MODERN ACADEMY

FOR ENGINEERING & TECHNOLOGY

Computer and Information Technology Department

Speech to Speech Translation Using Deep Learning

Graduation Project 2021

Supervised by: Dr. Abdel-Moniem Fouda

17350 بيتر بشرى عزيز

17534 محمود احمد احمد حسن مناع

17743 عبدالرحمن سند عبدالرحمن

19253 عبدالله احمد محمد

17782 سلمى هشام عثمان

17647 مريم لطفي محمد

Speech to Speech Translation Using Deep Learning

Graduation Project

The Research Structure

THE RESEARCH STRUCTURE III

ACKNOWLEDGEMENT IV

ABSTRACT V

TABLE OF CONTENTS VI

LIST OF FIGURES XI

LIST OF ACRONYMS/ABBREVIATIONS XIII

1. CHAPTER ONE: GENERAL INTRODUCTION 1

2. CHAPTER TWO: SPEECH TO SPEECH SYSTEM ANALYSIS 11

3. CHAPTER THREE: SPEECH TO SPEECH SYSTEM DESIGN 96

4. CHAPTER FOUR: SPEECH TO SPEECH PROTOTYPE 112

5. CHAPTER FIVE: SYSTEM IMPLEMENTATION 119

6. CONCLUSIONS AND FUTURE WORK 173

REFERENCES 176

APPENDICES 178

APPENDIX 1: MATHEMATICS FOR DEEP LEARNING 178

CONTACT US 179

Acknowledgement

First of all, we would like to express our sincere gratitude to our supervisor Dr Abdel-Moniem Fouda for the support and guidance throughout this thesis.

Abstract

An average person speaks 11000–25000 words per day making speech the most common way of expressing ourselves. Be it a conversation, dialogue, speech, presentations or any general talks, we use speech to make other as well as ourselves understand thoughts and actions. If either of the side is unaware of the language of communication, the cycle will be incomplete. So, we need a system that can bridge this language barrier. Speech to speech translation is one such system that can play important role by facilitating communication between persons speaking different languages.

Table of Contents

1. CHAPTER ONE: GENERAL INTRODUCTION 1

1.1 INTRODUCTION 2
1.2 HISTORY 3
1.3 OBJECTIVE 4
1.4 PROBLEM STATEMENT 5
1.5 BENEFITS AND EFFECT ON SOCIETY 6
1.6 TOOLBOX 7
1.6.1 Programming Languages 7
Python 7
1.6.2 Machine Learning Libraries 7
Scikit-Learn 7
1.6.3 Deep Learning Frameworks 7
TensorFlow, Pytorch 7
1.6.4 Natural Language Processing Libraries 7
Genism 7
spaCy 7
NLTK 7
1.6.5 Visualization 8
Matplotlib 8
1.6.6 Audio 8
librosa 8
1.6.7 GPU 8
CUDA 8
1.6.8 Google API 8
Text-to-Speech 8
1.7 BUSINESS MODEL 9

2. CHAPTER TWO: SPEECH TO SPEECH SYSTEM ANALYSIS 11

2.1 HOW SYSTEM WORKS 12
2.2 UNDERSTANDING AUDIO DATA REPRESENTATION 13
2.2.1 Spectrogram 14
2.3 LINEAR NEURAL NETWORK 15
2.3.1 Basics of Neural Networks 16
2.3.2 Linear Regression 16
2.3.3 Basic Elements of Linear Regression 16
2.3.4 Linear Model 17
2.4 MULTILAYER PERCEPTRON 19
2.4.1 Layers 20
2.4.1.1 Input layer 20
2.4.1.2 Hidden layer 20
2.4.1.3 Output layer 21
2.4.2 Forward and backward Propagation 22
2.4.2.1 Forward Propagation 22
2.4.2.1.1 Computational Graph of Forward Propagation 23
2.4.2.2 Backpropagation 24
2.4.2.2.1 Computing Gradients 24
2.4.3 Activation Functions 26
2.4.3.1 Role of the Activation Function in a Neural Network Model 26

2.4.3.2 Types of Activation Functions 27
2.4.3.3 Common Nonlinear Activation Functions 27
2.4.3.3.1 Sigmoid / Logistic 27
2.4.3.3.2 TanH / Hyperbolic Tangent 28
2.4.3.3.3 ReLU (Rectified Linear Unit) 28
2.4.3.3.4 SoftMax 29
2.4.4 Loss Functions 30
2.4.4.1 Regression Losses 30
2.4.4.1.1 Mean Square Error/Quadratic Loss/L2 Loss 30
2.4.4.1.2 Root Mean Square Error (RMSE) 31
2.4.4.1.3 Mean Absolute Error/L1 Loss 31
2.4.4.1.4 Mean Bias Error 31
2.4.4.1.5 Cross Entropy Loss/Negative Log Likelihood 31
2.4.5 Optimizers 33
2.4.5.1 Cost Function 33
2.4.5.2 Stochastic Gradient Descent 33
2.4.5.3 Adam Optimizer 34
2.4.5.3.1 How Adam works? 34
2.4.5.3.2 Momentum: 34
2.4.5.4 Root Mean Square Propagation (RMSP) 34
2.4.6 Model's Problems 36
2.4.6.1 Overfitting 36
2.4.6.2 Underfitting 36
2.4.7 Regularization 38
2.4.7.1 What is Regularization? 38
2.4.7.2 How does Regularization help reduce Overfitting? 39
2.4.7.3 Different Regularization Techniques in Deep Learning 40
2.4.7.3.1 L2 & L1 regularization 40
2.4.7.3.2 Dropout 41
2.4.8 Model Evaluation 43
2.4.8.1 Classification Metrics 43
2.4.8.2 Confusion matrix 44
2.4.8.3 Accuracy 46
2.4.8.4 Precision 46
2.4.8.5 Recall 47
2.4.8.6 Receiver Operating Characteristics (ROC) 48
2.4.8.7 R-Squared 49
2.5 SEQUENCE MODELS 50
2.5.1 Recurrent Neural Network (RNN) 50
2.5.2 Different types of RNNs 52
2.5.2.1 Many-to-many 52
2.5.2.2 Many-to-one 52
2.5.2.3 One-to-many 53
2.5.3 Backpropagation through time 54
2.5.3.1 Limitations of backpropagation through time: 55
2.6 MODERN SEQUENCE MODELS 56
2.6.1 GRU 56
2.6.1.1 Reset Gate and Update Gate 57
2.6.1.2 Candidate Hidden State 57
2.6.1.3 Hidden State 57
2.6.2 Long Short-Term Memory (LSTM): 58
2.6.2.1 Gated Memory Cell 58
2.6.2.2 Input Gate, Forget Gate, and Output Gate 58

2.6.2.3 Candidate Memory Cell 59
2.6.2.4 Memory Cell 60
2.6.2.5 Hidden State 60
2.6.3 Bidirectional Recurrent Neural Networks: 62
2.6.3.1 Bidirectional Model 62
2.7 BEAM SEARCH 63
2.8 PRE-TRAINING 65
2.8.1 One-hot Vector 66
2.8.2 Word Embedding 67
2.8.2.1.1 Vector Similarity 69
2.8.2.1.2 Cosine Similarity 69
2.8.3 Word2Vec 70
2.8.3.1 CBOW Model 70
2.8.3.2 SkipGram 72
2.8.4 GloVe 73
2.9 SEQUENCE TO SEQUENCE MODELS 75
2.9.1 Encoder Decoder 75
2.9.1.1 Encoder GRU 76
2.9.1.1.1 Explanation for X 77
2.9.1.1.2 Explanation for H 77
2.9.1.1.3 Explanation for Y 77
2.9.1.1.4 Summary of the Encoder 78
2.9.1.2 Decoder GRU — Training Mode 78
2.9.1.2.1 Summary of Encoder-Decoder Training Process 79
2.9.1.3 Decoder GRU — Inference Mode 79
2.9.1.3.1 Inference Algorithm 81
2.9.2 Attention Mechanism in Deep Learning 81
2.9.2.1 Summary of Encoder-Decoder Training Process 82
2.9.2.2 Why the name Attention? 83
2.9.2.3 Attention Mechanism 83
2.10 CONNECTIONIST TEMPORAL CLASSIFICATION 88
2.11 TIME DISTRIBUTED LAYER 92

3. CHAPTER THREE: SPEECH TO SPEECH SYSTEM DESIGN 96

3.1 MODEL 1: SIMPLE RNN 97
3.1.1 Pseudo Code 98
3.1.1.1.1 Model's parameters: 98
3.1.1.1.2 Main acoustic input: 98
3.1.1.1.3 Recurrent layer: 98
3.1.1.1.4 Softmax activation layer: 98
3.1.1.1.5 Specify the model 98
3.2 MODEL 2: RNN + TIMEDISTRIBUTED DENSE 99
3.2.1 Pseudo Code 99
3.3 MODEL 3: MODEL DEEPER RNN + TIMEDISTRIBUTED DENSE 101
3.3.1 Pseudo Code 101
3.4 MODEL 4: BIDIRECTIONAL RNN + TIMEDISTRIBUTED DENSE 103
3.4.1 Pseudo Code 103
3.4.1.1.1 Model's parameters: 103
3.4.1.1.2 Main acoustic input: 103
3.4.1.1.3 bidirectional recurrent layer: 103
3.4.1.1.4 TimeDistributed layer: 103
3.4.1.1.5 softmax activation layer: 103

3.4.1.1.6 Specify the model: 104
3.5 MODEL 5: CNN + RNN + TIMEDISTRIBUTED DENSE: 105
3.5.1 Pseudo Code 105
3.5.1.1.1 Model's parameters: 105
3.5.1.1.2 Main acoustic input: 105
3.5.1.1.3 Convolutional layer: 106
3.5.1.1.4 Batch normalization: 106
3.5.1.1.5 Recurrent layer: 106
3.5.1.1.6 Batch normalization: 106
3.5.1.1.7 Time Distributed (Dense (output_dim)) layer: 106
3.5.1.1.8 Softmax activation layer: 106
3.6 DEEP SPEECH 2 ARCHITECTURE 107
3.6.1 Comparison between different Models and Architecture 108
3.6.1.1 RNN with Batch Normalization 108
3.6.1.2 Simple RNN vs. GRU 108
3.6.1.3 CNN 109
3.6.2 Pseudo Code 109
4. CHAPTER FOUR: SPEECH TO SPEECH PROTOTYPE 112
4.1 DATA SYNTHESIS: CREATING A SPEECH DATASET 113
4.2 GENERATING A SINGLE TRAINING EXAMPLE 113
4.3 MODEL 115
4.3.1 Build the model 115
4.3.1.1 Implementing the model 116
4.3.2 Fit the model 117
4.3.3 Test the model 117
5. CHAPTER FIVE: SYSTEM IMPLEMENTATION 119
5.1 DESCRIPTION OF DATASET 120
5.1.1 LibriSpeech Structure 120
5.1.2 How Data Prepared 123
5.1.3 Output 125
5.2 DATA GENERATOR 125
5.3 CHARACTERS MAPPING 134
5.4 DATA CONVERSION 135
5.5 MODEL DESIGN 136
5.5.1 Simple RNN Model 137
5.5.2 RNN + TimeDistributed Dense Model 139
5.5.3 Deeper RNN Model 141
5.5.4 Bidirectional Model 143
5.5.5 CNN RNN Model 145
5.5.6 Final Model – Combination of all Previous Models 148
5.6 TRAINING PROCESS 150
5.6.1 ctc-lambda-function 150
5.6.2 add_ctc_loss 150
5.6.3 train_model 151
5.7 MODEL PREDICTION 155
5.8 AUDIO RECORDING 156
5.9 APIS 157

5.9.1 Watson Translation API *157*
5.9.1.1 Authentication 157
5.9.1.2 Access between services 158
5.9.2 Watson Speech 2 Text *159*
5.9.3 Synthesizer *160*
5.10 MODELS PERFORMANCE 162
5.11 RESULTS 163
5.11.1 Result of First Model Simple RNN *163*
5.11.2 Result of Second Model RNN Time Distributed *164*
5.11.3 Result of Third Model RNN_CNN_TimeDist *165*
5.11.4 Result of 4th Model Deeper RNN *166*
5.11.5 Result of 5th BIRNN with Time Distributed *167*
5.11.6 Result of Final Model With 10 Hours of Training *168*
5.11.7 Result of Final Model With 360 Hours of Training *169*
5.11.8 Result of Final Model With 1000 Hours of Training *170*
5.12 LIVE TRANSCRIPTION SENTENCE BY SENTENCE 171

6. CONCLUSIONS AND FUTURE WORK 173

6.1 CONCLUSION 174
6.2 FUTURE TASKS 174
6.2.1 Future Tasks: Voice Characteristics *174*
6.2.2 Future Tasks: High Precision All-Purpose Translation System *174*
6.2.3 Future Tasks: Simultaneous Interpretation *174*
6.2.4 Future Tasks: Context-Based Translation *175*
6.2.5 Future Tasks: Spoken Arabic Language *175*

List of Figures

CH1 1 Neural Network 3
CH1 2 SpeechToSpeech Business Model 9
CH2 1 How System Works 12
CH2 2 Audio in Time Domain 13
CH2 3 Audio in Frequency Domain 14
CH2 4 Spectrogram 14
CH2 5 Simple Neural Network 15
CH2 6 Multilayer neural network with 1 hidden layer 19
CH2 7 The input layer for image recognition tasks consists of a 3D input feature matrix. 20
CH2 8 Computational Graph of Forward Propagation 23
CH2 9 Activation Gate 26
CH2 10 Sigmoid activation function 27
CH2 11 TanH activation function 28
CH2 12 Relu activation function 28
CH2 13 \Gradient Descent 33
CH2 14 Adam Optimizer 35
CH2 15 Overfitted models 38
CH2 16 Training set vs. Test set Error 38
CH2 17 Overfitting in the training set 39
CH2 18 Well fitted model 40
CH2 19 Neural network before applying dropout technique 41
CH2 20 Neural network after applying dropout technique 42
CH2 21 Confusion matrix for binary classification 44
CH2 22 Confusion matrix without normalization 45
CH2 23 Normalized confusion matrix 46
CH2 24 Roc Curve 48
CH2 25 Before RNN 50
CH2 26 RNN Model 51
CH2 27 RNN-Many to Many 52
CH2 28 RNN-Many to One 52
CH2 29 RNN-One to Many 53
CH2 30 RNN-Many to Many Encoder-Decoder 53
CH2 31 GRU Architecture 56
CH2 32 Input Gate, Forget Gate, and Output Gate 58
CH2 33 Candidate Memory Cell 59
CH2 34 Memory Cell 60
CH2 35 Hidden State 61
CH2 36 Bidirectional Model 62
CH2 37 Beam Search 64
CH2 38 from Pretraining to final model 65
CH2 39 deep learning architectures for different downstream natural language processing applications 66
CH2 40 One-hot vector represent 66
CH2 41 Embedding Matrix 67
CH2 42 visualization of word embedding 68
CH2 43 Cosine Similarity 69
CH2 44 Illustration of a word embedding vector space in 2-D 69
CH2 45 CBOW & SkipGram model 70

CH2 46 CBOW Model with one word of context 71
CH2 47 CBOW Model with Multiple word of context 71
CH2 48 Skip_gram model 72
CH2 49Co-occurrence probabilities and ratio with the words ice and steam in function 74
CH2 50 Encoder Decoder architecture 75
CH2 51 Encoder GRU 76
CH2 52 Decoder GRU — Training Mode 78
CH2 53 Summary of Encoder-Decoder Training Process 79
CH2 54 Output Sequence of Encoder in Inference Mode 80
CH2 55 Output Sequence of Decoder in Inference Mode 81
CH2 56 Summary of Encoder-Decoder Training Process 82
CH2 57 Attention Mechanism 83
CH2 58 CTC 88
CH3 1 SIMPLE RNN ARCH 97
CH3 2 RNN + TimeDistributed Dense 99
CH3 3 Deeper RNN + TimeDistributed Dense 101
CH3 4 Bidirectional RNN + TimeDistributed Dense 103
CH3 5 CNN + RNN + TimeDistributed Dense 105
CH3 6 Deep Speech 2 Architecture 107
CH4 1 Trigger Word Model ARCH 115
CH5 1 Model 1 Result 163
CH5 2 Model 2 Result 164
CH5 3 Model 3 Result 165
CH5 4 Model 4 Result 166
CH5 5 Model 5 Result 167
CH5 6 Final Model 10 Hours of Training 168
CH5 7 Final Model 360 Hours of Training 169
CH5 8 CH5 7 Final Model 1000 Hours of Training 170
CH5 9 Live Transcription 171

List of Acronyms/Abbreviations

NLP	Natural Language Processing
S2S	Speech to Speech
MLP	Multi-Layered Perceptron
RNN	Recurrent Neural Network
BPTT	Backpropagation Through Time
API	Application Programming Interface
ASR	Automatic Speech Recognition
NMT	Neural Machine Translation
GRUs	Gated Recurrent Units
LSTM	Long Short-Term Memory
BRNN	Bidirectional Recurrent Neural Networks
FFT	Fast Fourier Transform
Glove	Global Vectors for Word Representation
CBOW	Continuous Bag-Of-Words
SG	Skip Gram
CTC	Connectionist Temporal Classification

1. CHAPTER ONE:
GENERAL INTRODUCTION

1.1 Introduction

Owing to this era of global and border less economy, the information exchange has become inevitable and speech is the traditional and best way for the same. The global scenario adds the demand of communication among speakers of different languages. This is actually one of the great challenges before the Information Technologists to overcome language barriers across the global community, and enable them to express themselves in real time. Further, the task of bridging the Digital-divide can never be accomplished in real sense without breaking the language barriers with an intelligent system or machines. Speech translation technology—being able to speak and have one's words translated automatically into the other person's language has long been a dream of humankind. In the recent reductions, speech to speech translation has been placed as one of the ten technologies that will change the world. All the languages prevailing in our world have different origins and set of their native mother tongue is also quite discrete. According to a study, it is quite cumbersome and costly for an 'adult' to learn new language than a child. Almost all worldly population finds acquisition of foreign languages extremely difficult due to the geographical factors. Thus, speech to speech translation technology would be a great boon to all. Manual translation has been limited to important official documents, news items and some award-winning literary works. There exists a huge backlog of materials that needs to be translated for administration, education, commerce, tourism etc. Technological support in the form of machine aids for translation is of great importance

In this chapter we will Discussing history of speech2speech Translator and the problem statement and how it will be improving our communication and how we will use it in various fields, then we will talk about our toolbox which enabling us to do this application.

1.2 History

You must be quite familiar with speech recognition systems. They are ubiquitous these days – from Apple's Siri to Google Assistant. These are all new advents though brought about by rapid advancements in technology.

Did you know that the exploration of speech recognition goes way back to the 1950s? That's right – these systems have been around for over 50 years! We have prepared a neat illustrated timeline for you to quickly understand how Speech Recognition systems have evolved over the decades:

The first speech recognition system, Audrey, was developed back in 1952 by three Bell Labs researchers. Audrey was designed to recognize only digits.

Just after 10 years, IBM introduced its first speech recognition system IBM Shoebox, which was capable of recognizing 16 words including digits. It could identify commands like "Five plus three plus eight plus six plus four minus nine, total," and would print out the correct answer, i.e., 17.

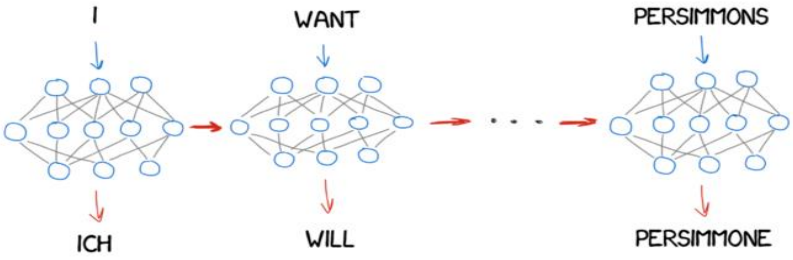
The Defense Advanced Research Projects Agency (DARPA) contributed a lot to speech recognition technology during the 1970s. DARPA funded for around 5 years from 1971-76 to a program called Speech Understanding Research and finally, Harpy was developed which was able to recognize 1011 words. It was quite a big achievement at that time.

In the 1980s, the Hidden Markov Model (HMM) was applied to the speech recognition system. HMM is a statistical model which is used to model the problems that involve sequential information. It has a pretty good track record in many real-world applications including speech recognition.

In 2001, Google introduced the Voice Search application that allowed users to search for queries by speaking to the machine. This was the first voice-enabled application which was very popular among the people. It made the conversation between the people and machines a lot easier.

By 2011, Apple launched Siri that offered a real-time, faster, and easier way to interact with the Apple devices by just using your voice. As of now, Amazon's Alexa and Google's Home are the most popular voice command based virtual assistants that are being widely used by consumers across the globe.

RNNs are used everywhere — Siri's speech recognition (it's parsing the sequence of sounds, where the next depends on the previous), keyboard's tips (memorize the prior, guess the next), music generation, and even chatbots.



CH1 1 Neural Network

In two years, neural networks surpassed everything that had appeared in the past 20 years of translation. Neural translation contains 50% fewer word order mistakes, 17% fewer lexical mistakes, and 19% fewer grammar mistakes. The neural networks even learned to harmonize gender and case in different languages. And no one taught them to do so.

The most noticeable improvements occurred in fields where direct translation was never used. Statistical machine translation methods always worked using English as the key source. Thus, if you translated from Russian to German, the machine first translated the text to English and then from English to German, which leads to a double loss.

Neural translation doesn't need that — only a decoder is required so it can work. That was the first time that direct translation between languages with no common dictionary became possible.

1.3 Objective

Develop an application that perform speech to Text transcribe spoken language into text then Translate it to another Language then read it with a machine voice. Also, we can use our Speech2Text as Speech recognition to Execute Some Commands. We will Develop a Speech2Text From Scratch Depend on Machine Learning, then we will use an APIs to translate and read translated Language.

1.4 Problem Statement

Language barrier is the primary challenge for effective cross-cultural communication in education, health care, and business collaboration. Despite the increasing need for a solution, language remains “the biggest barrier to intercultural collaboration. Though human interpreters a bridge the communication gap between people with different languages, the services accessible only to a privileged few because of its cost and availability. For example, international organizations like the United Nations pay a freelance interpreter over $600 per day. As a result, only a handful of spoken encounters between humans are interpreted.

Speech- to speech translation technology is significant because it enables speakers of different languages from around the world to communicate, erasing the language divide in global business and cross-cultural exchange. Achieving speech translation would have tremendous scientific, cultural, and economic value for humankind.

Engineering automatic speech recognition (ASR) for speech to speech (S2S) translation systems, especially targeting languages and domains that do not have readily available spoken language resources, is immensely challenging due to several reasons. In addition to contending with the conventional data-hungry speech acoustic and language modeling needs, these designs have to accommodate varying requirements imposed by the domain needs and characteristics, target device and usage modality (such as phrase-based, or spontaneous free form interactions, with or without visual feedback) and huge spoken language variability arising due to socio-linguistic and cultural differences of the users.

Over the past few decades, there has been tremendous development in machine learning paradigms used in automatic speech recognition (ASR) for home automation to space exploration. Though commercial speech recognizers are available for certain well-defined applications like dictation and transcription, many issues in ASR like recognition in noisy environments, multilingual recognition, and multi-modal recognition are yet to be addressed effectively.

In This project, an important issue in speech translation is to minimize the negative effect of speech recognition errors on machine translation by using Multiple deep learning models to optimize speech recognition accuracy.

1.5 Benefits and Effect on Society

Popular Application like Skype use speech-to-speech translator to help their users break the language barriers by showing translated conversations instantly.

Online video streaming services, like YouTube, use speech recognition technology to generate automatic captions for their contributors. Not only can help video producers drastically increase the scope of their audience reach with a much lower translation cost, but audiences around the world can also enjoy more videos in the original voice with translated subtitles in no time!

Traveling abroad can fun and exciting. Not only does it allow you to experience new cultures but it can also help you to form new friendships, become more independent, and develop your communications. However, while visiting different countries can be an enlightening experience, some people are concerned that it may take them too far out of their comfort zone. This is especially the case when it comes to language barriers.

Electronic translators are being developed every day. They typically come in the form of apps and devices. And they have proven to be very convenient for people who are interested in speaking to people of other nationalities without having to be fluent in their language.

The application could also be used in international conferences and will make it easier to translate a language to another without using any human resources.

It is Versatile

While hiring a human translator can be a good idea at times, there’s often a limit to how many languages they know. And this means that if you want to communicate with someone who speaks a language other than what your interpreter is able to speak, you won’t be able to effectively communicate with them. The good news is you don’t have to worry about this with a smart voice translator.

speech to speech translator applications usually offer multiple languages. If you want to speak several languages then simply download them or change categories to suit your particular needs. This is much easier than attempting to use a phrasebook or dictionary. With just one touch of a button, you’ll be able to speak between several cultures.

1.6 Toolbox

1.6.1 Programming Languages

Python

Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Also, Python have a Huge Machine learning and Deep Learning Libraries and Frameworks, and Active Community.

1.6.2 Machine Learning Libraries

Scikit-Learn

Sklearn Provide many Functions to solve many problems in machine learning, such as Preprocessing, Classification, Regression, Clustering, Dimensionality Reduction and Model Selection.

1.6.3 Deep Learning Frameworks

TensorFlow, Pytorch

TensorFlow and pytorch is an end-to-end open source platform for machine and Deep learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

1.6.4 Natural Language Processing Libraries

Genism

The fastest library for training of vector embeddings – Python or otherwise. The core algorithms in Gensim use battle-hardened, highly optimized & parallelized C routines. Gensim can process arbitrarily large corpora, using data-streamed algorithms. There are no "dataset must fit in RAM" limitations.

spaCy

spaCy is the best way to prepare text for deep learning. It interoperates seamlessly with TensorFlow, PyTorch, scikit-learn, Gensim and the rest of Python's awesome AI ecosystem. With spaCy, you can easily construct linguistically sophisticated statistical models for a variety of NLP problems.

NLTK

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries

1.6.5 Visualization

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

1.6.6 Audio

librosa

librosa is a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems.

1.6.7 GPU

CUDA

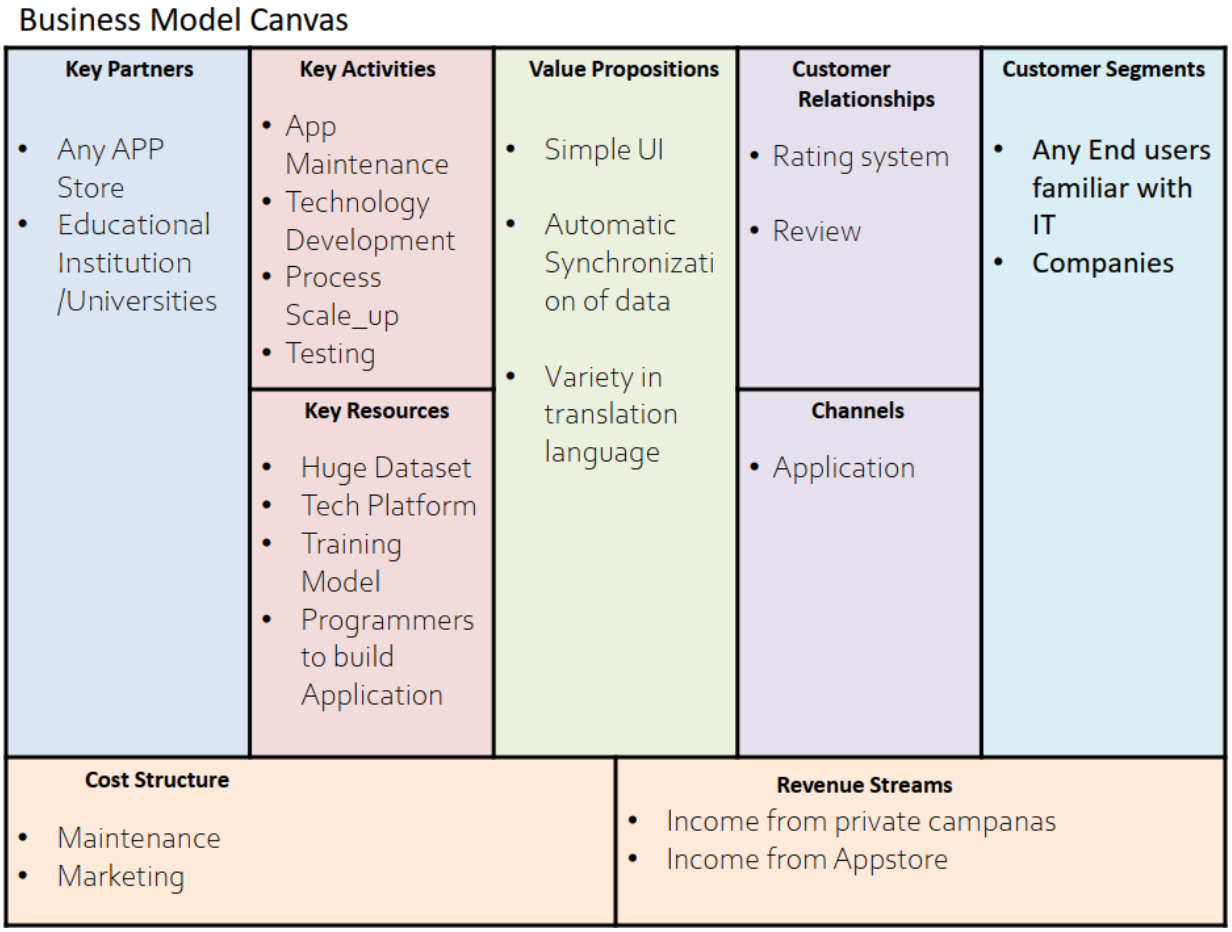
CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

1.6.8 Google API

Text-to-Speech

Convert text into natural-sounding speech using an API powered by Google's AI technologies.

1.7 Business Model



CH1 2 SpeechToSpeech Business Model

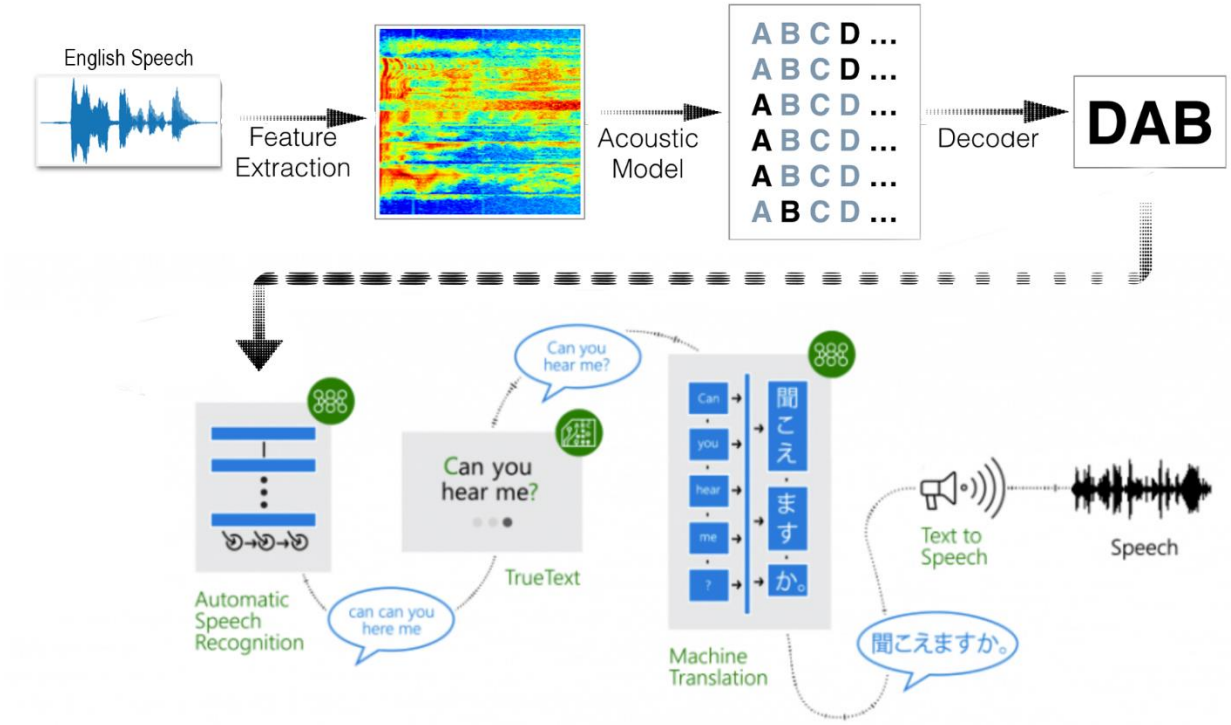
2. CHAPTER TWO: SPEECH TO SPEECH SYSTEM ANALYSIS

In this chapter we will analyze Speech2Speech Translator System and explain how components in system works in details, also we will introduce different architecture for same problem and we will try these in our system to find best model.

This chapter built as Bottom-up approach so, each topic depends on its previous topic.

2.1 How System Works

The Raw speech audio is pre-processed to one of two feature representations that are commonly used for ASR then acoustic model which accepts audio features as input and returns a probability distribution over all potential transcriptions. converted text is translated using neural machine translation (NMT) to Target Language. Then Text-to-speech, or voice synthesis, creates computer-generated audio output from the translated text, in next chapter we will explain how acoustic model works in details.

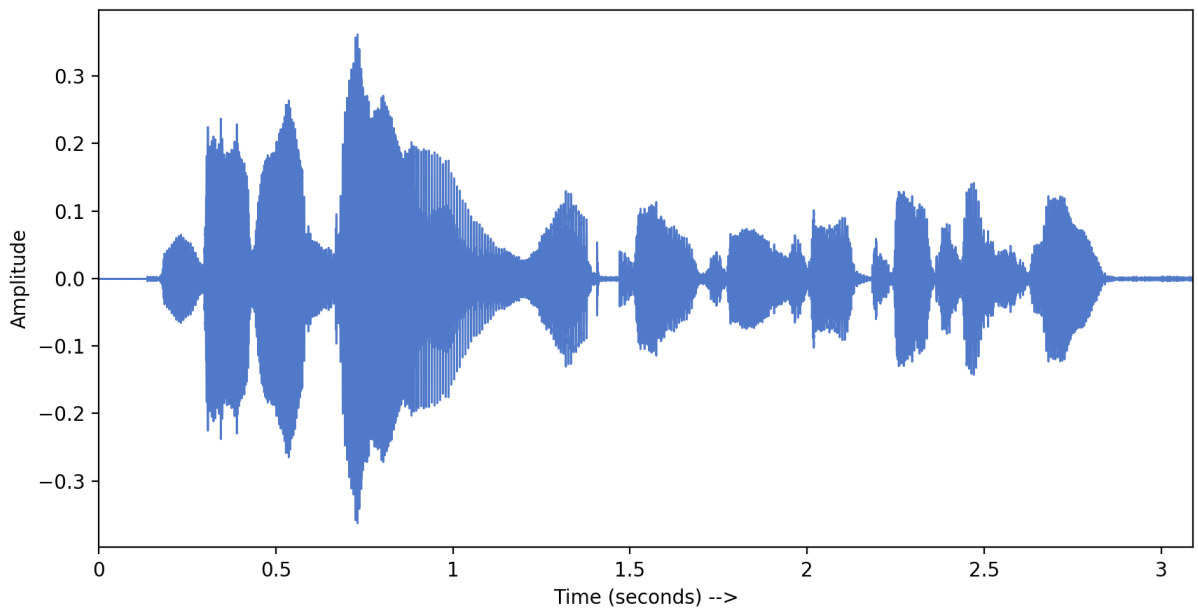


CH2 1 How System Works

2.2 Understanding Audio data Representation

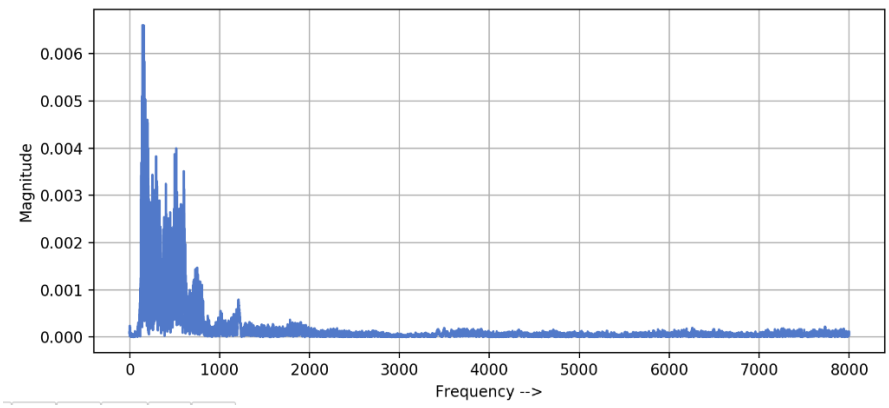
What really is an audio recording? A microphone records little variation in air pressure over time, and it is these little variations in air pressure that your ear also perceives as sound. You can think of an audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone. We will use audio sampled at 44100 Hz (or 44100 Hertz). This means the microphone gives us 44100 numbers per second. Thus, a 10 second audio clip is represented by 441000

(44100hz*10s) It is quite difficult to figure out from this "raw" representation of audio whether the words were said. In order to help your sequence model more easily learn to detect words, we will compute a spectrogram of the audio. The spectrogram tells us how much different frequencies are present in an audio clip at a moment in time.



CH2 2 Audio in Time Domain

This visualization is called the time-domain representation of a given signal. This shows us the loudness (amplitude) of sound wave changing with time. here amplitude = 0 represents silence. These amplitudes are not very informative, as they only talk about the loudness of audio recording. To better understand the audio signal, it is necessary to transform it into the frequency-domain. The frequency-domain representation of a signal tells us what different frequencies are present in the signal. Fourier Transform is a mathematical concept that can convert a continuous signal from time-domain to frequency-domain.

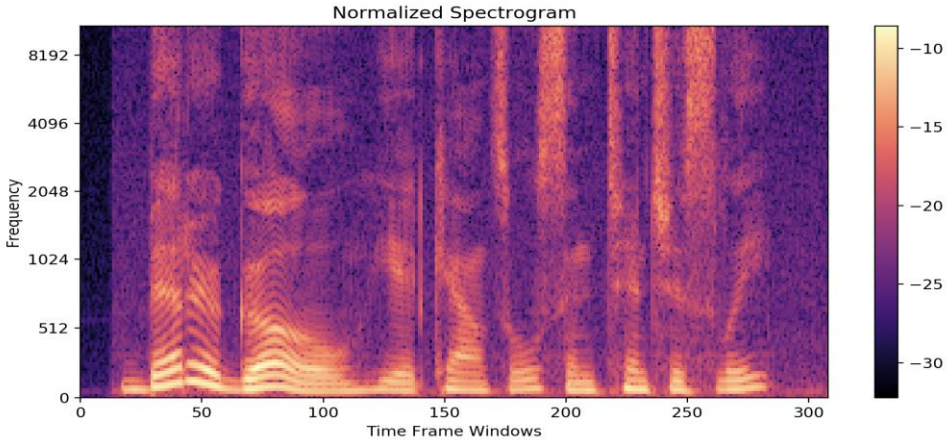


CH2 3 Audio in Frequency Domain

In speech2text we need both Frequency and Time Domain so, we will use spectrogram which combines both.

2.2.1 Spectrogram

Suppose You have an audio file in which someone is speaking a phrase (for example: How are you). Your recognition system should be able to predict these three words in the same order (1. 'how', 2. 'are', 3. 'you'). frequency values which will serve as features for our recognition system. But when we applied FFT to our signal, it gave us only frequency values and we lost the track of time information. Now our system won't be able to tell what was spoken first if we use these frequencies as features. We need to find a different way to calculate features for our system such that it has frequency values along with the time at which they were observed.
Here Spectrograms come into the picture.



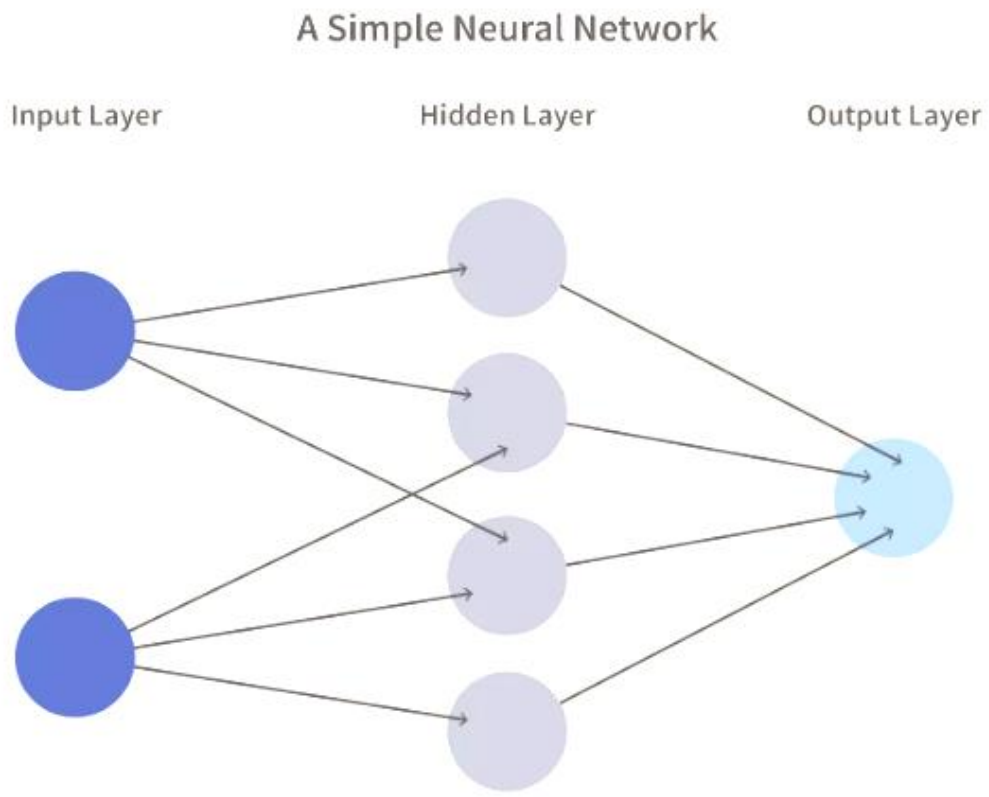
CH2 4 Spectrogram

Visual representation of frequencies of a given signal with time is called Spectrogram. In a spectrogram representation plot — one axis represents the time; the second axis represents frequencies and the colors represent magnitude (amplitude) of the observed frequency at a particular time.

