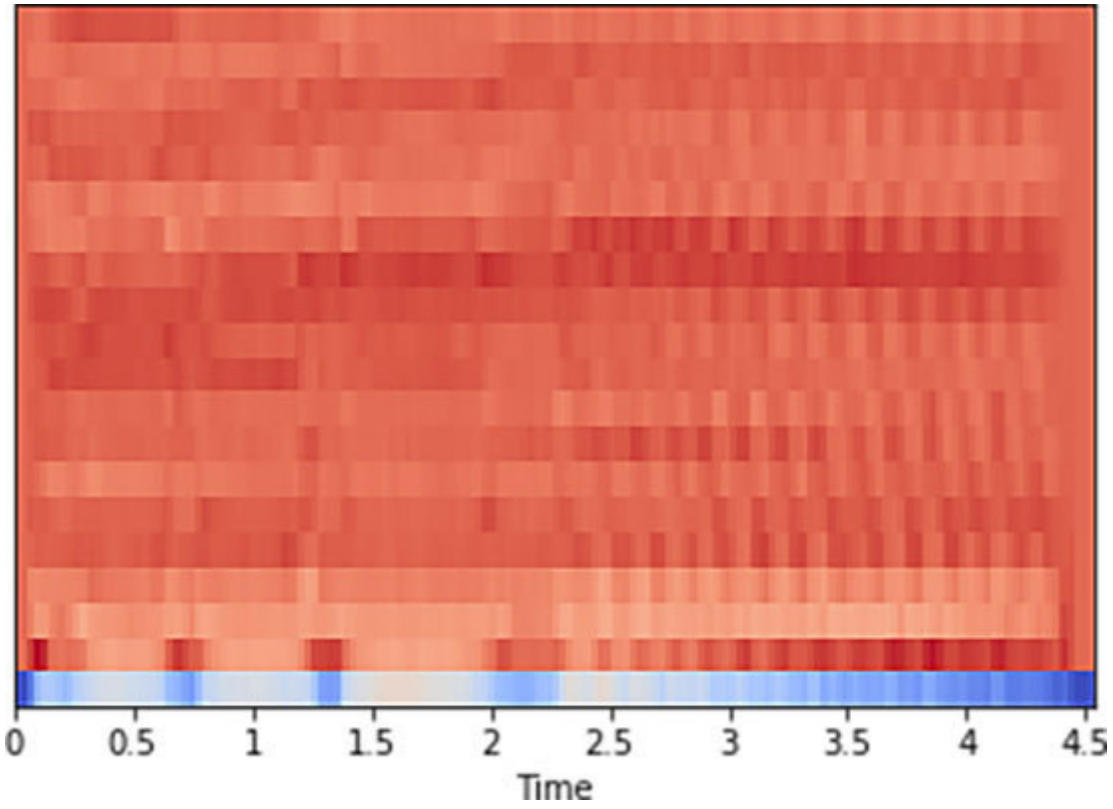


Figure 10.4: *MFCC*

Spectral Centroid

It is about finding the center of the mass of the frequency distribution in the given audio. If the audio ends with a higher frequency, the spectrum will shift high toward the end. If the audio is uniform throughout, the spectral centroid will be toward the center. Spectral centroid can be calculated as follows:

```
spectral_centroids = librosa.feature.spectral_centroid(time_series, sampling_rate)[0]
spectral_centroids.shape
# Computing the time variable for visualization
frames = range(len(spectral_centroids))
time_frame = librosa.frames_to_time(frames)
# Normalising the spectral centroid for visualisation
def normalize(time_series, axis=0):
return sklearn.preprocessing.minmax_scale(time_series, axis=axis)
#Plotting the Spectral Centroid along the waveform
librosa.display.waveplot(time_series, sr=sampling_rate, alpha=0.4)
plt.plot(time_frame, normalize(spectral_centroids), color='green')
Frames_to_time converts frame to time.time[i] ==
```

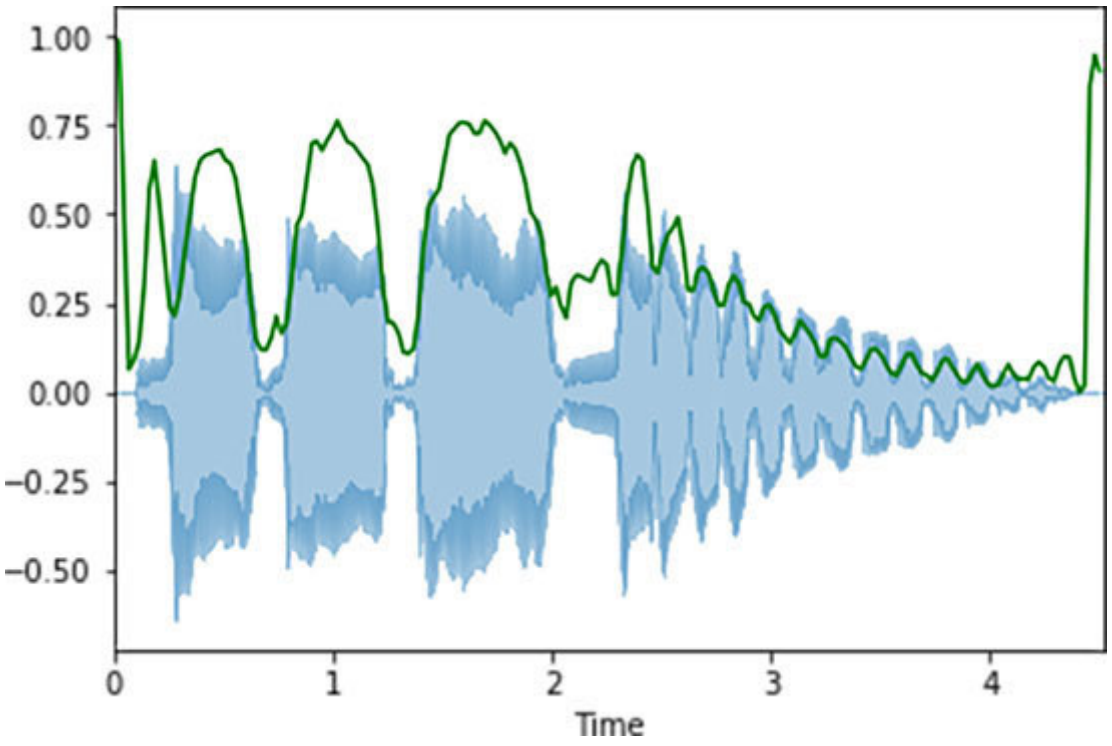


Figure 10.5: *Spectral centroid*

Spectral Rolloff

Spectral rolloff where the frequency lower than the given percentage of the average frequency lies. If we define cutoff = 85%, only the frequency lower than 85% is provided:

```
spectral_rolloff = librosa.feature.spectral_rolloff(time_series, sampling_rate)[0]
frames = range(len(spectral_rolloff))
time_frame = librosa.frames_to_time(frames)
librosa.display.waveplot(time_series, sr=sampling_rate, alpha=0.4)
plt.plot(time_frame, normalize(spectral_rolloff), color='red')
```