Now we have an Audio file as Spectrogram which is a 2D matrix representing the frequency magnitudes along with time for a given signal

2.3 Linear Neural Network

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so, the network generates the best possible result without needing to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems.



CH2 5 Simple Neural Network

2.3.1 Basics of Neural Networks

A neural network works similarly to the human brain's neural network. A "neuron" in a neural network is a mathematical function that collects and classifies information according to a specific architecture. The network bears a strong resemblance to statistical methods such as curve fitting and regression analysis.

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may be nonlinear.

In a multi-layered perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. For instance, the patterns may comprise a list of quantities for technical indicators about a security; potential outputs could be "buy," "hold" or "sell."

Hidden layers fine-tune the input weightings until the neural network's margin of error is minimal. It is hypothesized that hidden layers extrapolate salient features in the input data that have predictive power regarding the outputs. This describes feature extraction, which accomplishes a utility like statistical techniques such as principal component analysis.

2.3.2 Linear Regression

Regression refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable. In the natural sciences and social sciences, the purpose of regression is most often to characterize the relationship between the inputs and outputs. Machine learning, on the other hand, is most often concerned with prediction.

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting length of stay (for patients in the hospital), demand forecasting (for retail sales), among countless others.

2.3.3 Basic Elements of Linear Regression

Linear regression may be both the simplest and most popular among the standard tools to regression. Dating back to the dawn of the 19th century, linear regression flows from a few simple assumptions. First, we assume that the relationship between the independent variables x and the dependent variable y is linear, i.e., that y can be expressed as a weighted sum of the elements in x, given some noise on the observations. Second, we assume that any noise is well-behaved (following a Gaussian distribution).

To motivate the approach, let us start with a running example. Suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years). To develop a model for predicting house prices, we would need to get our hands on a dataset consisting of sales for which we know the sale price, area, and age for each home. In the terminology of machine learning, the dataset is called a training dataset or training set, and each row (here the data corresponding to one sale) is called an example (or data point, data instance, sample). The thing we are trying to predict (price) is called a label (or target). The independent variables (age and area) upon which the predictions are based are called features (or covariates).

Typically, we will use n to denote the number of examples in our dataset. We index the data examples by i, denoting each input as $x^{(i)}=[x_1^{(i)}, x_2^{(i)}]$^$T$ and the corresponding label as $y^{(i)}$.

2.3.4 Linear Model

The linearity assumption just says that the target (price) can be expressed as a weighted sum of the features (area and age):

$$price = warea \cdot area + wage \cdot age + b$$

In previous section $w.area$ and $w.age$ are called weights, and b is called a bias (also called an offset or intercept). The weights determine the influence of each feature on our prediction and the bias just says what value the predicted price should take when all of the features take value 0. Even if we will never see any homes with zero area, or that are precisely zero years old, we still need the bias or else we will limit the expressivity of our model. As we spoke is an affine transformation of input features, which is characterized by a linear transformation of features via weighted sum, combined with a translation via the added bias.

Given a dataset, our goal is to choose the weights w and the bias b such that on average, the predictions made according to our model best fit the true prices observed in the data. Models whose output prediction is determined by the affine transformation of input features are linear models, where the affine transformation is specified by the chosen weights and bias.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form like this is common. In machine learning, we usually work with high-dimensional datasets, so it is more convenient to employ linear algebra notation. When our inputs consist of dd features, we express our prediction $y^\wedge$ (in general the "hat" symbol denotes estimates) as

$$y^\wedge = w1x1+\ldots+wdxd + b.$$

Collecting all features into a vector $x \in Rd$ and all weights into a vector $w \in Rd$, we can express our model compactly using a dot product:

$$y^{=w}\top x + b$$

the vector xx corresponds to features of a single data example. We will often find it convenient to refer to features of our entire dataset of n examples via the design matrix $X \in Rn \times d$. Here, X contains one row for every example and one column for every feature.

For a collection of features X, the predictions $y^\wedge \in Rn$ can be expressed via the matrix-vector product:

$$y^\wedge = Xw + b$$

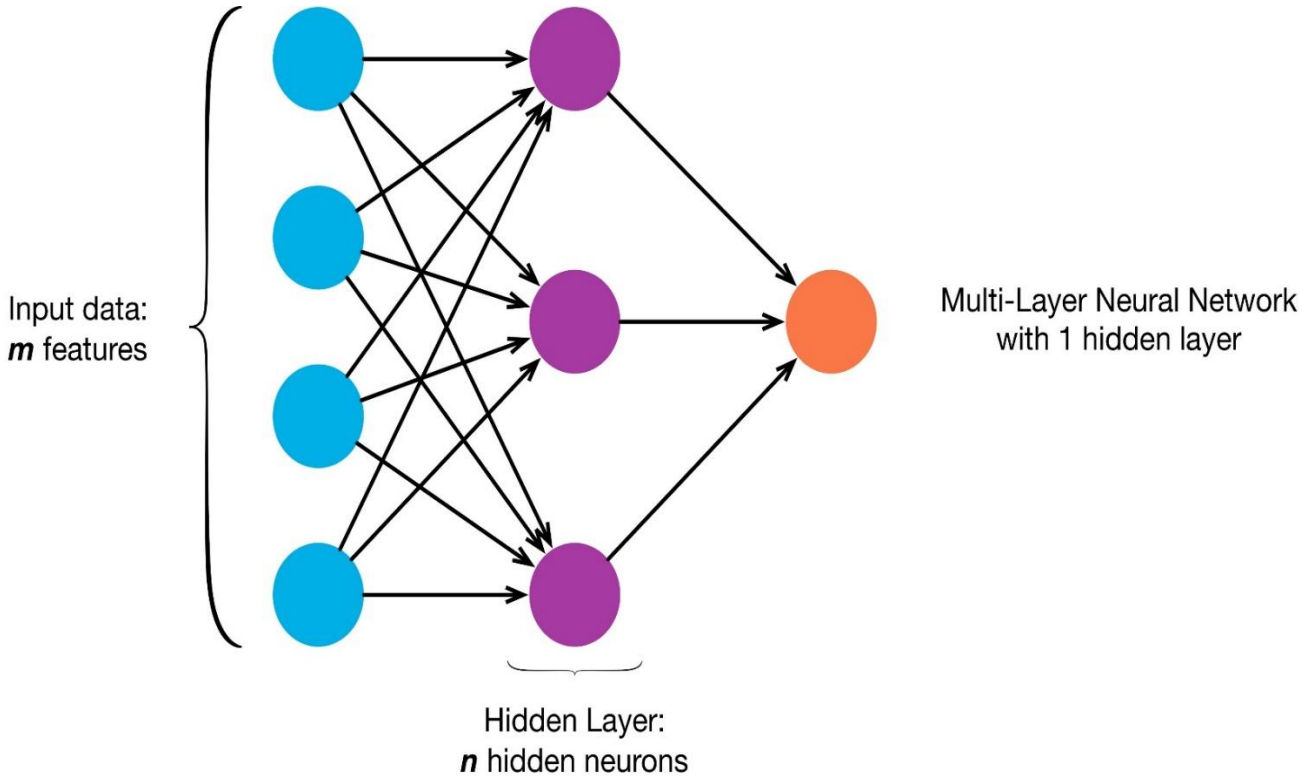
where broadcasting is applied during the summation. Given features of a training dataset X and corresponding (known) labels y, the goal of linear regression is to find the weight vector w and the bias term b that given features of a new data example sampled from the same distribution as X, the new example's label will (in expectation) be predicted with the lowest error.

Even if we believe that the best model for predicting y given x is linear, we would not expect to find a real-world dataset of n examples where $y(i)$ exactly equals $w\top x(i) + b$ for all $1 \leq i \leq n$ For example, whatever instruments we use to observe the features X and labels y might suffer small amount of measurement error. Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

Before we can go about searching for the best parameters (or model parameters) w and b, we will need two more things: (i) a quality measure for some given model; and (ii) a procedure for updating the model to improve its quality.

2.4 Multilayer Perceptron

The field of artificial neural networks is often just called neural networks or multi-layer perceptrons after perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks.
It is a field that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that we can use to model difficult problems.



CH2 6 Multilayer neural network with 1 hidden layer

The power of neural networks comes from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm.
The predictive capability of neural networks comes from the hierarchical or multi-layered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example, from lines, to collections of lines to shapes.

2.4.1 Layers

2.4.1.1 Input layer

The input layer consists of a list of input features. For a single image, the input layer consists of a single array of 3 dimensions: width and height of the image (measured as the number of pixels) and the number of color channels, i.e. 1 channel for black and white and 3 or more channels for color images.

CH2 7 The input layer for image recognition tasks consists of a 3D input feature matrix.

2.4.1.2 Hidden layer

Hidden layer(s) are the secret of the neural network. They allow you to model complex data thanks to their nodes/neurons. They are “hidden” because the true values of their nodes are unknown in the training dataset. In fact, we only know the input and output. Each neural network has at least one hidden layer. Otherwise, it is not a neural network. Networks with multiple hidden layers are called deep neural networks. The most common type of hidden layer is the fully connected layer.

Here, each neuron is connected to all the others in two adjacent layers. It is not connected to the ones in the same layer. Convolutional layers are another type of hidden layers that are very prominent when dealing with images.

Neurons are the processing units of the network. Each neuron weighs and sums the different inputs and passes them through an activation function. The role of the activation function is to buffer the data before it is fed to the next layer. You can change the activity of your neuron.

2.4.1.3 Output layer

The output layer is the final layer with neurons. This is where the data comes out of your model. So, the number of neurons needs to be exactly the number of outputs you want, i.e. the questions you want to answer. If we want to know if the object in the image either a dog or cat, the number of output neurons is exactly 2. One is for the probability of being a dog and the other for being a cat.

- A regression problem may have a single output neuron and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the class 1. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.
- A multi-class classification problem may have multiple neurons in the output layer, one for each class (e.g. three neurons for the three classes in the famous iris flowers classification problem). In this scenario, a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

2.4.2 Forward and backward Propagation

So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the deep learning framework.

The automatic calculation of gradients (automatic differentiation) profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. Surprisingly often, academic papers had to allocate numerous pages to deriving update rules. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

In this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*). To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs. To start, we focus our exposition on a one-hidden-layer MLP with weight decay (L2 regularization).

2.4.2.1 Forward Propagation

Forward propagation (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious

For the sake of simplicity, let us assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h,

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden variable h is also an intermediate variable. Assuming that the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector of length q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

Assuming that the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y).$$

According to the definition of L2 regularization, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right),$$

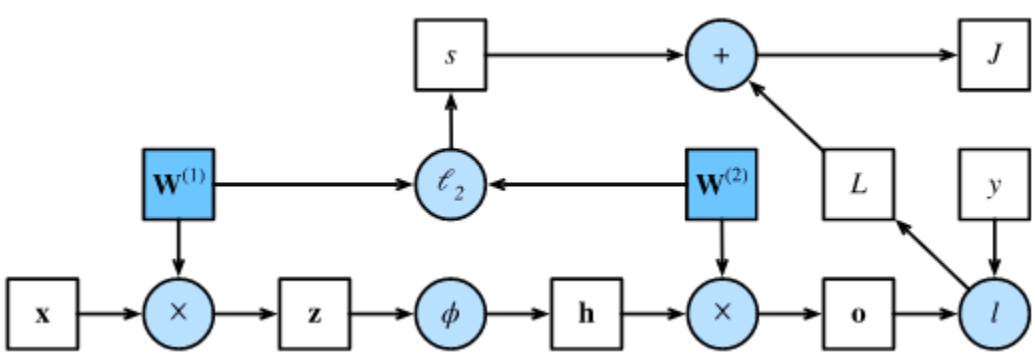
where the Frobenius norm of the matrix is simply the L2 norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

We refer to JJ as the *objective function* in the following discussion.

2.4.2.1.1 Computational Graph of Forward Propagation

Plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation. contains the graph associated with the simple network described above, where squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward.



CH2 8 Computational Graph of Forward Propagation

2.4.2.2 Backpropagation

Backpropagation is the heart of every neural network. Firstly, we need to make a distinction between backpropagation and optimizers.

Backpropagation is for calculating the gradients efficiently, while optimizers is for training the neural network, using the gradients computed with backpropagation. In short, all backpropagation does for us is compute the gradients. Nothing more.

We always start from the output layer and propagate backwards, updating weights and biases for each layer.

The idea is simple: adjust the weights and biases throughout the network, so that we get the desired output in the output layer. Say we wanted the output neuron to be 1.0, then we would need to nudge the weights and biases so that we get an output closer to 1.0.

We can only change the weights and biases, but activations are direct calculations of those weights and biases, which means we indirectly can adjust every part of the neural network, to get the desired output — except for the input layer, since that is the dataset that you input.

2.4.2.2.1 Computing Gradients

Now, before the equations, let's define what each variable means. We have already defined some of them, but it's good to summarize. Some of this should be familiar to you, if you read the post.

Pay attention to the notation used between L, L-1 and *l*. I intentionally mix it up, so that you can get an understanding of how both of them work.

L = Last Layer	w = Weight (connections from one neuron to another)
L-1 = Second Last Layer	
(..etc)	a = Activation (all the neurons)
l = layer	
y = The output value we wish to predict	b = Bias (added value to each activation)
C = Cost Function	$z = w \times a + b$

And now we will show how we calculate the parameters

$$z^{(L)} = w^{(L)} \times a + b$$

$$a^{(L)} = \sigma\left(z^{(L)}\right)$$

$$C = (a^{(L)} - y)^2$$

Then we will show how we update the parameters after backpropagation

$$w^{(l)} = w^{(l)} - \textbf{learning rate} \times \frac{\partial C}{\partial w^{(l)}}$$

$$b^{(l)} = b^{(l)} - \textbf{learning rate} \times \frac{\partial C}{\partial b^{(l)}}$$

2.4.3 Activation Functions

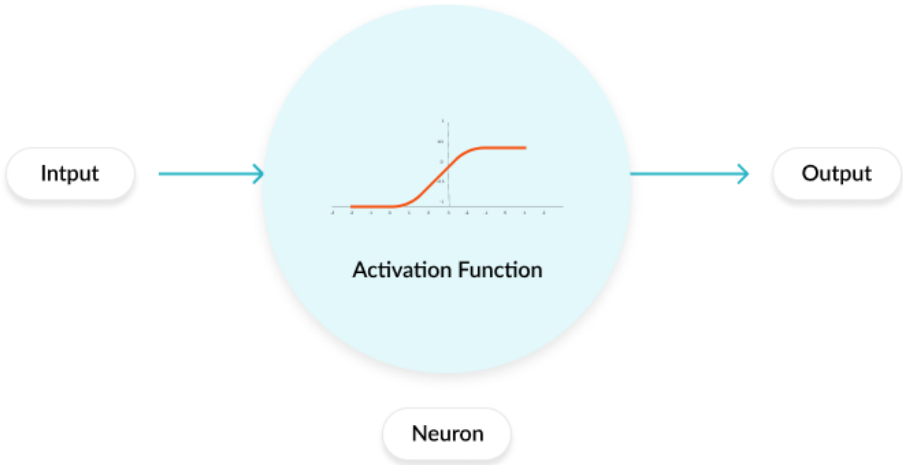
Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.

An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model, which places an increased computational strain on the activation function, and its derivative function.

2.4.3.1 Role of the Activation Function in a Neural Network Model

In a neural network, numeric data points, called inputs, are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

The activation function is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Or it can be a transformation that maps the input signals into output signals that are needed for the neural network to function.



CH2 9 Activation Gate

Increasingly, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

2.4.3.2 Types of Activation Functions

- Binary Step Function
- Linear Activation Function
- Non-Linear Activation Functions

Non-Linear Activation Functions

Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.

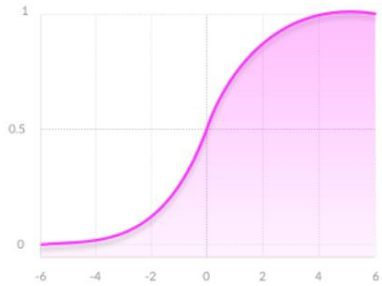
Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear.

Non-linear functions address the problems of a linear activation function:

1) They allow backpropagation because they have a derivative function which is related to the inputs.
2) They allow "stacking" of multiple layers of neurons to create a deep neural network. Multiple hidden layers of neurons are needed to learn complex data sets with high levels of accuracy

2.4.3.3 Common Nonlinear Activation Functions

2.4.3.3.1 Sigmoid / Logistic



CH2 10 Sigmoid activation function

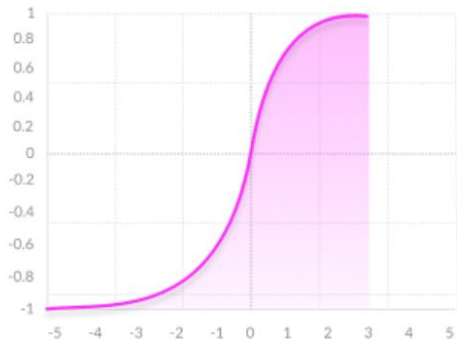
Advantages

- Smooth gradient, preventing "jumps" in output values.
- Output values bound between 0 and 1, normalizing the output of each neuron.
- Clear predictions—For X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

Disadvantages

- Vanishing gradient—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
- Outputs not zero centered.
- Computationally expensive

2.4.3.3.2 TanH / Hyperbolic Tangent



CH2 11 TanH activation function

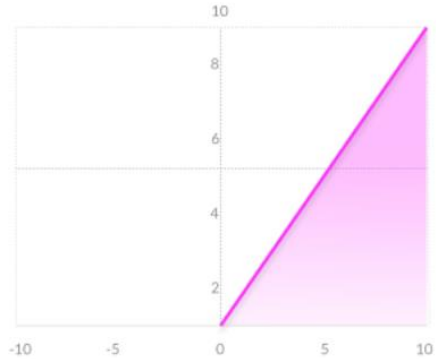
Advantages

- Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
- Otherwise like the Sigmoid function.

Disadvantages

- Like the Sigmoid function

2.4.3.3.3 ReLU (Rectified Linear Unit)



CH2 12 Relu activation function

Advantages

- Computationally efficient—allows the network to converge very quickly
- Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation

Disadvantages

- The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

2.4.3.3.4 SoftMax

$$\sigma(z)j= \frac{ezj}{\sum_{k=1}^{k} ezk} \quad for\ j= 1....k.$$

Advantages

- Able to handle multiple classes only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
- Useful for output neurons—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

2.4.4 Loss Functions

Machines learn by means of a loss function. It's a method of evaluating how well specific algorithm models the given data. If predictions deviate too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction. In this article we will go through several loss functions and their applications in the domain of machine/deep learning.

There's no one-size-fits-all loss function to algorithms in machine learning. There are various factors involved in choosing a loss function for specific problem such as type of machine learning algorithm chosen, ease of calculating the derivatives and to some degree the percentage of outliers in the data set.

Broadly, loss functions can be classified into two major categories depending upon the type of learning task we are dealing with — Regression losses and Classification losses. In classification, we are trying to predict output from set of finite categorical values (i.e., Given large data set of images of hand-written digits) categorizing them into one of 0–9 digits. Regression, on the other hand, deals with predicting a continuous value for example given floor area, number of rooms, size of rooms, predict the price of room.

2.4.4.1 Regression Losses

2.4.4.1.1 Mean Square Error/Quadratic Loss/L2 Loss

As the name suggests, mean square error is measured as the average of squared difference between predictions and actual observations. It's only concerned with the average magnitude of error irrespective of their direction.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

$\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ are the predicted values
$y_1, y_2, \ldots, y_n$ are the observed values
n is the total number of observations/rows in the dataset

However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. Plus, MSE has nice mathematical properties which makes it easier to calculate gradients.

2.4.4.1.2 Root Mean Square Error (RMSE)

It is the standard deviation of the errors which occur when a prediction is made on a dataset. This is the same as MSE (Mean Squared Error) but the root of the value is considered while determining the accuracy of the model.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

2.4.4.1.3 Mean Absolute Error/L1 Loss

Mean absolute error, on the other hand, is measured as the average of sum of absolute differences between predictions and actual observations. Like MSE, this as well measures the magnitude of error without considering their direction.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

Unlike MSE, MAE needs more complicated tools such as linear programming to compute the gradients. Plus MAE is more robust to outliers since it does not make use of square.

2.4.4.1.4 Mean Bias Error

This is much less common in machine learning domain as compared to its counterpart. This is same as MSE with the only difference that we don't take absolute values.

$$MBE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)}{n}$$

Clearly there's a need for caution as positive and negative errors could cancel each other out. Although less accurate in practice, it could determine if the model has positive bias or negative bias.

2.4.4.1.5 Cross Entropy Loss/Negative Log Likelihood

This is the most common setting for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label.

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

Notice that when actual label is 1 (y(i) = 1), second half of function disappears whereas in case actual label is 0 (y(i) = 0) first half is dropped off. In short, we are just multiplying the log of the actual predicted probability for the ground truth class. An important aspect of this is that cross entropy loss penalizes heavily the predictions that are confident but wrong.

2.4.5 Optimizers

Optimizers is how the neural networks learn, using backpropagation to calculate the gradients.

Many factors contribute to how well a model performs. The way we measure performance, as may be obvious to some, is by a cost function.

2.4.5.1 Cost Function

The cost function gives us a value, which we want to optimize. There are too many cost functions to mention them all, but one of the more simple and often used cost functions is the sum of the squared differences.

$$C = 1/n\sum(yi - \hat{}yi)2$$

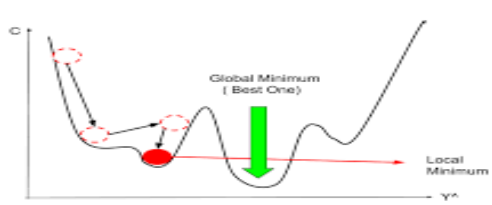
Where y is what we want the output to be and y^ being the actual predicted output from a neural network. Basically, for every sample n, we start summing from the first example i=1 and over all the squares of the differences between the output we want y and the predicted output y^ for each observation.

There are obviously many factors contributing to how well a particular neural network performs. Complexity of model, hyperparameters (learning rate, activation functions etc.), size of dataset and more.

2.4.5.2 Stochastic Gradient Descent

In Stochastic Gradient Descent, we take a mini-batch of random sample and perform an update to weights and biases based on the average gradient from the mini-batch. The weights for each mini-batch is randomly initialized to a small value, such as 0.1. The biases are initialized in many different ways; the easiest one being initialized to 0.

1) Define a cost function, with a vector as input (weight or bias vector)
2) Start at a random point along the x-axis and step in any direction.
   Ask, which way should we step to decrease the cost function most quickly?
3) Calculate the gradient using backpropagation
4) Step in the opposite direction of the gradient — we calculate gradient ascent, therefore we just put a minus in front of the equation or move in the opposite direction, to make it gradient descent.



CH2 13 \Gradient Descent

2.4.5.3 Adam Optimizer

Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the 'gradient descent with momentum' algorithm and the 'RMSP' algorithm.

2.4.5.3.1 How Adam works?

Adam optimizer involves a combination of two gradient descent methodologies:

2.4.5.3.2 Momentum:

This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.

$$w_{t+1} = w_t - \alpha m_t$$

where,

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right]$$

m_t = aggregate of gradients at time t [current] (initially, $m_t = 0$)
m_{t-1} = aggregate of gradients at time t-1 [previous]
W_t = weights at time t
W_{t+1} = weights at time t+1
α_t = learning rate at time t
∂L = derivative of Loss Function
∂W_t = derivative of weights at time t
β = Moving average parameter (const, 0.9)

2.4.5.4 Root Mean Square Propagation (RMSP)

Root mean square prop or RMSprop is an adaptive learning algorithm that tries to improve AdaGrad. Instead of taking the cumulative sum of squared gradients like in AdaGrad, it takes the 'exponential moving average'.

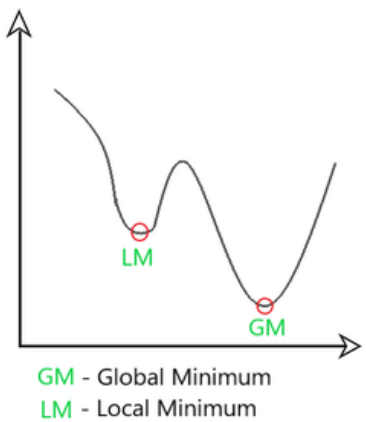
$$w_{t+1} = w_t - \frac{\alpha_t}{(v_t+\varepsilon)^{1/2}} * \left[\frac{\delta L}{\delta w_t}\right]$$

where,

$$v_t = \beta v_{t-1} + (1-\beta) * \left[\frac{\delta L}{\delta w_t}\right]^2$$

W_t = weights at time t
W_{t+1} = weights at time t+1
α_t = learning rate at time t
∂L = derivative of Loss Function
∂W_t = derivative of weights at time t
V_t = sum of square of past gradients. [i.e sum($\partial L/\partial Wt$-1)] (initially, $V_t = 0$)
β = Moving average parameter (const, 0.9)
ϵ = A small positive constant (10^{-8})

Adam Optimizer inherits the strengths or the positive attributes of the above two methods and builds upon them to give a more optimized gradient descent.



CH2 14 Adam Optimizer

Here, we control the rate of gradient descent in such a way that there is minimum oscillation when it reaches the global minimum while taking big enough steps (step-size) so as to pass the local minima hurdles along the way. Hence, combining the features of the above methods to reach the global minimum efficiently.

2.4.6 Model's Problems

2.4.6.1 Overfitting

Overfitting occurs when our model becomes really good at being able to classify or predict on data that was included in the training set, but is not as good at classifying data that it wasn't trained on. So essentially, the model has overfit the data in the training set.

Solutions:

- Adding More Data to The Training Set
- Reduce the Complexity of The Model
- Increase Dropout

2.4.6.2 Underfitting

it is said to be underfitting when it's not even able to classify the data it was trained on, let alone data it hasn't seen before.

Solutions:

1) Increase The Complexity Of The Model:

One thing we can do is increase the complexity of our model. This is the exact opposite of a technique we gave to reduce overfitting. If our data is more complex, and we have a relatively simple model, then the model may not be sophisticated enough to be able to accurately classify or predict on our complex data.

We can increase the complexity of our model by doing things such as:

- Increasing the number of layers in the model.
- Increasing the number of neurons in each layer.
- Changing what type of layers we're using and where.

2) Add More Features To The Input Samples

Another technique we can use to reduce underfitting is to add more features to the input samples in our training set if we can. These additional features may help our model classify the data better.

For example, say we have a model that is attempting to predict the price of a stock based on the last three closing prices of this stock. So our input would consist of three features:

- day 1 close
- day 2 close
- day 3 close

If we added additional features to this data, like, maybe the opening prices for these days, or the volume of the stock for these days, then perhaps this may help our model learn more about the data and improve its accuracy.

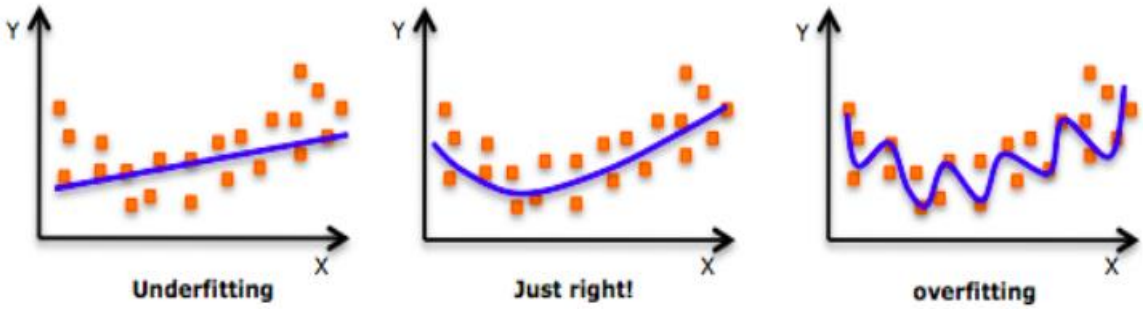
3) Reduce Dropout

2.4.7 Regularization

One of the most common problem data science professionals face is to avoid **overfitting**. When facing a situation where the model performed exceptionally well on training set but, was not able to predict test data.

2.4.7.1 What is Regularization?

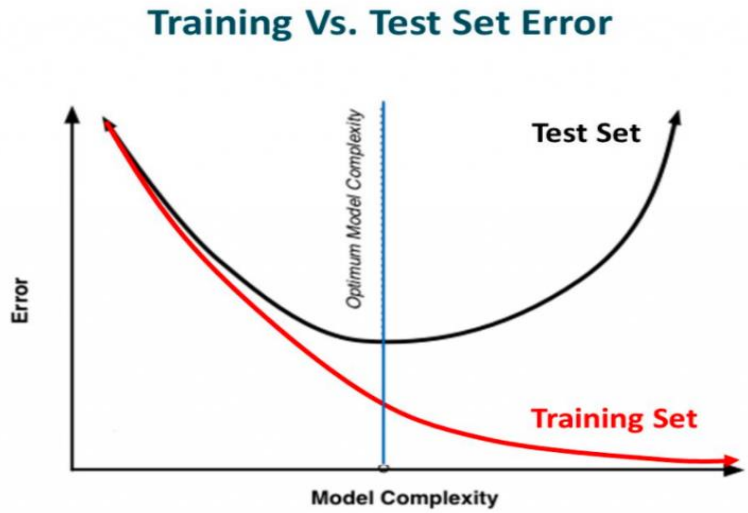
Before we deep dive into the topic, refer to the next figure:



CH2 15 Overfitted models

As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data.

In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.



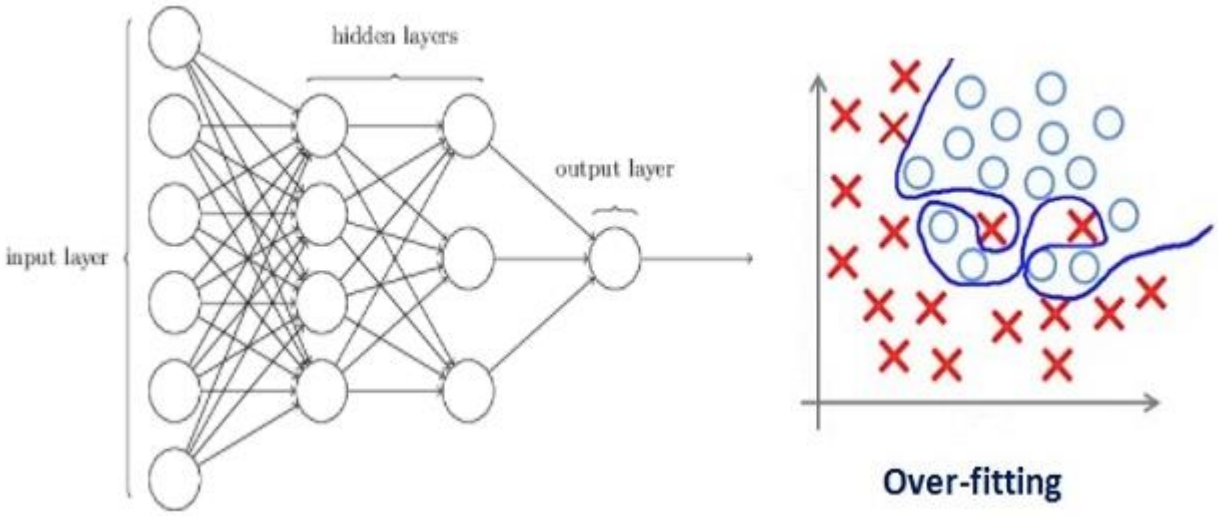
CH2 16 Training set vs. Test set Error

If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.

Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

2.4.7.2 How does Regularization help reduce Overfitting?

Let's consider a neural network which is overfitting on the training data.

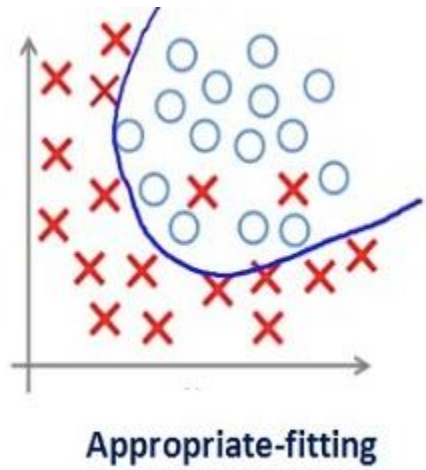


CH2 17 Overfitting in the training set

If you have studied the concept of regularization in machine learning, you will have a fair idea that regularization penalizes the coefficients. In deep learning, it penalizes the weight matrices of the nodes.

Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero. This will result in a much simpler linear network and slight underfitting of the training data.

Such a large value of the regularization coefficient is not that useful. We need to optimize the value of regularization coefficient in order to obtain a well-fitted model as shown in the image below.



CH2 18 Well fitted model

2.4.7.3 Different Regularization Techniques in Deep Learning

Now that we have an understanding of how regularization helps in reducing overfitting, we'll learn a few different techniques in order to apply regularization in deep learning.

2.4.7.3.1 L2 & L1 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2.

In L2, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).

In L1, we have:

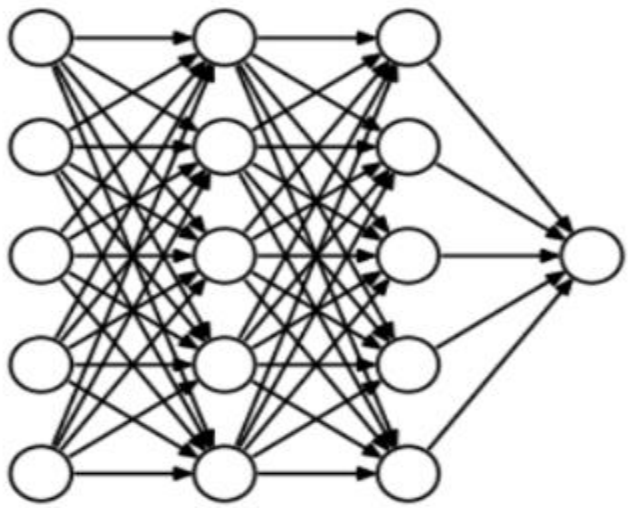
$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. **Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.**

2.4.7.3.2 Dropout

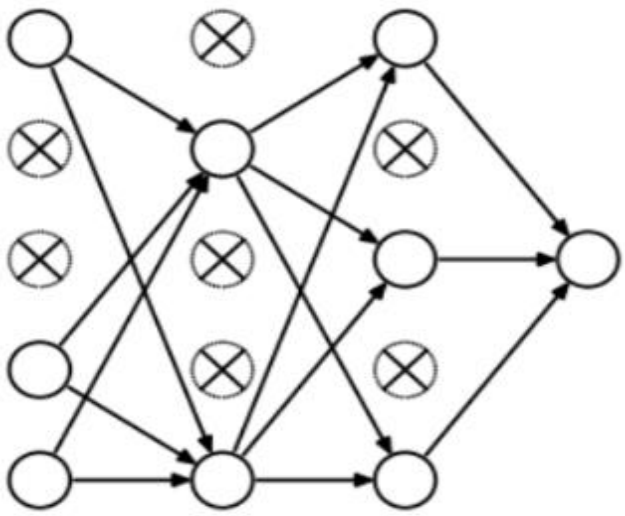
This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

To understand dropout, let's say our neural network structure is akin to the one shown below:



CH2 19 Neural network before applying dropout technique

So, what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.



CH2 20 Neural network after applying dropout technique

So, each iteration has a different set of nodes and this results in a different set of outputs. It can also be thought of as an ensemble technique in machine learning. Ensemble models in machine learning combine the decisions from multiple models to improve the overall performance. This can be achieved in various ways, which you will discover in this article.

This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. Dropout can be applied to both the hidden layers as well as the input layers. Due to these reasons, dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

2.4.8 Model Evaluation

The most important thing that can be done to properly evaluate the model is to not train the model on the entire dataset. A typical train/test split would be to use 70% of the data for training and 30% of the data for testing.

It's important to use new data when evaluating our model to prevent the likelihood of overfitting to the training set. However, sometimes it's useful to evaluate our model as we're building it to find that best parameters of a model - but we can't use the test set for this evaluation or else we'll end up selecting the parameters that perform best on the test data but maybe not the parameters that generalize best. To evaluate the model while still building and tuning the model, we create a third subset of the data known as the validation set. A typical train/test/validation split would be to use 60% of the data for training, 20% of the data for validation, and 20% of the data for testing.

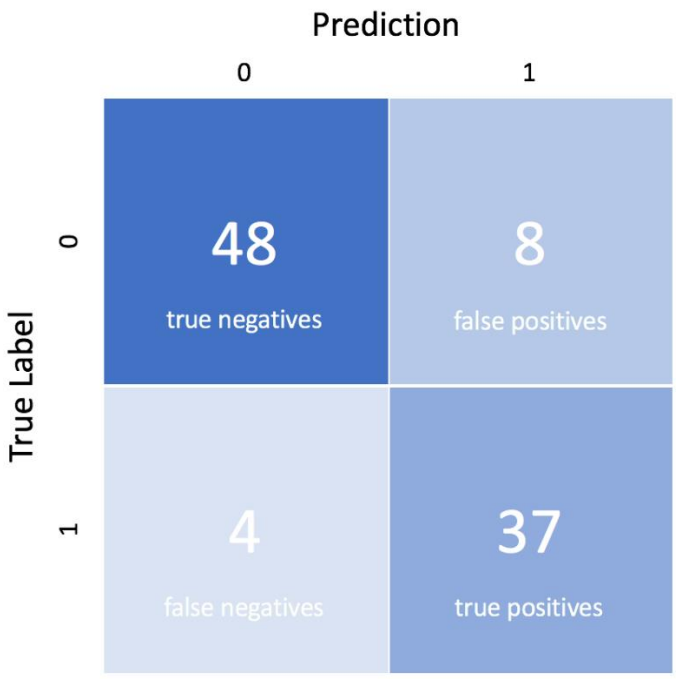
I'll also note that it's very important to shuffle the data before making these splits so that each split has an accurate representation of the dataset.

2.4.8.1 Classification Metrics

When performing classification predictions, there's four types of outcomes that could occur.

- True positives are when you predict an observation belongs to a class and it actually does belong to that class.
- True negatives are when you predict an observation does not belong to a class and it actually does not belong to that class.
- False positives occur when you predict an observation belongs to a class when in reality it does not.
- False negatives occur when you predict an observation does not belong to a class when in fact it does.

These four outcomes are often plotted on a confusion matrix. The following confusion matrix is an example for the case of binary classification. You would generate this matrix after making predictions on your test data and then identifying each prediction as one of the four possible outcomes described above.

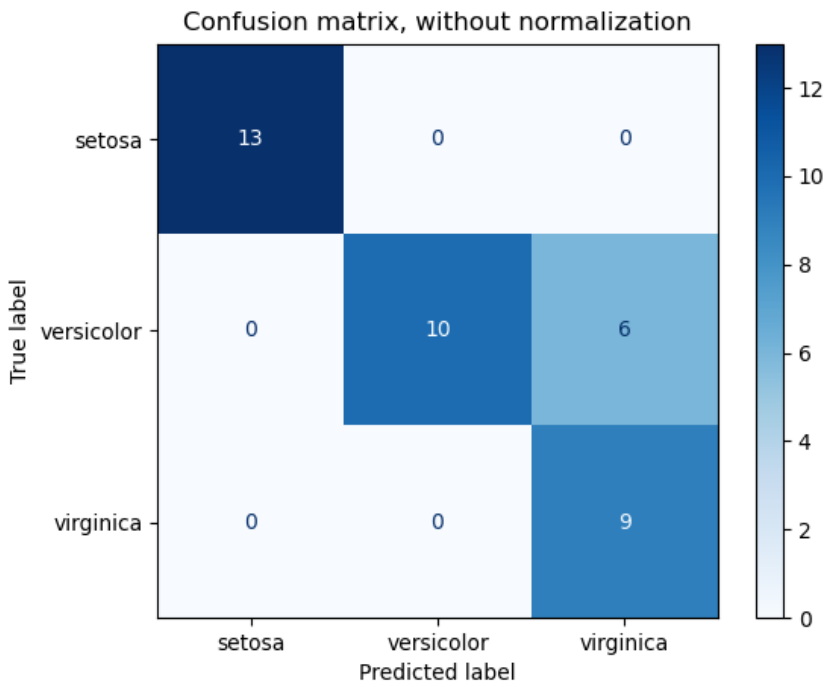


CH2 21 Confusion matrix for binary classification

You can also extend this confusion matrix to plot multi-class classification predictions.

2.4.8.2 Confusion matrix

Example of confusion matrix usage to evaluate the quality of the output of a classifier on the iris data set.



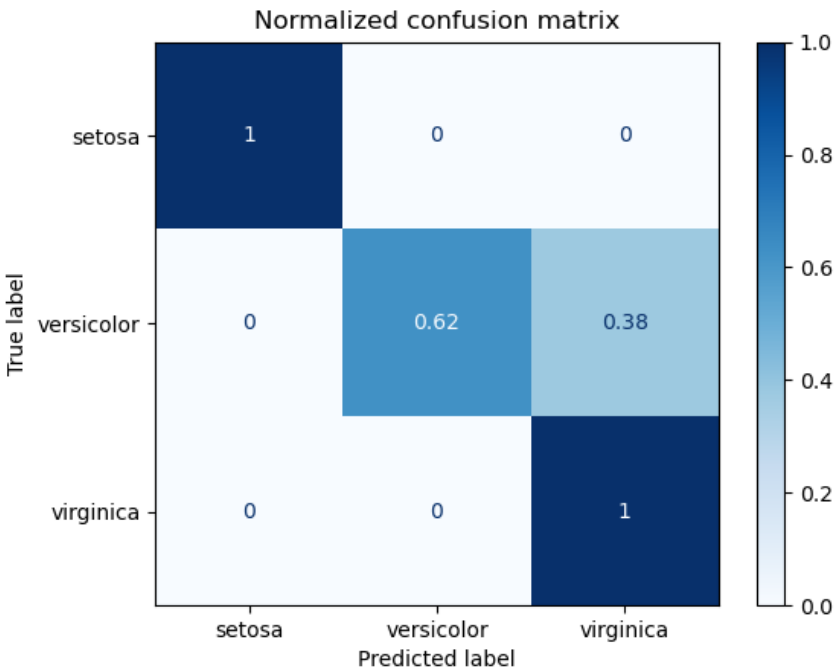
CH2 22 Confusion matrix without normalization

The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions.

The figures show the confusion matrix with and without normalization by class support size (number of elements in each class). This kind of normalization can be interesting in case of class imbalance to have a more visual interpretation of which class is being misclassified.

Here the results are not as good as they could be as our choice for the regularization parameter C was not the best.

```
Confusion matrix, without normalization
[[13 0 0]
[ 0 10 6]
[ 0 0 9]]

Confusion matrix, without normalization
[[1.0 0. 0.]
[ 0. 0.62 0.38]
[ 0. 0. 1.]]
```



CH2 23 Normalized confusion matrix

The three main metrics used to evaluate a classification model are accuracy, precision, and recall.

2.4.8.3 Accuracy

is defined as the percentage of correct predictions for the test data. It can be calculated easily by dividing the number of correct predictions by the number of total predictions.

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{all predictions}}$$

2.4.8.4 Precision

is defined as the fraction of relevant examples (true positives) among all of the examples which were predicted to belong in a certain class.

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

2.4.8.5 Recall

is defined as the fraction of examples which were predicted to belong to a class with respect to all of the examples that truly belong in the class.

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Precision and recall are useful in cases where classes aren't evenly distributed. The common example is for developing a classification algorithm that predicts whether or not someone has a disease. If only a small percentage of the population (let's say 1%) has this disease, we could build a classifier that always predicts that the person does not have the disease, we would have built a model which is 99% accurate and 0% useful.

However, if we measured the recall of this useless predictor, it would be clear that there was something wrong with our model. In this example, recall ensures that we're not overlooking the people who have the disease, while precision ensures that we're not misclassifying too many people as having the disease when they don't. Obviously, you wouldn't want a model that incorrectly predicts a person has cancer (the person would end up in a painful and expensive treatment process for a disease they didn't have) but you also don't want to incorrectly predict a person does not have cancer when in fact they do. Thus, it's important to evaluate both the precision and recall of a model.

Ultimately, it's nice to have one number to evaluate a machine learning model just as you get a single grade on a test in school. Thus, it makes sense to combine the precision and recall metrics; the common approach for combining these metrics is known as the **f-score**.

$$F_\beta = \left(1 + \beta^2\right) \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

The β parameter allows us to control the tradeoff of importance between precision and recall. β<1 focuses more on precision while β>1 focuses more on recall.

2.4.8.6 Receiver Operating Characteristics (ROC)

is another common tool used to understand a model's performance is a curve. A ROC curve visualizes an algorithm's ability to discriminate the positive class from the rest of the data.

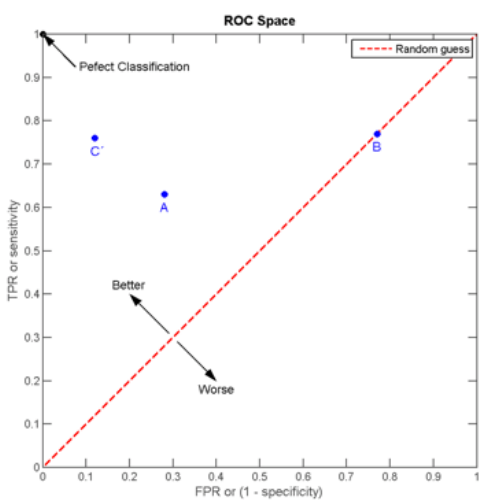
$$\text{TPR} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{FPR} = \frac{\text{false positives}}{\text{false positives} + \text{true negatives}}$$

This can be done by plotting the True Positive Rate against the False Positive Rate for varying prediction thresholds.

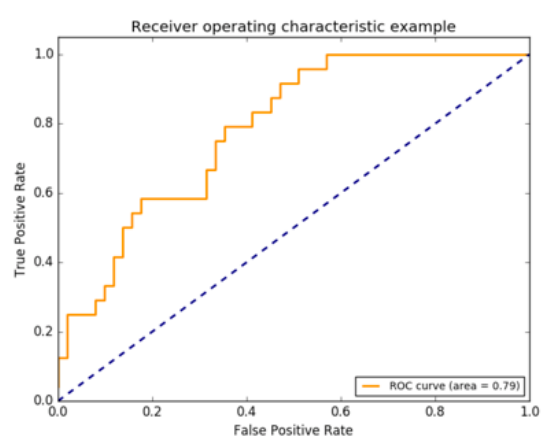
For classifiers which only produce factor outcomes (i.e., directly output a class), there exists a fixed TPR and FPR for a trained model. However, other classifiers, such as logistic regression, has the capability of giving a probabilistic output (i.e., the chance that a given observation belongs to the positive class). For these classifiers, we can specify the probability threshold by which above that amount we'll predict the observation belongs to the positive class.

Classifiers with factor output



* Where A, B, and C are different models

Classifiers with probabilistic output



The behavior of the classifier can be adjusted to achieve a performance anywhere on the curve by adjusting the probability threshold by which we predict the positive class.

The area under the curve (AUC) is a single-value metric for which attempts to summarize a ROC curve to evaluate the quality of a classifier. As the name implies, this metric approximates the area under the ROC curve for a given classifier. Recall that the ideal curve hugs the upper left-hand corner as closely as possible, giving us the ability to identify all true positives while avoiding false positives; this ideal model would have an AUC of 1.

2.4.8.7 R-Squared

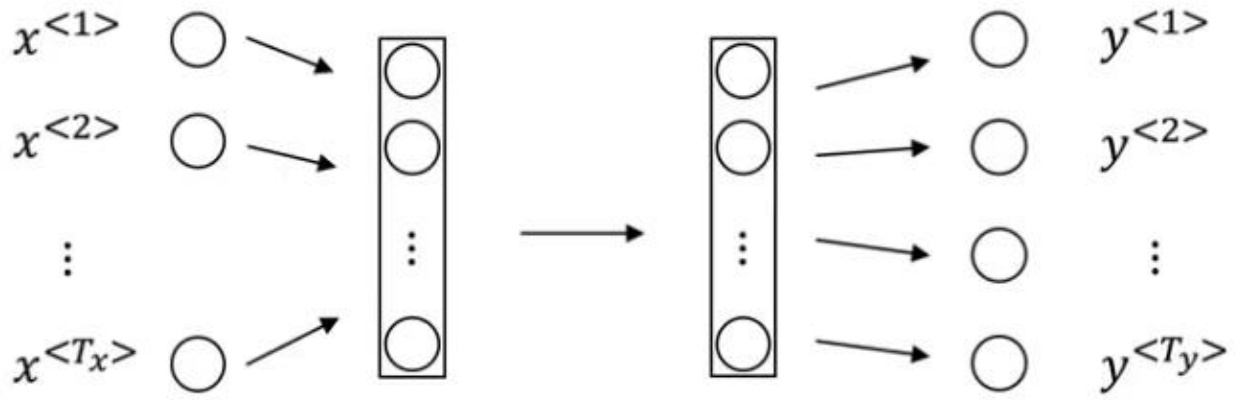
It is also known as the coefficient of determination. This metric gives an indication of how good a model fits a given dataset. It indicates how close the regression line (i.e., the predicted values plotted) is to the actual data values. The R squared value lies between 0 and 1 where 0 indicates that this model doesn't fit the given data and 1 indicates that the model fits perfectly to the dataset provided.

2.5 Sequence models

Sequence models, in supervised learning, can be used to address a variety of applications including financial time series prediction, speech recognition, music generation, sentiment classification, machine translation and video activity recognition. The only constraint is that either the input or the output is a sequence. In other words, you may use sequence models to address any type of supervised learning problem which contains a time series in either the input or output layers. Traditional feedforward neural networks do not share features across different positions of the network. In other words, these models assume that all inputs (and outputs) are independent of each other. This model would not work in sequence prediction since the previous inputs are inherently important in predicting the next output. For example, if you were predicting the next word in a stream of text, you would want to know at least a couple of words before the target word. Traditional neural networks require the input and output sequence lengths to be constant across all predictions. sequence model networks can directly address this problem.

2.5.1 Recurrent Neural Network (RNN)

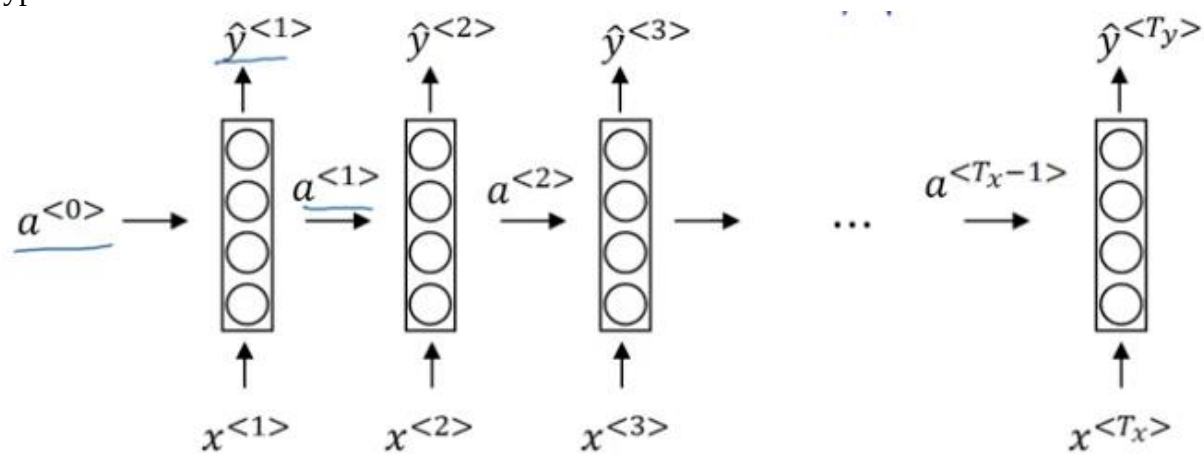
We use Recurrent Neural Networks to learn mapping from X to Y, when either X or Y, or both X and Y, are some sequences. But why can't we just use a standard neural network for these sequence problems? Suppose we build the below neural network:



CH2 25 Before RNN

There are primarily two problems with this:

Inputs and outputs do not have a fixed length, i.e., some input sentences can be of 10 words while others could be <> 10. The same is true for the eventual output We will not be able to share features learned across different positions of text if we use a standard neural network We need a representation that will help us to parse through different sentence lengths as well as reduce the number of parameters in the model. This is where we use a recurrent neural network. This is how a typical RNN looks like:



CH2 26 RNN Model

A RNN takes the first word (x<1>) and feeds it into a neural network layer which predicts an output (y'<1>). This process is repeated until the last time step x<Tx> which generates the last output y'<Ty>. This is the network where the number of words in input as well as the output are same. The RNN scans through the data in a left to right sequence. Note that the parameters that the RNN uses for each time step are shared. We will have parameters shared between each input and hidden layer (Wax), every timestep (Waa) and between the hidden layer and the output (Wya). So if we are making predictions for x<3>, we will also have information about x<1> and x<2>. A potential weakness of RNN is that it only takes information from the previous timesteps and not from the ones that come later. This problem can be solved using bi-directional RNNs which we will discuss later. For now, let's look at forward propagation steps in a RNN model:

a<0> is a vector of all zeros and we calculate the further activations similar to that of a standard neural network:

$$a < 0 > \ = \ 0$$
$$a < 1 > \ = \ g(Waa * a < 0 > \ + \ Wax * x < 1 > \ + \ ba)$$
$$y < 1 > \ = \ g'(Wya * a < 1 > \ + \ by)$$

Similarly, we can calculate the output at each time step. The generalized form of these formulae can be written as:

$$a^{<t>} = g(W_{aa}a^{<t>} + W_{ax}x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

We can write these equations in an even simpler way:

$$a < T > \ = \ g(Wa\,[a < T - 1 >, x < T >] \ + \ by)$$

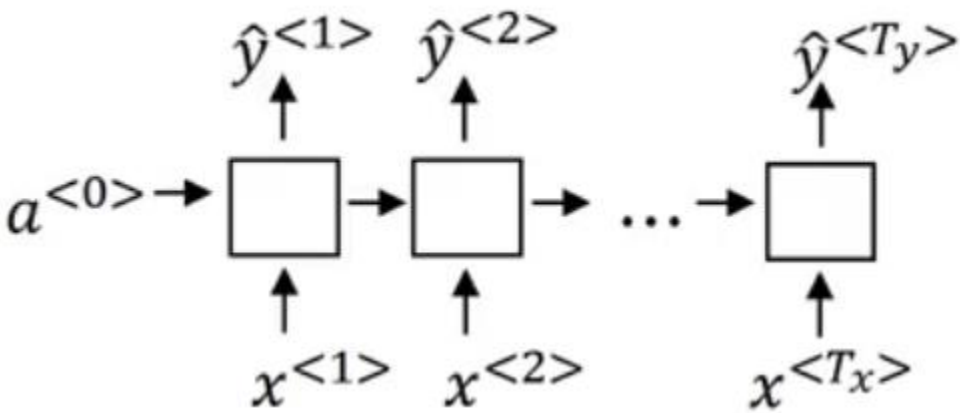
We horizontally stack Waa and Wya to get $Wa.a < t - 1 >$ and $x < t >$ are stacked vertically. Rather than carrying around 2 parameter matrices, we now have just 1 matrix. And that, in a nutshell, is how forward propagation works for recurrent neural networks.

2.5.2 Different types of RNNs

We can have different types of RNNs to deal with use cases where the sequence length differs. These problems can be classified into the following categories:

2.5.2.1 Many-to-many

The name entity recognition examples we saw earlier fall under this category. We have a sequence of words, and for each word, we have to predict whether it is a name or not. The RNN architecture for such a problem looks like this:

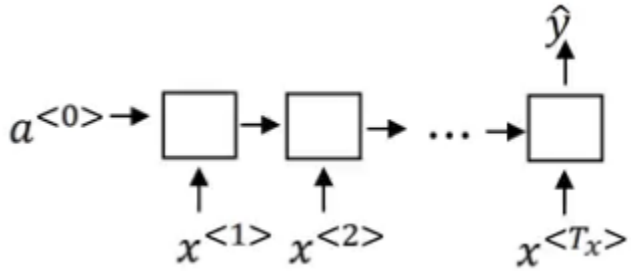


CH2 27 RNN-Many to Many

For every input word, we predict a corresponding output word.

2.5.2.2 Many-to-one

Consider the sentiment classification problem. We pass a sentence to the model and it returns the sentiment or rating corresponding to that sentence. This is a many-to-one problem where the input sequence can have varied length, whereas there will only be a single output. The RNN architecture for such problems will look something like this:

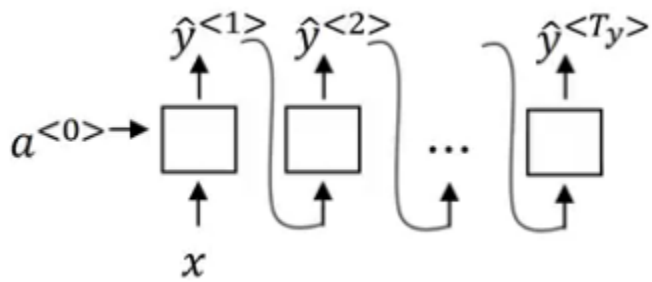


CH2 28 RNN-Many to One

Here, we get a single output at the end of the sentence.

2.5.2.3 One-to-many

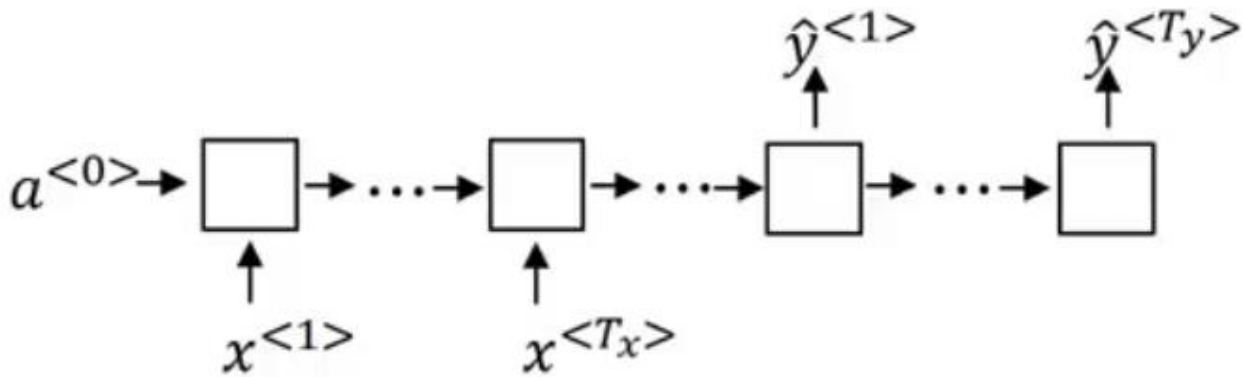
Consider the example of music generation where we want to predict the lyrics using the music as input. In such scenarios, the input is just a single word (or a single integer), and the output can be of varied length. The RNN architecture for this type of problems looks like the below:



CH2 29 RNN-One to Many

There is one more type of RNN which is popularly used in the industry. Consider the machine translation application where we take an input sentence in one language and translate it into another language. It is a many-to-many problem but the length of the input sequence might or might not be equal to the length of output sequence.

In such cases, we have an encoder part and a decoder part. The encoder part reads the input sentence, and the decoder translates it to the output sentence:



CH2 30 RNN-Many to Many Encoder-Decoder

2.5.3 Backpropagation through time

Backpropagation is a training algorithm that is used for training neural networks. When training a neural network, we are actually tuning the weights of the network to minimize the error with respect to the already available true values(labels) by using the Backpropagation algorithm. It is a supervised learning algorithm as we find errors with respect to already given labels. The general algorithm is as follows:

Present a training input pattern and propagate it through the network to get an output.

Compare the predicted outputs to the expected outputs and calculate the error.

Calculate the derivatives of the error with respect to the network weights.

Use these calculated derivatives to adjust the weights to minimize the error.

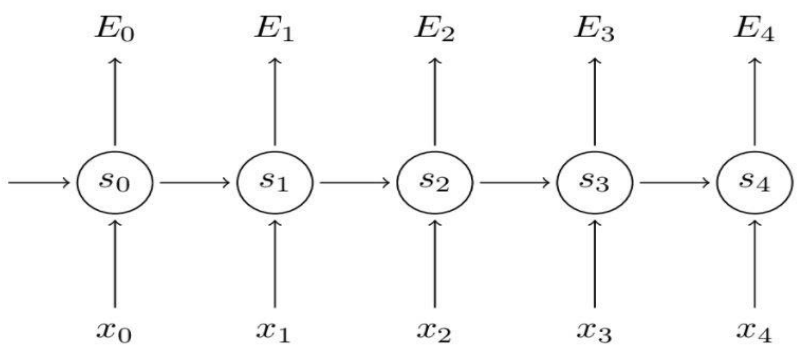
Repeat.

Now the questions that should come to your mind are:

What is the total loss for this network?

How do we update the weights Whh ,Wxh and Why?

The total loss is simply the sum of the loss's overall timestamps. For example, in the figure below, En is the loss at each time stamp and instead of h to denote cell state



In general, we can say the total error is equal to summation over all the errors across timestamps t,

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

Where $Et(yt, yt)$ is the error made at timestamp t. Now we have to calculate the gradient of this loss with respect to Whh ,Wxh and Why. For simplicity I am going to denote Whh as ,Wxh as U and Why .as V. It is quite easy to calculate the derivative of loss with respect to V as it only depends on the values at the current time step. But when it comes to calculating the derivative of loss with respect to W and U, it becomes a bit tricky.

$$\frac{\partial\ (\mathrm{E})}{\partial W} = \frac{\partial\ (\mathrm{E})}{\partial s_4}\frac{\partial s_4}{\partial W}$$

The first part of this equation, i.e., derivative of E w.r.t s4 can be easily calculated as it only depends on V. But to calculate the derivative of s4 with respect to W is not so easy.

$$s_4 = \sigma(Ws_3 + b)$$

The value of s4depends on s3 which itself is a function W, therefore we cannot calculate the derivative of s4 keeping s3 as constant. In such networks, the derivative has two parts, implicit and explicit. In explicit part, we assume all other inputs as constant whereas in implicit part we sum over all the indirect paths. Therefore, we calculate the derivative as:

$$\begin{aligned}
\frac{\partial s_4}{\partial W} &= \underbrace{\frac{\partial^+ s_4}{\partial W}}_{\text{explicit}} + \underbrace{\frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial W}}_{\text{implicit}} \\
&= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\Big[\underbrace{\frac{\partial^+ s_3}{\partial W}}_{\text{explicit}} + \underbrace{\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W}}_{\text{implicit}}\Big] \\
&= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\Big[\frac{\partial^+ s_2}{\partial W} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W}\Big] \\
&= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial^+ s_2}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\Big[\frac{\partial^+ s_1}{\partial W}\Big]
\end{aligned}$$

For simplicity we will short-circuit some of the paths

$$\frac{\partial s_4}{\partial W} = \frac{\partial s_4}{\partial s_4}\frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_2}\frac{\partial^+ s_2}{\partial W} + \frac{\partial s_4}{\partial s_1}\frac{\partial^+ s_1}{\partial W} = \sum_{k=1}^{4}\frac{\partial s_4}{\partial s_k}\frac{\partial^+ s_k}{\partial W}$$

Now we can easily put this value in the previous formula to calculate the derivative of E with respect to W. In a similar manner, you can also calculate the derivative of E with respect to U. Now that we have all the three derivatives, we can easily update our weights. This algorithm is called backpropagation through time or BPTT for short as we used values across all the timestamps to calculate the gradients.

2.5.3.1 Limitations of backpropagation through time:

When using BPTT (backpropagation through time) in RNN, we generally encounter problems such as exploding gradient and vanishing gradient.

To avoid exploding gradient, we simply use a method called gradient clipping where at each timestamp, we can check if the gradient > threshold and if it is, we normalize it. This helps to tackle exploding gradient.

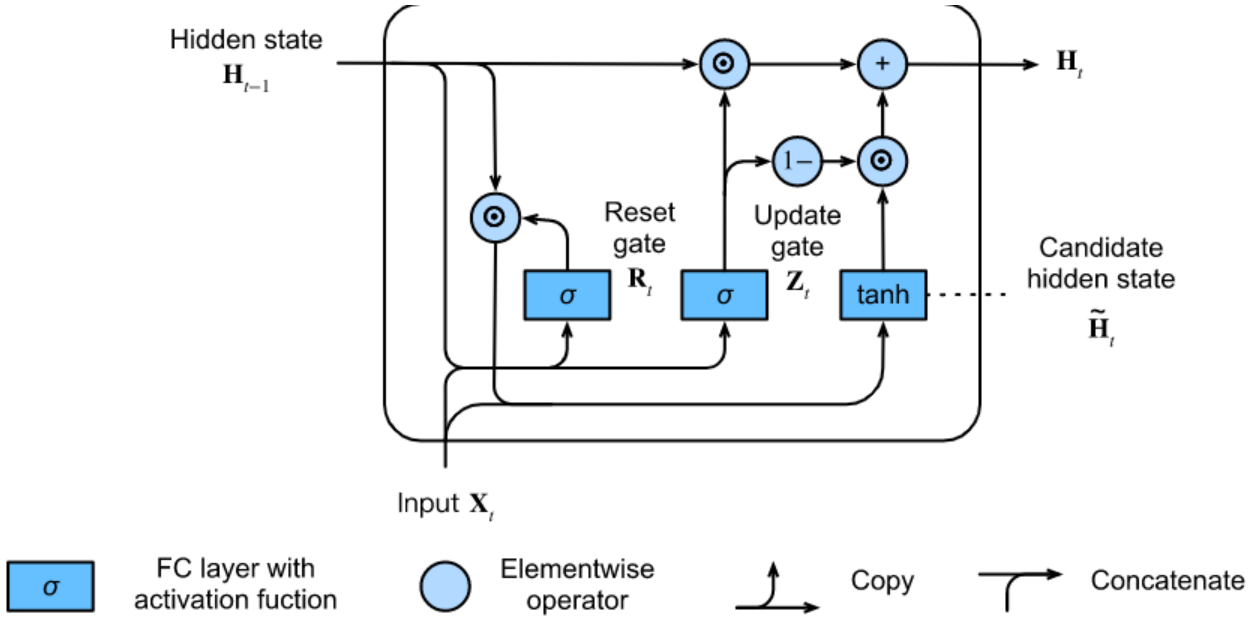
2.6 Modern Sequence Models

We have introduced the basics of RNNs, which can better handle sequence data. For demonstration, we implemented RNN-based language models on text data. However, such techniques may not be sufficient for practitioners when they face a wide range of sequence learning problems nowadays.

We will begin by introducing two of such widely-used networks, namely gated recurrent units (GRUs) and long short-term memory (LSTM). Furthermore, we will expand the RNN architecture with a single unidirectional hidden layer that has been discussed so far. We will describe deep architectures with multiple hidden layers, and discuss the bidirectional design with both forward and backward recurrent computations.

2.6.1 GRU

The key distinction between vanilla RNNs and GRUs is that the latter support gating of the hidden state. This means that we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset. These mechanisms are learned and they address the concerns listed above. For instance, if the first token is of great importance, we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed.



CH2 31 GRU Architecture

2.6.1.1 Reset Gate and Update Gate

The first thing we need to introduce are the reset gate [R] and the update gate [Z]. We engineer them to be vectors with entries in (0,1) such that we can perform convex combinations. For instance, a reset gate would allow us to control how much of the previous state we might still want to remember. Likewise, an update gate would allow us to control how much of the new state is just a copy of the old state.

$$Rt = \sigma(XtWxr + Ht - 1Whr + br),$$
$$Zt = \sigma(XtWxz + Ht - 1Whz + bz)$$

where $Wxr, Wxz \in Rd \times h$ and $Whr, Whz \in Rh \times h$ are weight parameters and $br, bz \in R1 \times h$ are biases. We use sigmoid to transform input values to the interval (0,1).

2.6.1.2 Candidate Hidden State

The intermediate memory unit or candidate hidden state combines the information from the previous hidden state with the input.

$$H{\sim}t = tanh(XtWxh + (Rt \odot Ht - 1)Whh + bh)$$

where $Wxh \in Rd \times h$ and $Whh \in R\ h \times h$ are weight parameters $bh \in R1 \times h$ is the bias, and the symbol $\odot$ is the Hadamard (elementwise) product operator. Here we use a nonlinearity in the form of tanh to ensure that the values in the candidate hidden state remain in the interval (−1,1).

2.6.1.3 Hidden State

Finally, we need to incorporate the effect of the update gate Zt. This determines the extent to which the new hidden state $Ht \in Rn \times h$ is just the old state $Ht - 1$ and by how much the new candidate state $H{\sim}t$ is used. The update gate Zt can be used for this purpose, simply by taking elementwise convex combinations between both $Ht - 1\ and\ H{\sim}t$. This leads to the final update equation for the GRU:

$$Ht = Zt \odot Ht - 1 + (1 - Zt) \odot H{\sim}t.$$

Whenever the update gate Zt is close to 1, we simply retain the old state. In this case the information from Xt is essentially ignored, effectively skipping time step t in the dependency chain. In contrast, whenever Zt is close to 0, the new latent state Ht approaches the candidate latent state $H{\sim}t$. These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances. For instance, if the update gate has been closed to 1 for all the time steps of an entire subsequence, the old hidden state at the time step of its beginning will be easily retained and passed to its end, regardless of the length of the subsequence.

2.6.2 Long Short-Term Memory (LSTM):

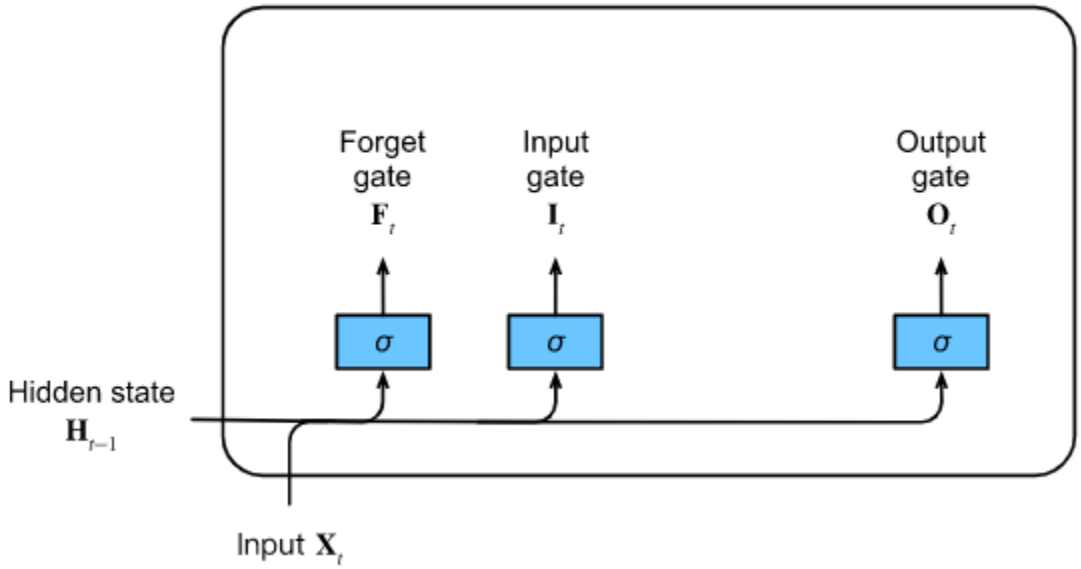
The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time. One of the earliest approaches to address this was the long short-term memory (LSTM) [Hochreiter & Schmidhuber, 1997]. It shares many of the properties of the GRU. Interestingly, LSTMs have a slightly more complex design than GRUs but predates GRUs by almost two decades.

2.6.2.1 Gated Memory Cell

Arguably LSTM's design is inspired by logic gates of a computer. LSTM introduces a memory cell (or cell for short) that has the same shape as the hidden state (some literatures consider the memory cell as a special type of the hidden state), engineered to record additional information. To control the memory cell we need a number of gates. One gate is needed to read out the entries from the cell. We will refer to this as the output gate. A second gate is needed to decide when to read data into the cell. We refer to this as the input gate. Last, we need a mechanism to reset the content of the cell, governed by a forget gate. The motivation for such a design is the same as that of GRUs, namely to be able to decide when to remember and when to ignore inputs in the hidden state via a dedicated mechanism. Let us see how this works in practice.

2.6.2.2 Input Gate, Forget Gate, and Output Gate

Just like in GRUs, the data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Fig. 9.2.1. They are processed by three fully-connected layers with a sigmoid activation function to compute the values of the input, forget. And output gates. As a result, values of the three gates are in the range of (0, 1).



CH2 32 Input Gate, Forget Gate, and Output Gate

Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $Xt \in Rn \times d$ and the hidden state of the previous time step is $Ht - 1 \in Rn \times h$. Correspondingly, the gates at time step t are defined as follows: the input gate is $It \in Rn \times h$, the forget gate is $Ft \in Rn \times h$, and the output gate is $Ot \in Rn \times h$. They are calculated as follows:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),$$

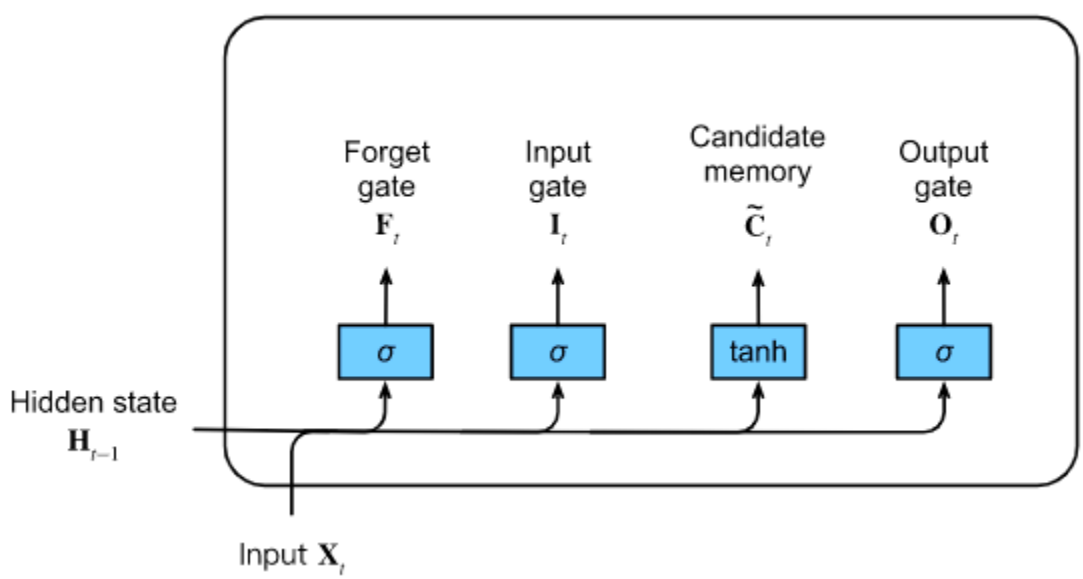
where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ are bias parameters.

2.6.2.3 Candidate Memory Cell

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the candidate memory cell $C{\sim}t \in Rn \times h$. Its computation is similar to that of the three gates described above, but using a tanh function with a value range for (−1, 1) as the activation function. This leads to the following equation at time step t:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter.



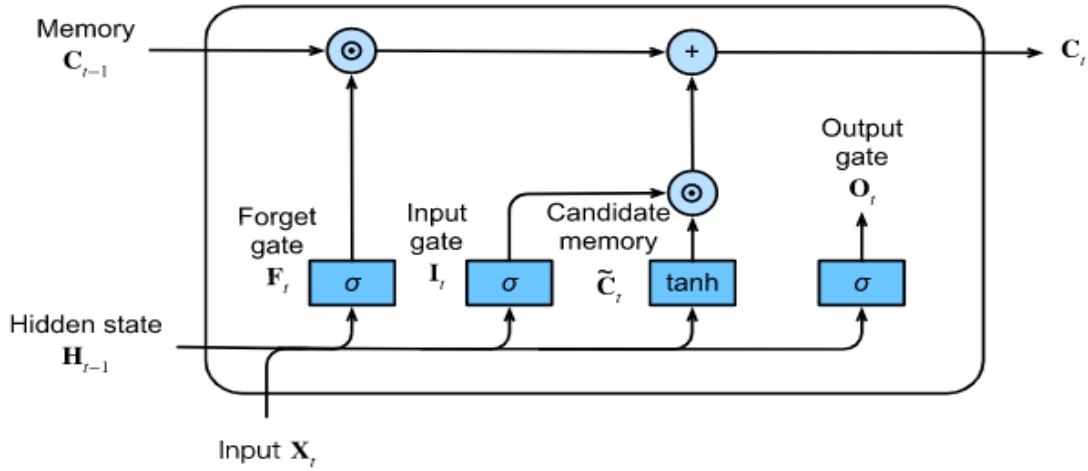
CH2 33 Candidate Memory Cell

2.6.2.4 Memory Cell

In GRUs, we have a mechanism to govern input and forgetting (or skipping). Similarly, in LSTMs we have two dedicated gates for such purposes: the input gate It governs how much we take new data into account via C~t and the forget gate Ft addresses how much of the old memory cell content $Ct - 1 \in Rn \times h$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells ***Ct−1*** will be saved over time and passed to the current time step. This design is introduced to alleviate the vanishing gradient problem and to better capture long range dependencies within sequences.



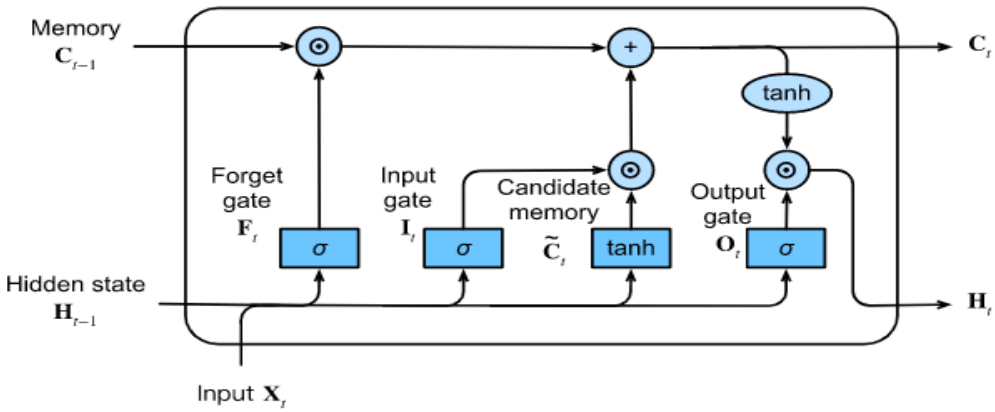
CH2 34 Memory Cell

2.6.2.5 Hidden State

Last, we need to define how to compute the hidden state $\boldsymbol{Ht} \in \boldsymbol{Rn} \times \boldsymbol{h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the ***tanh*** of the memory cell. This ensures that the values of $\boldsymbol{Ht}$ are always in the interval (−1, 1).

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

Whenever the output gate approximates 1 we effectively pass all memory information through to the predictor, whereas for the output gate close to 0 we retain all the information only within the memory cell and perform no further processing.



CH2 35 Hidden State

2.6.3 Bidirectional Recurrent Neural Networks:

In sequence learning, so far, we assumed that our goal is to model the next output given what we have seen so far, e.g., in the context of a time series or in the context of a language model. While this is a typical scenario, it is not the only one we might encounter. To illustrate the issue, consider the following three tasks of filling in the blank in a text sequence:

I am ___.

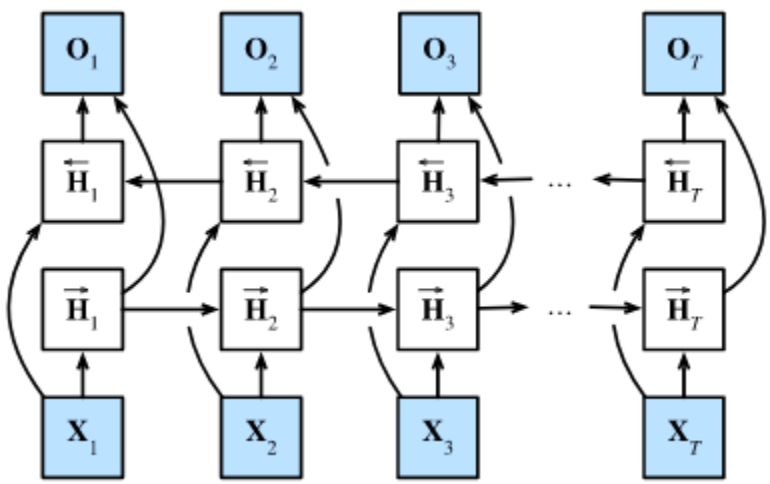
I am ___ hungry.

I am ___ hungry, and I can eat half a pig.

Depending on the amount of information available, we might fill in the blanks with very different words such as "happy", "not", and "very". Clearly the end of the phrase (if available) conveys significant information about which word to pick. A sequence model that is incapable of taking advantage of this will perform poorly on related tasks. For instance, to do well in named entity recognition (e.g., to recognize whether "Green" refers to "Mr. Green" or to the color) longer-range context is equally vital.

2.6.3.1 Bidirectional Model

We need to modify the RNN design that we have seen so far. Fortunately, this is easy conceptually. Instead of running an RNN only in the forward mode starting from the first token, we start another one from the last token running from back to front. Bidirectional RNNs add a hidden layer that passes information in a backward direction to more flexibly process such information.



CH2 36 Bidirectional Model

For any time step $\boldsymbol{t}$, given a minibatch input $\boldsymbol{Xt} \in \boldsymbol{Rn} \times \boldsymbol{d}$ (number of examples: $\boldsymbol{n}$, number of inputs in each example: $\boldsymbol{d}$) and let the hidden layer activation function be $\boldsymbol{\phi}$. In the bidirectional architecture, we assume that the forward and backward hidden states for this time step are $\boldsymbol{H} \rightarrow \boldsymbol{t} \in \boldsymbol{Rn} \times \boldsymbol{h}$ and $\boldsymbol{H} \leftarrow \boldsymbol{t} \in \boldsymbol{Rn} \times \boldsymbol{h}$, respectively, where $\boldsymbol{h}$ is the number of hidden units. The forward and backward hidden state updates are as follows:

$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),$$

where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all the model parameters.