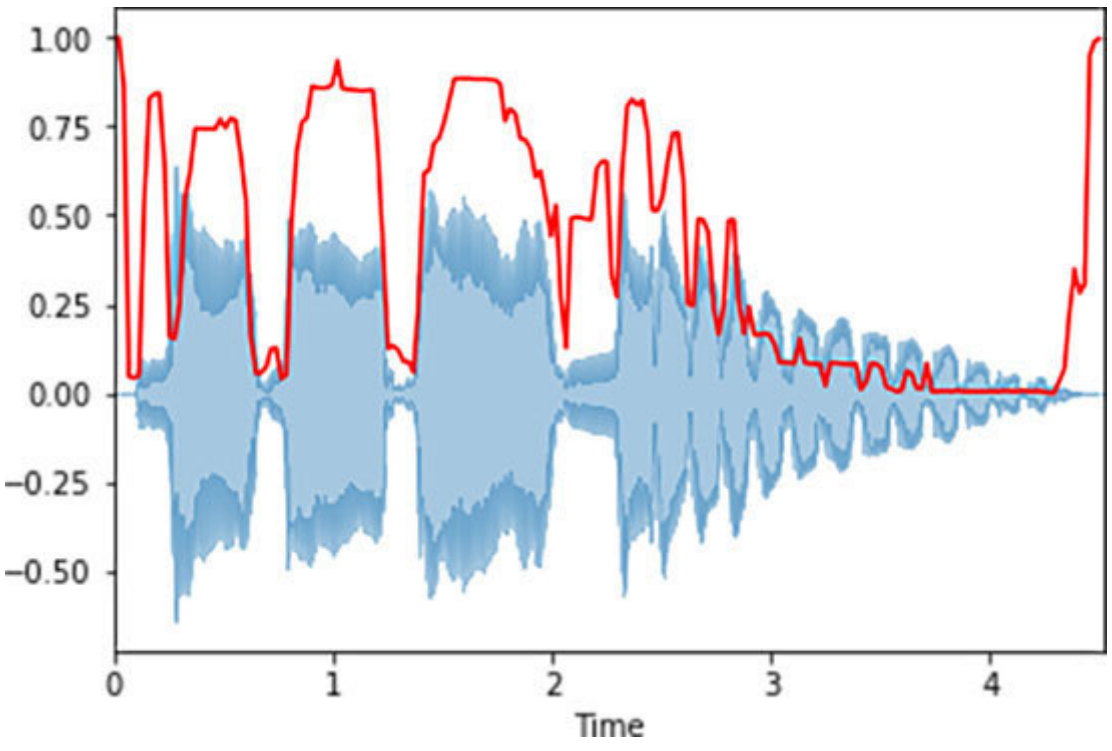


Figure 10.6: *Spectral Rolloff*

Librosa extracts many more features than the one discussed here. These include chromastft, Constant Q cromagram, Croma energy normalize - A chroma variant, RMS value for each frame, Spectral Bandwidth, Spectral Contrast, Spectral Flatness, Nth order polynomial to the column of the spectrogram, Zero Crossing Rate, and Tempogram. You can refer to the Librosa documentation to explore these features in detail. I have provided the code at where you can reproduce the implementation described earlier.

Refer to Librosa documentation: *https://librosa.github.io/librosa/*

Training a Small Network

We will be use the UrbanSound8K dataset to demonstrate how speech recognition can be done and the components usually required in such pipelines.

The UrbanSound8K dataset has 8732 labeled sound recordings of 10 classes, namely, and These files are in the .wav format. The UrbanSound8K dataset is available at Ensure that you download this dataset while you run this recipe's implementation.

The speech recognition pipeline will have the following steps:

Feature extraction

Constructing CNN model

Training and estimating performance on the test set

Let's learn each one by one.

Feature Extraction

Various features are extracted with the help of the Librosa library. These features are discussed in detail in the previous recipe.

Melspectrogram: Compute a mel-scaled spectrogram

MFCC (Mel-frequency cepstral coefficients)

Compute a chromagram from a waveform or power spectrogram

Constant-Q chromagram

Computes the chroma variant “Chroma Energy Normalized” (CENS)

All these features are then stacked, and the final feature shape is [m, 40, 5, 1]. This shape is compatible with CNN layers that we will use in our model.

Constructing the CNN Model

Model is a very simple convolutional network with various layers like convolution 2D, batch normalization, maxpooling, linear/dense layers along with the Relu activation function. The model accepts shape 40, 5, 1] as produced by the data loader, where is the batch size. Two convolutional transformations with intermediate batch normalization and ReLu activation are applied to it. Eventually, the final shape is converged in to shape 10], where 10 is the number of class, and *m* is the batch size. This model is too simple to produce exceptionable accuracy, but it will provide an idea of how voice recognition pipelines are designed:

```
class simple_network(nn.Module):
def __init__(self):
super(simple_network, self).__init__()
self.conv1 = nn.Conv2d(in_channels=40, out_channels=64,
kernel_size=3, padding=1,stride=1)
self.bn1 = nn.BatchNorm2d(64)
self.relu = nn.ReLU()
self.drop= nn.Dropout(0.2)
self.conv2 = nn.Conv2d(in_channels=64, out_channels=128,
kernel_size=3, padding=1, stride=1)
self.bn2 = nn.BatchNorm2d(128)
self.maxpool = nn.MaxPool2d(kernel_size=2, padding=1)
self.dense1 = nn.Linear(in_features=128*3, out_features=128*2)
```

```
self.dense2 = nn.Linear(in_features=128*2, out_features=10)
def forward(self, input_):
conv1_out = self.conv1(input_)


conv1_out = self.bn1(conv1_out)
relu_applied_1 = self.relu(conv1_out)
maxpol_out = self.maxpool(relu_applied_1)
conv_2_out = self.conv2(maxpol_out)
conv_2_out = self.bn2(conv_2_out)
relu_applied_2 = self.relu(conv_2_out)
drop_applied = self.drop(relu_applied_2)
dense1_out =
self.dense1(drop_applied.view(drop_applied.shape[0],drop_applied
.shape[1]*drop_applied.shape[2]))
relu_applied_3 = self.relu(dense1_out)
drop_applied = self.drop(relu_applied_3)
dense2_out = self.dense2(drop_applied)
relu_applied_4 = self.relu(dense2_out)
drop_applied = self.drop(relu_applied_4)
return torch.softmax(drop_applied, dim =1)
```

Training and Estimating Performance on the Test Set

Training is done using Adam optimizer and the binary cross-entropy loss function. The convergence of the network on the test dataset and its performance on test data as noted by TensorBoard is plotted as follows:

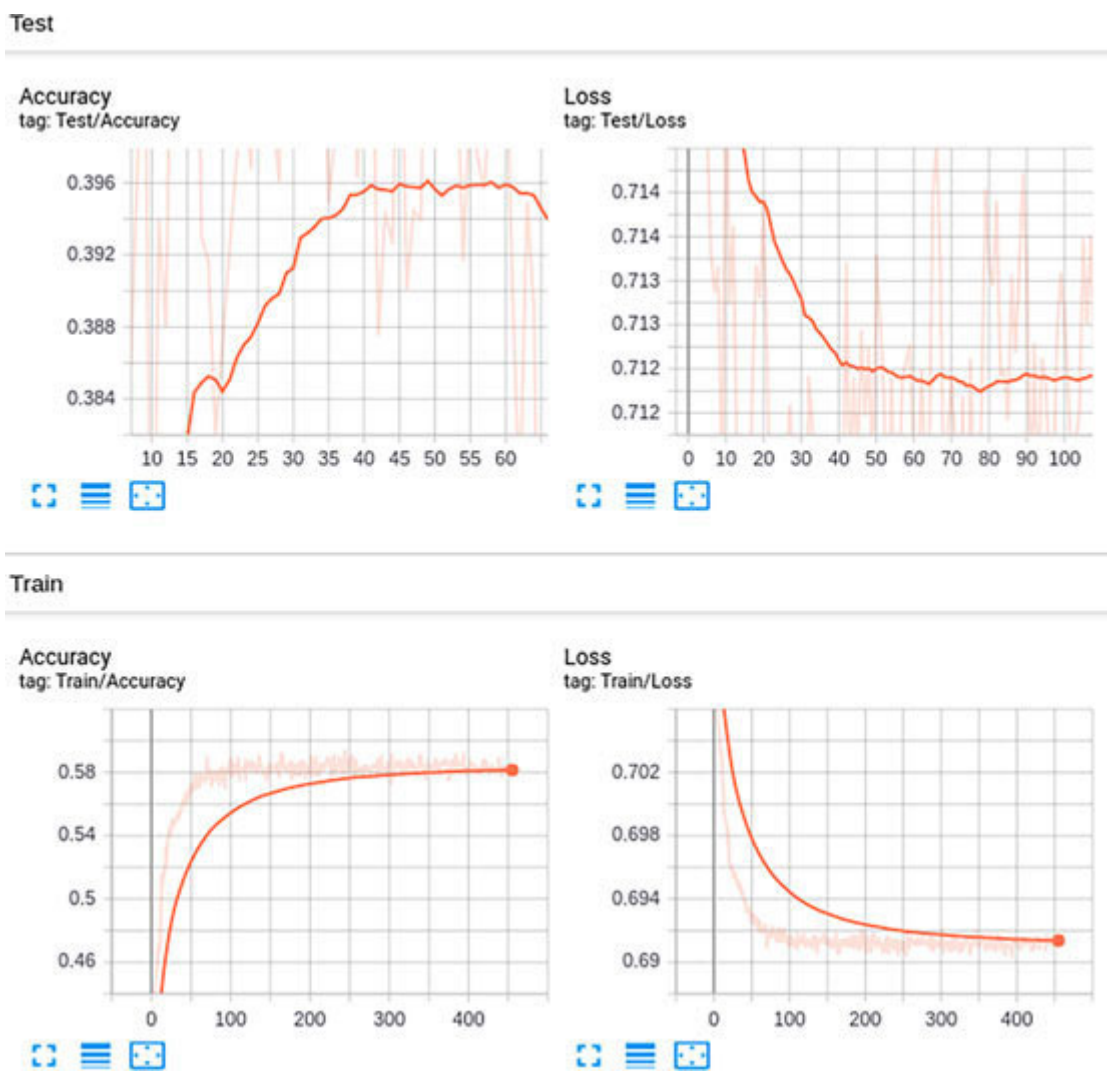


Figure 10.7: *Performance of the network on the UrbanSound8K dataset.*

The performance is not up to the mark, but the network is learning, and changing architecture and parameters may provide better accuracy.

Well, this was a very basic pipeline with a very basic feature set and a basic CNN model. Here, the shape of the input data was 40, 5, 1], to make it compatible with the CNN layer. You can reshape it to 40, 5] to make it compatible with the RNN layer. In the modified shape, there are 40-time steps, and each one has five features. Construct a network with RNN layers and see if you get significant improvement in the test accuracies.

Speech commands: A public dataset for single-word speech recognition dataset. This dataset has 10,000 train and 1,000 test recording. The output has ten classes 0-9, and 997 speakers produced this dataset. This dataset is freely available at Try with this dataset with the same or different features and models to reinforce your skills in the speech processing tasks.

Understanding Speech to Text

Speech to text is an interesting area of research; it has a unique end application. Some of the real-world applications are listed here:

Save time with automatic transcription

No need to wait for the customer care executive to receive your call; just leave your note

Write important emails and notes on the fly with voice commands

Well-known state-of-the-art text to speech applications rely on several handcrafted features and often don't work end to end. Some systems are very good in some areas, but porting them to another area might require complete rework. In this recipe, we will learn about an end-to-end automated speech to text implementation. DeepSpeech was released by the Baidu research lab in 2014. An updated version of DeepSpeech, named DeepSpeech-2, was released in 2015. Both the architecture allows end-to-end training of neural network architecture capable of producing state-of-the-art results.

We will start by understanding the model behind DeepSpeech; the DeepSpeech model has RNN components. In this training set, signal will be treated as utterance x and labels The entire training sample can be mathematically represented as X = Spectrogram is used as the feature of the model, where T are the total equally divided features present in the form of the time slice It can be mathematically represented as t = 1, ..., In a spectrogram for any point, p represents the power of the frequency in an audio frame at time The next schematic diagram explains the feature for DeepSpeech. The end goal is to predict any character given the time slice The probability of predicting a character given the time slice is given by = This probability is calculated by optimizing the neural network, which has five hidden layers; all of them are RNN layers. The first three layers are not recurrent, and for any layer the output is denoted as At the very first layer, the input is provided, which is also denoted as For the first three layers, all the recurrent layers operate independently. For any time slice in the spectrogram at time the output of the non-RNN unit is calculated as follows:

=

Where,

= weight for the given layer

= bias for the given layer

= input from the previously hidden layer

g = Clipped ReLu activation can be given by

The fourth layer is the bidirectional recurrent layer, which provides two outputs: a forward state and a backward state

$$= -$$

$$= -$$

The forward layer takes the utterances from t = 1 to t = whereas the backward layer takes utterances from t = T to t = 1.

Finally, Softmax is applied to the final output = + It shows the probability of the presence of the given character at a given utterance. After calculating the probability of the given character, we calculate the CTC loss on top of this output to measure the error in the prediction. The overall network architecture is as follows:

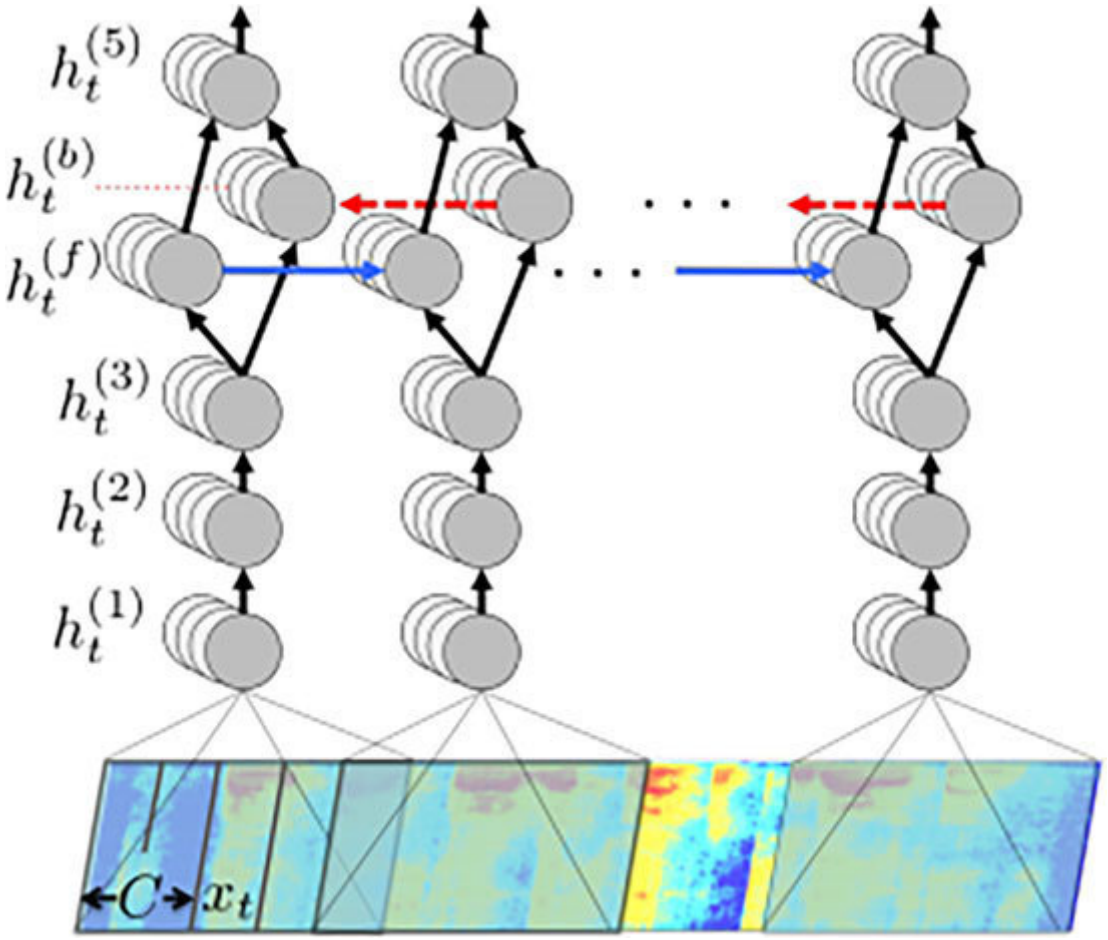


Figure 10.8: *Network architecture for DeepSpeech - 1*
Source: *https://arxiv.org/pdf/1412.5567.pdf*

The network is trained with stochastic gradient descent and keeping momentum = 0.99, decreasing the learning rate by a constant factor. Dropout is used for regularizing the learning. A year after this model was released, the Baidu research optimized it further and proposed a newer model that was released as DeepSpeech - 2. The newer model has several layers. The initial convolution layers followed by one or more convolution layers and then a few fully-connected layers. CTC loss is used as a loss measure. The schematic diagram of the network architecture is as follows:

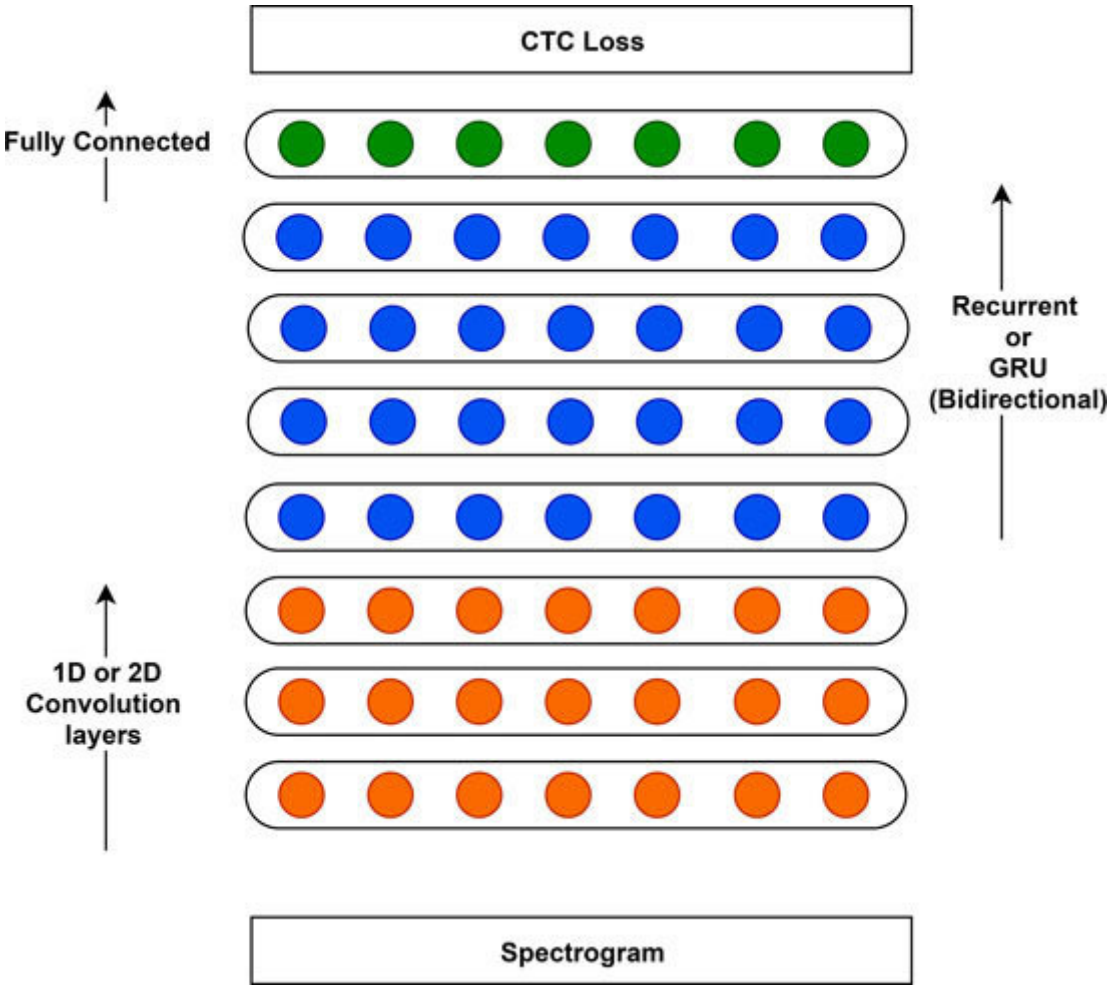


Figure 10.9: *Network architecture for DeepSpeech – 2.*

This network is trained with batch normalization. The paper of DeepSpeech - 2 describes how different arrangement performs differently. Some of the different arrangements are listed here:

Different RNN layers like GRU and LSTM are used

Variable layers of LSTM and CNN are used

Different stride with CNN

Unidirectional and bidirectional model

In the next session, we will see how to use DeepSpeech-2 to translate speech to text.

We will use DeepSpeechPyTorch implementation of the DeepSpeech -2.

Installation

The DeepSpeech can be installed and used just like other packages, or we can directly download its Docker container and start using it without installation. Docker image can be constructed using a Docker file that is provided along with the repository.

Cloning the repository

git clone https://github.com/SeanNaren/DeepSpeech.PyTorch.git &&DeepSpeech.PyTorch

Using the container

sudonvidia-Docker build -t DeepSpeech2.Docker.
sudonvidia-Docker run -ti -v $pwd:/data:/workspace/data -p 8888:8888 --net=host --ipc=host DeepSpeech2.Docker # Opens a Jupyter notebook, mounting the /data drive in the container

We won't discuss the installation from the source, as using Docker is the simplest option available here.

Datasets

DeepSpeech-2 currently supports the AN4, TEDLIUM, Voxforge, and LibriSpeech datasets. The required dataset can be downloaded as follows:

```
# To download AN4 dataset
cd data; python an4.py
# To download TEDLIUM dataset
cd data; python ted.py
# To download Voxforge dataset
cd data; python voxforge.py
# To download AN4 dataset
cd data; python LibriSpeech.py
```

Pretrained Model

A trained model can be downloaded and used as it is or can be used to pre-trained model on which you may apply fine-tuning by your custom dataset. The pre-trained model is present here:
https://github.com/mozilla/DeepSpeech/releases/download/v0.6.1/DeepSpeech-0.6.1-models.tar.gz

Training

Training can be started by simply passing the Train and validation files:

python train.py --train-manifest data/train_manifest.csv --val-manifest data/val_manifest.csv

Visualizing Training

Training can be visualized by pointing tensorboard to the log directory:

```
python train.py --tensorboard --logdirlog_dir/
```

Dataset Augmentation

Similar to images, we can augment the speech dataset by applying augmentation techniques and injecting noise. In augmentation techniques, a small change in tempo and gain increases the robustness of the training. Use the -augment flag when training. One can also inject noise into the training run by specifying noise files location. The noise files can be provided to training by specifying --noise-dir Additionally, one can specify --noise_prob to determine the probability of the noise to be added. Also, --noise-max can be specified to choose a minimum or maximum noise to be added. The noise can be injected with the help of

python noise_inject.py --input-path /path/to/input.wav --noise-path /path/to/noise.wav --output-path /path/to/input_injected.wav --noise-level 0.5

Checkpoints and Continuing from Checkpoint

One can checkpoint the model at every N batch by specifying --checkpoint-per-batch argument:

python train.py --checkpoint --checkpoint-per-batch N # N is the number of batches to wait till saving a checkpoint at this batch.

You can continue from this checkpoint by specifying the

python train.py --continue-from models/DeepSpeech_checkpoint_epoch_N_iter_N.pth

Testing/Inference

The saved model can be used to evaluate the test data, which must be in the same format as the train data:

```
python test.py --model-path models/DeepSpeech.pth --test-manifest /path/to/test_manifest.csv --cuda
```

Transcription can be done by specifying the path to the audio file:

```
python transcribe.py --model-path models/DeepSpeech.pth --audio-path /path/to/audio.wav
```

Running a Server

The code is already provided with the script that exposes everything as a server and can be used as an API. The call to the server can be made as shown:

```
python server.py --host 0.0.0.0 --port 8000 # Run on one window
curl -X POST http://0.0.0.0:8000/transcribe -H "Content-type: multipart/form-data" -F "file=@/path/to/input.wav"
```

Instead of just running the model as a black box, it is advisable to go into detail by understanding model.py. Warp-CTC, a better implementation of the CTC loss is also supported.