Next, we concatenate the forward and backward hidden states $\boldsymbol{H} \rightarrow \boldsymbol{t}$ and $\boldsymbol{H} \leftarrow \boldsymbol{t}$ to obtain the hidden state $\boldsymbol{Ht} \in \boldsymbol{Rn} \times \boldsymbol{2h}$ to be fed into the output layer. In deep bidirectional RNNs with multiple hidden layers, such information is passed on as input to the next bidirectional layer. Last, the output layer computes the output $\boldsymbol{Ot} \in \boldsymbol{Rn} \times \boldsymbol{q}$ (number of outputs: $\boldsymbol{q}$):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

2.7 Beam Search

In natural language processing tasks such as caption generation, text summarization, and machine translation, the prediction required is a sequence of words.

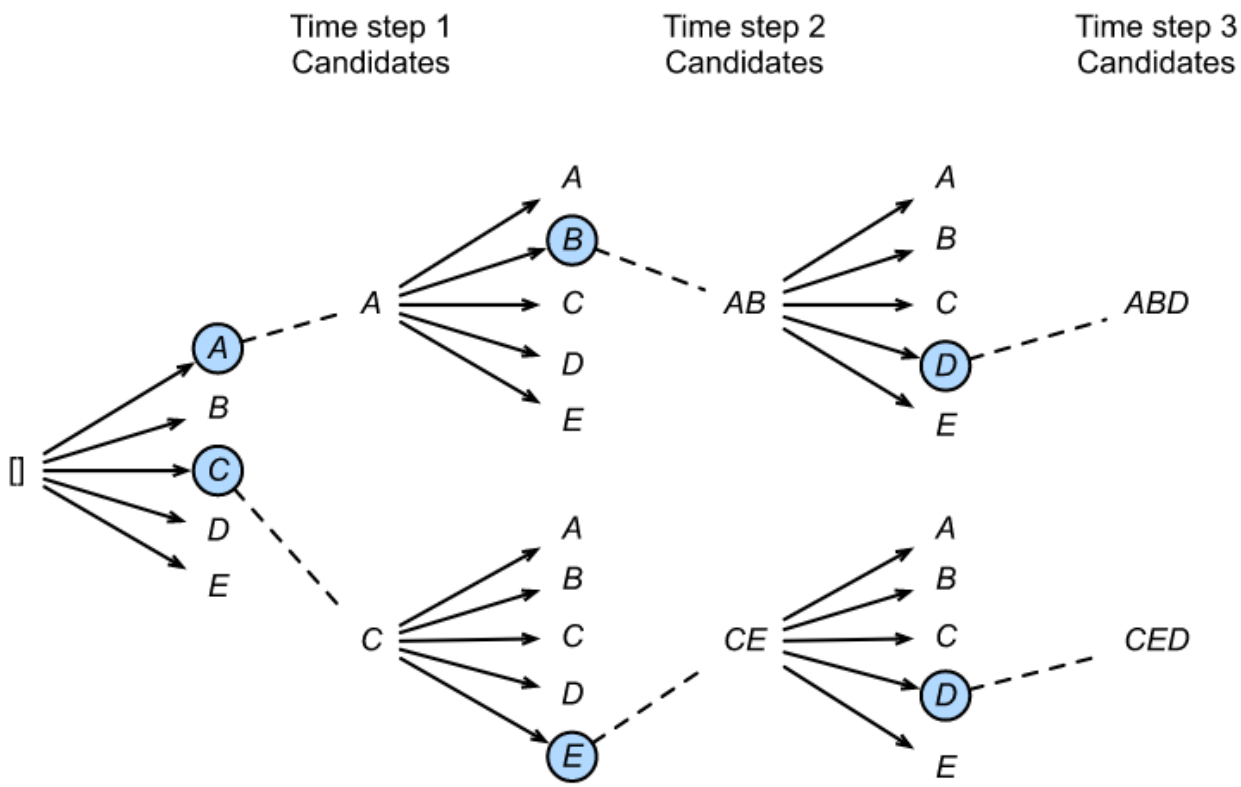
It is common for models developed for these types of problems to output a probability distribution over each word in the vocabulary for each word in the output sequence. It is then left to a decoder process to transform the probabilities into a final sequence of words.

You are likely to encounter this when working with recurrent neural networks on natural language processing tasks where text is generated as an output. The final layer in the neural network model has one neuron for each word in the output vocabulary and a SoftMax activation function is used to output a likelihood of each word in the vocabulary being the next word in the sequence.

Decoding the most likely output sequence involves searching through all the possible output sequences based on their likelihood. The size of the vocabulary is often tens or hundreds of thousands of words, or even millions of words. Therefore, the search problem is exponential in the length of the output sequence and is intractable (NP-complete) to search completely.

In practice, heuristic search methods are used to return one or more approximate or “good enough” decoded output sequences for a given prediction.

“As the size of the search graph is exponential in the source sentence length, we have to use approximations to find a solution efficiently.” [Handbook of Natural Language Processing and Machine Translation]



CH2 37 Beam Search

Beam search is an improved version of greedy search. It has a hyperparameter named beam size, k. At time step 1, we select k tokens with the highest conditional probabilities. Each of them will be the first token of k candidate output sequences, respectively. At each subsequent time step, based on the k candidate output sequences at the previous time step, we continue to select kk candidate output sequences with the highest conditional probabilities from k|Y|. possible choices.

$$\frac{\mathbf{1}}{\mathrm{L}\alpha}\mathrm{logP}(\mathrm{y}\mathbf{1},\ldots,\mathrm{yL}) = \frac{\mathbf{1}}{\mathrm{L}\alpha}\sum_{t'=1}^{L}\mathrm{logP}(\mathrm{yt}' \mid \mathrm{y}\mathbf{1},\ldots,\mathrm{yt}'-\mathbf{1},\mathrm{c})$$

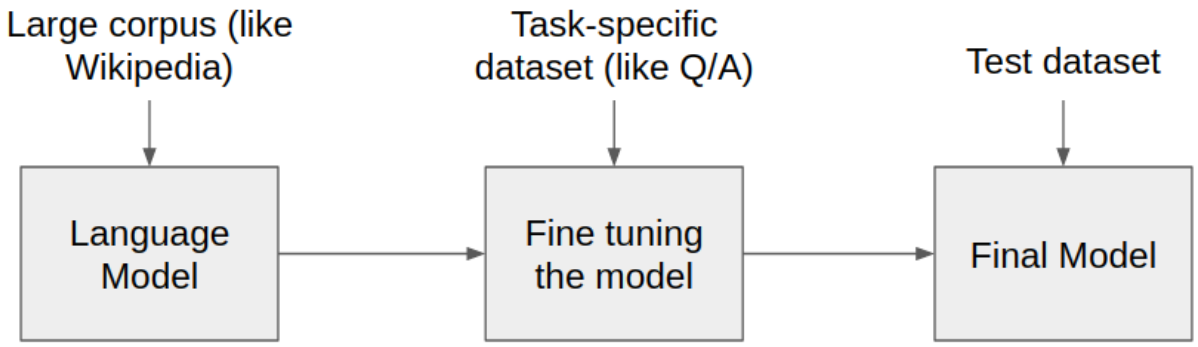
The computational cost of beam search is $O(k|Y|T')$. This result is in between that of greedy search and that of exhaustive search. In fact, greedy search can be treated as a special type of beam search with a beam size of 1. With a flexible choice of the beam size, beam search provides a tradeoff between accuracy versus computational cost.

2.8 Pre-training

in AI refers to training a model with one task to help it form parameters that can be used in other tasks.

The concept of pre-training is inspired by human beings. Thanks to an innate ability, we do not have to learn everything from scratch. Instead, we transfer and reuse our old knowledge of what we have learned in the past to understand new knowledge and handle a variety of new tasks.

In AI, pre-training imitates the way human beings process new knowledge. That is: using model parameters of tasks that have been learned before to initialize the model parameters of new tasks. In this way, the old knowledge helps new models successfully perform new tasks from old experience instead of from scratch. so The intuition behind pre-trained language models is to create a black box which understands the language and can then be asked to do any specific task in that language. The idea is to create the machine equivalent of a 'well-read' human being.

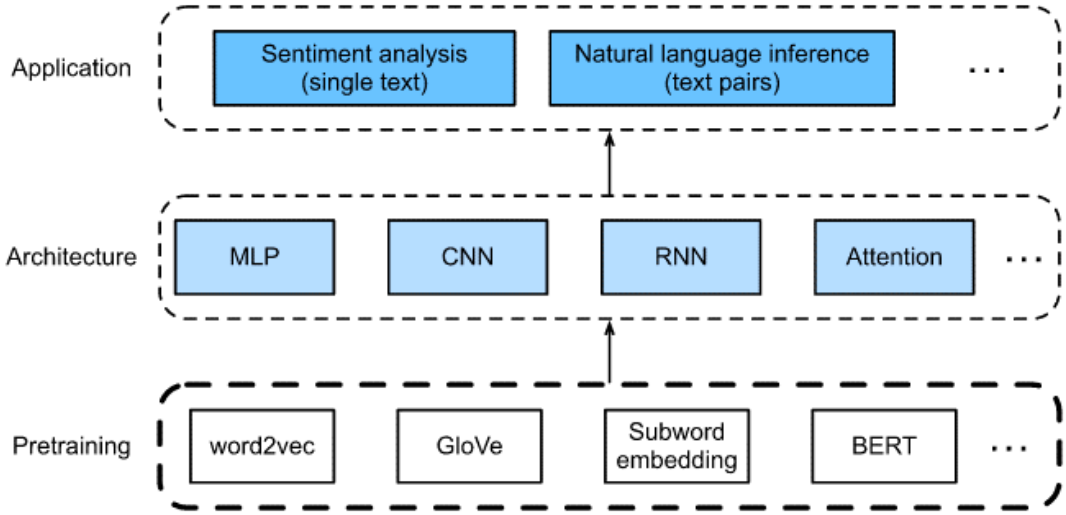


CH2 38 from Pretraining to final model

The language model is first fed a large amount of unannotated data. This lets the model learn the usage of various words and how the language is written in general. The model is now transferred to an NLP task where it is fed another smaller task-specific dataset, which is used to fine tune and create the final model capable of performing the task.

Natural language processing studies interactions between computers and humans using natural languages. In practice, it is very common to use natural language processing techniques to process and analyze text (human natural language) data, such as language models and machine translation models.

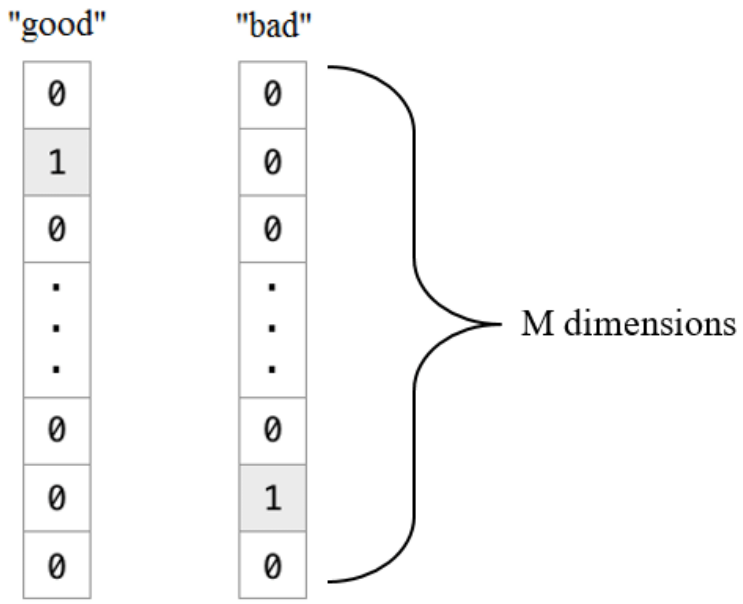
To understand text, we can begin with its representation, such as treating each word or subword as an individual text token. the representation of each token can be pretrained on a large corpus, using word2vec, GloVe, subword embedding, or BERT models. After pretraining, representation of each token can be a vector, however, it remains the same no matter what the context is.



CH2 39 deep learning architectures for different downstream natural language processing applications.

2.8.1 One-hot Vector

We used one-hot vectors to represent words (characters are words). Assume the number of different words in a dictionary (the dictionary size) is M, each word can correspond one-to-one with consecutive integers from 0 to M−1. These integers that correspond to words are called the indices of the words. We assume that the index of a word is i. In order to get the one-hot vector representation of the word, we create a vector of all 0s with a length of M and set element i to 1. In this way, each word is represented as a vector of length M.



CH2 40 One-hot vector represent

Although one-hot word vectors are easy to construct, they are usually not a good choice. One of the major reasons is that the one-hot word vectors cannot accurately express the similarity between different words, such as the cosine similarity that we commonly use. For the vectors x,y∈Rd, their cosine similarities are the cosines of the angles between them.

$$\frac{\mathbf{x}^{\mathbf{T}} * \mathbf{y}}{\| \mathbf{x} \| \| \mathbf{y} \|} \in [-\mathbf{1}, \mathbf{1}]$$

Since the cosine similarity between the one-hot vectors of any two different words is 0.

Although one-hot word vectors are easy to construct, they are usually not a good choice. One of the major reasons is that the one-hot word vectors cannot accurately express the similarity between different words. One-hot encoded vectors are high dimensional and sparse. Assume that we are doing Natural Language Processing (NLP) and have a dictionary of 2000 words. This means that, when using one-hot encoding, each word will be represented by a vector containing 2000 integers. And 1999 of these integers are zeros. In a big dataset, this approach is not computationally efficient.

2.8.2 Word Embedding

The first step in using an embedding layer is to encode this sentence by indices. In this case we assign an index to each unique word. The sentence than looks like this:

Deep learning is very deep

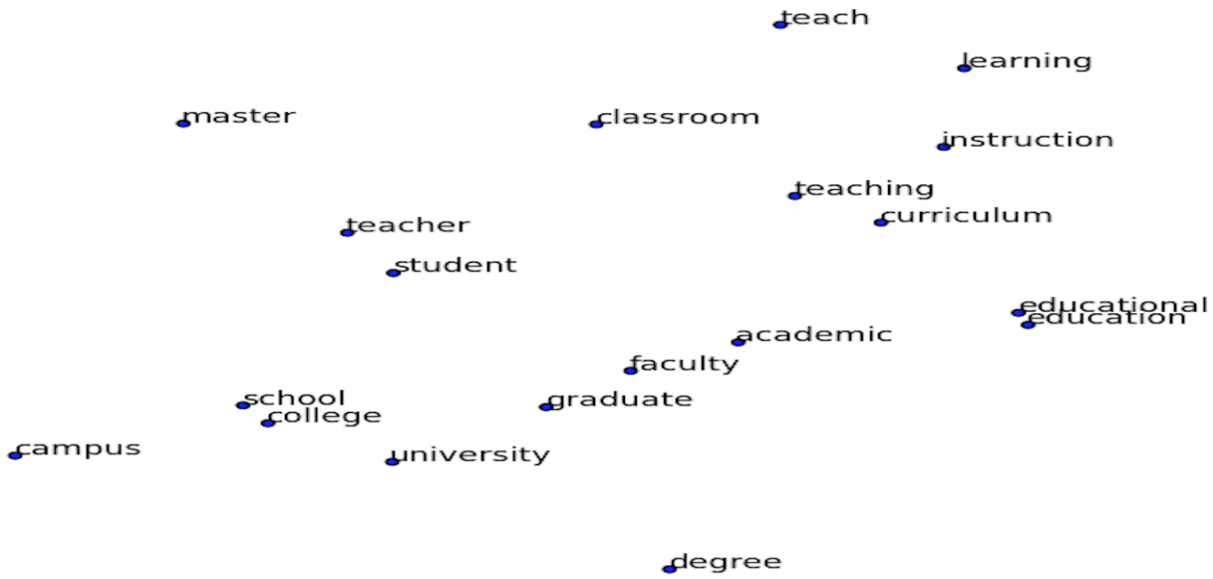
[1 2 3 4 1]

The embedding matrix gets created next. We decide how many 'latent factors' are assigned to each index. Basically, this means how long we want the vector to be. General use cases are lengths like 32 and 50. Assign 6 latent factors per index in this post to keep it readable. The embedding matrix than looks like this:

Indices	Latent Factors					
1	.32	.02	.48	.21	.56	.15
2	.65	.23	.41	.57	.03	.92
3	.45	.87	.89	.45	.12	.01
4	.65	.21	.25	.45	.78	.82

CH2 41 Embedding Matrix

So, instead of ending up with huge one-hot encoded vectors we can use an embedding matrix to keep the size of each vector much smaller. In short, all that happens is that the word "deep" gets represented by a vector [.32, .02, .48, .21, .56, .15]. However, not every word gets replaced by a vector. Instead, it gets replaced by index that is used to look-up the vector in the embedding matrix. Once again, this is computationally efficient when using very big datasets. Because the embedded vectors also get updated during the training process of the deep neural network, we can explore what words are similar to each other in a multi-dimensional space.



CH2 42 visualization of word embedding

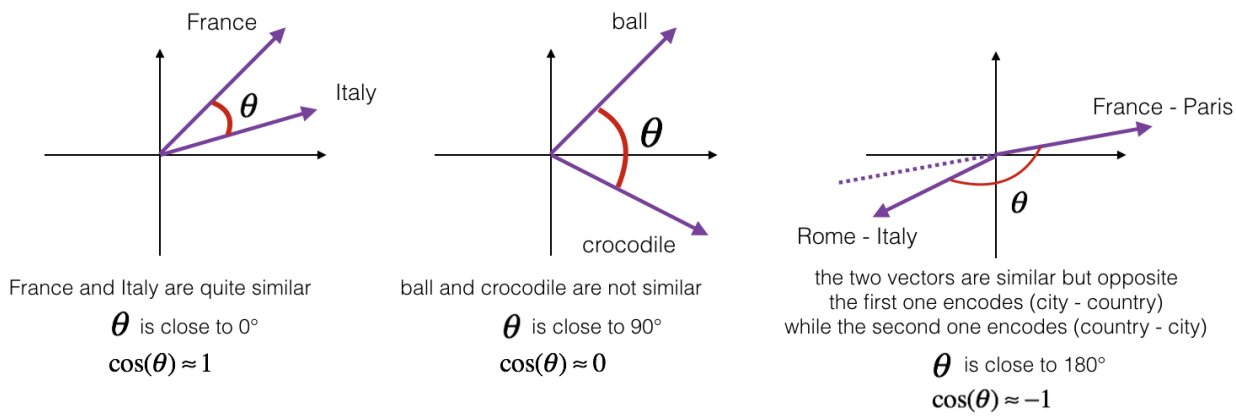
2.8.2.1.1 Vector Similarity

Generated word embeddings need to be compared in order to get semantic similarity between two vectors. There are few statistical methods are being used to find the similarity between two vectors. which are:

2.8.2.1.2 Cosine Similarity

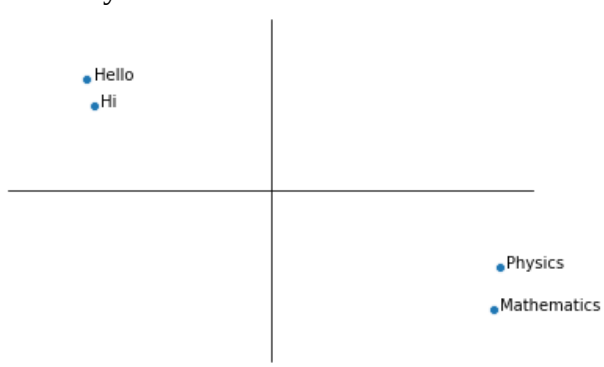
It is the most widely used method to compare two vectors. It is a dot product between two vectors. We would find the cosine angle between the two vectors. For degree 0, cosine is 1 and it is less than 1 for any other angle.

$$\boldsymbol{sim(A,B) = \cos(\theta) = \frac{A.B}{||A||||B||}}$$



CH2 43 Cosine Similarity

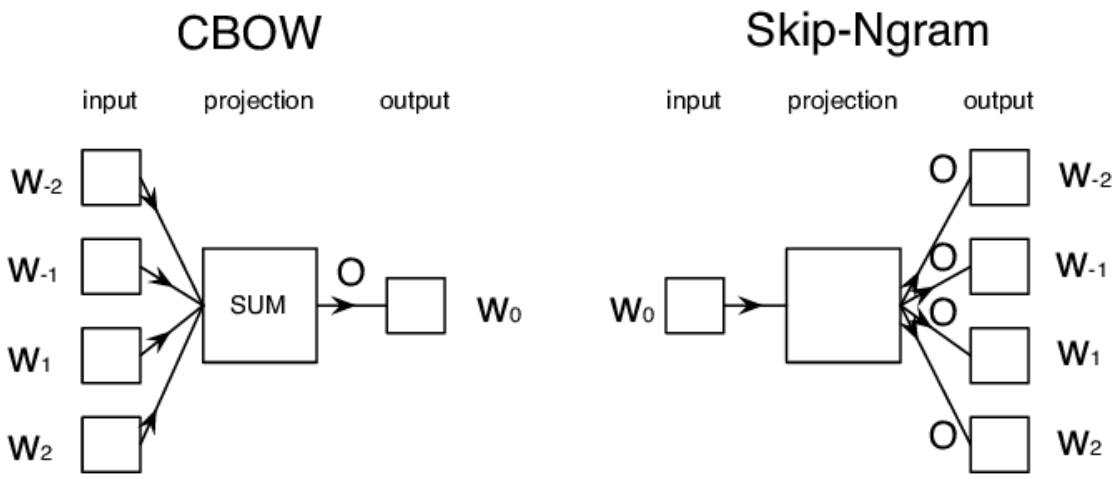
Word embeddings are vector representations of words which model semantic similarity through each word proximity to other words in the vector space. For example, words such as "hi" and "hello" will have similar coordinates to each
other, which in turn will have very different coordinates to the word mathematics.



CH2 44 Illustration of a word embedding vector space in 2-D

2.8.3 Word2Vec

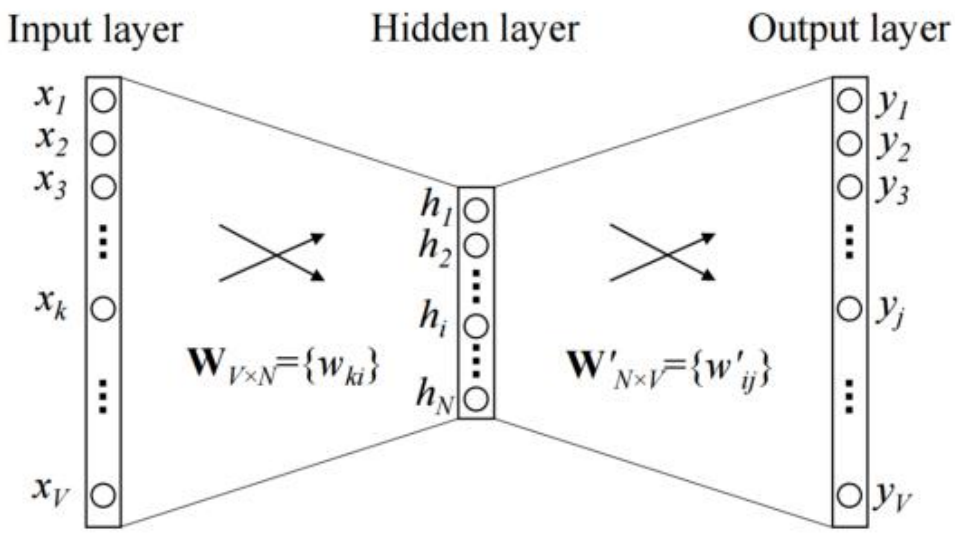
Innovation with an efficient Word Embedding implementation: the Word2Vec. They first demonstrate that a shallower and more efficient model 2 allows to be trained on much larger amounts of data (Speed increased by 1000!). This efficiency is given by removing the hidden layer, present on the previous architectures. Two versions of Word2Vec were proposed: Continuous Bag-of-words (CBOW) and SkipGram (SG).The first one predicts the central word based on a window of words surrounding it and the second one does the exact opposite: it predicts the context based on the central word. These architectures contain only one hidden layer, which leads to trainings that are more efficient.



CH2 45 CBOW & SkipGram model

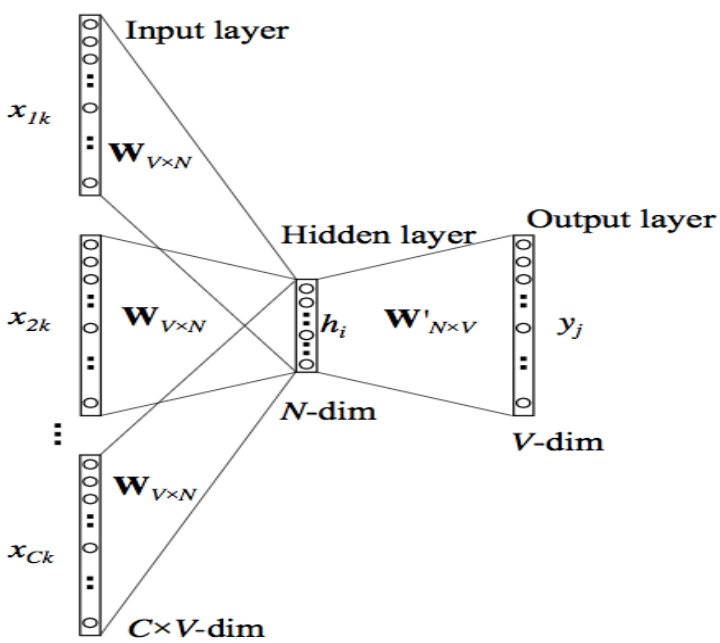
2.8.3.1 CBOW Model

This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider our example: Have a great day.
the input to the Neural Network be the word, great. Notice that here we are trying to predict a target word (day) using a single context input word great. More specifically, we use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (day). In the process of predicting the target word, we learn the vector representation of the target word. The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.



CH2 46 CBOW Model with one word of context

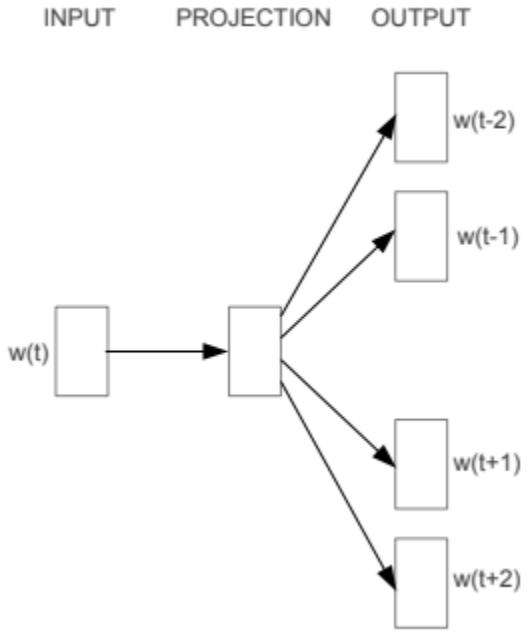
(Wv*n) is the weight matrix that maps the input x to the hidden layer (V*N dimensional matrix) (W`n*v) is the weight matrix that maps the hidden layer outputs to the final output layer (N*V dimensional matrix) The hidden layer neurons just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer. The above model used a single context word to predict the target. We can use multiple context words to do the same.



CH2 47 CBOW Model with Multiple word of context

2.8.3.2 SkipGram

Skip-gram is used to predict the context word for a given target word. It's reverse of CBOW algorithm. Here, target word is input while context words are output. As there is more than one context word to be predicted which makes this problem difficult.



CH2 48 Skip_gram model

w(t) is the target word or input given. There is one hidden layer which performs the dot product between the weight matrix and the input vector w(t). No activation function is used in the hidden layer. Now the result of the dot product at the hidden layer is passed to the output layer. Output layer computes the dot product between the output vector of the hidden layer and the weight matrix of the output layer. Then we apply the softmax activation function to compute the probability of words appearing to be in the context of w(t) at given context location.

Probability function:

In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability. We assume that the word is indexed as i in the dictionary, its vector is represented as $v_i \in R^d$ when it is the central target word, and $u_i \in R^d$ when it is a context word. Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary. The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product

$$\boldsymbol{p(wo|wc) = \frac{\exp\,(uo^T * vc)}{\sum_{i \in v} \exp\,(ui^T * vc)}}$$

I the word predicted on the o the context position; wo is the actual word present on the o the context position; w(c) is the only input word; and u(o,i) is the I the value in the U vector when predicting the word for o the context position.

In the training process, we are going to learn the model parameters by maximizing the likelihood function, which is also known as maximum likelihood estimation. This is equivalent to minimizing the following loss function

$$l = -\log P(Wc, 1, Wc, 2, Wc, 3, \ldots\ldots Wc, C|Wo) = -log \prod_{c=1}^{C} P(Wi, c|Wo)$$

2.8.4 GloVe

GloVe was developed by Stanford and stands for Global Vectors for Word Representation. It aims to conciliate the word prediction models with the word statistics over a whole corpus. They provide a model that takes into account co-occurrence statistics of the corpus, and the efficiency of prediction-based methods.

Given the words i = ice and j = steam, we want to study a ratio of co-occurrence probabilities with some probe word k = solid. We can expect the co-occurrence between the word i and the word k (Pik) being great over Pjk. There, the ratio Pik Pjk should be great.

We get ratio by the following formula:

$$\boldsymbol{f(Wi, Wj, W\tilde{}k) = \frac{Pik}{Pjk}}$$

w_k=	solid	gas	water	fashion
$p_1 = P(w_k \mid \text{ice})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{steam})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

CH2 49Co-occurrence probabilities and ratio with the words ice and steam in function

We will be able to observe phenomena such as: (previous Figure)

For a word wk that is related to “ice” but not to “steam”, such as wk=solid we would expect a larger conditional probability ratio, like the value 8.9 in the last row of the table above.

For a word wk that is related to “steam” but not to “ice”, such as wk=gas we would expect a smaller conditional probability ratio, like the value 0.085 in the last row of the table above.

For a word wk that is related to both “ice” and “steam”, such as wk=water we would expect a conditional probability ratio close to 1, like the value 1.36 in the last row of the table above.

For a word wk that is related to neither “ice” or “steam”, such as wk=fashion, we would expect a conditional probability ratio close to 1, like the value 0.96 in the last row of the table above.

Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: f(wi,wj,wk)=f((wi−wj)⊤wk). After exchanging index j with k, we will be able to see that function f satisfies the condition f(x)f(−x)=1, so one possibility could be f(x)=exp(x).

$$\boldsymbol{f(Wi, Wj, W\tilde{}k) = \frac{\exp\,(W^T i, Wk)}{\exp\,(w^T j, Wk)} = \frac{Pik}{Pjk}}$$

2.9 Sequence to Sequence Models

Recurrent Neural Networks (RNN) or more precisely (LSTM/GRU) are very effective in solving complex sequence-related problems given a large amount of data. They have real-time applications in speech recognition, Natural Language Processing (NLP) problems, time series forecasting, etc.

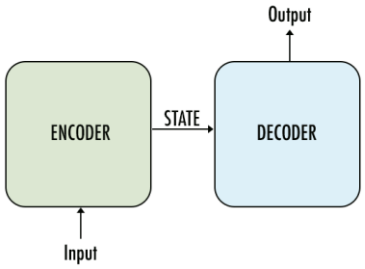
Sequence to Sequence (seq2seq) models are a special class of Recurrent Neural Network architectures typically used (but not restricted) to solve complex Language related problems like Machine Translation, Question Answering, creating Chatbots, Text Summarization, etc.

The seq2seq model is normally composed of an encoder-decoder architecture, where the encoder processes the input sequence and encodes/compresses/summarizes the information into a context vector (also called the "thought vector") of a fixed length.

The decoder is then initialized with this context vector, using which it starts generating the transformed output.

2.9.1 Encoder Decoder

The most common architecture used to build Seq2Seq models is Encoder-Decoder architecture.



CH2 50 Encoder Decoder architecture

a. Both encoder and the decoder are typically LSTM models or GRU models.

b. Encoder reads the input sequence and summarizes the information in something called the internal state vectors, **But** in the case of LSTM these are called the hidden state (ht) and cell state (Ct) vectors.

c. We discard all the intermediate states of the encoder and use only its final states

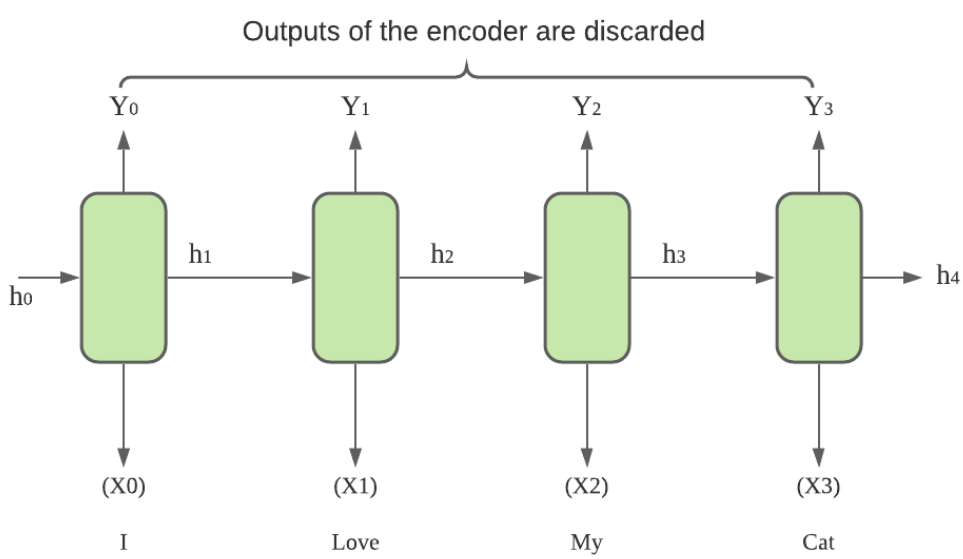
d. The decoder is an LSTM whose initial states are initialized to the final states of the Encoder LSTM. Using these initial states, the decoder starts generating the output sequence.

e. the encoder summarizes the input sequence into state vectors , which are then fed to the decoder which starts generating the output sequence.

Note: For illustrative purposes, we will explain encoder and decoder (using GRU) by giving an example that will simplify how it works. We will take the problem of Machine Translation (translating a text from one language to another, in our case from English to Italy).
Input (English) Sentence → "I love my cat".
Target (Italian) Sentence → "amo il mio gatto".

We will use a (GRU) layer. The reason being that LSTM has two internal states (hidden state (ht) and cell state (Ct)), But (GRU) has only one internal state (hidden state (ht)). This will help simplify the concept and explanation.

2.9.1.1 Encoder GRU



CH2 51 Encoder GRU

The GRU reads the data one sequence after the other. if the input is a sequence of length ‘k’, we say that GRU reads it in ‘k’ time steps, In this case, K = 4.

X: Input sequence at a time step.
Y: Output sequence at a time step.
h: the hidden state at each time step.
h0: the initial state of the encoder is initialized to zero or using some random method.

Important: Technically these components (X, h,) are vectors of floating-point numbers.

2.9.1.1.1 Explanation for X

We will break the sentence (I love my cat) by words as this scheme is more common in real-world applications. So, referring to the (Encoder GRU) diagram, we have the following input:
X1 = "I" / X2 = "love" / X3 = "my" / X4= "Cat".

2.9.1.1.2 Explanation for H

In very simple terms, they remember what the GRU has read (learned) till now. For example h3 => This vector will remember that the network has read “I love my ” till now.

Similarly, we can say that h4 will contain the summary of the entire input sentence since this is where the sentence ends (at time step 4). This state coming out of the last time step is also called the “Thought vectors” as they summarize the entire sequence in a vector form.

Important: The size of both of these vectors is equal to the number of units (neurons) used in the GRU cell.

2.9.1.1.3 Explanation for Y

These are the output (predictions) of the GRU model at each time step.

Depending on the context of the problem they might sometimes be used or sometimes be discarded. In our case, we have nothing to output unless we have read the entire English sentence. Because we will start generating the output sequence (equivalent Italy sentence) once we have read the entire English sentence. So we will discard the Y of the Encoder for our problem.

in the case of word-level language models, each Y is a probability distribution over the entire vocabulary which is generated by using a SoftMax activation. each Y is a vector of size “vocabulary size” representing a probability distribution.

2.9.1.1.4 Summary of the Encoder

We will read the input sequence (English sentence) word by word and preserve the internal state of the GRU network generated after the last time step hk (assuming the sentence has ‘k’ words). This vector (state hk) is called the encoding of the input sequence, as it encodes (summarize) the entire input in a vector form. Since we will start generating the output once we have read the entire sequence, outputs (Y) of the Encoder at each time step are discarded.

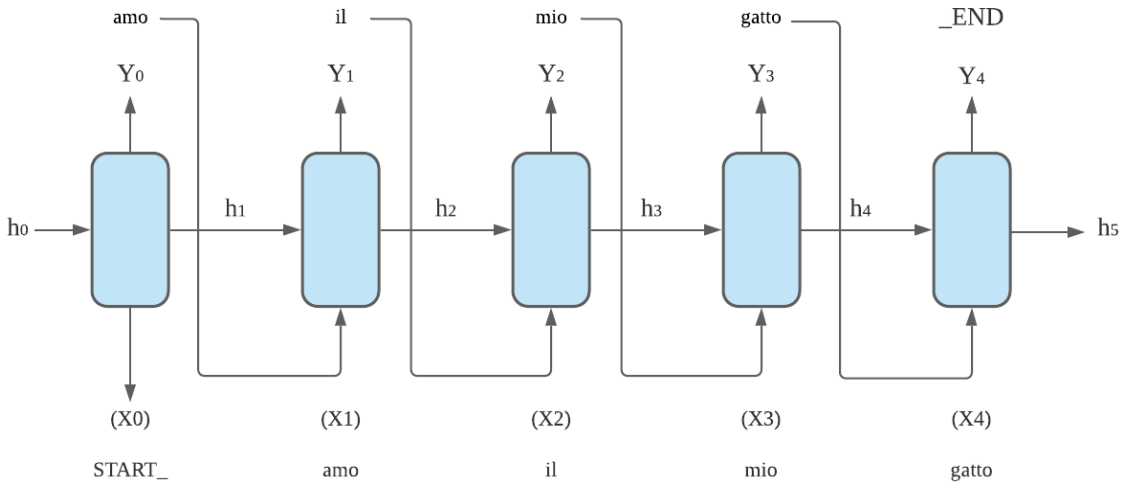
2.9.1.2 Decoder GRU — Training Mode

Unlike the Encoder GRU which has the same role to play in both the training phase as well as in the inference phase (will be explained later), the Decoder GRU has a slightly different role to play in both of these phases.

Recall that given the input sentence "I love my cat" the goal of the training process is to train (teach) the decoder to output "Amo il Mio Gatto". Just as the Encoder scanned the input sequence word by word, similarly the Decoder will generate the output sequence word by word.

For some technical reasons (explained later) we will add two tokens in the output sequence as follows:

Output sequence : “START_ amo il mio gatto _END”



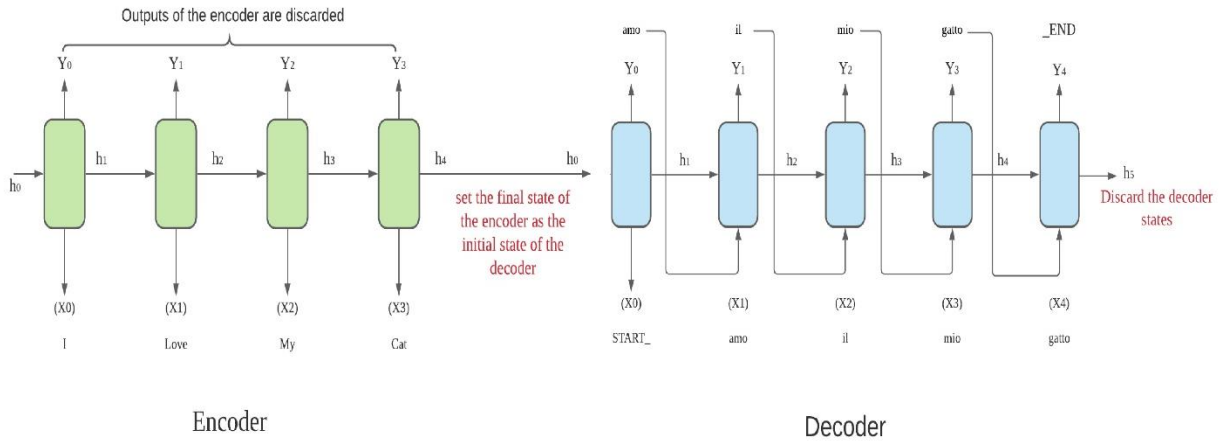
CH2 52 Decoder GRU — Training Mode

The most important point is that the initial states (h0) of the decoder are set to the final states of the encoder. This intuitively means that the decoder is trained to start generating the output sequence depending on the information encoded by the encoder. The translated Italian sentence must depend on the given English sentence.

In the first-time step, we provide the START_ token so that the decoder starts generating the next token (the actual first word of Italy sentence). And after the last word in the Italy sentence, we make the decoder learn to predict the _END token. This will be used as the stopping condition during the inference procedure, basically, it will denote the end of the translated sentence and we will stop the inference loop (more on this later).

Finally, the loss is calculated on the predicted outputs from each time step and the errors are back propagated through time in order to update the parameters of the network. Training the network over longer period with sufficiently large amount of data results in pretty good predictions (translations).