Take a look at the following links for reference:

DeepSpeech: Scaling up end-to-end speech recognition: *https://arxiv.org/pdf/1412.5567.pdf*

DeepSpeech 2: End-to-end speech recognition in English and Mandarin: *https://arxiv.org/pdf/1512.02595.pdf*

Understanding Text to Speech

Synthesizing artificial human speech from text using computational techniques is generally known as Text To Speech or TTS. TTS has many applications, and we've been using some of them for years. Some real-life applications are:

Converts article to voice to listen to them on-the-go

For learning on-the-go during daily commute and exercising

Play.ht: Similar to Narro, helps enhance productivity while commuting and exercising or gymming

A modern TTS system is complex and relies on hand-engineered features. This makes the process of the TTS system labor-intensive and complex. Baidu proposes a newer model that uses end-to-end deep learning to train a model named DeepVoice. The first publication of DeepVoice was in 2017, and three versions have been introduced since, claiming state-of-the-art performance in the TTS domain. In this recipe, we will understand DeepVoice 1 to get a basic idea of how such systems work, and then we will learn how to use DeepVoice-3 implementation on the real data.

The text to speech system is not as simple as speech to text. To understand TTS, we must understand a few terms. A grapheme is the smallest unit of a writing system for a given language, and a single grapheme may not carry any meaning. Graphemes can be alphabetic characters or combinations, numerical digits, punctuation marks, Chinese characters, and typographic ligatures. A graphene is often noted in the angle bracket. The phoneme is the unit of the sound that distinguishes one word from another in a language-specific manner. TTS system has the following sub-models:

Graphene to phoneme model: Convert written text to phonemes.

A segmentation model: Locate the phoneme boundaries in the given voice dataset. Given an audio file with phoneme by phoneme translation, the segmentation model identifies where the audio begins and ends for a given phoneme.

Phoneme duration model: Predict the temporal duration of each phoneme. It ensures that the translation is more human-like.

The fundamental frequency model predicts whether the phoneme is be converted to voice or note; this takes care of silent phonemes.

The audio synthesis model combines the output of graphene to phoneme, modulate phoneme duration and fundamental frequency to synthesize audio at high sampling rate.

Now, we will look at each model in detail.

Grapheme to Phoneme Model

The grapheme to phoneme had encoder-decoder architecture. The model has multilayered, bidirectional GRU. The architecture is similar to the language translation model. This model uses teacher forcing to generate phonemes. Bidirectional layers with 1024 units are used.

The Segmentation Model

This model is trained to output alignment between the given utterance and the sequence of target phonemes. This is very similar to speech to text, where the alignment of the given audio with target phonemes is the end goal. Here, CTC loss has been used to train the model.

Phoneme Duration and Fundamental Frequency Model

Here, two models are combined into one. This model takes the utterances generated by the previous m model and produces the phoneme duration and the probability of the phone being voiced. The model has two fully-connected layers with 256 units, each followed by a unidirectional recurrent layer with 128 GRU. The GRU output is given to a fully-connected layer.

Audio Synthesis Model

This model is similar to the Wavenet model, which was first proposed by DeepMind to synthesize a realistic human-like voice. In addition to Wavenet, researchers use Quasi RNN architecture for better speed than normal RNN. The details of Wavenet are beyond the scope of the book, but you can refer to the original paper. The following is a schematic diagram of the overall network:

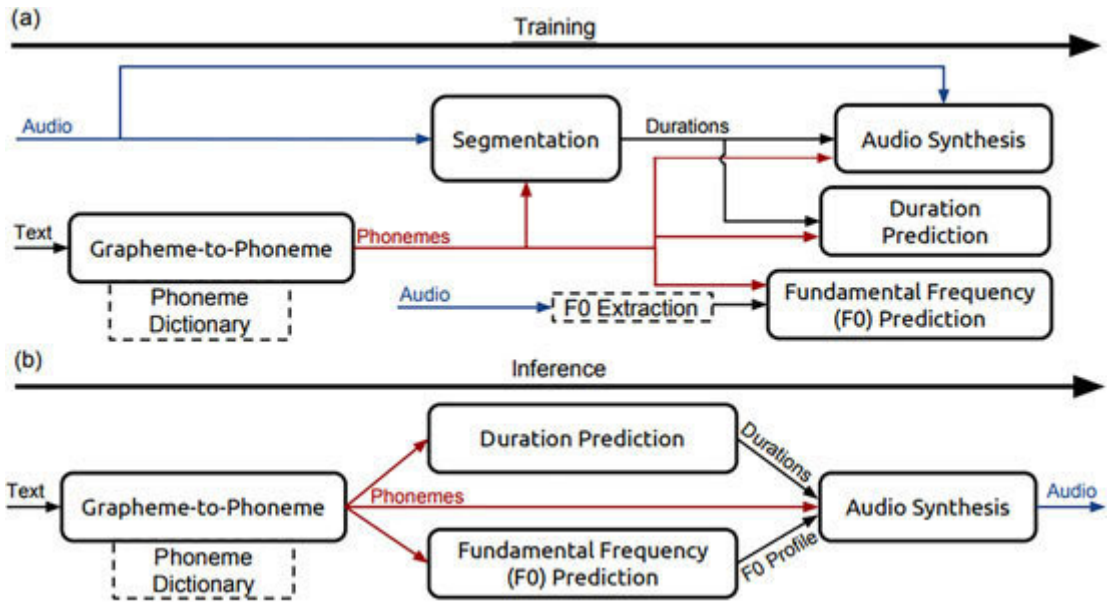


Figure 10.10: *The DeepVoice-Model a) Training procedure b) Inference Procedure.*

Source: *DeepVoice: Real-time Neural TTS.*

This was a basic model, and Baidu published DeepVoice 2 and DeepVoice-3 models after this one. In the next section,

we will see how to use DeepVoice Model 3 to generate voice from the text practically.

We will use DeepVoice PyTorch-based implementation of the text to speech synthesis model.

Download Dataset

One can use all of the following datasets for the text to speech model training:

LJSpeech *https://keithito.com/LJ-Speech-Dataset/*

VCTK
http://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html

JSUT
https://sites.google.com/site/shinnosuketakamichi/publication/jsu

From here onward, we will see how to preprocess the **LJSpeech** dataset, and we will discuss all the preprocessing as well as training commands accordingly.

Installation

PyTorch DeepVoice 3 has the following dependencies and must be installed before proceeding further:

Python 3

CUDA >= 8.0

PyTorch>= v0.4.0

nnmnkwii>= v0.0.11

Then, you need to clone the repo and install the package:

git clone https://github.com/r9y9/DeepVoice3_PyTorch && cd DeepVoice3_PyTorch
pip install -e ".[bin]"

Test to speech is still under research, and each language requires critical hyperparameter settings. For the same purpose, different hyperparameter files are given at the present/ folder of the repository. Each command for preprocessing, train, and synthesis take the respective setting

as an option The same --preset= must be used throughout the process.

Preprocessing

We will preprocess the LJSpeech dataset:

python preprocess.py --preset=presets/DeepVoice3_ljspeech.json ljspeech ~/data/LJSpeech-1.0/./data/ljspeech

The preceding command will extract features like mel-spectrograms and linear spectrograms in

Training

python train.py --preset=presets/DeepVoice3_ljspeech.json --data-root=./data/ljspeech/

Model alignment and and files will be saved in ./checkpoints directory at every 1000 steps.

Monitoring using TensorBoard

tensorboard --logdir=log

Using the model for synthesis

Any checkpoint can be used for the text to speech synthesis, as shown:

```
python synthesis.py${checkpoint_path.pth} ${text_list.txt} ${output_dir} --preset=
```

If you have limited data, you can load the pre-trained model and fine tune your dataset:

dataset:
dataset:
dataset:
dataset:

To use this pre-trained model, you must check out the specific git commit, as follows:

```
git checkout ${commit_hash}
```

Then, you can directly synthesize from the checkpoint, as shown here:

```
# pretrained model
(20180505_DeepVoice3_checkpoint_step000640000.pth)
# hparams (20180505_DeepVoice3_ljspeech.json)
git checkout 4357976
python synthesis.py --preset=20180505_DeepVoice3_ljspeech.json
20180505_DeepVoice3_checkpoint_step000640000.pth
sentences.txt output_dir
```

Here are the reference links for further details:

DeepVoice 3: Scaling text-to-speech with convolutional sequence learning: *https://arxiv.org/pdf/1710.07654.pdf*

DeepVoice: Real-time neural text-to-speech: *https://arxiv.org/pdf/1702.07825.pdf*

Wavenet: A generative model for raw audio: *https://arxiv.org/pdf/1609.03499.pdf*

Conclusion

In this chapter, we saw how speech is also related to the NLP, and it's more like continuously flowing data. We experimented with the preprocessing treatment required for audio data, and then we covered two of the most-used applications in the realm of speech processing: speech to text and text to speech. In both, CNN and RNN models are used, as we saw with the DeepVoice and DeepSpeech models. We also saw that these networks use CTC loss, which we previously understood in detail. This space is continuously evolving, and if the domain excites you, you can explore similar models like NvidiaTacotron and Waveglow.