2.9.1.2.1 Summary of Encoder-Decoder Training Process

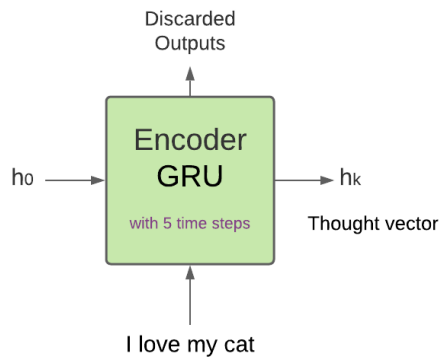


CH2 53 Summary of Encoder-Decoder Training Process

2.9.1.3 Decoder GRU — Inference Mode

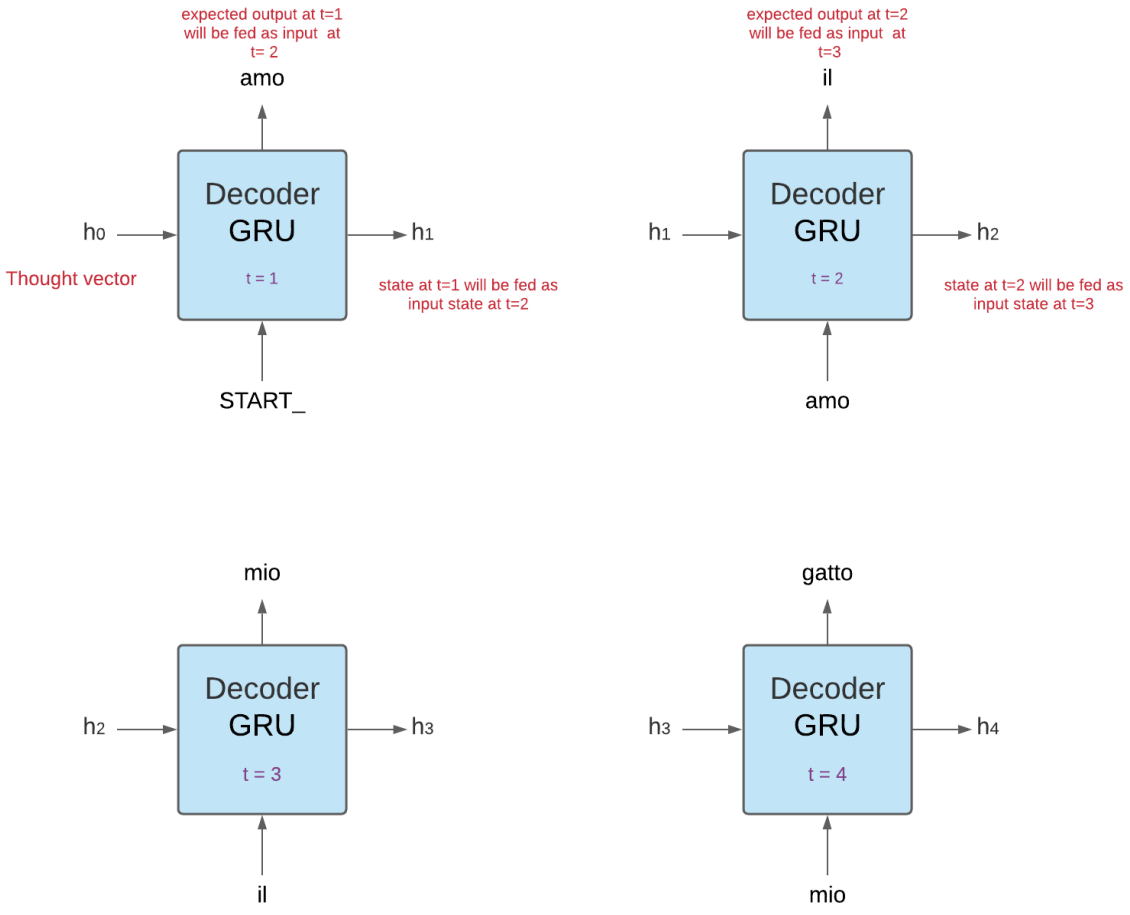
As already stated the Encoder GRU plays the same role in reading the input sequence (English sentence) and generating the thought vectors (hk).
However, the decoder now has to predict the entire output sequence (italy sentence) given this thought vector.

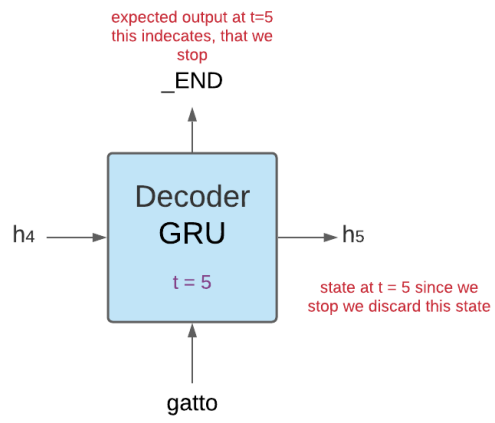
Step 1: Encode the input sequence into the Thought Vectors:



CH2 54 Output Sequence of Encoder in Inference Mode

Step 2: Start generating the output sequence in a loop, word by word:





CH2 55 Output Sequence of Decoder in Inference Mode

2.9.1.3.1 Inference Algorithm

1. During inference, we generate one word at a time. the Decoder GRU is called in a loop, every time processing only one time step.
2. The initial states of the decoder are set to the final states of the encoder.
3. The initial input to the decoder is always the START_ token.
4. At each time step, we preserve the states of the decoder and set them as initial states for the next time step.
5. At each time step, the predicted output is fed as input in the next time step.
6. We break the loop when the decoder predicts the END_ token.

2.9.2 Attention Mechanism in Deep Learning

Attention is one of the most influential ideas in the Deep Learning community. Even though this mechanism is now used in various problems like image captioning and others, it was initially designed in the context of Neural Machine Translation using Seq2Seq Models.

So as we explained in The seq2seq models is normally composed of an encoder-decoder architecture where the encoder processes the input sequence and summarizes the information into a “thought vector” of a fixed length.

A critical and apparent disadvantage of this fixed-length context vector design is the incapability of the system to remember longer sequences. Often is has forgotten the earlier parts of the sequence once it has processed the entire the sequence. The attention mechanism was born to resolve this problem.

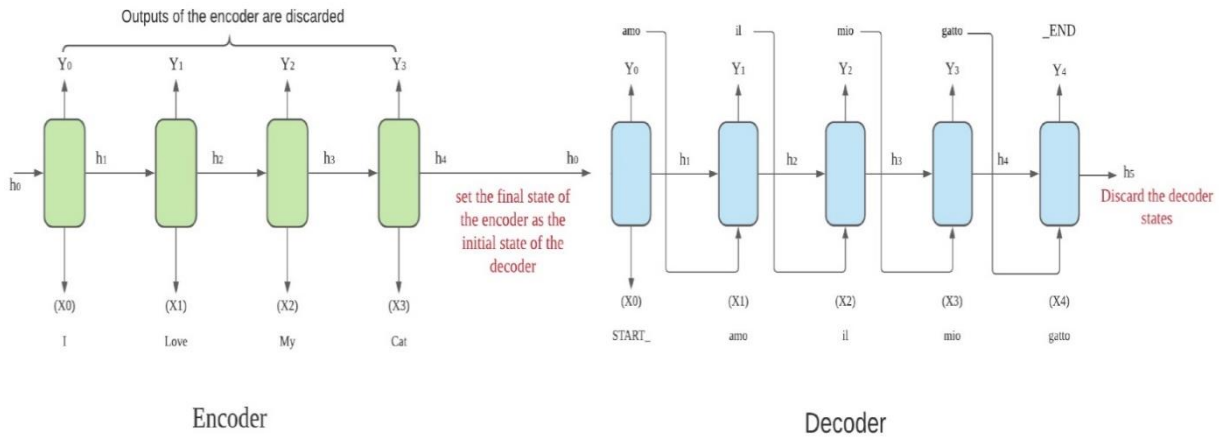
Note: For illustrative purposes, we will borrow the same example that we used to explain Seq2Seq models.

We will take the problem of Machine Translation (translating a text from one language to another, in our case from English to Italy).

Input (English) Sentence → "I love my cat".

Target (Italian) Sentence → "amo il mio gatto".

2.9.2.1 Summary of Encoder-Decoder Training Process



CH2 56 Summary of Encoder-Decoder Training Process

In the traditional Seq2Seq model, we discard all the intermediate states of the encoder and use only its final states (vector) to initialize the decoder. This technique works good for smaller sequences, however as the length of the sequence increases, a single vector becomes a bottleneck and it gets very difficult to summarize long sequences into a single vector. This observation was made empirically as it was noted that the performance of the system decreases drastically as the size of the sequence increases.

The central idea behind Attention is not to throw away those intermediate encoder states but to utilize all the states to construct the context vectors required by the decoder to generate the output sequence.

To understand Attention Mechanism well, we need to know

2.9.2.2 Why the name Attention?

Imagine you are translating “I love my cat” to “ amo il mio gatto ”.

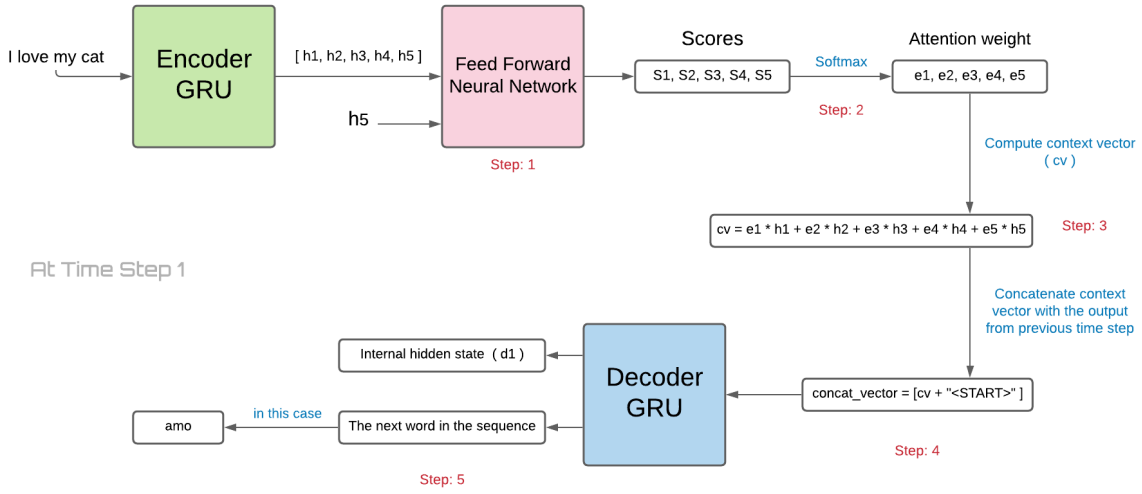
When you predict " amo " , its obvious that the result of the word " I love " present in the input English sentence regardless of the rest of the sentence. We say that while predicting " gatto ", we pay more Attention to the word " cat " in the input sentence.

Similarly while predicting the word " il mio " . we pay more Attention to the word " my " in the input sentence. So That's way the name “Attention”.

As human beings we are quickly able to understand these mappings between different parts of the input sequence and corresponding parts of the output sequence. However its not that straight forward for artificial neural network to automatically detect these mappings.

the Attention mechanism is developed to “learn” these mappings through Gradient Descent and Back-propagation.

2.9.2.3 Attention Mechanism



CH2 57 Attention Mechanism

Before we start decoding, we first need to encode the input sequence into a set of internal states (in our case h1, h2, h3, h4 and h5).

Now the hypothesis is that, the next word in the output sequence is dependent on the current state of the decoder as well as on the hidden states of the encoder at each time step, we consider these two things.

At time step 1, we can break the entire process into five steps:

Step 1 — Compute a score each encoder state

Since we are predicting the first word itself, the decoder does not have any current internal state. For this reason, we will consider the last state of the encoder (i.e., h5) as the previous decoder state.

Now using these two components (all the encoder states and the current state of the decoder), we will train a simple feed forward neural network.

Recall we are trying to predict the first word in the target sequence i.e. " amo ". As per the idea behind attention, we do not need all the encoder states to predict this word, but we need encoder states which store information about the word " i love " in the input sequence.

As discussed previously these intermediate encoder states store the local information of the input sequence. So it is highly likely that the information of the word " i love " will be present in the states, let's say, h1 and h2.

we want our decoder to pay more attention to the states h1 and h2 while paying less attention to the remaining states of the encoder.

For this reason we train a feed forward neural network which will learn to identify relevant encoder states by generating a high score for the states for which attention is to be paid while low score for the states which are to be ignored.

Let s1, s2, s3, s4 and s5 be the scores generated for the states h1, h2, h3, h4 and h5 correspondingly. Since we assumed that we need to pay more attention to the states h1 and h2 and ignore h3, h4 and h5 to predict " amo ". we expect the above neural to generate scores such that s1 and s2 are high while s3, s4 and s5 are relatively low.

Step 2— Compute the attention weights

Once these scores are generated, we apply a softmax on these scores to produce the attention weights e1, e2, e3 ,e4 and e5 as shown above.

The advantage of applying softmax :

1. All the weights lie between 0 and 1.
2. All the weights sum to 1.

we get a nice probabilistic interpretation of the attention weights.In our case we would expect values like (for example) e1 = 0.75, e2 = 0.2, e3 = 0.02, e4 = 0.02, e5 = 0.01

This means that while predicting the word " amo ". the decoder needs to put more attention on the states h1 and h2 (since values of e1 and e2 are high) while ignoring the states h3, h4 and h5 (since the values of e3, e4 and e5 are very small).

Step 3— **Compute the context vector**

Once we have computed the attention weights, we need to compute the context vector (thought vector) which will be used by the decoder in order to predict the next word in the sequence. Calculated as follows:

$$context_vector = e1 * h1 + e2 * h2 + e3 * h3 + e4 * h4 + e5 * h5$$

If the values of e1 and e2 are high and those of e3, e4 and e5 are low then the context vector will contain more information from the states h1 and h2 and relatively less information from the states h3, h4, and h5.

Step 4— Concatenate context vector with output of previous time step

Finally the decoder uses the below two input vectors to generate the next word in the sequence :

1. The context vector.
2. The output word is generated from the previous time step.

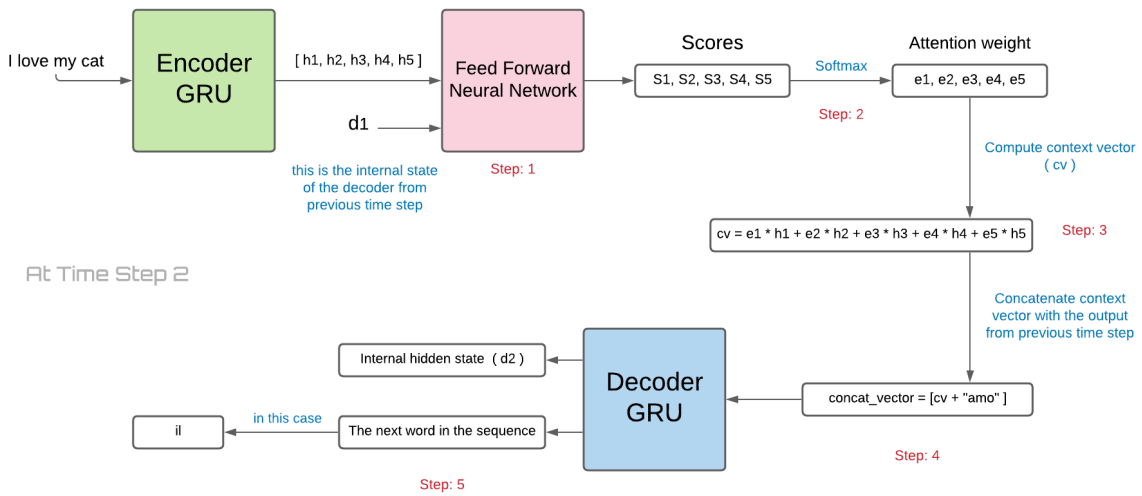
We simply concatenate these two vectors and feed the merged vector to the decoder.

Note : that for the first time step, since there is no output from the previous time step, we use a special <START> token for this purpose.

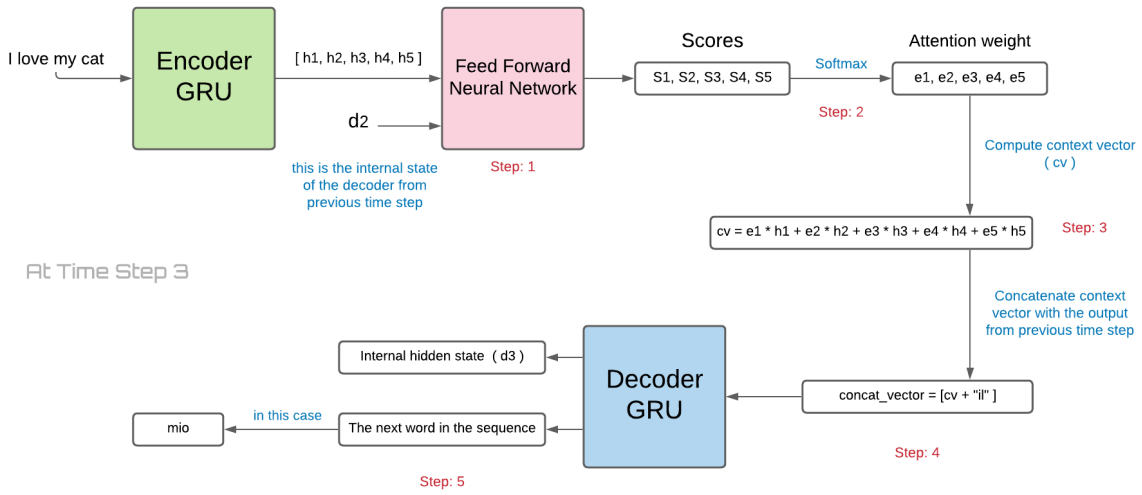
Step 5— Decoder Output

The decoder then generates the next word in the sequence (in this case, it is expected to generate " amo "). and along with the output, the decoder will also generate an internal hidden state, and lets call it as “d1”.

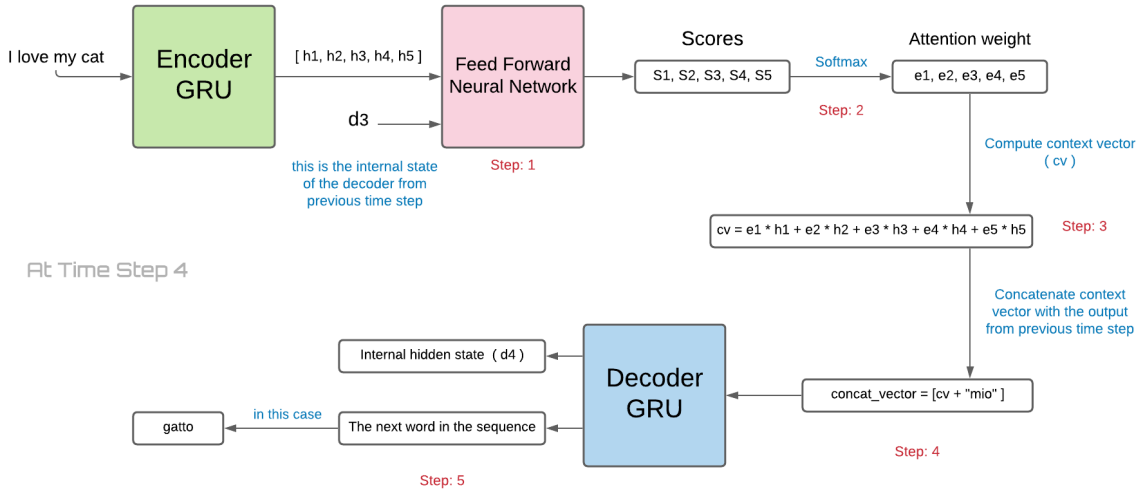
Decoding at time step 2:



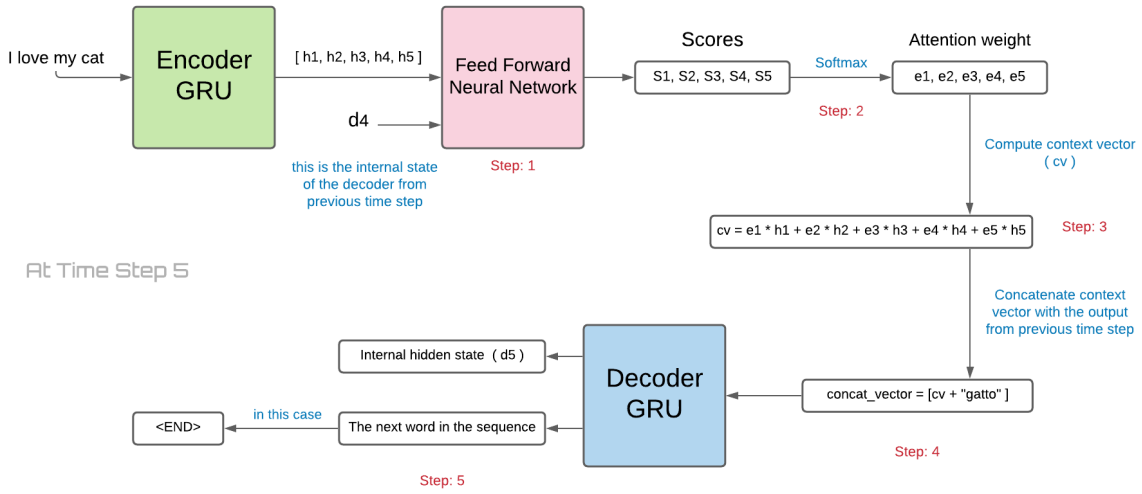
Decoding at time step 3:



Decoding at time step 4:



Decoding at time step 5:



Once the decoder outputs the <END> token, we stop the generation process.

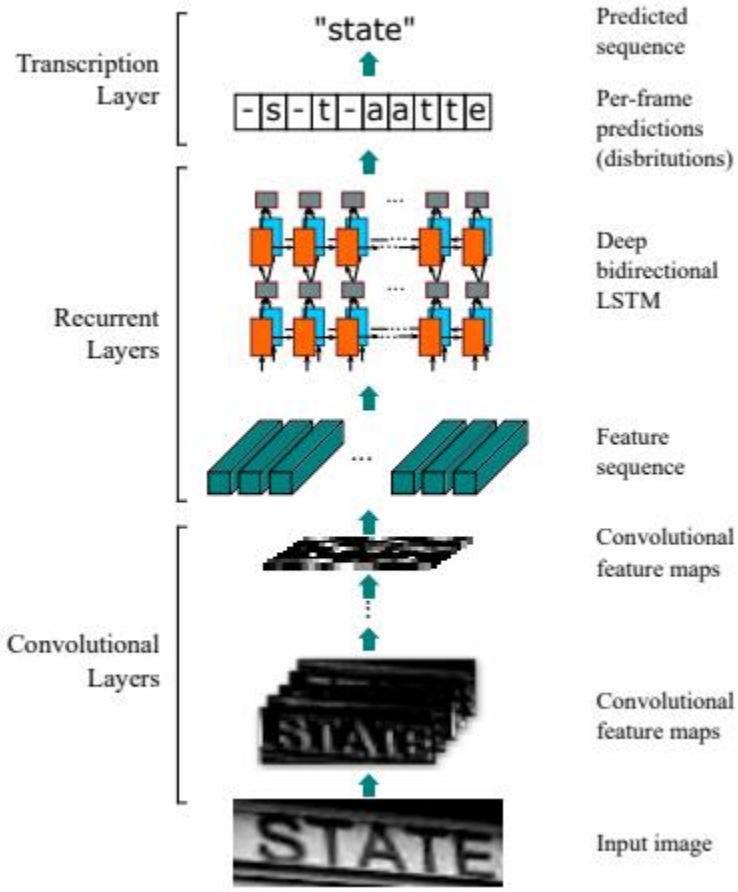
Note: that unlike the fixed context vector used for all the decoder time steps in case of the traditional Seq2Seq models, here in case of Attention, we compute a separate context vector for each time step by computing the attention weights every time.

By using this mechanism our model is able to find interesting mappings between different parts of the input sequence and corresponding parts of the output sequence.

2.10 Connectionist Temporal Classification

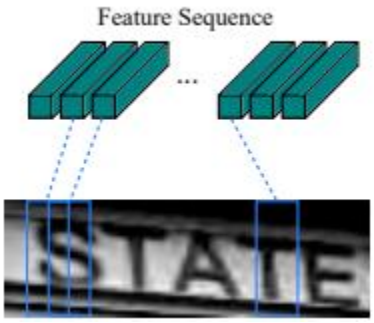
CTC is an algorithm used to tackle a key issue faced when training seq-2-seq models, which is, when the length of the input and output sequences do not match.

Word Recognition: The problem is to predict the word present in an image. One way to solve this is by treating it as a sequence-to-sequence problem and using a hybrid architecture of Convolutional and Recurrent Neural Nets. The architecture of CRNN is as follows:



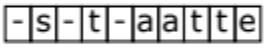
CH2 58 CTC

Here, the input sequence is the convolutional feature map and the output will be a character sequence.



One can ask what could be the temporal dependency here in case of images. Well since the feature maps are split and the predictions are made for every column, it is highly unlikely that one complete character will fall inside a single column. Hence a character's pattern learning depends not only on the current column but also on columns before and after it.

1. input image



2. predicted sequence from RNN

"state"

3. decoded output

While training such a network, all we have is **<1. Input image, 3. Decoded output>** pair as a training instance. But the neural net gives out predicted sequence as output whose length is not equal to that of the decoded output.

So in order to train this net, we need to know what the target sequence is.

Decoded output → "STATE"

Say the length of the target sequence/predicted sequence (i.e. output from the RNN) is 14. Now let's guess the target sequence to be "SSSTTAAAATTEEE".

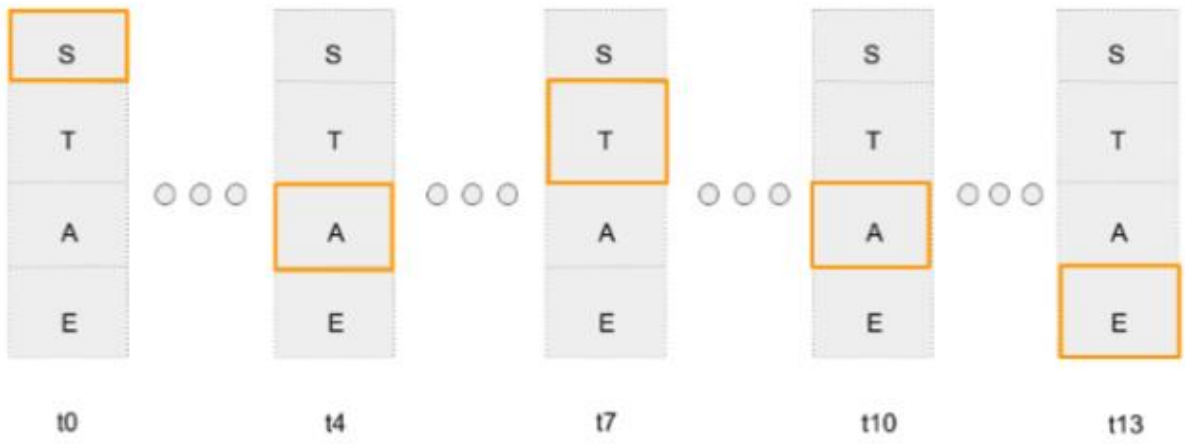
Given that we have a target sequence, we can train the network with that sequence for the first step, then use the network's predictions as the new target sequence(for next step) and iterate it over and over during training.

But there is one key problem here. How do we ensure that the network's predicted target sequence when decoded gives us the desired output, "STATE" in our case?

What if the RNN outputs "RRRRAAAEEELLLL"? We can't use this output sequence as the new target sequence for the next step in the training process. Hence we need to impose few restrictions on the sequence predicted by the RNN.

Constraint 1: Instead of taking into account the predictions of entire vocabulary (alphabets A to Z), consider only the alphabets that is part of the decoded output.

In our case, the alphabets S,T,A,E are part of the decoded output "STATE".

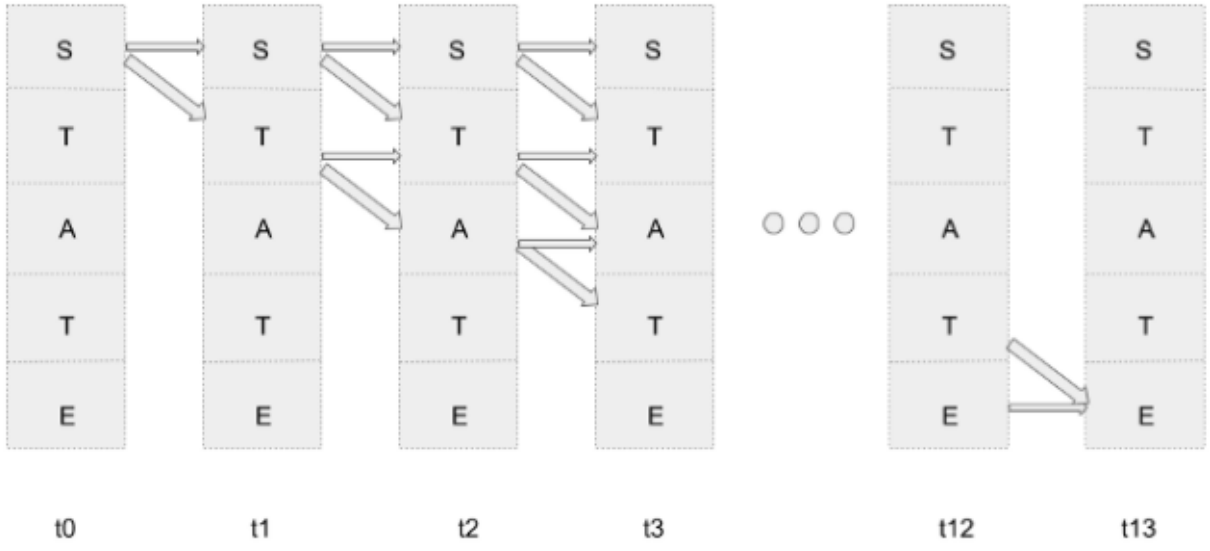


Okay, this now ensures that our network produces a predicted sequence containing only the alphabets in the decoded output. But there is no constraint on the sequence in which the alphabets should be outputted.

For eg: the RNN outputs “SSAAAATTTAAEEE”, which still contains only the target alphabets but when decoded produces SATAE instead of STATE.

Constraint 2: To ensure the network outputs a sequence which when decoded gives us the desired collapsed output, let’s impose the following constraint:

Not only consider the alphabets S,T,A,E but also order the individual alphabets in the sequence in which they occur in the decoded output(even if the alphabets have to be repeated).



And now fix that the first output should always be the top left symbol and last output should always be the bottom right symbol and the sequence should always be from top to bottom strictly i.e no upward movement. Every path from the top-left symbol to the bottom-right symbol is considered a valid sequence/alignment, since it will always decode to a valid 'decoded_sequence'.

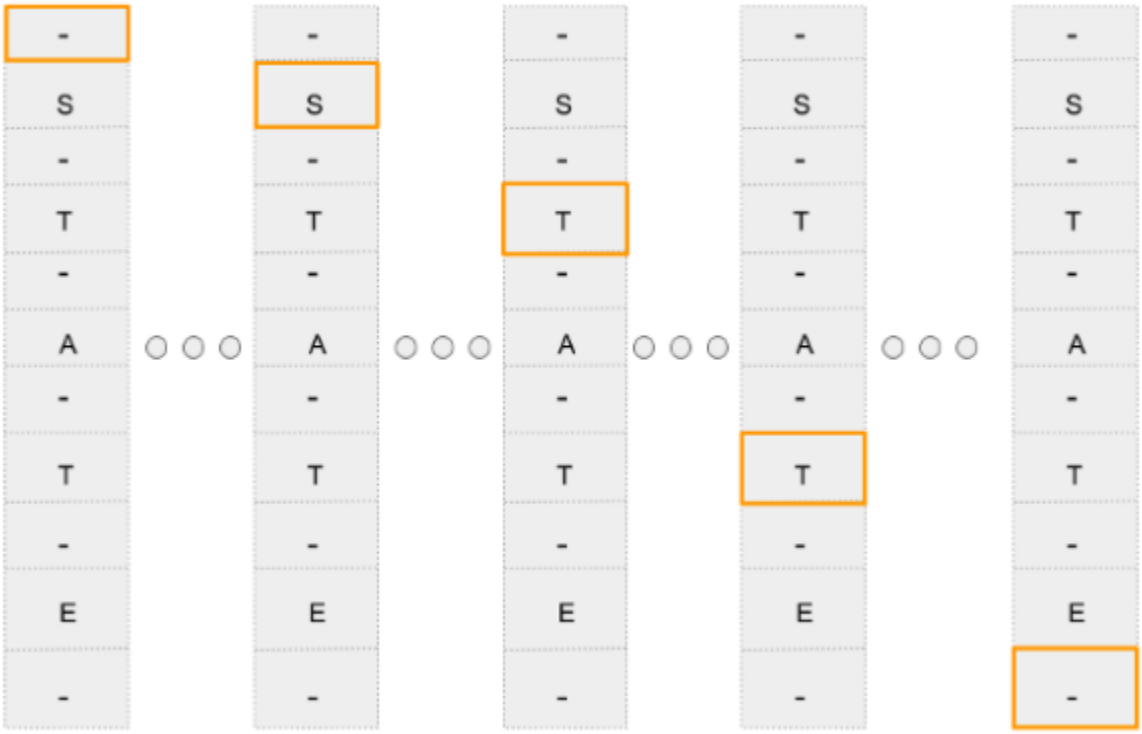
Lastly, consider the case when the output has repeated characters. Let the output word be "LETTER" and the predicted_sequence be "LLLLEETTTTEERR". Now what happens when this predicted_sequence is collapsed?

We get the decoded output as "LETER" which is undesirable.

Therefore inorder to account for repetitive characters in the output domain, blank character(-) is included as part of the vocabulary. Basically a neural-net outputs a blank character when none of the symbols in the vocabulary is present at that timestep.

So instead of producing "LLLLEETTTTEERR", an RNN with blank as part of the vocabulary might now output "-L-EE-TT-T-E-R" which when decoded produces "LETTER".

Since a blank character has been included and it can happen to occur anywhere in the target_sequence, we arrange the target alphabets + blank as follows



2.11 Time Distributed Layer

If you already worked with Convolution Neural Networks (C-NN), you've probably created models that take only one image to be classified.

But what if you need to present several images that are chronologically ordered to detect movements, actions, directions…? What we need is to be able to inject a sequence as input, and to make predictions of what that sequence is showing.

C-NN receives only one image and we make convolution on that one. The goal is to find features on those images that will help to make a "prediction" to guess the object on that image.

For example, we present an image of a cat, and the model says "it's a cat".

But now, we don't want to make that. We want… but we need to make "more" because the expected result is not "a cat", but "a cat is jumping".

You realize that what we need is to find features to find the cat, then use these features in "LSTM" to try to react on movements (or distortions). Finding a jumping cat now needs:

- Find the cat on each frame
- Then check if the cat movement corresponds to the "jumping" action

So we need to repeat several "cat" detection, and only after that we need to check the "action", the "movements".

Images should not be fully connected in one convolution

The problem is that we've got not only "one image" to manage, but we've also got potentially 2, 5,… 10 images to inject in the model. And we need to make convolutions for each image.

And because everything is never that simple, that's not the only problem to solve 😟.

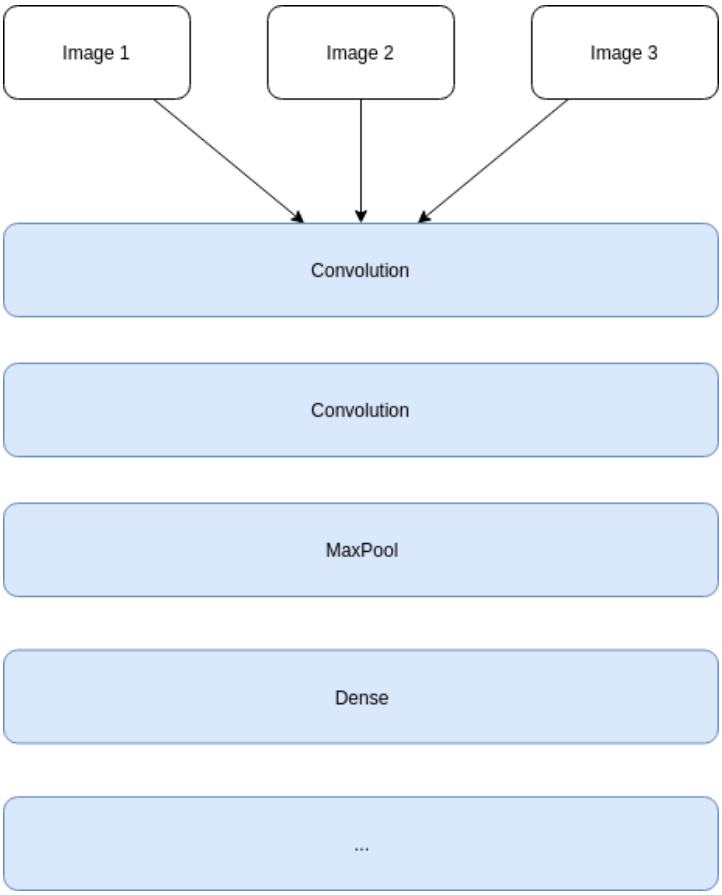
The first problem, the input shape — and the unwanted "image merging"

If we take a common Sequential neural network, each input is connected to the entire neuron list that resides in the first layer. For only one image, it's OK — but with several images, we need to avoid merging several images in one because the whole pixel list of all images will be sent to the first layer.

That means that if I send several images in the first layer, then I will merge the images. And this is not what I need!

Anyway, if you try to make that, what is the "shape" to set up to that layer?

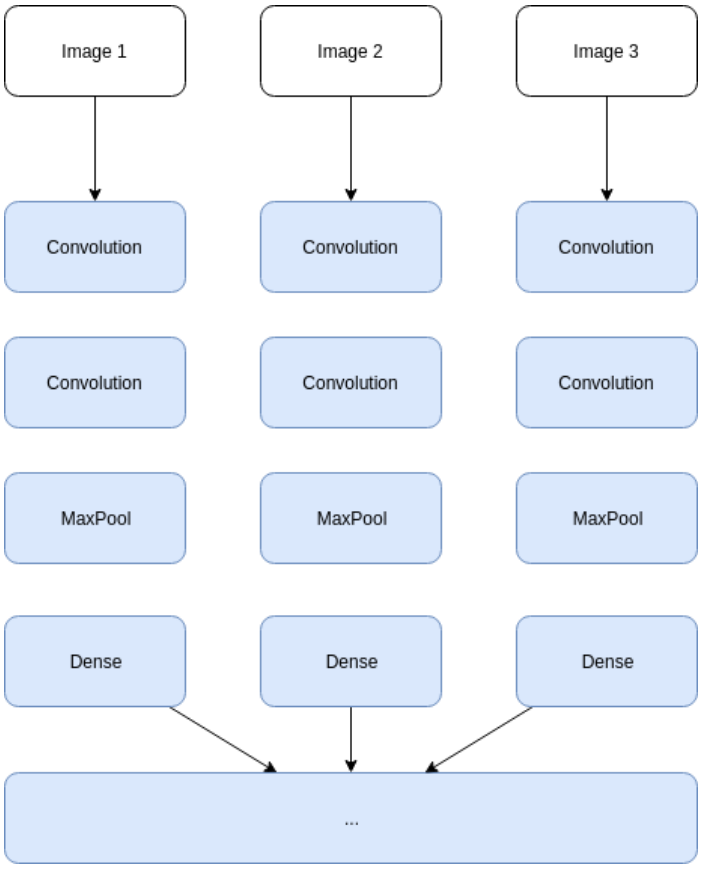
The next image is what we don't want to do :



Let's take an example of 5 images with 224x224 pixels in grayscale (one channel), Conv2D cannot use a (5, 224, 224, 1) shape (it requires 3 dimensions), and Conv3D is not made to manage that kind of input. No, we need to split the network to have one filter list for each image input (or any other structural data)

We need to pass one image to one "convolution block", the second to another, and so on…

So this is what we need to do:



Now, each image has got its own “convolutions flow” and we can, finally, continue the detection. It’s like we’ve got one neural network per image, then a last one to make inferences.

If we separately train each “convolution flow”, so we will have several unwanted behaviors:

We need very long training because we need to train several convolution flows (one per input image)

Each convolution flows, for one sequence, can have several different weights, and so different features detection that are not linked

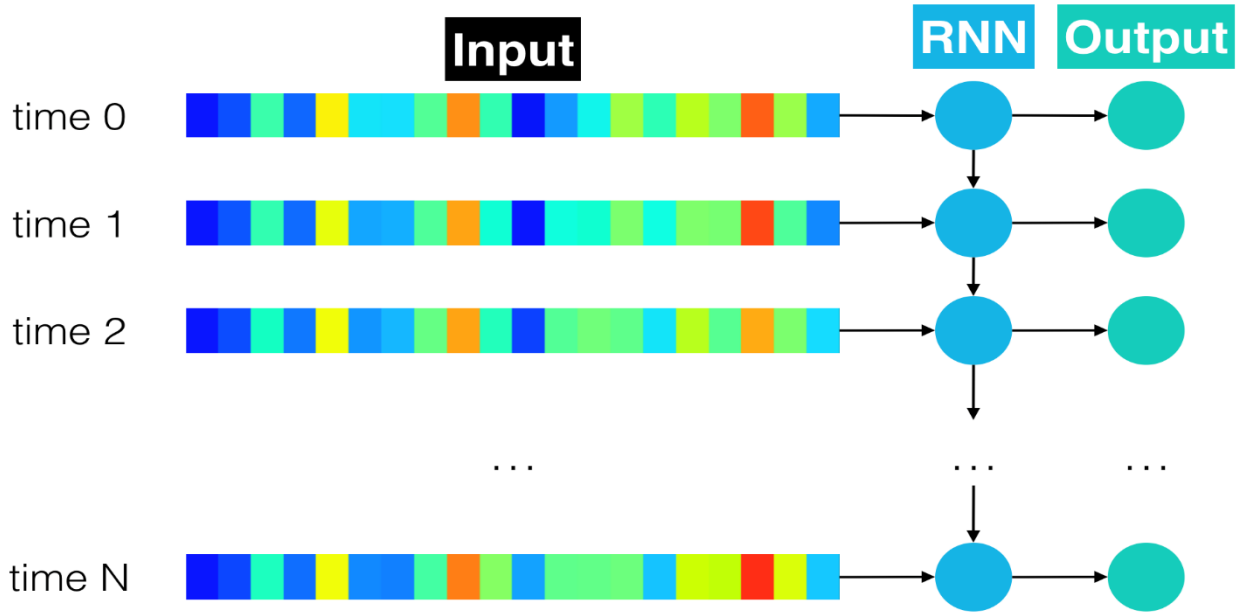
Some convolution flows will not detect what others flows can We need to be sure that the entire list of convolution flows can find the same features. It’s possible that some convolution flows detect something else, but we need to reduce that risk. TimeDistributed layer apply the same layer to several inputs. And it produce one output per input to get the result in time. That’s all!

3. CHAPTER THREE: SPEECH TO SPEECH SYSTEM DESIGN

3.1 Model 1: Simple RNN

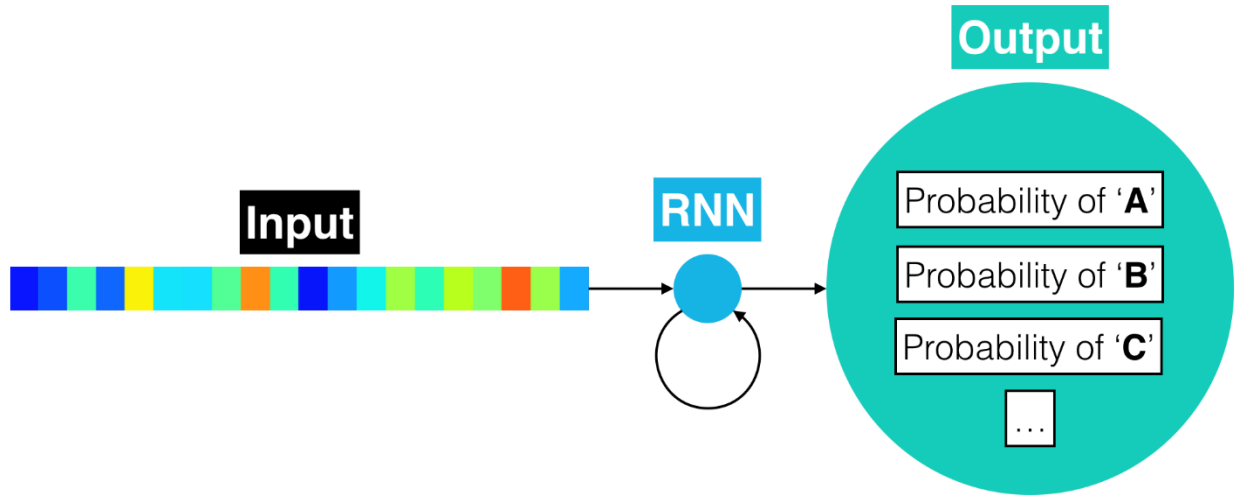
The first acoustic model you will use is an RNN, Given their effectiveness in modeling sequential data.

As shown in the figure below, the RNN will take the time sequence of audio features as input.



CH3 1 SIMPLE RNN ARCH

At each time step, the speaker pronounces one of 28 possible characters, including each of the 26 letters in the English alphabet, along with a space character (" "), and an apostrophe ('), the output of the RNN at each time step is a vector of probabilities with 29 entries.



3.1.1 Pseudo Code

3.1.1.1.1 Model's parameters:

```
simple_rnn_model (input_dim, output_dim)
```

3.1.1.1.2 Main acoustic input:

```
input_data = Input (name='the_input', shape= (None, input_dim))
```

3.1.1.1.3 Recurrent layer:

```
simp_rnn = GRU (output_dim, return_sequences=True, implementation=2,
name='rnn') (input_data)
```

3.1.1.1.4 Softmax activation layer:

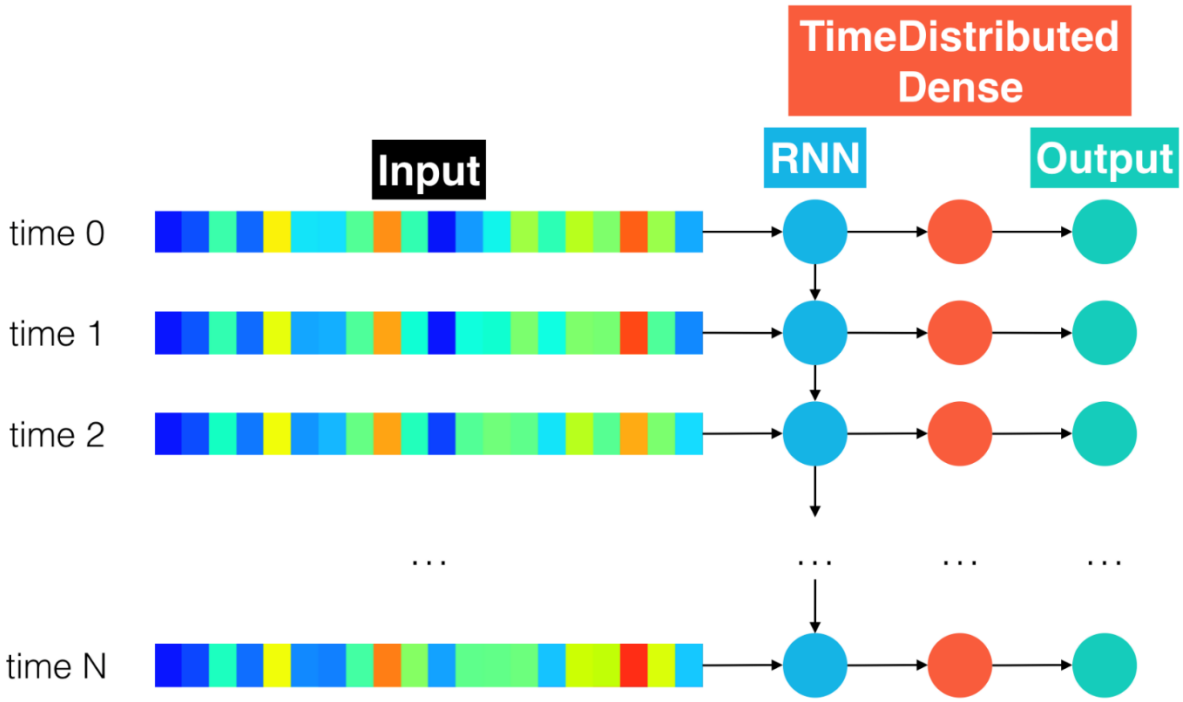
```
y_pred = Activation ('softmax', name='softmax') (simp_rnn)
```

3.1.1.1.5 Specify the model

```
model = Model (inputs=input_data, outputs=y_pred)
model.output_length = lambda x: x
```

3.2 Model 2: RNN + TimeDistributed Dense

The **TimeDistributed** layer will be used to find more complex patterns in the dataset. Batch normalization will be added to the recurrent layer to reduce training times. The unrolled snapshot of the architecture is depicted below.



CH3 2 RNN + TimeDistributed Dense

While building a sequential model - usually the second dimension (one after sample dimension) - is related to a time dimension. This means that if for example, the data is 5-dim with (sample, time, width, length, channel) you could apply a convolutional layer using TimeDistributed which is applicable to 4-dim with (sample, width, length, channel) along a time dimension (applying the same layer to each time slice) in order to obtain 5-D output.

In case of speech recognition, the first layer of the neural network will be an RNN (SimpleRNN, LSTM, or GRU) that takes the time sequence of audio features as input.

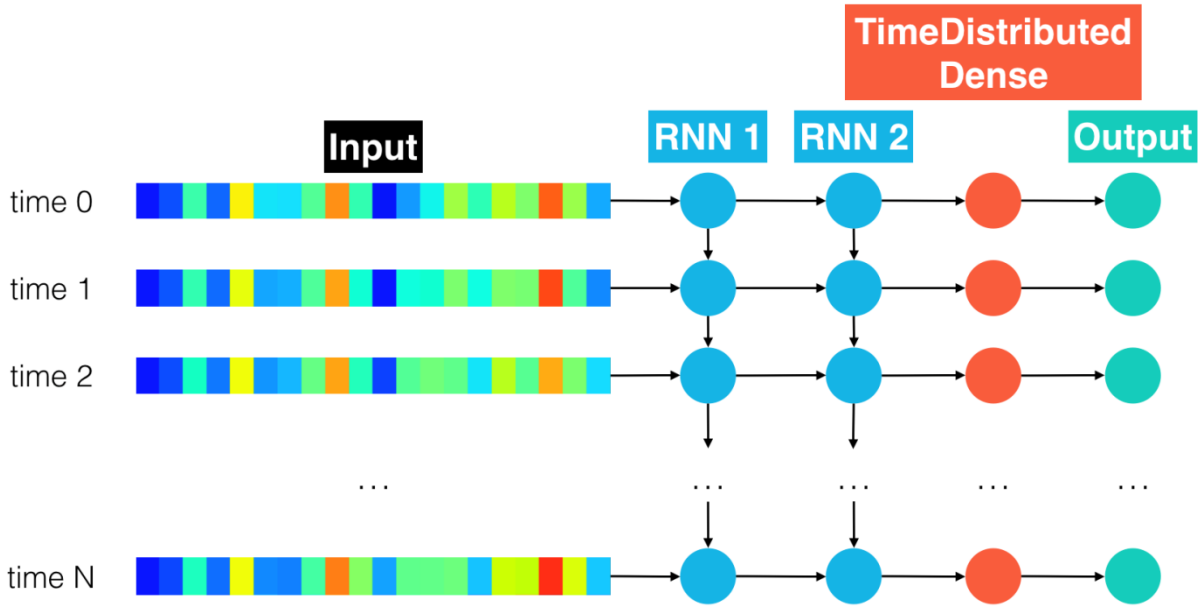
3.2.1 Pseudo Code

```
First, define a function named RNN which takes Input_dimension, units,
activation, and output_dim
Build the recurrent neural network for speech
```

```
Input_data = Input(name, shape)
Add recurrent layer
Simple_RNN = GRU(units, activation, name)
Add Batch normalization to the recurrent layer to reduce training times
Batch_norm = BatchNormalization()(Simple_RNN)
Add a TimeDistributed Dense output dimemsion layer
Time_dense = TimeDistributed(Dense(output_dim))( Batch_norm)
Add softmax activation layer
y_pred = Activation('softmax', name='softmax')(Time_dense)
Choose the model
model = Model(inputs=Input_data, outputs=y_pred)
Return model results
return model

Next step is to call the function we have just initialized
Model = RNN(input_dimension, units, activation=’relu’)

To train the neural network we specify input_to_softmax
Train_model(input_to_softmax= Model, Loss_history, Save_model_path)
```

3.3 Model 3: Model Deeper RNN + TimeDistributed Dense

The architecture in deep_rnn_model that utilizes a variable number recur_layers of recurrent layers. The figure below shows the architecture that should be returned if recur_layers=2. In the figure, the output sequence of the first recurrent layer is used as input for the next recurrent layer.



CH3 3 Deeper RNN + TimeDistributed Dense

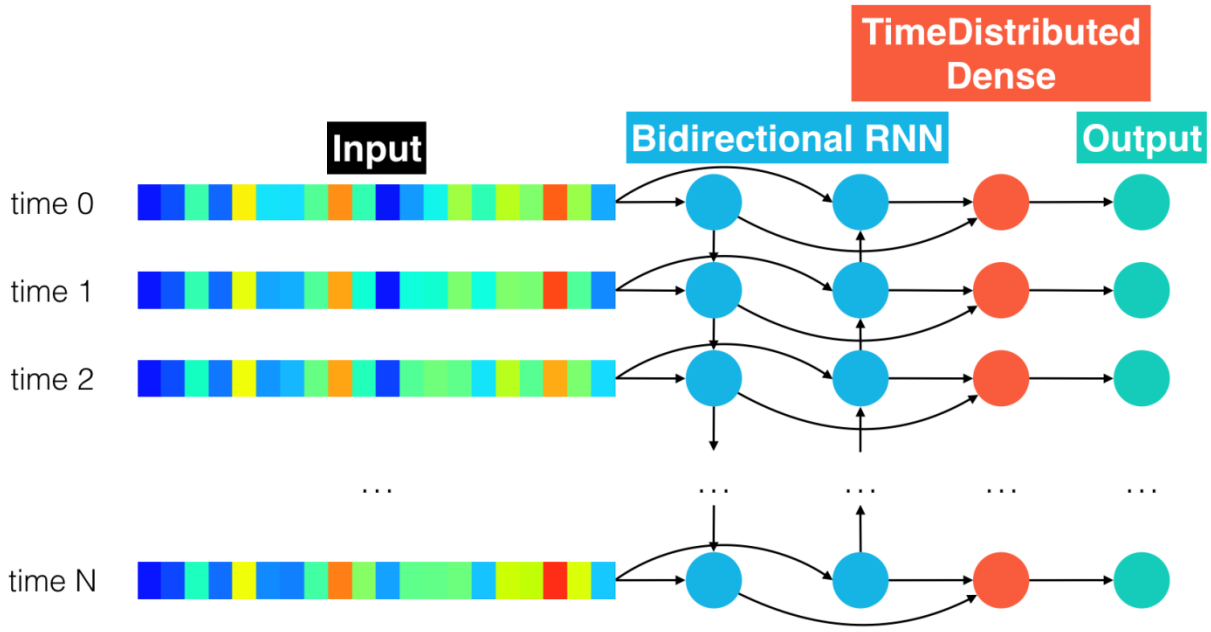
3.3.1 Pseudo Code

```
Model's parameters:
input_dim, units, recur_layers, output_dim=29
Main acoustic input:
input_data = Input(name='the_input', shape=(None, input_dim))

Recurrent layers:
Each Recurrent laye with batch normalization
bn_rnn = input_data
   for i in range(recur_layers):
      layer_name='rnn_'+str(i)
                simp_rnn = GRU(units, activation='relu',
                  return_sequences=True, implementation=2,
name=layer_name)(bn_rnn)
      bn_rnn = BatchNormalization()(simp_rnn)
```

```
TimeDistributed layer:
time_dense = TimeDistributed(Dense(units=output_dim))(bn_rnn)

softmax activation layer:
y_pred = Activation('softmax', name='softmax')(time_dense)

Specify the model:
model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())

model_3 = deep_rnn_model(input_dim=161, units=200, recur_layers=2)
```

3.4 Model 4: Bidirectional RNN + TimeDistributed Dense

For this architecture specify an architecture that uses a single bidirectional RNN layer, before a (TimeDistributed) dense layer.



CH3 4 Bidirectional RNN + TimeDistributed Dense

3.4.1 Pseudo Code

3.4.1.1.1 Model's parameters:

```
input_dim, units, output_dim=29
```

3.4.1.1.2 Main acoustic input:

```
input_data = Input(name='the_input', shape=(None, input_dim))
```

3.4.1.1.3 bidirectional recurrent layer:

```
bidir_rnn =  Bidirectional(GRU(output_dim, return_sequences=True,
implementation=2, name='rnn'), merge_mode='concat')(input_data)
```

3.4.1.1.4 TimeDistributed layer:

```
time_dense = TimeDistributed(Dense(units=output_dim))(bidir_rnn)
```

3.4.1.1.5 softmax activation layer:

```
y_pred = Activation('softmax', name='softmax')(time_dense)
```

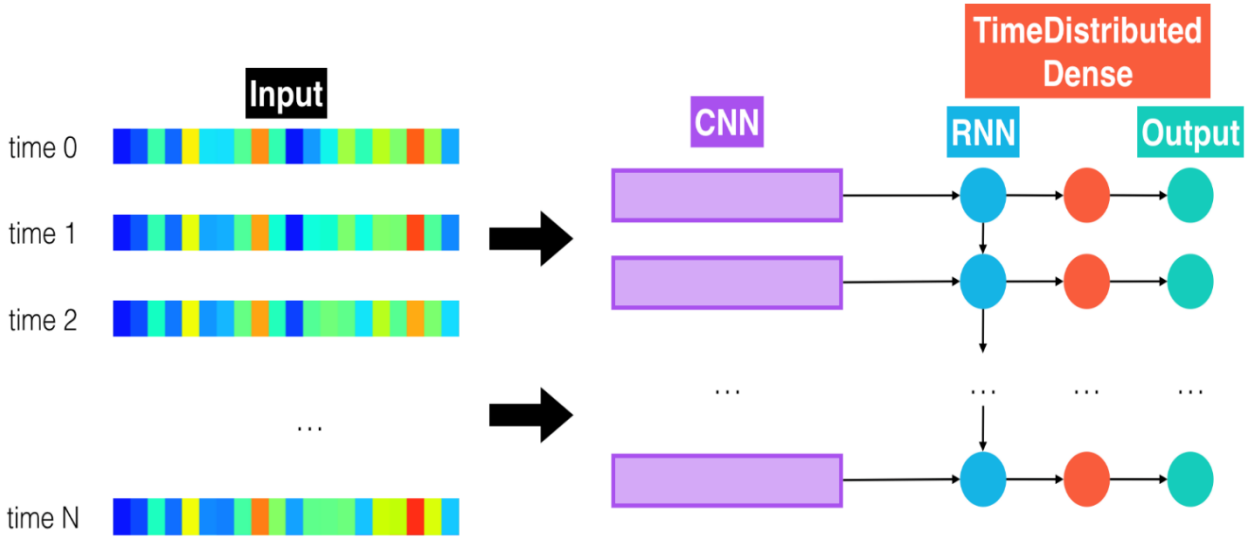
3.4.1.1.6 Specify the model:

```
model = Model(inputs=input_data, outputs=y_pred)
model.output_length=lambda x: x
```

```
model_4 = bidirectional_rnn_model(input_dim=161, units=200)
```

3.5 Model 5: CNN + RNN + TimeDistributed Dense:

The architecture in cnn_rnn_model adds an additional level of complexity, by introducing a 1D convolution layer.



CH3 5 CNN + RNN + TimeDistributed Dense

In this model we have add CNN layer at the beginning (before RNN layer). This way we are analyzing spectrogram or MFCC features with convolutional kernels. With this strategy, we are able to get only important features and give them to RNN part of the network.

3.5.1 Pseudo Code

3.5.1.1.1 Model's parameters:

```
cnn_rnn_model
(input_dim,filters,kernel_size, conv_stride,conv_border_mode, units, output_dim=
29)
```

3.5.1.1.2 Main acoustic input:

```
input_data = Input(name='the_input', shape=(None, input_dim))
```

3.5.1.1.3 Convolutional layer:

Defined argument of convolutional layer, the number of filters, their size, the stride size, the padding and the activation function.

Note: input from the previous layer

```
conv_1d = Conv1D(filters, kernel_size,
                 strides=conv_stride,
            padding=conv_border_mode,
                 activation='relu',
                 name='conv1d')(input_data)
```

3.5.1.1.4 Batch normalization:

The batch normalization layer which takes the output of the previous layer (Convolutional layer) and normalize it to make sure all our features have the same scale.

```
bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)
```

3.5.1.1.5 Recurrent layer:

```
simp_rnn = SimpleRNN(units, activation='relu',return_sequences=True,
implementation=2, name='rnn') (bn_cnn)
```

3.5.1.1.6 Batch normalization:

```
bn_rnn = BatchNormalization()(simp_rnn)
```

3.5.1.1.7 Time Distributed (Dense (output_dim)) layer:

```
time_dense = TimeDistributed (Dense (units=output_dim))(bn_rnn)
```

3.5.1.1.8 Softmax activation layer:

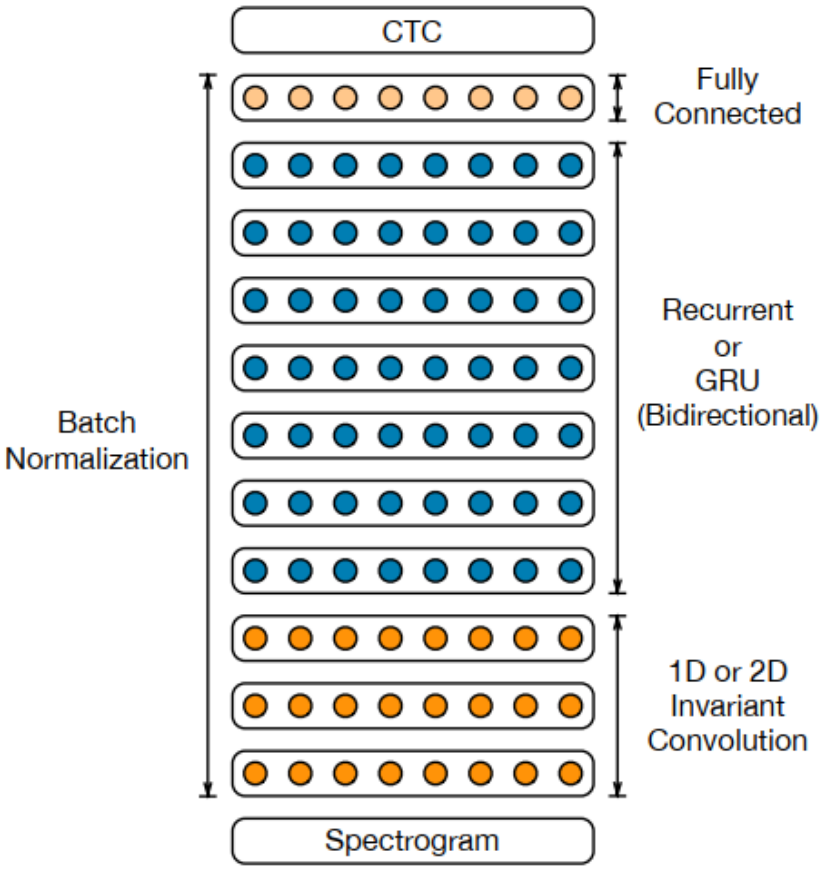
The output layer with a softmax activation

```
y_pred = Activation ('softmax', name='softmax')(time_dense)
```

```
model_2 = cnn_rnn_model(input_dim=161,
                filters=200,
                kernel_size=11,
                conv_stride=2,
                conv_border_mode='valid',
                units=200)
```

3.6 Deep Speech 2 Architecture

A simple multi-layer model with a single recurrent layer cannot exploit thousands of hours of labelled speech. In order to learn from datasets this large, we increase the model capacity via depth. We explore architectures with up to 11 layers including many bidirectional recurrent layers and convolutional layers. These models have nearly 8 times the amount of computation per data example as the models in Deep Speech 1 making fast optimization and computation critical. In order to optimize these models successfully, we use Batch Normalization for RNNs and a novel optimization curriculum we call SortaGrad. We also exploit long strides between RNN inputs to reduce computation per example by a factor of 3. This is helpful for both training and evaluation, though requires some modifications in order to work well with CTC. Finally, though many of our research results make use of bidirectional recurrent layers, we find that excellent models exist using only unidirectional recurrent layers—a feature that makes such models much easier to deploy. Taken together these features allow us to tractably optimize deep RNNs and improve performance by more than 40%.



CH3 6 Deep Speech 2 Architecture

3.6.1 Comparison between different Models and Architecture

This result gathered from previous works in internet we collected it to decide which model we will fit our data on it. There are a lot of hyper parameters in each Model we can tune such as: number of units, layers, GRU or LSTM architecture, using CTC or Attention model, using CNN or not etc.

So, we make this comparison.

3.6.1.1 RNN with Batch Normalization

Comparison of WER on a training and development set for various depths of RNN, with and without BatchNorm. The number of parameters is kept constant as the depth increases, thus the number of hidden units per layer decreases. All networks have 38 million parameters. The architecture "M RNN, N total" implies 1 layer of 1D convolution at the input, M consecutive bidirectional RNN layers, and the rest as fully-connected layers with N total layers in the network.

Architecture	Hidden Units	Train		Dev	
		Baseline	BatchNorm	Baseline	BatchNorm
1 RNN, 5 total	2400	10.55	11.99	13.55	14.40
3 RNN, 5 total	1880	9.55	8.29	11.61	10.56
5 RNN, 7 total	1510	8.59	7.61	10.77	9.78
7 RNN, 9 total	1280	8.76	7.68	10.83	9.52

3.6.1.2 Simple RNN vs. GRU

Comparison of development set WER for networks with either simple RNN or GRU, for various depths. All models have batch normalization, one layer of 1D-invariant convolution, and approximately 38 million parameters.

Architecture	Simple RNN	GRU
5 layers, 1 Recurrent	14.40	10.53
5 layers, 3 Recurrent	10.56	8.00
7 layers, 5 Recurrent	9.78	7.79
9 layers, 7 Recurrent	9.52	8.19

3.6.1.3 CNN

Comparison of WER for various arrangements of convolutional layers. In all cases, the convolutions are followed by 7 recurrent layers and 1 fully connected layer. For 2D-invariant convolutions the first dimension is frequency and the second dimension is time. All models have BatchNorm, SortaGrad, and 35 million parameters.

Architecture	Channels	Filter dimension	Stride	Regular Dev	Noisy Dev
1-layer 1D	1280	11	2	9.52	19.36
2-layer 1D	640, 640	5, 5	1, 2	9.67	19.21
3-layer 1D	512, 512, 512	5, 5, 5	1, 1, 2	9.20	20.22
1-layer 2D	32	41x11	2x2	8.94	16.22
2-layer 2D	32, 32	41x11, 21x11	2x2, 2x1	9.06	15.71
3-layer 2D	32, 32, 96	41x11, 21x11, 21x11	2x2, 2x1, 2x1	8.61	14.74

3.6.2 Pseudo Code

In this Model we will Combine all of Previous Architectures with Customized HyperParameters.

```
Final_model(input_dim, filters, kernel_size, conv_stride, conv_border_mode,
output_dim, recurrent_layers, recurrent_units)
```

Here we defined the model's parameters.

```
input_data = Input(name='the_input', shape=(None, input_dim))
```

Here we define the input layer, which takes as an arguments the dimensions of the inputs.

```
conv_1d = Conv1D(filters, kernel_size,
                 strides=conv_stride,
                 padding=conv_border_mode,
                 activation='relu',
                 name='conv1d')(input_data)
```

Here we defined our first convolutional layer, which takes as an argument the number of filters, their size, the stride size, the padding and the activation function.

```
bn_conv = BatchNormalization(name='bn_conv_1d')(conv_1d)
```

And here we define the batch normalization layer which takes the output of the convolutional layer and normalize it to make sure all our features have the same scale.

```
for i in range(recurrent_layers):
    layer='bidir_rnn_'+str(i)
```

```
bidir_rnn = Bidirectional(GRU(units=recurrent_units,
                              activation='relu',
                              return_sequences=True,
                              implementation=2,
                              name=layer,
                              dropout=0.2),
                          merge_mode='concat')(bidir_rnn)
bidir_rnn=BatchNormalization(name='bn_'+layer)(bidir_rnn)
```

At these previous lines of code, we defined an arbitrary number of GRU Layers, and they are also bidirectional, and then we followed that with a batch normalization layer as before.

```
time_dense = TimeDistributed(Dense(units=output_dim,
name='fc_final'))(bidir_rnn)
```

Last line of code here, we defined a time distributed dense layer, which take each output from the batch normalization layer and apply to them a fully connected layer all at the same time.

```
y_pred = Activation('softmax', name='softmax')(time_dense)
```

Finally, we define the output layer or y_pred with a softmax activation function to make sure all our predictions are between 0 and 1 and they all add to 1.

4. CHAPTER FOUR: SPEECH TO SPEECH PROTOTYPE

In This Chapter We'll Introduce Simple Model and One of Our Stages in Speech2Speech Translator Project, It's A Trigger Word Simply It's Word Which Power Up System Our Trigger Word Will Be An "Activate" Word, Every Time It Hears You Say "Activate," It's Turn on Something.

In this Prototype we will:

- Structure a speech recognition project
- Synthesize and process audio recordings to create train/dev datasets
- Train a trigger word detection model and make predictions

4.1 Data synthesis: Creating a speech dataset

Let's start by building a dataset for your trigger word detection algorithm. A speech dataset should ideally be as close as possible to the application you will want to run it on. In this case, you'd like to detect the word "activate" in working environments (library, home, offices, open-spaces ...). You thus need to create recordings with a mix of positive words ("activate") and negative words (random words other than activate) on different background sounds. Let's see how you can create such a dataset.

We need Positive Examples ("Activate") word, Negative Examples any word else and noise. Also, we need to Merge all together in same clip to be like a real example and indicate place of activate word in clip by label.

4.2 Generating a single training example

Because speech data is hard to acquire and label, we will synthesize our training data using the audio clips of activates, negatives, and backgrounds. It is quite slow to record lots of 10 second audio clips with random "activates" in it. Instead, it is easier to record lots of positives and negative words, and record background noise separately

To synthesize a single training example, we will:

- Pick a random 10 second background audio clip
- Randomly insert 0-4 audio clips of "activate" into this 10sec clip
- Randomly insert 0-2 audio clips of negative words into this 10sec clip

Because we had synthesized the word "activate" into the background clip, we know exactly when in the 10sec clip the "activate" makes its appearance. Later that this makes it easier to generate the labels as well.

```
# Load audio segments using pydub
activates, negatives, backgrounds = load_raw_audio()
```

```
def insert_audio_clip(background, audio_clip, previous_segments):
    """
    Insert a new audio segment over the background noise at a random time step, ensuring that the
    audio segment does not overlap with existing segments.

    Arguments:
    background -- a 10 second background audio recording.
    audio_clip -- the audio clip to be inserted/overlaid.
    previous_segments -- times where audio segments have already been placed

    Returns:
    new_background -- the updated background audio
    """

    # Get the duration of the audio clip in ms
    segment_ms = len(audio_clip)

    ### START CODE HERE ###
    # Step 1: Use one of the helper functions to pick a random time segment onto which to insert
    # the new audio clip. (≈ 1 line)
    segment_time = get_random_time_segment(segment_ms)

    # Step 2: Check if the new segment_time overlaps with one of the previous_segments. If so, kee
p
    # picking new segment_time at random until it doesn't overlap. (≈ 2 lines)
    while is_overlapping(segment_time, previous_segments):
        segment_time = get_random_time_segment(segment_ms)

    # Step 3: Add the new segment_time to the list of previous_segments (≈ 1 line)
    previous_segments.append(segment_time)
    ### END CODE HERE ###

    # Step 4: Superpose audio segment and background
    new_background = background.overlay(audio_clip, position = segment_time[0])

    return new_background, segment_time
```

4.3 Model

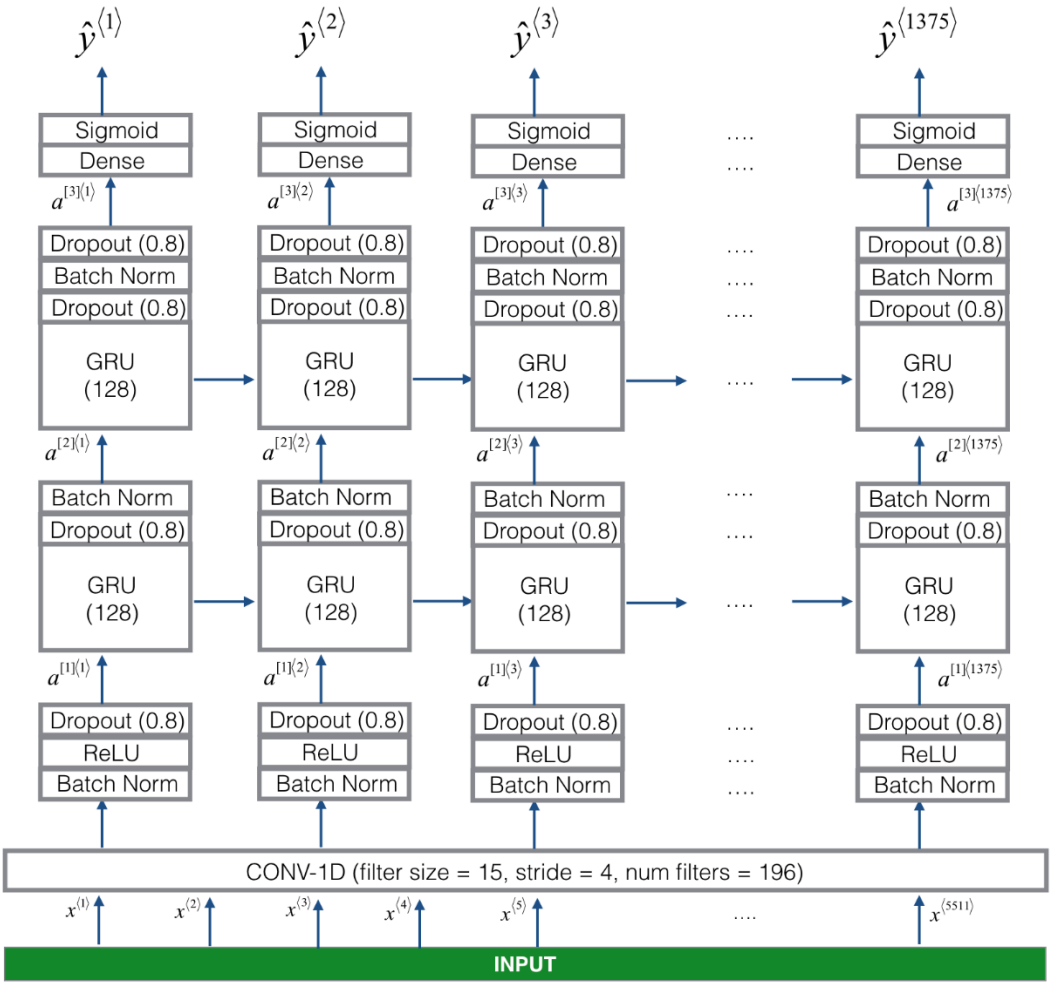
Now that we've built a dataset, lets write and train a trigger word detection model!

The model will use 1-D convolutional layers, GRU layers, and dense layers. Let's load the packages that will allow us to use these layers in Keras.

```
from keras.callbacks import ModelCheckpoint
from keras.models import Model, load_model, Sequential
from keras.layers import Dense, Activation, Dropout, Input, Masking, TimeDistributed, LSTM, Conv1D
from keras.layers import GRU, Bidirectional, BatchNormalization, Reshape
from keras.optimizers import Adam
```

4.3.1 Build the model

Here is the architecture we will use.



CH4 1 Trigger Word Model ARCH

4.3.1.1 Implementing the model

Step 1: CONV layer. Use Conv1D() to implement this, with 196 filters, a filter size of 15 (kernel_size=15), and stride of 4.

Step 2: First GRU layer. To generate the GRU layer, use:

X = GRU(units = 128, return_sequences = True)(X)

Setting return_sequences=True ensures that all the GRU's hidden states are fed to the next layer. Remember to follow this with Dropout and BatchNorm layers.

Step 3: Second GRU layer. This is similar to the previous GRU layer (remember to use return_sequences=True), but has an extra dropout layer.

Step 4: Create a time-distributed dense layer as follows:

X = TimeDistributed(Dense(1, activation = "sigmoid"))(X)

This creates a dense layer followed by a sigmoid, so that the parameters used for the dense layer are the same for every time step.

```
def model(input_shape):
    """
    Function creating the model's graph in Keras.

    Argument:
    input_shape -- shape of the model's input data (using Keras conventions)

    Returns:
    model -- Keras model instance
    """

    X_input = Input(shape = input_shape)

    ### START CODE HERE ###

    # Step 1: CONV layer (≈4 lines)
    X = Conv1D(196, kernel_size=15, strides=4)(X_input)                    # CONV1D
    X = BatchNormalization()(X)                          # Batch normalization
    X = Activation('relu')(X)                        # ReLu activation
    X = Dropout(0.8)(X)                          # dropout (use 0.8)

    # Step 2: First GRU Layer (≈4 lines)
    X = GRU(units = 128, return_sequences = True)(X) # GRU (use 128 units and return the sequ
ences)
    X = Dropout(0.8)(X)                          # dropout (use 0.8)
    X = BatchNormalization()(X)                          # Batch normalization
```

```
    # Step 3: Second GRU Layer (≈4 lines)
    X = GRU(units = 128, return_sequences = True)(X)   # GRU (use 128 units and return the seq
uences)
    X = Dropout(0.8)(X)                                # dropout (use 0.8)
    X = BatchNormalization()(X)                                # Batch normalization
    X = Dropout(0.8)(X)                                # dropout (use 0.8)

    # Step 4: Time-distributed dense layer (≈1 line)
    X = TimeDistributed(Dense(1, activation = "sigmoid"))(X) # time distributed  (sigmoid)

    ### END CODE HERE ###

    model = Model(inputs = X_input, outputs = X)

    return model
```

4.3.2 Fit the model

```
opt = Adam(lr=0.0001, beta_1=0.9, beta_2=0.999, decay=0.01)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=["accuracy"])
model.fit(X, Y, batch_size = 5, epochs=1)
```

4.3.3 Test the model

```
loss, acc = model.evaluate(X_dev, Y_dev)
print("Dev set accuracy = ", acc)
```

```
25/25 [==============================] - 4s
Dev set accuracy =  0.945629119873
```

We've come to the end of this Prototype!

- Data synthesis is an effective way to create a large training set for speech problems, specifically trigger word detection.
- Using a spectrogram and optionally a 1D conv layer is a common pre-processing step prior to passing audio data to an RNN, GRU or LSTM.
- An end-to-end deep learning approach can be used to built a very effective trigger word detection system.

Full code [18].

5. CHAPTER FIVE: SYSTEM IMPLEMENTATION

5.1 Description of Dataset

Dataset is the core of our project, so we chose LibriSpeech to be suitable for training and testing Which LibriSpeech database provides large-scale clean speech utterances.

LibriSpeech is a corpus of read speech, based on LibriVox's public domain audio books. Its purpose is to enable the training and testing of automatic speech recognition (ASR) systems.

5.1.1 LibriSpeech Structure

The corpus is split into several parts to enable users to selectively download subsets of it, according to their needs. The subsets with "clean" in their name are supposedly "cleaner"(at least on average), than the rest of the audio and US English accented. That classification was obtained using very crude automated means and should not be considered completely reliable. The subsets are disjoint, i.e., the audio of each speaker is assigned to exactly one subset.

The parts of the corpus that has been worked on are as follows:

* dev-clean, test-clean - development and test set containing "clean" speech.

When extracted, each of the {dev,test} sets re-creates LibriSpeech's root directory, containing some metadata, and a dedicated subdirectory for the subset itself. The audio for each individual speaker is stored under a dedicated subdirectory in the subset's directory, and each audio chapter read by this speaker is stored in separate sub subdirectory.

The main meta info about the speech is listed in the READERS and the CHAPTERS:

- READERS.TXT contains information about speaker's gender and total amount of audio in the corpus.

The meaning of the fields in left-to-right order is as follows:

reader_id: the ID of the reader in the LibriVox's database

gender: 'F' for female, 'M' for male

subset: the corpus subset to which the reader's audio is assigned.

duration: total number of minutes of speech by the reader, included in the corpus.

name: the name under which the reader is registered in LibriVox

```
;ID   |SEX| SUBSET            |MINUTES| NAME
14    | F | train-clean-360   | 25.03 | Kristin LeMoine
16    | F | train-clean-360   | 25.11 | Alys AtteWater
17    | M | train-clean-360   | 25.04 | Gord Mackenzie
19    | F | train-clean-100   | 25.19 | Kara Shallenberg
20    | F | train-other-500   | 30.07 | Gesine
22    | F | train-clean-360   | 25.14 | Michelle Crandall
23    | F | train-clean-360   | 25.23 | Anita Roy Dobbs
25    | M | train-other-500   | 30.16 | John Gonzalez
26    | M | train-clean-100   | 25.08 | Denny Sayers
27    | M | train-clean-100   | 20.14 | Sean McKinley
28    | F | train-clean-360   | 25.03 | Kristin Hughes
29    | M | train-other-500   | 30.10 | Linton
30    | F | train-clean-360   | 25.19 | Annie Coleman Rothenberg
31    | M | train-other-500   | 23.79 | Martin Clifton
32    | F | train-clean-100   | 24.01 | Betsie Bush
36    | M | train-other-500   | 25.85 | Chip
37    | M | train-other-500   | 30.04 | wolvrin
38    | M | train-clean-360   | 25.11 | R. Francis Smith
39    | F | train-clean-100   | 25.05 | Sherry Crowther
40    | F | train-clean-100   | 25.04 | Vicki Barbour
```

- CHAPTERS.TXT has information about the per-chapter audio durations.
The meaning of the fields in left-to-right order is as follows:
chapter_id: the ID of the chapter in the LibriVox's database.
reader_id: the ID of the reader in the LibriVox's database.
duration: how many minutes of this chapter are used in the corpus.
subset: the corpus subset to which this chapter is assigned.
project_id: the LibriVox project ID.
book_id: The Project Gutenberg's ID for the book on which the LibriVox project is based.
chapter_title: the title of the chapter on LibriVox.
project_title: the title of the LibriVox project.