In the next chapter, we will learn about the operational efficiency of your application.

CHAPTER 11

The Road Ahead

The goal of this book is to provide you an entire landscape of NLP and apply Deep learning for daily use cases. This chapter covers the must-have skills for a deep learning enthusiast. This chapter is mainly about scaling the experiment to the production capable solution using various optimization techniques. We have come a long way, covering most of the basic concepts and their implementation. Conceptual knowledge of these implementations will help you in their real-life application. The previous chapters covered the concepts and applications, but there is more to it. Deployment is an essential end goal attached to real-life applications. When it comes to the Deep Learning project, around 90% of research is not scalable, or no thought process is given to scalability, so it fails or ends up with un-optimized resource utilization. Deep Learning algorithms are computed hungry, and resources must be utilized properly during training or inferencing.

This chapter mainly focuses on resource utilization and deployment of the deep learning application. There is no doubt that GPU is better when it comes to Deep Learning. GPU technology developed as an alternative power source in the past few decades. Many companies are offering GPU compute, but there is no match for Nvidia GPU and its

software support for Deep Learning. GPU with proper software optimization, if utilized properly, allows great speed for a deployment/training pipeline. The next part of the discussion is divided into two sections: training and inferencing in the most efficient way.

Structure

Efficient training

Parallel data loading

Utilizing hardware resources

Efficient deployment

Hardware-related optimizations

Objective

Learning about the upcoming features as well as efficient deployment and hardware-related optimization.

Efficient Training

Most Deep Learning related real-life applications require multiple GPU. Efficient training can be ensured by focusing on the following aspects of the training pipeline:

Parallel data loading

Utilizing hardware resources

Let's understand each of these in detail.

Parallel Data Loading

Most of the deep learning architecture, be it MXNet, PyTorch, or TensorFlow, has parallel data loaders. Parallel data loaders help remove the bottleneck of data flow. Usually, CPU or GPU waits for data to be loaded and for computation to happen, which results in poor training performance. In the above framework, data loaders are provided by default and help gain a performance boost. Data loaders can use multiple cores at a time for parallel data loading to multiple GPUs. The following are the classes for data loader in various frameworks:

torch.utils.data.Dataset and torch.utils.data.DataLoader

torch.utils.data

mx.gluon.data

These data loaders help load data and also help complete intermediate operations like image augmentation, text cleaning, tokenization, vectorization, feature generation, adding noise, and filtering. Another advantage of such a data loader pipeline is that the same data loader can be used while inferencing and it helps minimize the variability in data injection.

Utilizing Hardware Resources

For the last few years, it has become common to have multiple GPUs attached to a single CPU. These multiple GPUs can be utilized simultaneously for training, but you need to change the training process a little for this. Conceptually, parallel training can be done by exploring data parallelism or model parallelism:

Data Here, one model is replicated in multiple GPU, and a different instance of the data is provided to each model. After forward pass, the collective loss from the multiple GPU is calculated, and backpropagation is carried out in each model. Data parallelism is very simple to implement and is the most common type of parallelism used these days.

Model This involves keeping different layers of the model in different GPUs, and operations are carried out in different GPUs with the same set of data. It is complex and needs lots of thought process and implementation. Majority of the large models like Bert, GPT, and single-shot detectors are trained this way.

As mentioned earlier, data parallelism can be implemented easily, so most frameworks support it. Data parallelism can be implemented as one of the following options:

In one machine with multiple GPUs.

In multiple machines with multiple GPUs attached.

Synchronous distributed All workers are synchronized at the start of the new batch, and the server waits to receive gradient from each worker after each batch. The drawback is that the server has to wait for each worker to finish the job. If one worker fails or takes a long time, it makes others wait.

Asynchronous distributed Here, the server maintains a key-value pair of updates, and it updates the parameters and starts with a new batch as soon as any worker finishes the batch, without waiting for others to finish. This mode is faster than synchronous distributed training.

The following are the various classes present in the several frameworks for data parallelism:

torch.nn.DataParallel

MXNet multiple GPU context

tf.distribute. Strategy

Horovod is another great library for distributed training using TensorFlow, Keras, PyTorch, and Apache MXNet. Horovod is

super easy to use, and it provides great speed up with changes in a few lines.

Efficient Deployment

Deployment is the most important part, whereby the interaction with customers or end-users occurs. The smoother the interaction, the better the business. In deployment, the efficiency of the product in terms of hardware cost matters the most. A large model occupies more space in the disk and main memory and often requires more compute resources. Major research on the deployment side is being carried out to shrink model size with an increase in the performance in terms of the number of data points processed per unit of time, without much degradation in its accuracy. These optimizations work by utilizing the following methods, individually or in combination:

Decreasing the floating point precision

Fusing layers

Getting rid of the backpropagation parameters

Most machine learning models do not require high precision for training/prediction, and decreasing precision has very little to no effect on the model performance. Most neural networks work in FP32, but in 2017, Nvidia explored that neural networks can be trained with lower precision, keeping the

same performance. Training neural network in half(FP16) precision requires additional steps like loss scaling. According to one of the white papers by Nvidia, the model trained with FP32 and FP16 has the same performance with loss scaling. Nvidia Tensor cores explore the previously discussed opportunity to greatly enhance training and inference performance.

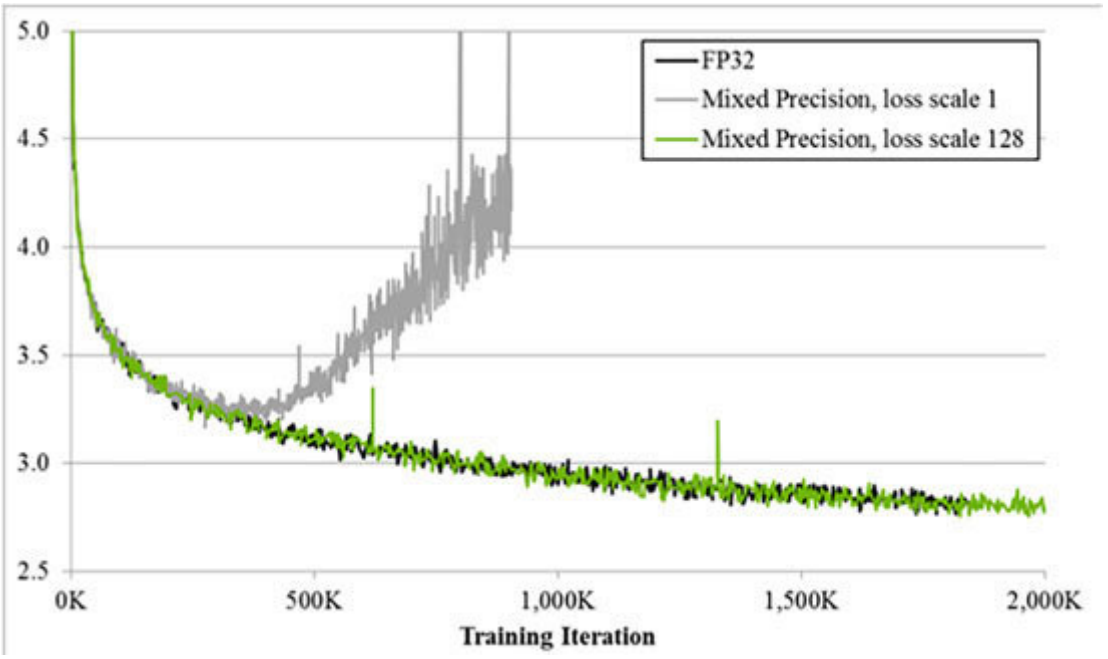


Figure 11.1: *Mixed precision training*

Training curves for the bigLSTM English language model shows the benefits of mixed-precision training techniques. The Y-axis is training loss. Mixed precision without loss scaling (grey) diverges after a while, whereas mixed precision with loss scaling (green) matches the single-precision model (black). **Source:** *https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html*

One of the open-source projects by Nvidia is named Apex, which is being developed to allow support for mixed-precision training on Nvidia GPU. Apex works on the following mechanism:

A library that supports mixed-precision training

FP 16 Wraps the existing PyTorch optimizer in low precision and automatically handles the weight updates and loss-scaling. Plus, adding an apex to the existing model only requires two lines of code change.