;ID	READER	MINUTES	SUBSET	PROJ.	BOOK ID	CH. TITLE	PROJECT TITLE
1	110	19.77	train-other-500	53	1023	In Chancery	Bleak House
2	110	10.30	train-other-500	53	1023	In Fashion	Bleak House
159	4174	7.67	train-other-500	68	2184	Letter XXV	Unbeaten Tracks in Japan
198	19	8.42	train-clean-100	219	121	Chapter 01	Northanger Abbey
199	98	11.68	train-clean-360	219	121	Chapter 02	Northanger Abbey
200	173	11.25	train-other-500	219	121	Chapter 03	Northanger Abbey
201	44	7.57	train-other-500	219	121	Chapter 04	Northanger Abbey
204	92	12.76	train-other-500	219	121	Chapter 07	Northanger Abbey
205	20	12.82	train-other-500	219	121	Chapter 08	Northanger Abbey
207	44	18.33	train-other-500	219	121	Chapter 10	Northanger Abbey
208	14	12.95	train-clean-360	219	121	Chapter 11	Northanger Abbey
209	198	8.20	train-clean-100	219	121	Chapter 12	Northanger Abbey
212	14	12.09	train-clean-360	219	121	Chapter 15	Northanger Abbey
214	37	6.19	train-other-500	219	121	Chapter 17	Northanger Abbey
215	37	8.28	train-other-500	219	121	Chapter 18	Northanger Abbey
216	403	6.30	train-clean-100	219	121	Chapter 19	Northanger Abbey
218	89	12.78	train-clean-100	219	121	Chapter 21	Northanger Abbey
219	89	12.30	train-clean-100	219	121	Chapter 22	Northanger Abbey
222	40	15.59	train-clean-100	219	121	Chapter 25	Northanger Abbey
224	246	7.56	train-clean-360	219	121	Chapter 27	Northanger Abbey
225	37	15.58	train-other-500	219	121	Chapter 28	Northanger Abbey
227	19	16.77	train-clean-100	219	121	Chapter 30	Northanger Abbey
348	728	30.21	train-other-500	381	84	00 - Letters	Frankenstein, or The Modern Prometheus
352	166	10.65	train-clean-360	381	84	04 - Chapter 5	Frankenstein, or The Modern Prometheus
354	66	13.84	train-other-500	381	84	06 - Chapter 7	Frankenstein, or The Modern Prometheus

- The file BOOKS.TXT makes contains the title for each book, whose text is used in the corpus, and its Project Gutenberg ID.

1	United States Declaration of Independence	
11	Alice's Adventures in Wonderland	
12	Through the Looking-Glass	
13	The Hunting of the Snark	
16	Peter Pan	
17	The Book of Mormon	Church of Jesus Christ of Latter-day Saints, Joseph, Jr. Smith, Joseph Smith
20	Paradise Lost	
23	Narrative of the Life of Frederick Douglass, an American Slave	Frederick Douglass, Frederick Augustus Washington Bailey, Frederick Augustus Washington Baly
24	O Pioneers!	
32	Herland	
33	The Scarlet Letter	
35	The Time Machine	
36	The War of the Worlds	
44	The Song of the Lark	
45	Anne of Green Gables	
46	A Christmas Carol in Prose; Being a Ghost Story of Christmas	
47	Anne of Avonlea	

5.1.2 How Data Prepared

Data preparation is the step after data collection in the machine learning life cycle and it is the process of cleaning and transforming the raw data you collected. By doing so, we will have a much easier time when it comes to analyzing and modeling our data.

```
"""
Use this script to create JSON-Line description files that can be used to
train deep-speech models through this library.
This works with data directories that are organized like LibriSpeech:
data_directory/group/speaker/[file_id1.wav, file_id2.wav, ...,
                              speaker.trans.txt]
Where speaker.trans.txt has in each line, file_id transcription
"""
#Importing Libraries

from __future__ import absolute_import, division, print_function
import argparse  # importing  Parser for command-line options, arguments and sub-
commands
import json      # importing json to work with json file
import os        #importing os module
import wave      #importing to read and write wav files


def main(data_directory, output_file):
    labels = []
    durations = []
    keys = []
    for group in os.listdir(data_directory):
        if group.startswith('.'):
            continue

       #Define speaker_path
        speaker_path = os.path.join(data_directory, group)
        for speaker in os.listdir(speaker_path):
            if speaker.startswith('.'):
                continue
            labels_file = os.path.join(speaker_path, speaker,
                                       '{}-{}.trans.txt'
                                       .format(group, speaker))
             # Spliting the data
            for line in open(labels_file):
                split = line.strip().split()
                file_id = split[0]
```

```
            label = ' '.join(split[1:]).lower()
            audio_file = os.path.join(speaker_path, speaker,
                                      file_id) + '.wav'

            audio = wave.open(audio_file)  #Define audio for audio file
            duration = float(audio.getnframes()) / audio.getframerate()   #Calculating
duration
            audio.close()
            keys.append(audio_file)
            durations.append(duration)
            labels.append(label)

#Saving data in output file
    with open(output_file, 'w') as out_file:
        for i in range(len(keys)):
            line = json.dumps({'key': keys[i], 'duration': durations[i],
                               'text': labels[i]})
            out_file.write(line + '\n')


if __name__ == '__main__':

    parser = argparse.ArgumentParser()    #creating an ArgumentParser object

    #Filling an ArgumentParser with information about program arguments
    parser.add_argument('data_directory', type=str,
                        help='Path to data directory')

    parser.add_argument('output_file', type=str,
                        help='Path to output file')

    #calling parse_args() will return an object with two attributes, integers and
accumulate.
    args = parser.parse_args()
    main(args.data_directory, args.output_file)
```

5.1.3 Output


```
{
  "key": "D:\\Practicing\\DataSets\\LibriSpeech\\test-clean\\1089\\134686\\1089-134686-0000.wav",
  "duration": 10.435,
  "text": "he hoped there would be stew for dinner turnips and carrots and bruised potatoes and fat
} {
  "key": "D:\\Practicing\\DataSets\\LibriSpeech\\test-clean\\1089\\134686\\1089-134686-0001.wav",
  "duration": 3.275,
  "text": "stuff it into you his belly counselled him"
} {
  "key": "D:\\Practicing\\DataSets\\LibriSpeech\\test-clean\\1089\\134686\\1089-134686-0002.wav",
  "duration": 6.625,
  "text": "after early nightfall the yellow lamps would light up here and there the squalid quarter
} {
  "key": "D:\\Practicing\\DataSets\\LibriSpeech\\test-clean\\1089\\134686\\1089-134686-0003.wav",
  "duration": 2.68,
  "text": "hello bertie any good in your mind"
} {
```


5.2 Data Generator

class that is used to featurize audio clips, and provide them to the network for training or testing.

```
# importing json to work with json file
import json
# importing numpy to work with tensors and arrays and provid
# mathematical solutions
import numpy as np
# import random to generate rondom numbers
import random
# importing mfcc to convert audio to mfcc spectrum
from python_speech_features import mfcc
import librosa
# import wav to work with .wav files
import scipy.io.wavfile as wav
# import matplotlib to provides plotting
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

from utils import calc_feat_dim, spectrogram_from_file, text_to_int_sequence
from utils import conv_output_length

RNG_SEED = 123
```

```
"""
Defines a class that is used to featurize audio clips, and provide
them to the network for training or testing.
"""


class AudioGenerator():
# This function Initializes the parameter of this class

    def __init__(self, step=10, window=20, max_freq=8000, mfcc_dim=13,
        minibatch_size=20, desc_file=None, spectrogram=True, max_duration=20.0,
        sort_by_duration=False):
        """
        Params:
            step (int): Step size in milliseconds between windows (for spectrogram ONLY)
            window (int): FFT window size in milliseconds (for spectrogram ONLY)
            max_freq (int): Only FFT bins corresponding to frequencies between
                [0, max_freq] are returned (for spectrogram ONLY)
            desc_file (str, optional): Path to a JSON-line file that contains
                labels and paths to the audio files. If this is None, then
                load metadata right away
        """




      # initializing parameters
      self.feat_dim = calc_feat_dim(window, max_freq)
        self.mfcc_dim = mfcc_dim
        self.feats_mean = np.zeros((self.feat_dim,))
        self.feats_std = np.ones((self.feat_dim,))
        self.rng = random.Random(RNG_SEED)
        if desc_file is not None:
            self.load_metadata_from_desc_file(desc_file)
        self.step = step
        self.window = window
        self.max_freq = max_freq
        self.cur_train_index = 0
        self.cur_valid_index = 0
        self.cur_test_index = 0
        self.max_duration=max_duration
        self.minibatch_size = minibatch_size
        self.spectrogram = spectrogram
        self.sort_by_duration = sort_by_duration
```

```
# this function gets a batch of the audio data from the path they are saved into, it takes
 # an argument partition which refers to which dataset it would take a batch from, it also
 # calculates the max length , initialize the arrays and then returns the arrays ready to
 # be processed
    def get_batch(self, partition):
        """ Obtain a batch of train, validation, or test data
        """
        if partition == 'train':
            audio_paths = self.train_audio_paths
            cur_index = self.cur_train_index
            texts = self.train_texts
        elif partition == 'valid':
            audio_paths = self.valid_audio_paths
            cur_index = self.cur_valid_index
            texts = self.valid_texts
        elif partition == 'test':
            audio_paths = self.test_audio_paths
            cur_index = self.test_valid_index
            texts = self.test_texts
        else:
            raise Exception("Invalid partition. "
                "Must be train/validation")

        features = [self.normalize(self.featurize(a)) for a in
            audio_paths[cur_index:cur_index+self.minibatch_size]]

        # calculate necessary sizes
        max_length = max([features[i].shape[0]
            for i in range(0, self.minibatch_size)])
        max_string_length = max([len(texts[cur_index+i])
            for i in range(0, self.minibatch_size)])

        # initialize the arrays
        X_data = np.zeros([self.minibatch_size, max_length,
            self.feat_dim*self.spectrogram + self.mfcc_dim*(not self.spectrogram)])
        labels = np.ones([self.minibatch_size, max_string_length]) * 28
        input_length = np.zeros([self.minibatch_size, 1])
        label_length = np.zeros([self.minibatch_size, 1])

        for i in range(0, self.minibatch_size):
            # calculate X_data & input_length
            feat = features[i]
            input_length[i] = feat.shape[0]
```

```
            X_data[i, :feat.shape[0], :] = feat

            # calculate labels & label_length
            label = np.array(text_to_int_sequence(texts[cur_index+i]))
            labels[i, :len(label)] = label
            label_length[i] = len(label)

        # return the arrays
        outputs = {'ctc': np.zeros([self.minibatch_size])}
        inputs = {'the_input': X_data,
                  'the_labels': labels,
                  'input_length': input_length,
                  'label_length': label_length
                 }
        return (inputs, outputs)

# this function take as an argument the partition and then it shuffles the data
    def shuffle_data_by_partition(self, partition):
        """ Shuffle the training or validation data
        """
        if partition == 'train':
            self.train_audio_paths, self.train_durations, self.train_texts = shuffle_data(
                self.train_audio_paths, self.train_durations, self.train_texts)
        elif partition == 'valid':
            self.valid_audio_paths, self.valid_durations, self.valid_texts = shuffle_data(
                self.valid_audio_paths, self.valid_durations, self.valid_texts)
        else:
            raise Exception("Invalid partition. "
                "Must be train/validation")

# this function sorts the data by increasing duration, it takes as an argument the
# partition
    def sort_data_by_duration(self, partition):
        """ Sort the training or validation sets by (increasing) duration
        """
        if partition == 'train':
            self.train_audio_paths, self.train_durations, self.train_texts = sort_data(
                self.train_audio_paths, self.train_durations, self.train_texts)
        elif partition == 'valid':
            self.valid_audio_paths, self.valid_durations, self.valid_texts = sort_data(
                self.valid_audio_paths, self.valid_durations, self.valid_texts)
        else:
            raise Exception("Invalid partition. "
```

```
            "Must be train/validation")

 # this funtion obtains a batch of training data, inside it, it calls the get_batch function
    def next_train(self):
        """ Obtain a batch of training data
        """
        while True:
            ret = self.get_batch('train')
            self.cur_train_index += self.minibatch_size
            if self.cur_train_index >= len(self.train_texts) - self.minibatch_size:
                self.cur_train_index = 0
                self.shuffle_data_by_partition('train')
            yield ret

# this funtion obtains a batch of validation data, inside it, it calls the get_batch function
    def next_valid(self):
        """ Obtain a batch of validation data
        """
        while True:
            ret = self.get_batch('valid')
            self.cur_valid_index += self.minibatch_size
            if self.cur_valid_index >= len(self.valid_texts) - self.minibatch_size:
                self.cur_valid_index = 0
                self.shuffle_data_by_partition('valid')
            yield ret

# this funtion obtains a batch of test data, inside it, it calls the get_batch function
    def next_test(self):
        """ Obtain a batch of test data
        """
        while True:
            ret = self.get_batch('test')
            self.cur_test_index += self.minibatch_size
            if self.cur_test_index >= len(self.test_texts) - self.minibatch_size:
                self.cur_test_index = 0
            yield ret

# this funtion loads the train data from the corpus
    def load_train_data(self, desc_file='train_corpus.json'):
        self.load_metadata_from_desc_file(desc_file, 'train')
        self.fit_train()
        if self.sort_by_duration:
            self.sort_data_by_duration('train')
```

```
# this funtion loads the validation data from the corpus
    def load_validation_data(self, desc_file='valid_corpus.json'):
        self.load_metadata_from_desc_file(desc_file, 'validation')
        if self.sort_by_duration:
            self.sort_data_by_duration('valid')

# this funtion loads the test data from the corpus
    def load_test_data(self, desc_file='test_corpus.json'):
        self.load_metadata_from_desc_file(desc_file, 'test')

# this function takes as an argument the file path and partition, and opens the json file,
# and assigns the paths in it to their corresponding variables
    def load_metadata_from_desc_file(self, desc_file, partition):
        """ Read metadata from a JSON-line file
            (possibly takes long, depending on the filesize)
        Params:
            desc_file (str):  Path to a JSON-line file that contains labels and
                paths to the audio files
            partition (str): One of 'train', 'validation' or 'test'
        """
        audio_paths, durations, texts = [], [], []
        with open(desc_file) as json_line_file:
            for line_num, json_line in enumerate(json_line_file):
                try:
                    spec = json.loads(json_line)
                    if float(spec['duration']) > self.max_duration:
                        continue
                    audio_paths.append(spec['key'])
                    durations.append(float(spec['duration']))
                    texts.append(spec['text'])
                except Exception as e:
                    # Change to (KeyError, ValueError) or
                    # (KeyError,json.decoder.JSONDecodeError), depending on
                    # json module version
                    print('Error reading line #{}: {}'
                              .format(line_num, json_line))
        if partition == 'train':
            self.train_audio_paths = audio_paths
            self.train_durations = durations
            self.train_texts = texts
        elif partition == 'validation':
            self.valid_audio_paths = audio_paths
```

```
            self.valid_durations = durations
            self.valid_texts = texts
        elif partition == 'test':
            self.test_audio_paths = audio_paths
            self.test_durations = durations
            self.test_texts = texts
        else:
            raise Exception("Invalid partition to load metadata. "
             "Must be train/validation/test")

# this function takes as an argument the number of samples and calculates the mean
# and standard deviation of the features
    def fit_train(self, k_samples=100):
        """ Estimate the mean and std of the features from the training set
        Params:
            k_samples (int): Use this number of samples for estimation
        """
        k_samples = min(k_samples, len(self.train_audio_paths))
        samples = self.rng.sample(self.train_audio_paths, k_samples)
        feats = [self.featurize(s) for s in samples]
        feats = np.vstack(feats)
        self.feats_mean = np.mean(feats, axis=0)
        self.feats_std = np.std(feats, axis=0)

# For a given audio clip, calculate the corresponding feature
    def featurize(self, audio_clip):
        """ For a given audio clip, calculate the corresponding feature
        Params:
            audio_clip (str): Path to the audio clip
        """
        if self.spectrogram:
            return spectrogram_from_file(
                audio_clip, step=self.step, window=self.window,
                max_freq=self.max_freq)
        else:
            (rate, sig) = wav.read(audio_clip)
            return mfcc(sig, rate, numcep=self.mfcc_dim)

# Center a feature using the mean and std
    def normalize(self, feature, eps=1e-14):
        """ Center a feature using the mean and std
        Params:
            feature (numpy.ndarray): Feature to normalize
```

```
        """
        return (feature - self.feats_mean) / (self.feats_std + eps)

# Shuffle the data (called after making a complete pass through

 # training or validation data during the training process)
# take the audio paths and their durations and transcriptions and shuffle them
def shuffle_data(audio_paths, durations, texts):
    """ Shuffle the data (called after making a complete pass through
        training or validation data during the training process)
    Params:
        audio_paths (list): Paths to audio clips
        durations (list): Durations of utterances for each audio clip
        texts (list): Sentences uttered in each audio clip
    """
    p = np.random.permutation(len(audio_paths))
    audio_paths = [audio_paths[i] for i in p]
    durations = [durations[i] for i in p]
    texts = [texts[i] for i in p]
    return audio_paths, durations, texts

# sort data by duration, takes in audio paths, duration, transcripts and return a sorted
# version of them
def sort_data(audio_paths, durations, texts):
    """ Sort the data by duration
    Params:
        audio_paths (list): Paths to audio clips
        durations (list): Durations of utterances for each audio clip
        texts (list): Sentences uttered in each audio clip
    """
    p = np.argsort(durations).tolist()
    audio_paths = [audio_paths[i] for i in p]
    durations = [durations[i] for i in p]
    texts = [texts[i] for i in p]
    return audio_paths, durations, texts

# Visualizing the data point in the training set at the supplied index
def vis_train_features(index=0):
    """ Visualizing the data point in the training set at the supplied index
    """
    # obtain spectrogram
    audio_gen = AudioGenerator(spectrogram=True)
    audio_gen.load_train_data()
    vis_audio_path = audio_gen.train_audio_paths[index]
```

```
    vis_spectrogram_feature =
audio_gen.normalize(audio_gen.featurize(vis_audio_path))
    # obtain mfcc
    audio_gen = AudioGenerator(spectrogram=False)
    audio_gen.load_train_data()
    vis_mfcc_feature = audio_gen.normalize(audio_gen.featurize(vis_audio_path))
    # obtain text label
    vis_text = audio_gen.train_texts[index]
    # obtain raw audio
    vis_raw_audio, _ = librosa.load(vis_audio_path)
    # print total number of training examples
    print('There are %d total training examples.' % len(audio_gen.train_audio_paths))
    # return labels for plotting
    return vis_text, vis_raw_audio, vis_mfcc_feature, vis_spectrogram_feature,
vis_audio_path


def plot_raw_audio(vis_raw_audio):
    # plot the raw audio signal
    fig = plt.figure(figsize=(12,3))
    ax = fig.add_subplot(111)
    steps = len(vis_raw_audio)
    ax.plot(np.linspace(1, steps, steps), vis_raw_audio)
    plt.title('Audio Signal')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
    plt.show()

def plot_mfcc_feature(vis_mfcc_feature):
    # plot the MFCC feature
    fig = plt.figure(figsize=(12,5))
    ax = fig.add_subplot(111)
    im = ax.imshow(vis_mfcc_feature, cmap=plt.cm.jet, aspect='auto')
    plt.title('Normalized MFCC')
    plt.ylabel('Time')
    plt.xlabel('MFCC Coefficient')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    plt.colorbar(im, cax=cax)
    ax.set_xticks(np.arange(0, 13, 2), minor=False);
    plt.show()

def plot_spectrogram_feature(vis_spectrogram_feature):
```

```
# plot the normalized spectrogram
fig = plt.figure(figsize=(12,5))
ax = fig.add_subplot(111)
im = ax.imshow(vis_spectrogram_feature, cmap=plt.cm.jet, aspect='auto')
plt.title('Normalized Spectrogram')
plt.ylabel('Time')
plt.xlabel('Frequency')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(im, cax=cax)
plt.show()
```

5.3 Characters Mapping

Defines two dictionaries for converting between text and integer sequences.

```
char_map_str = """
' 0
<SPACE> 1
a 2
b 3
c 4
d 5
e 6
f 7
g 8
h 9
i 10
j 11
k 12
l 13
m 14
n 15
o 16
p 17
q 18
r 19
s 20
t 21
u 22
v 23
w 24
```

```
x 25
y 26
z 27
"""
# the "blank" character is mapped to 28

char_map = {}
index_map = {}
for line in char_map_str.strip().split('\n'):
    ch, index = line.split()
    char_map[ch] = int(index)
    index_map[int(index)+1] = ch
index_map[2] = ' '
```

5.4 Data Conversion

```
#this function take the characters and convert it to int then return it
def text_to_int_sequence(text):
    """ Convert text to an integer sequence """
    int_sequence = []
    for c in text:
        if c == ' ':
            ch = char_map['<SPACE>']
        else:
            ch = char_map[c]
        int_sequence.append(ch)
    return int_sequence
```

```
#this function takes the int and map it into characters
def int_sequence_to_text(int_sequence):
    """ Convert an integer sequence to text """
    text = []
    for c in int_sequence:
        ch = index_map[c]
        text.append(ch)
    return text
```

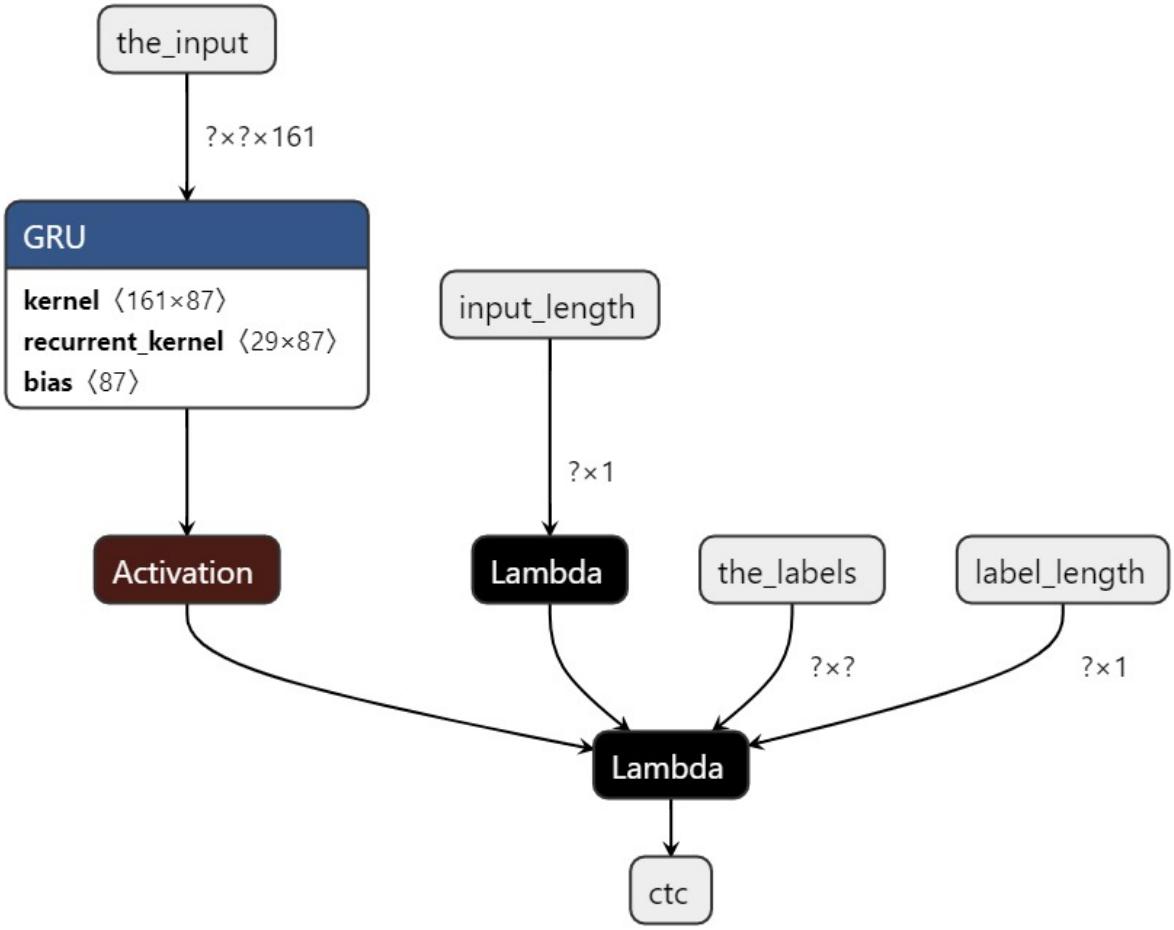
5.5 Model Design

in this Section we will talk about Model Implementation of Designs which mentioned in CHAPTER 4.

Before Discuss each Model Implementation we import Required Libraries

```
from keras import backend as K
from keras.models import Model
from keras.layers import (BatchNormalization, Conv1D, Dense, Input,
    TimeDistributed, Activation, Bidirectional, SimpleRNN, GRU, LSTM, Dropout)
```

5.5.1 Simple RNN Model

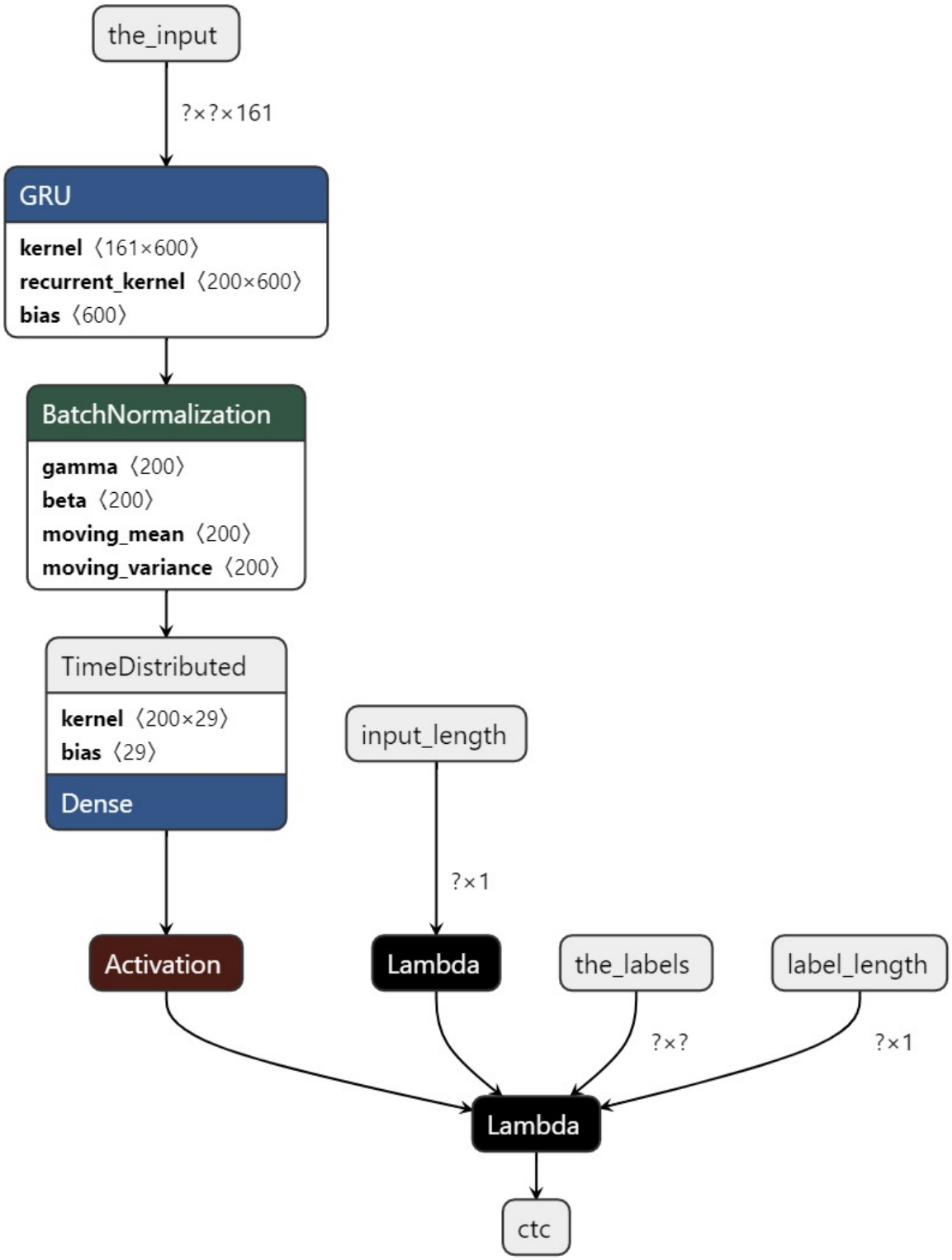


The Simple_RNN_MODEL Function takes input dimension which 13 or 169 according to selected features spectrogram or MFCC. And produce 29 output according to 26 letters and other special characters.

```
def simple_rnn_model(input_dim, output_dim=29):
    """ Build a recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add recurrent layer
    simp_rnn = GRU(output_dim, return_sequences=True, name='rnn')(input_data)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(simp_rnn)
```

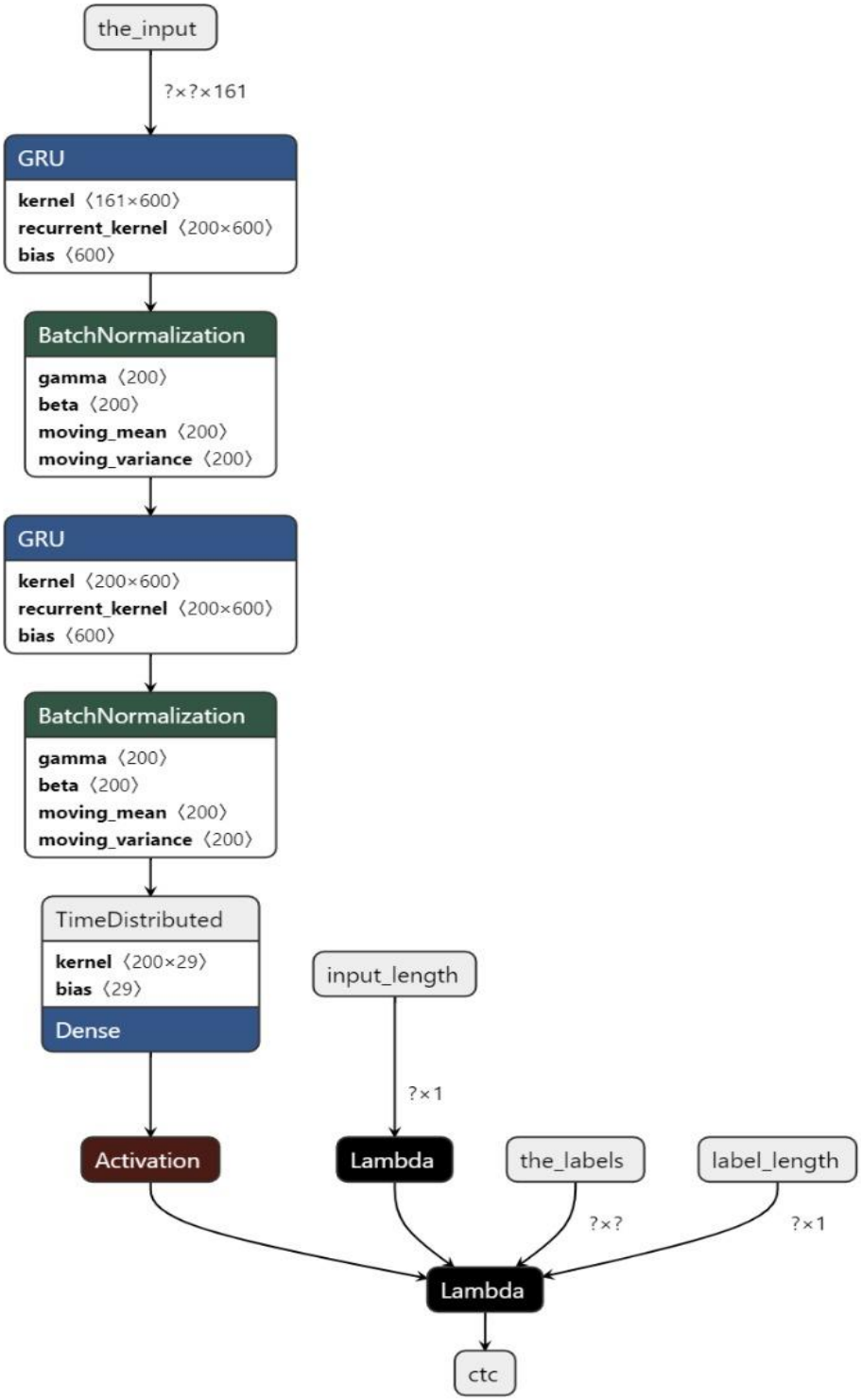
```
# Specify the model
model = Model(inputs=input_data, outputs=y_pred)

# specify output to feed ctc later
model.output_length = lambda x: x

# print Summary of Model's Parameters
print(model.summary())

# return Model Object
return model
```

5.5.2 RNN + TimeDistributed Dense Model



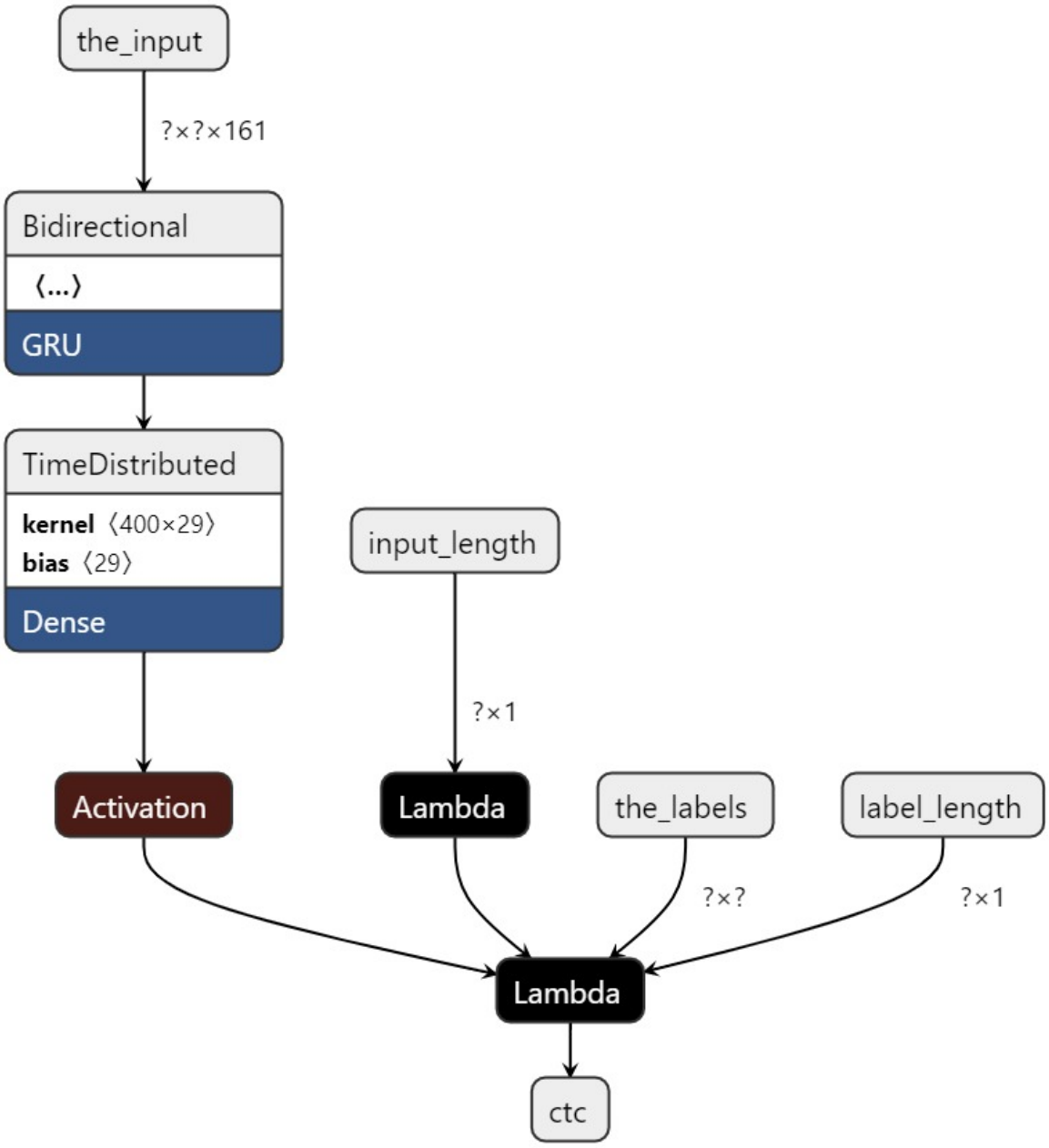
```
def rnn_model(input_dim, units, activation, output_dim=29):
    """ Build a recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add recurrent layer
    simp_rnn = GRU(units, activation=activation,
        return_sequences=True, name='rnn')(input_data)
    #  Add batch normalization
    bn_rnn = BatchNormalization(name='bn_rnn')(simp_rnn)
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim), name='time_dense')(bn_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model
```

5.5.3 Deeper RNN Model



```
def deep_rnn_model(input_dim, units, recur_layers, output_dim=29):
    """ Build a deep recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add recurrent layers, each with batch normalization

    dummy_layer = input_data
    for n in range(recur_layers):
        # Add RNN layer with batch normalization
        rnn = GRU(units, activation='relu', return_sequences=True,
                  name='rnn_{}'.format(n))(dummy_layer)
        bn_rnn = BatchNormalization(name='bn_rnn_{}'.format(n))(rnn)
        dummy_layer = bn_rnn
    last_layer = dummy_layer
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim), name='time_dense')(last_layer)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model
```

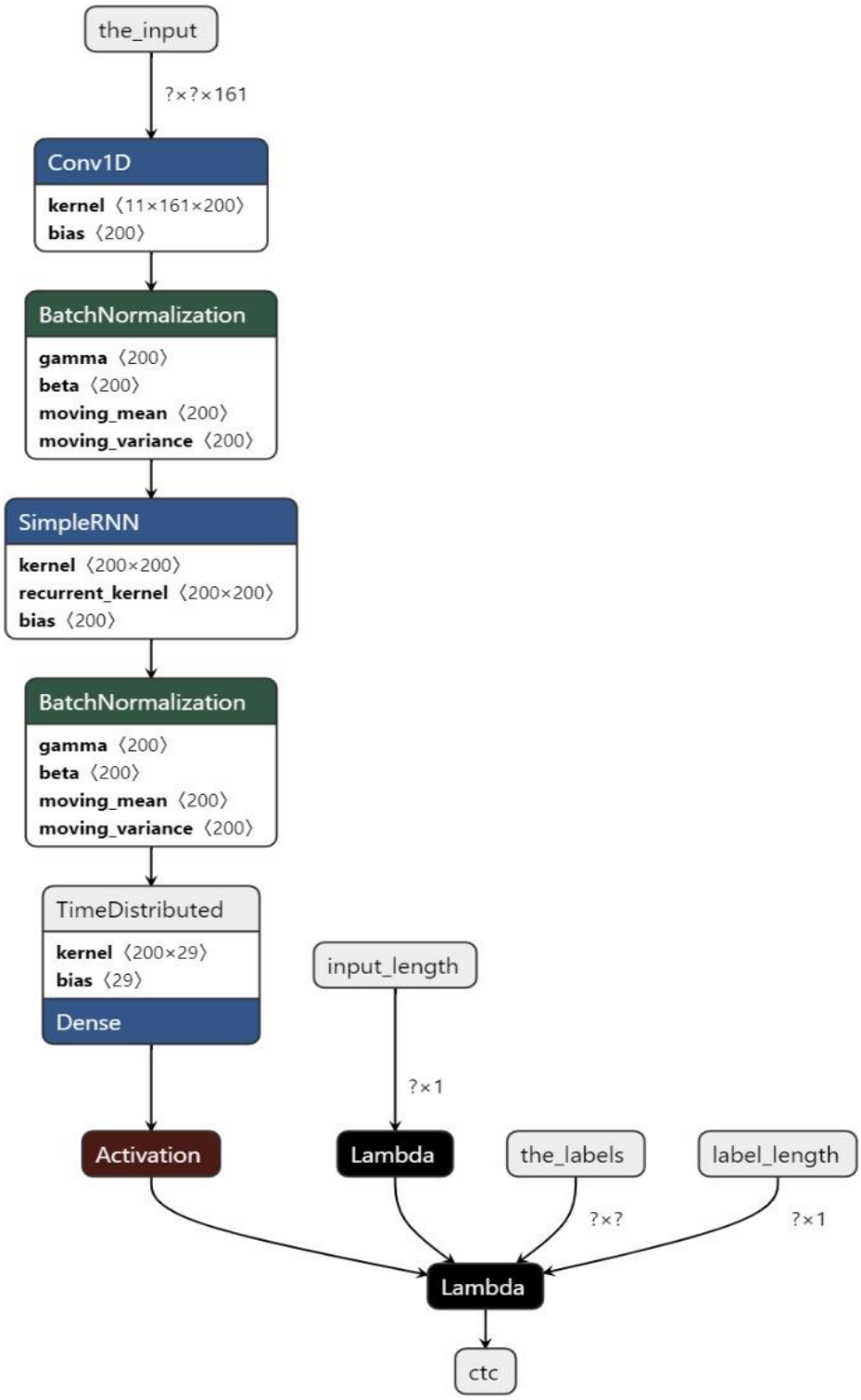
5.5.4 Bidirectional Model




```
def bidirectional_rnn_model(input_dim, units, output_dim=29):
    """ Build a bidirectional recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    #  Add bidirectional recurrent layer
```

```
    bi_rnn = Bidirectional(GRU(units, activation="relu", return_sequences=True),
name="bi_rnn")(input_data)
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim), name='time_dense')(bi_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model
```