By mechanism, the Apex divides all the functions into three parts:

White Function that can be ported into FP16 and will provide a great speedup

Porting such a function will not provide much speedup

Everything All the leftover functions where FP32 - > FP16, conversion cost is high, are left as it is.

In Apex, various other mechanisms like monkey patching are used and a dictionary is maintained to minimize casting operations on variables that have already been cast.

Another candidate that is gaining popularity for deployment is TensorRT. Nvidia-TensorRT has multiple mechanisms to optimize network performance while shrinking its overall size and computational requirement. TensorRT works on the following mechanisms:

Layers with unused output are eliminated to avoid unnecessary computation.

Wherever possible, convolution, bias, and ReLU layers are fused to form a single layer.

Horizontal layer fusion, or layer aggregation improves performance by combining layers that take the same source tensor and apply the same operations with similar parameters.

Backpropagation parameters for the gradient update are not required when performing inference. Removing such parameters also helps shrink the model size. TensorFlow model freezing works on this mechanism.

These graph operations do not change the overall accuracy of the model but help carry out operations faster and in a power-efficient manner.

Hardware-related Optimizations

Now, it is obvious that GPU is the power source for the deep learning related models. GPU is a physical device attached to the motherboard via PCIe lane. A system with multiple GPUs attached to one motherboard is common these days. PCIe has a maximum theoretical bandwidth of 30GB/s, and it proves to be a real bottleneck in a multi-GPU system. When multiple GPUs are connected in the network, such systems are often interconnected by CAT-6 Ethernet or optic-fiber networks. Such connections are not efficient and prove to be a bottleneck. For this purpose, specialized machines are available to accelerate deep learning training. This device relies on the **InfiniBand** connection between GPU-GPU and GPU-CPU. InfiniBand is much faster for data transfer as compared to the traditional data transfer system. NVLink uses InfiniBand along with software-level enhancements to greatly bypass the bottleneck.

IBM Power8 CPU, where CPU can be directly connected to the GPU using NVLink.

Multiple Nvidia-GPUs can be connected using multiple NVlinks provides 10X+ performance than the PCI bandwidth.

The following software-level improvements enhance the performance of such specialized systems:

Remote Direct Memory Access Provides granular access across GPU.

Multi-process service Allows multiprocessing on the GPU based on the time required and is criticality associated with the process. It ensures that one process does not occupy all the resources while other processes die out.

Address translation GPU can directly access the CPU's memory page table. It aids synchronous data pulling from the main memory.

Take a look at the following links:

Data parallelism: *https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html*

Demystifying parallel and distributed deep learning: An in-depth concurrency analysis: *https://arxiv.org/pdf/1802.09941.pdf*

Run MXNet on Multiple CPU/GPUs with data parallelism: *https://mxnet.incubator.apache.org/versions/master/faq/multi_devices.html*

Conclusion

This chapter brought operational efficiency in your application. We briefly visited techniques like mixed-precision training, which significantly improves the pace of the training. We also saw how an accelerator framework like TensorRT can be used for accelerated inference. This chapter briefly introduced you to enterprise-grade solutions to scale out your solution to millions of customers.

Index

A

abstractive text summarization 253
address translation service 373
advance RNN Units 122
affix stemmers
about 48
reference link 48
Ag News corpus
reference link 175
Annotated Transformer
reference link 262
Apex mechanism
functions 371
working 371
asynchronous distributed training 369
attention mechanism
about 155
decoder
Auto Keras tool 282
auto-ml tool
advantages 282
Auto-sklearn tool 282
Auto-WEKA tool 282

B

batches
translating, with Seq2Seq 146
BatchNorm

benefits 182
Batch-wise Matrix Multiplication (BMM) 263
beam search
bias
about 12
cause 13
problem, tackling 14
bias-variance problem
about 15
managing 18
measures 18
Bidirectional Encoder Representations from Transformers (BERT)
about 230
using
Bilingual Evaluation Understudy Score (BLEU) score 158
binary labels 6
bit pair encoding
Bootstrap Aggregating 288

C

character-based embedding
about 106
advantages 105
generation

character level CNN
using
character representation
Comma Separated Values (CSV) 64
components, TorchText

TorchText.data 63
TorchText.datasets 63
TorchText.vocab 63
confusion matrix 10
connectionist temporal classification (CTC)
about 306
working 307
contextual vectors
about 220
pre-trained model, using
contextual vectors, components
Bidirectional Language Model (biLM) 219
token representation 219
Continuous Bag of Words (CBOW) 87
convblock function
layers 183
Convolutional Neural Network (CNN) 163
convolution block 183
convolution layer
about 172
for inputs 166
for inputs 167
convolution operations 163
co-occurrence matrix
about 81
constructing

disadvantages 84
count-based approaches 80
CTC loss

about 306
calculating 308
example 307
usage 310
CTC-warp
installing 309

D

data
downloading 312
loading 298
parsing, from JSON format 44
parsing, from XML format 45
passing, from JSON format 45
pre-processing 298
data learning
about
binary labels 6
fake image data 6
machine learning model 7
model, coding with PyTorch 8
model convergence, confirming 9
perceptron model, implementing 5
data parallelism
about 369
asynchronous distributed training 369

frameworks 370
reference link 373
synchronous distributed training 369
data processing 42

data retrieval 42
dataset
downloading 119
LJSpeech dataset, preprocessing 364
monitoring, with TensorBoard 364
predicting 303
predicting, with CNN 304
predicting, with RCNN 306
preparing 303
pre-processing 169
pre-trained model, using 365
PyTorch DeepVoice installing 363
training 364
dataset description 297
decoder
about 148
with batching 142
decoder batch processing
implementing 147
decoder module
decoder phase 142
decoding 309
Deep Convolution Generative Adversarial Network (DC-GAN) 336
deep convolution network
using 182
deeper network

training 189
DeepSpeech

reference link 360
DeepSpeech 2
reference link 360
DeepVoice
reference link 365
DeepVoice 3
reference link 365
DenseNet 194
derived measures 10
Docker
learning 345
Docker container
commands 345
document classification
multi-document summarization 250
single document summarization 250
Document Object Model (DOM) 42
documents 79

E

efficient deployment
efficient training
about 368
hardware resources, utilizing 369
parallel data loading 368
ElasticNet
about 23

implementing 28
embedding 170
embedding concept

implementing
encoder
about 147
with batching 142
encoder batch processing
implementing 147
encoder module 313
End of Sequence (EOS) 149
ensembling techniques
ensembling techniques, types
bagging 288
boosting 289
stacking 289
error/noise reduction
about 10
BLEU score 12
confusion matrix 10
derived measures 10
weighted loss function, defining 12
evaluation module 139
extrinsic evaluation 104

F

Facebook AI Research (FAIR) 106
fake image data 6
FastText embeddings

training
feature extraction techniques
Mel-Frequency Cepstral Coefficients (MFCC) 349
spectral centroid 350

spectral rolloff 352
Feed-forward Network (FFN) 163
flexible model
with dense layers 285
with RNN layers 286
flexible networks
creating 284
fully connected layers 166

G

GAN architecture
about 325
loss function 326
GAN architecture, variations
CycleGAN 329
music generation 330
PixelDTGAN 329
super-resolution 329
text to image 329
GAN challenges
diminished gradient 331
feature matching 332
historical averaging 334
hyper-parameter selection 331
minibatch discrimination 334

mode collapse 330
nash equilibrium 332
non-convergence 330
one-sided label smoothing 334
GAN-CLS

discriminator 338
discriminator loss 340
generation loss 340
generator 337
GAN components
about 323
discriminator function 324
generator function 324
Gated Recurrent Unit (GRU)
about 127
current memory content 129
final memory 129
implementing 127
reset gate 128
update gate 128
with PyTorch 130
Generative Adversarial Nets (GAN)
implementing, for MNIST
reference link 326
theory 335
Global Vectors (GloVe)
about 99
components
co-occurrence matrix, constructing 101
data preprocessing 101
learnable parameters, defining 99

loss function, defining 100
parameters, defining 101
reference link 105

Gradient Boosted Machine (GBM) 29
Grey-level co-occurrence matrix (GLCM) 84
GRU function
arguments 129

H

hardware-based inferencing 32
hardware-related optimizations 372
hardware resources
utilizing 369
high network
fundamental block 193
highway network 191
hybrid approach 48

I

image
generating, from description 341
image augmentation
about 313
implementing 313
image captioning
inference
about 30
hardware-based inferencing 32

software-based inferencing 30
InferSent
InfiniBand (IB) 372
intrinsic evaluation 104

L

language problem
language translation
building, with transformer
encoder 154
implementing 153
Lasso regression
implementing
Lasso regularization 23
Latent Dirichlet Allocation (LDA)
about 243
applying 242
data preparation 242
model, evaluating 244
output, visualizing 244
reference link 244
learning curve
about 19
data pre-process, loading 19
random forest regression, using 22
simple regression model, using 20
Learning Phrase Representations 127
learning principles
about 32

data related concepts
model related concepts 33
learning rate modifier 291
lemma 49
lemmatization 50
librosa documentation

reference link 352
LJSpeech dataset 363
Long Short-Term Memory (LSTM)
about 117
forget gate 124
gating mechanism
input gate 124
output gate 124
loss function
for sequence to sequence architecture
LSTM Units
modifying 127

M

machine learning model 7
masked language model 231
masking
functions 208
Mean Squared Error Loss (MSELoss) 61
Mean Squared Error (MSE) 5
Mel-Frequency Cepstral Coefficients (MFCC) 349
methods, Word2Vec
Continuous Bag of Words (CBOW) 89

Minimum Description Length (MDL) 32
model
coding, with PyTorch 8
model convergence
confirming 9
model parallelism 369
Multi-process service (MPS) 373

N

named entity recognition (NER)
building 267
character level features
word level features 271
Named Entity Resolution (NER) 52
Natural Language Inference (NLI) 301
Natural Language Processing (NLP)
about 222
problems 41
negative log likelihood (NLL) 93
network 188
network architecture
Neural Machine Translation 152
next sentence prediction model 232
NLTK tokenizer
multi-word expression tokenizer (MWETokenizer) 54
reference link 54
Regular Expressions tokenizer 54
Twitter-aware tokenizer 53
using 52

no pre-trained embedding
training
numbers
computing 79

O

Occam's Razor theory 32
optical character recognition (OCR) 306

P

padding 165
paragraphs 79
Paragraph Vector 213
Paragraph Vector - Distributed Memory model (PV-DM) 215
parallel data loading 368
path decoding algorithm 309
perceptron model
implementing 5
phonemes
about
audio file, loading 347
audio file, playing 347
audio signals, visualizing 349
feature extraction techniques 349
small network, training 352
polysemy 79
pooling layers 167
positional encoding 206

Principle Component Analysis (PCA) 87
PyTorch
about 55
components 56
features 55
installing
used, for coding model 8

R

random multi-model architecture 283
random multi-model deep learning (RMDL)

about 282
applying, on Reuter data 288
network architecture 282
reference link 288
using 286
RECIPE tool 282
Recognizing Textual Entailment (RTE) 301
Rectifier Linear Unit (Relu) 168
Recurrent-CNN architecture 303
Recurrent Convolutional Network (RCN) 301
Recurrent Convolutional Neural Network (RCNN)
about 301
application 302
Recurrent Neural Network (RNN) 117
Recurrent Units 114
Recursive Neural Network (RNN) 263
Regional Convolutional Neural Network (RCNN) 301
regularization

about 22
ElasticNet, implementing 28
Lasso regression, implementing
Lasso regularization 23
Ridge regularization 23
Remote Direct Memory Access (RDMA) 373
residual connection 207
ResNet
about 190
fundamental block 192
Reuter data

RMDL, applying on 288
Ridge regularization 23
rolling

S

Scikit learn functions
to build pipeline 17
sentence
converting, to vector
SentencePiece algorithm
about 281
features 277
sentiment analysis 263
attention mechanism 266
Seq2Seq
batching
used, for translating batches 146
Sequence encoder/decoder

implementing 135
sequence to sequence architecture
loss function 145
sequence to sequence model
short-term fourier transform 349
Short-Term Memory 116
Siamese network
about
layers type 296
Singular Vector Decomposition (SVD) 87
sister network
building 299

SkipThought 218
small network
training 352
snapshot
parameters 293
predicting
recording 292
snowball algorithm
about
reference link 48
software-based inferencing 30
software-level improvements
performance 373
Spacy tokenizer
reference link 54
using 52
spectral centroid 350
spectral rolloff 352
speech recognition pipeline

CNN model, constructing 353
feature extraction 352
performance, estimating on test set 355
performance, testing on test set 355
performance, training on test set 354
Speech to Text
about
checkpoint 359
dataset augmentation 359
datasets 358

DeepSpeech, installing 358
inference 360
pretrained model 358
server, running 360
testing 360
training 359
training, visualization 359
Stanford Natural Language Inference (SNLI) 301
stemming 45
stemming algorithm
production techniques 45
suffix stripping techniques 46
stem network 301
stochastic algorithm 48
stochastic gradient descent (SGD) 93
stride 165
subword-nmt 277
supervised embedding
training
synchronous distributed training 369

T

Tab Separated Values (TSV) 64
TensorBoard
embedding values, projecting on TensorboardX 74
image, projecting to TensorboardX 72
scalar values, displaying on TensorboardX 70
text, displaying on TensorboardX 72
visualizing 69
TensorboardX

reference link 74
TensorRT 31
TensorRT mechanisms
working 372
Term Frequency-Inverse Document Frequency (TF-IDF)
about 85
inverse document frequency 85
matrix, constructing 87
term frequency (TF) 85
text generation
network architecture
text summarization engine
abstractive summarization 250
building 251
extractive summarization 250
Text To Speech (TTS)
about 361
dataset, downloading 363
Text To Speech (TTS), sub-models
audio synthesis model 362
fundamental frequency model 362

graphene model 361
phoneme duration model 362
phoneme model 361
segmentation model 362
token 79
tokenization
about 50
high-level tokenization 51

low-level tokenization 51
NLTK tokenizer, using 51
Spacy tokenizer, using 52
topic modeling
about
LDA, applying 242
reference link 244
TorchText
components 63
data, loading 64
preprocessing 66
reference link 62
using 62
vectorization 68
TPOT tool 282
training
about 137
components 137
transformer
about
positional encoding
source and target masking
used, for building language translation

using

U

unigram language model 277
unrolling
unsupervised pretraining 323

V

Vapnik-Chervonenkis (VC) dimension 33
variance
techniques 15
variants 319
Vector Space Models (VSM) 87
version techniques, Word2Vec
negative sampling 98
sub-sampling 97
word pairs and phrases 97

W

Wavenet
reference link 365
web developers
techniques 45
web page
scrapping 44
weighted loss function

defining 12
use cases 11
Word2Vec
about 88
code implementation
Continuous Bag of Words (CBOW) 90
methods 89
SkipGram 91
version 96
word level CNN
using 169

WordPiece 277
word/token
converting