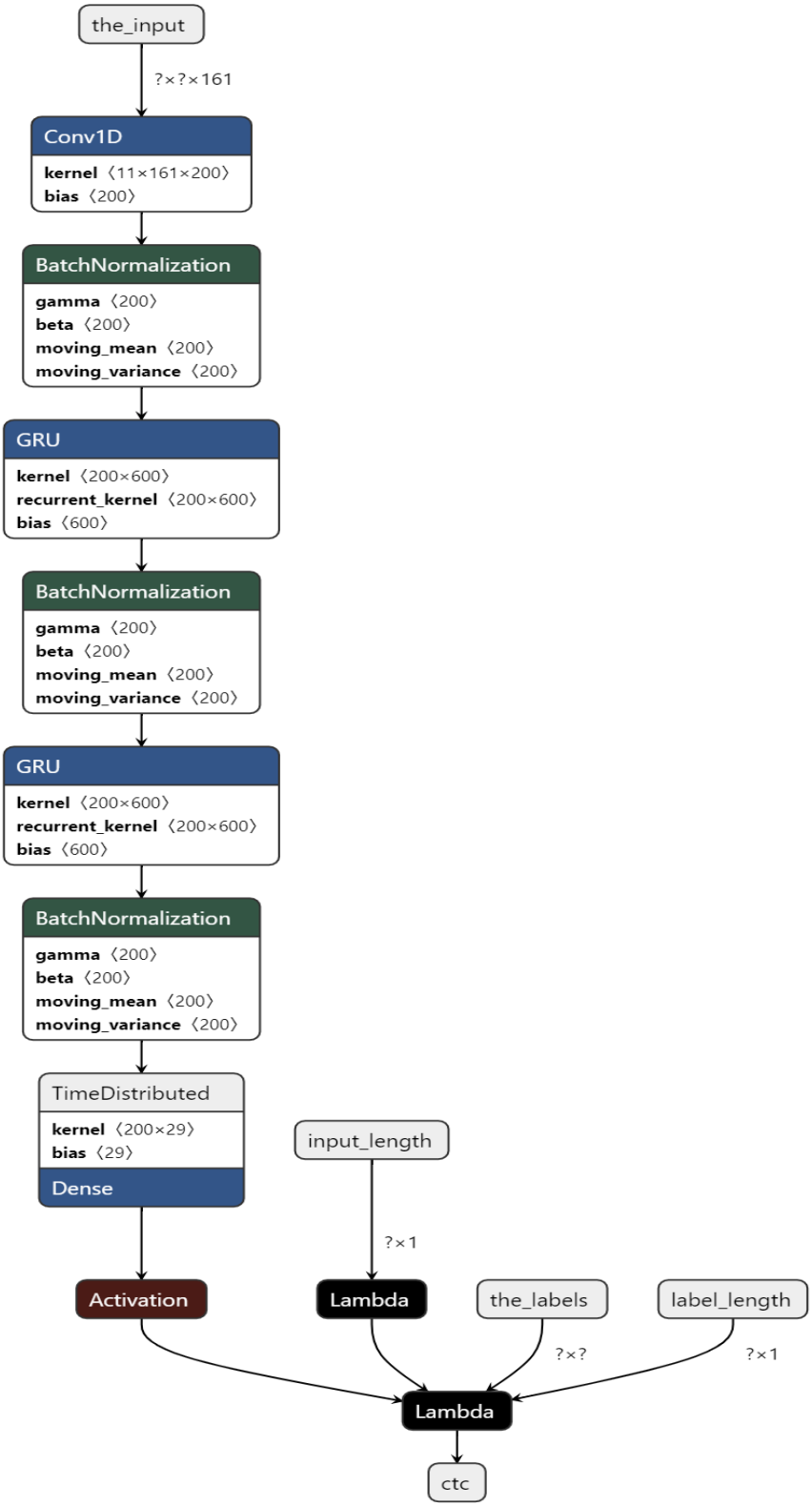
5.5.5 CNN RNN Model



```
def cnn_rnn_model(input_dim, filters, kernel_size, conv_stride,
    conv_border_mode, units, output_dim=29):
    """ Build a recurrent + convolutional network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add convolutional layer
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d')(input_data)
    # Add batch normalization
    bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)
    # Add a recurrent layer
    simp_rnn = SimpleRNN(units, activation='relu',
        return_sequences=True, name='rnn')(bn_cnn)
    # Add batch normalization
    bn_rnn = BatchNormalization(name='bn_rnn')(simp_rnn)
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim), name='time_dense')(bn_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)
    print(model.summary())
    return model

def cnn_output_length(input_length, filter_size, border_mode, stride,
                      dilation=1):
    """ Compute the length of the output sequence after 1D convolution along
        time. Note that this function is in line with the function used in
        Convolution1D class from Keras.
    Params:
        input_length (int): Length of the input sequence.
        filter_size (int): Width of the convolution kernel.
        border_mode (str): Only support `same` or `valid`.
        stride (int): Stride size used in 1D convolution.
        dilation (int)
    """
    if input_length is None:
```

```
        return None
    assert border_mode in {'same', 'valid'}
    dilated_filter_size = filter_size + (filter_size - 1) * (dilation - 1)
    if border_mode == 'same':
        output_length = input_length
    elif border_mode == 'valid':
        output_length = input_length - dilated_filter_size + 1
    return (output_length + stride - 1) // stride
```

5.5.6 Final Model – Combination of all Previous Models



```
def final_model(input_dim, filters, kernel_size, conv_stride,
    conv_border_mode, units, recur_layers, output_dim=29):

    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Specify the layers in your network
    # Add convolutional layer
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d')(input_data)
    # Add batch normalization
    bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)

    # Deep GRUs
    dummy_layer = bn_cnn
    for n in range(recur_layers):
        # Add RNN layer with batch normalization
        rnn = GRU(units, activation='relu', return_sequences=True, dropout = 0.3,
                  name='rnn_{}'.format(n))(dummy_layer)
        bn_rnn = BatchNormalization(name='bn_rnn_{}'.format(n))(rnn)
        dummy_layer = bn_rnn

    last_layer = dummy_layer
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim), name='time_dense')(last_layer)

    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)

    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)

    # Specify model.output_length (MAKE SURE IT IS CORRECT FOR CONV1D!)
    model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)

    print(model.summary())
    return model
```

5.6 Training Process

After implement the models came turn to training processing phase.

CTC, Connectionist temporal classification. it is used to solve the classification problem of time series data. Technology applied in speech recognition end-to-end solutions. Mainly to solve the following two problems

1. Solve the alignment problem of voice input and label. For a piece of speech input, it is converted into an acoustic spectrogram. The traditional acoustic model requires the pronunciation factors corresponding to each frame on the spectrogram. Using CTC as the loss function, only one input sequence and output sequence are needed.
2. CTC is a loss function used to measure the difference between the input sequence and the true output after passing through the neural network. For the pronunciation of (nihao), different people have different ways of pronunciation, it may be (nnnnniiiihhhaaaooo)... etc. CTC can measure the loss value of the input of different lengths after passing through the neural network and the actual result.

5.6.1 ctc-lambda-function

CTC is a loss of function, used to measure the input sequence after the neural network, and how much difference there is real output.

parameters:

y_pred: containing the prediction, or output of the softmax. (the model's calculated output).

Lable (y_true): containing the truth labels.

label_length: containing the sequence length for each batch item in `y_true`.

- example: sentance "Hello world" >> label_length is 11 because of the 11 characters of "Hello world"

input_length; containing the sequence length for each batch item in `y_pred`.

function return: CTC loss algorithm on each batch element

5.6.2 add_ctc_loss

Calculating CTC LOSS

Model groups layers into an object with training and inference features.

model = Model()

Once the model is created, you can config the model with losses and metrics with model.compile(), train the model with model.fit(), or use the model to do prediction with model.predict(). Will use it in the training function next.

5.6.3 train_model

Arguments:

input_to_softmax (Model): The acoustic model

pickle_path the model path .pickle

save_model_path the model path .h5

train_json,valid_json: Path to a JSON-line file that contains labels and paths to the audio files.

Stochastic gradient descent(SGD) optimizer:

- lr: float >= 0. Learning rate.
- decay: float >= 0. Learning rate decay over each update.
- momentum: float >= 0. Parameter updates momentum.
- nesterov: boolean. Whether to apply Nesterov momentum.
- clipnorm used with all optimizers to control gradient clipping

epochs: int, number of epochs

verbose=1 displays one line per epoch

steps:

audio_gen create from class AudioGenerator on data_generator.py

Add the training data and validation data to the generator

Calculate train and validation steps_per_epoch

Use model.compile to Configures the model for training.

ModelCheckpoint callback is used in conjunction with training using *model.fit()* to save a model or weights (in a checkpoint file) at some interval, so the model or weights can be loaded later to continue the training from the state saved.

Use all previous to Fits the model on data yielded batch-by-batch by a Python generator.

```
"""
Defines a functions for training a NN.
"""
# load the packages

from data_generator import AudioGenerator
import _pickle as pickle
```

```
from keras import backend as K
from keras.models import Model
from keras.layers import (Input, Lambda)
from tensorflow.keras import optimizers
from tensorflow.keras.optimizers import SGD
#from keras.optimizers import SGD
from keras.callbacks import ModelCheckpoint
import os



def ctc_lambda_func(args):
    '''
    CTC is a loss of function, used to measure the input sequence after the
    neural network, and how much difference there is real output.

    define ctc_lambda_func parameters:
    y_pred is the model's calculated output
    label_length is the seq length of each item in labels
    input_length= seq length of each item in y_pred
    '''
    y_pred, labels, input_length, label_length = args
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)


def add_ctc_loss(input_to_softmax):
    #INPUT definition used in calculating CTC LOSS

    # dtype 32-bit floating-point number type
    the_labels = Input(name='the_labels', shape=(None,), dtype='float32')
    # dtype 64-bit signed integer
    input_lengths = Input(name='input_length', shape=(1,), dtype='int64')
    label_lengths = Input(name='label_length', shape=(1,), dtype='int64')
    output_lengths = Lambda(input_to_softmax.output_length)(input_lengths)

    # CTC loss is implemented in a lambda layer
    loss_out = Lambda(ctc_lambda_func, output_shape=(1,), name='ctc')(
        [input_to_softmax.output, the_labels, output_lengths, label_lengths])

    # The final training model has 4 inputs, and the output is the ctc loss value.
    # Model groups layers into an object with training and inference features.
```

```
    model = Model(
        inputs=[input_to_softmax.input, the_labels, input_lengths, label_lengths],
        outputs=loss_out)
    return model


def train_model(input_to_softmax,
                pickle_path,
                save_model_path,
                train_json='train_corpus.json',
                valid_json='valid_corpus.json',
                minibatch_size=20,
                spectrogram=True, # change to False if you would like to use MFCC features
                mfcc_dim=13,
                optimizer=SGD(lr=0.02,
                              decay=1e-6,
                              momentum=0.9,
                              nesterov=True,
                              clipnorm=5),
                epochs=20,
                verbose=1,
                sort_by_duration=False,
                max_duration=10.0):

    '''
    Arguments:
        input_to_softmax (Model): The acoustic model
        pickle_path the model path .pickle
        save_model_path the model path h5
        train_json,valid_json: Path to a JSON-line file that contains labels
            and paths to the audio files.
        Stochastic gradient descent(SGD) optimizer:
            lr: float >= 0. Learning rate.
            decay: float >= 0. Learning rate decay over each update.
            momentum: float >= 0. Parameter updates momentum.
            nesterov: boolean. Whether to apply Nesterov momentum.
            clipnorm used with all optimizers to control gradient clipping
        epochs: int, number of epochs
        verbose=1 displays one line per epoch

    '''
    # create a class instance for obtaining batches of data
    audio_gen = AudioGenerator(minibatch_size=minibatch_size,
```

```
    spectrogram=spectrogram, mfcc_dim=mfcc_dim, max_duration=max_duration,
    sort_by_duration=sort_by_duration)

# add the training data to the generator
audio_gen.load_train_data(train_json)
audio_gen.load_validation_data(valid_json)

# calculate train steps_per_epoch
num_train_examples=len(audio_gen.train_audio_paths)
steps_per_epoch = num_train_examples//minibatch_size

# calculate validation_steps
num_valid_samples = len(audio_gen.valid_audio_paths)
validation_steps = num_valid_samples//minibatch_size

# add CTC loss to the NN specified in input_to_softmax
model = add_ctc_loss(input_to_softmax)

# CTC loss is implemented elsewhere, so use a dummy lambda function
# for the loss
# Configures the model for training.
model.compile(loss={'ctc': lambda y_true, y_pred: y_pred},
              optimizer=optimizer)

# make results/ directory, if necessary
if not os.path.exists('results'):
    os.makedirs('results')

# add checkpointer
checkpointer=ModelCheckpoint(filepath='results/'+save_model_path,verbose=0)

# use the model.fit to train the model
# Fits the model on data yielded batch-by-batch by a Python generator.
hist = model.fit_generator(generator=audio_gen.next_train(),
                           steps_per_epoch=steps_per_epoch,
                           epochs=epochs,
                           validation_data=audio_gen.next_valid(),
                           validation_steps=validation_steps,
                           callbacks=[checkpointer],
                           verbose=verbose)

# save model loss
```

```
    with open('results/'+pickle_path, 'wb') as f:
        pickle.dump(hist.history, f)
```

5.7 Model Prediction

This Functions Used to Make Prediction on New Sample to Use in Inference Stage.

```
from keras.backend.tensorflow_backend import set_session
import tensorflow as tf
import numpy as np
from data_generator import AudioGenerator
from keras import backend as K
from utils import int_sequence_to_text
# import NN architectures for speech recognition
from sample_models import *
# import function for training acoustic model
from train_utils import train_model
import pyaudio
import simpleaudio as sa
import numpy as np
from scipy.io.wavfile import read

def get_predictions(audio_path,  input_to_softmax, model_path):
    """ Print a model's decoded predictions
    Params:
        index (int): The example you would like to visualize
        partition (str): One of 'train' or 'validation'
        input_to_softmax (Model): The acoustic model
        model_path (str): Path to saved acoustic model's weights
    """
    # obtain the true transcription and the audio features

    # init Object From Audio Generator Class
    data_gen = AudioGenerator()
    # Load Data to Calculate Mean and Variance for Batch Normalization
    data_gen.load_train_data()
    # Normalize Input As Training Data
    data_point = data_gen.normalize(data_gen.featurize(audio_path))
    # obtain and decode the acoustic model's predictions
    input_to_softmax.load_weights(model_path)
    # Make Prediction
    prediction = input_to_softmax.predict(np.expand_dims(data_point, axis=0))
    output_length = [input_to_softmax.output_length(data_point.shape[0])]
    # Pass Result to CTC Decoder
    pred_ints = (K.eval(K.ctc_decode(
            prediction, output_length)[0][0])+1).flatten().tolist()
     return transcribted
```

5.8 Audio Recording

PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, you can easily use Python to play and record audio on a variety of platforms.

```
import pyaudio
import wave
import os

class RecAUD:


"""
CHUNK :  is the number of frames in the buffer.
RATE : is the number of samples collected per second.
FORMAT : is the format type
CHANNELS : Each frame will have 1 samples as “channels=1”
PY : instantiate PyAudio
FRAMES :  empty list for frams
ST  : start Flag
"""


    def __init__(self, chunk=1024, frmat=pyaudio.paInt16, channels=1, rate=16000,
py=pyaudio.PyAudio()):

        self.collections = []
        self.CHUNK = chunk
        self.FORMAT = frmat
        self.CHANNELS = channels
        self.RATE = rate
        self.p = py
        self.frames = []
        self.st = 1
     # define Stream object to write the WAV file to
        self.stream = self.p.open(format=self.FORMAT, channels=self.CHANNELS,
rate=self.RATE, input=True, frames_per_buffer=self.CHUNK)


    def start_record(self):
        #print("start_record")
        self.st = 1
        self.frames = []                                    #Clear Frams list
     # Open .Stream object to write the WAV file to
        stream = self.p.open(format=self.FORMAT, channels=self.CHANNELS,
```

```
rate=self.RATE, input=True, frames_per_buffer=self.CHUNK)
      #check state of the flag
       while self.st == 1:

           # Store data in chunks
           data = stream.read(self.CHUNK)
           self.frames.append(data)
           #print("* recording")

    # Stop and close the stream
       stream.close()

       # Save the recorded data as a WAV file
       wf = wave.open('test_recording.wav', 'wb')
       wf.setnchannels(self.CHANNELS)
       wf.setsampwidth(self.p.get_sample_size(self.FORMAT))
       wf.setframerate(self.RATE)
       wf.writeframes(b''.join(self.frames))
       wf.close()


 def stop(self):
       #print("STOP IN CLASS")
       self.st = 0                          #Set Flag to 0 to stop recordingstop recording
```

5.9 APIs

5.9.1 Watson Translation API

IBM Watson™ Language Translator translates text from one language to another. The service offers multiple IBM-provided translation models that you can customize based on your unique terminology and language. Use Language Translator to take news from across the globe and present it in your language, communicate with your customers in their own language, and more.

5.9.1.1 Authentication

You authenticate to the API by using IBM Cloud Identity and Access Management (IAM).

You can pass either a bearer token in an authorization header or an API key. Tokens support authenticated requests without embedding service credentials in every call. API keys use basic authentication.

For IBM Cloud instances, the SDK provides initialization methods for each form of authentication.

- Use the API key to have the SDK manage the lifecycle of the access token. The SDK requests an access token, ensures that the access token is valid, and refreshes it if necessary.
- Use the access token to manage the lifecycle yourself. You must periodically refresh the token.

5.9.1.2 Access between services

Your application might use more than one Watson service. You can grant access between services and you can grant access to more than one service for your applications.

For IBM Cloud services, the method to grant access between Watson services varies depending on the type of API key

- To grant access between IBM Cloud services, create an authorization between the services.
- To grant access to your services by applications without using user credentials, create a service ID, add an API key, and assign access policies.

When you give a user ID access to multiple services, use an endpoint URL that includes the service instance ID (for example, https://api.us-south.language-translator.watson.cloud.ibm.com/instances/6bbda3b3-d572-45e1-8c54-22d6ed9e52c2). You can find the instance ID in two places:

- By clicking the service instance row in the Resource list. The instance ID is the **GUID** in the details pane.
- By clicking the name of the service instance in the list and looking at the credentials URL.

```
from ibm_watson import LanguageTranslatorV3
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator
import json


#inserting api key
authenticator = IAMAuthenticator('JnYR2fdHtDpIFG2Lr8Bfcmfmm0w0kVuQgS-aV8iZNhyq')
language_translator = LanguageTranslatorV3(
    version='2018-05-01',
    authenticator=authenticator
)
```

```
# Setting the service url
language_translator.set_service_url('https://api.eu-gb.language-translator.watson.cloud.ibm.com/instances/71064c3a-5b61-4a85-b13d-3b090255999e')

"""
Creating functions That take En_text and The language to be translated into
Returns translated text
"""
def get_watson_translation(text,lang):
    translation = language_translator.translate(
        text=text,
        model_id=f'en-{lang}').get_result()

    return (translation["translations"][0]["translation"])
```

5.9.2 Watson Speech 2 Text

We use this API to measure our Performance.

The IBM Watson™ Speech to Text service provides APIs that use IBM's speech-recognition capabilities to produce transcripts of spoken audio. The service can transcribe speech from various languages and audio formats. In addition to basic transcription, the service can produce detailed information about many different aspects of the audio. It returns all JSON response content in the UTF-8 character set.

The service supports two types of models: previous-generation models that include the terms Broadband and Narrowband in their names, and beta next-generation models that include the terms Multimedia and Telephony in their names. Broadband and multimedia models have minimum sampling rates of 16 kHz. Narrowband and telephony models have minimum sampling rates of 8 kHz. The beta next-generation models currently support fewer languages and features, but they offer high throughput and greater transcription accuracy.

For speech recognition, the service supports synchronous and asynchronous HTTP Representational State Transfer (REST) interfaces. It also supports a WebSocket interface that provides a full-duplex, low-latency communication channel: Clients send requests and audio to the service and receive results over a single connection asynchronously.

The service also offers two customization interfaces. Use language model customization to expand the vocabulary of a base model with domain-specific terminology. Use acoustic model customization to adapt a base model for the acoustic characteristics of your audio. For language model customization, the service also supports grammars. A grammar is a formal language specification that lets you restrict the phrases that the service can recognize.

```
import json
from os.path import join, dirname
```

```
from ibm_watson import SpeechToTextV1
from ibm_watson.websocket import RecognizeCallback, AudioSource
import threading
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator

#inserting api key
authenticator =
IAMAuthenticator('vVWMZcTRxRDvaaXyGGWH0pIHHcHqFa8X_NHona-UvxDM')
service = SpeechToTextV1(authenticator=authenticator)
#Setting the service url
service.set_service_url('https://api.eu-gb.speech-to-
text.watson.cloud.ibm.com/instances/8f450cc1-8340-40a1-a018-49a07ed81e9b')

models = service.list_models().get_result()
#print(json.dumps(models, indent=2))

model = service.get_model('en-US_BroadbandModel').get_result()
#print(json.dumps(model, indent=2))


def get_prediction_ibm(path):
    with open(join(dirname(__file__), path),
            'rb') as audio_file:
        results = service.recognize(
                audio=audio_file,
                content_type='audio/wav',
                timestamps=True,
                word_confidence=True).get_result()


    return results['results'][0]['alternatives'][0]['transcript']
```

5.9.3 Synthesizer

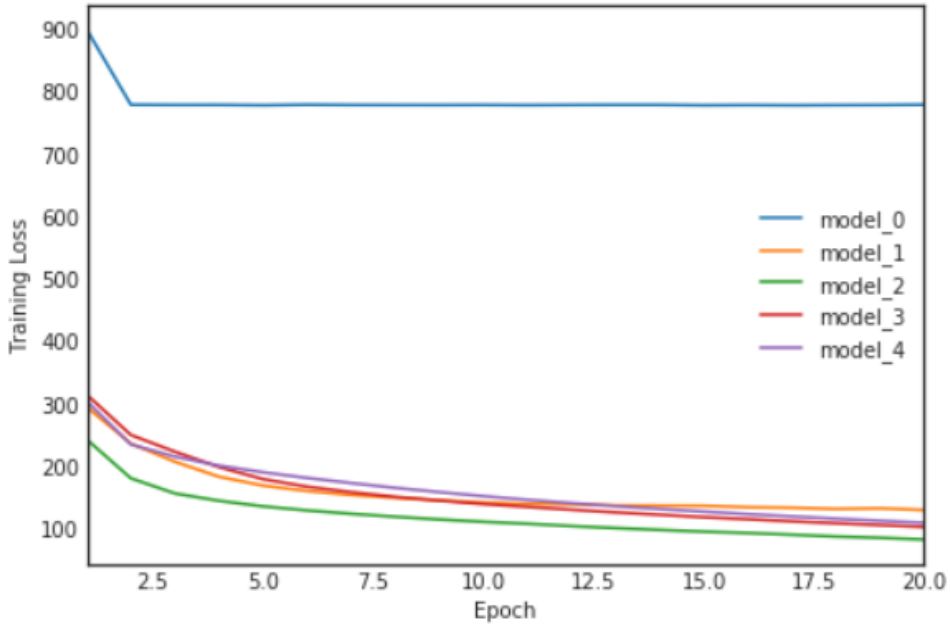
Used To Convert Text to Audio.

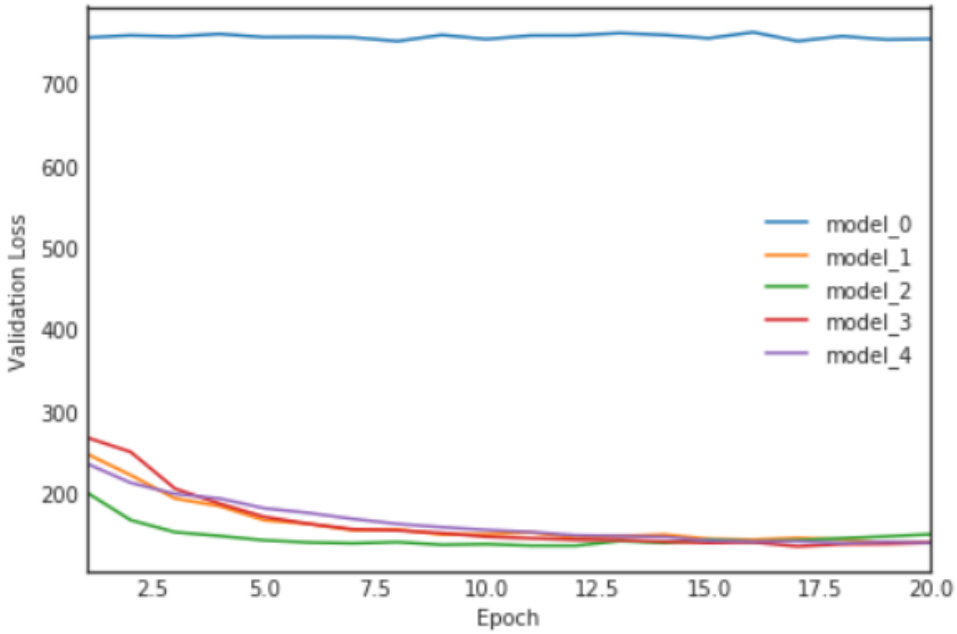
```
from audioplayer import AudioPlayer
from gtts import gTTS


# Playback stops when the object is destroyed (GC'ed), so save a reference to the
object for non-blocking playback.
```

```
def synthesize(text,language):
    try:
        tts = gTTS(text=text ,lang=language)
        tts.save("synthesized.mp3")
        AudioPlayer("./synthesized.mp3").play(block=True)
    except Exception as E:
        return "No Speech!"
```

5.10 Models Performance

Analyze Results Thats Came from Models.



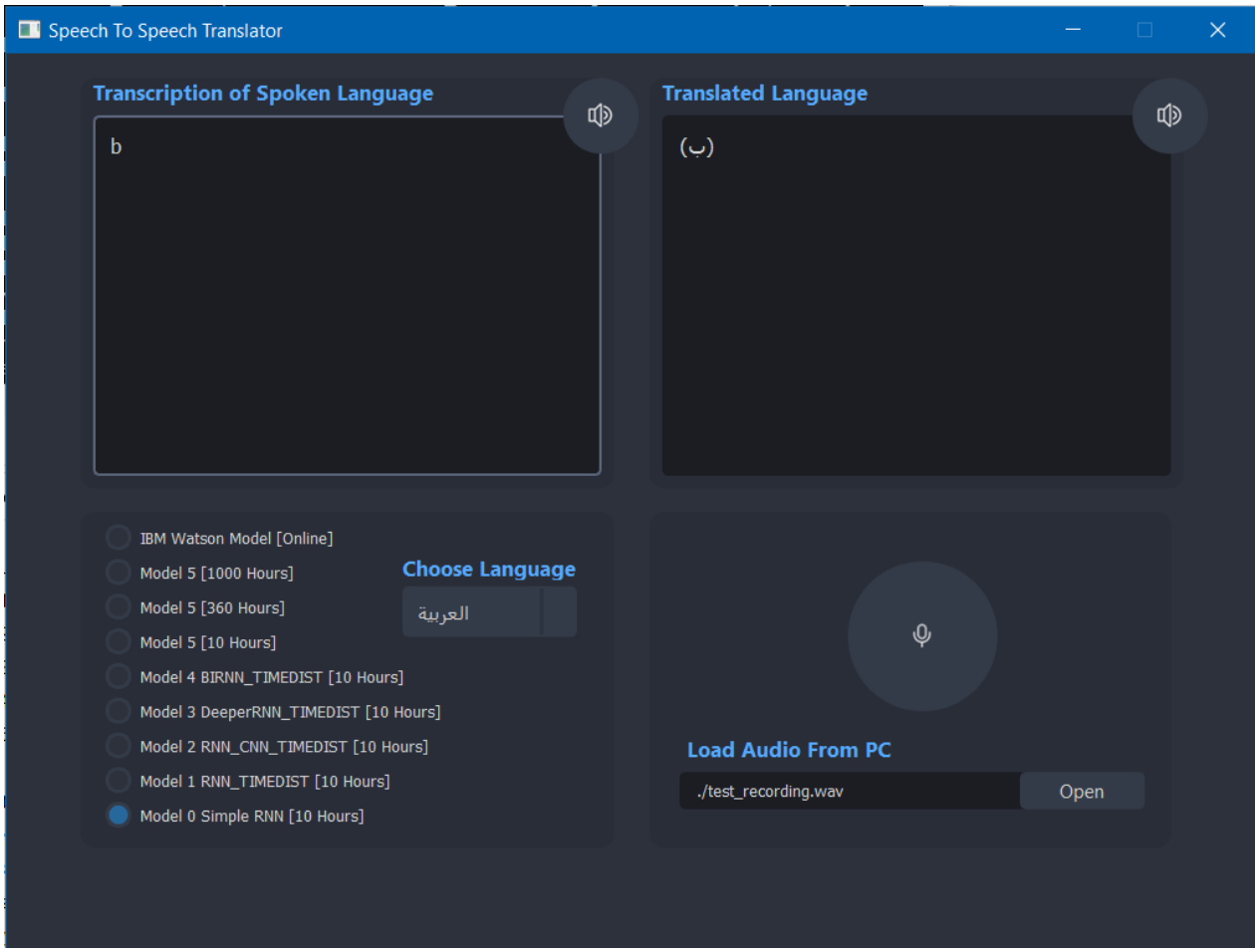


model_2 performs best as it has the lowest Training loss and moderate validation loss. In general, model with deep RNN structure(model_3) and Bi-directional RNNs(model_4) perform better than the single RNN model(model_1). Model_0(simple RNN) performs the worst. Model 2(CNN+RNN) really boosts the model performance because of the convolutional layer. Among all models, overfitting exists because the validation losses are much larger than training losses.

5.11 Results

After Testing All Models, We Obtained These Results.

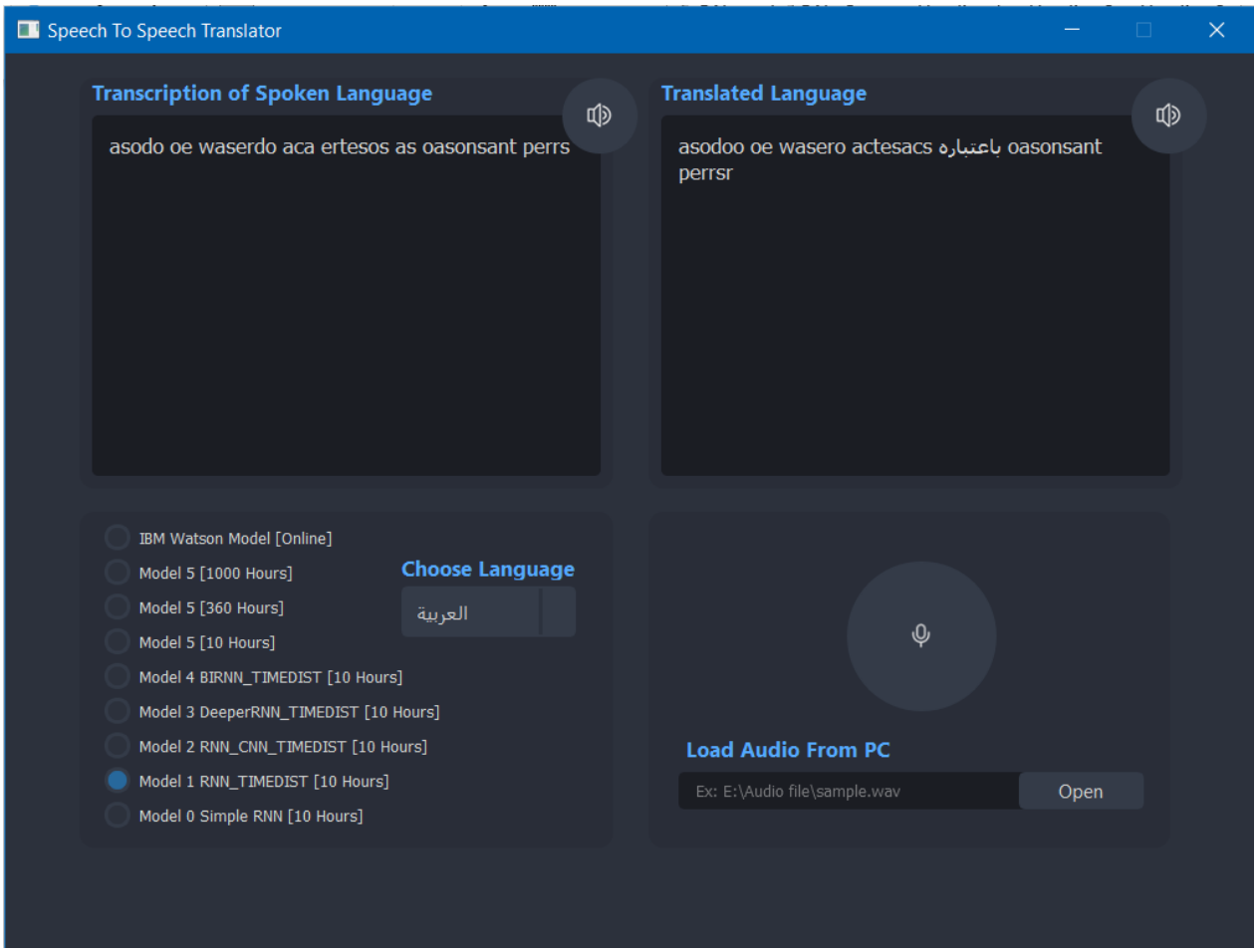
5.11.1 Result of First Model Simple RNN



CH5 1 Model 1 Result

This Model has least performance because it was trained on very small dataset and it has small Architecture not suitable to capture continuous Speech.

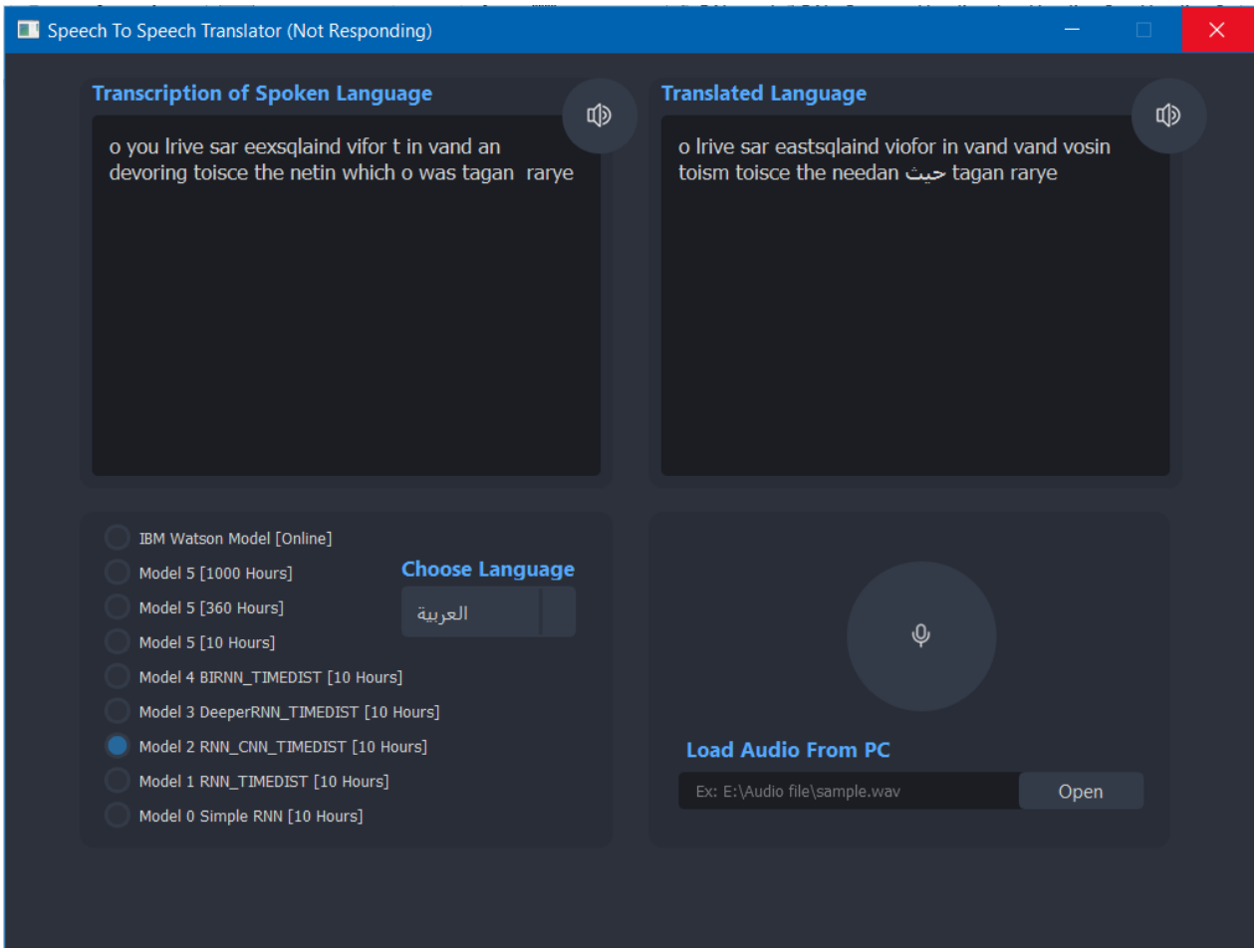
5.11.2 Result of Second Model RNN Time Distributed



CH5 2 Model 2 Result

This Model acts Better Than Previous One, because we Added Time Distributed Layer Which Capture Some of Continuous Speech, also it is Trained on Same Data.

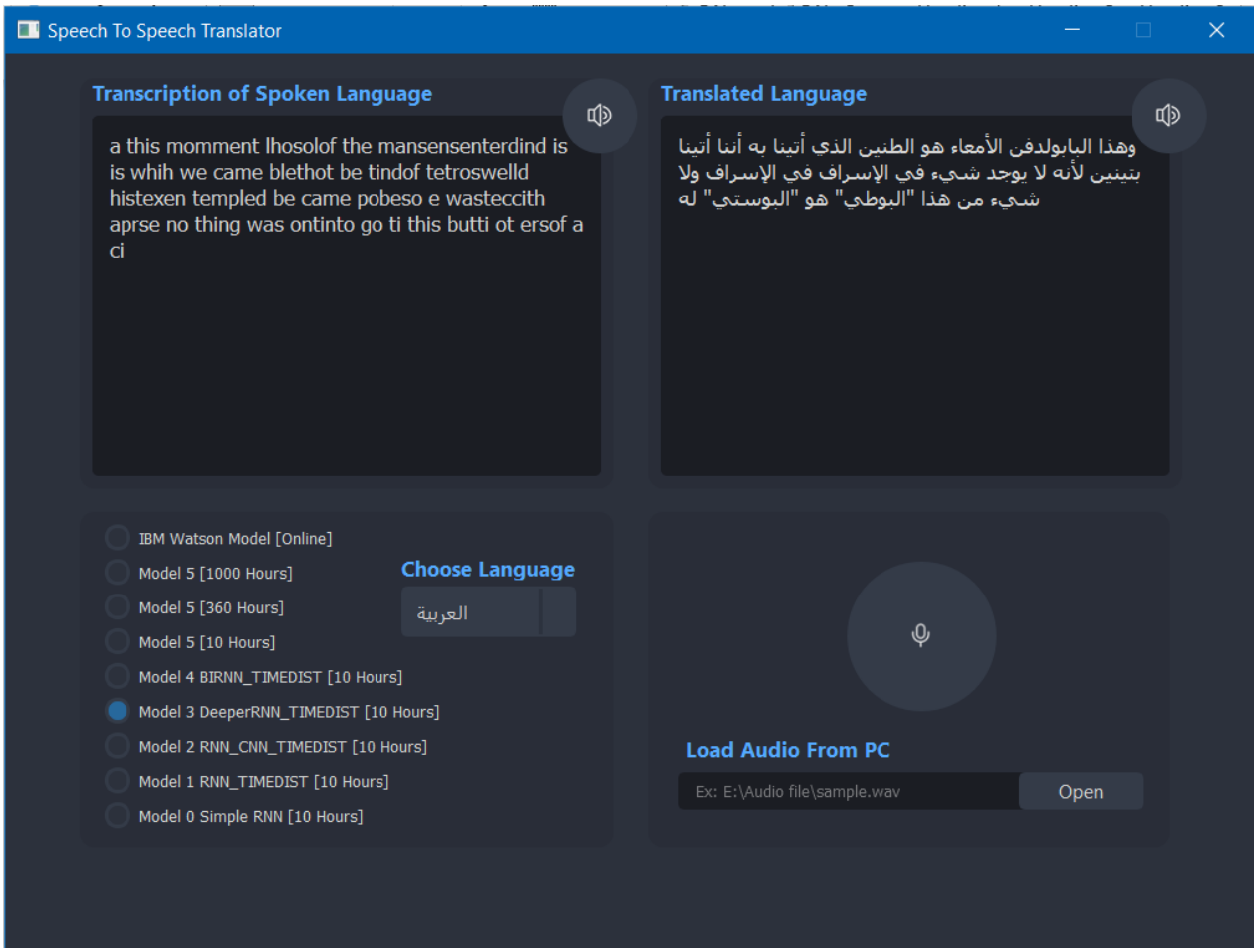
5.11.3 Result of Third Model RNN_CNN_TimeDist



CH5 3 Model 3 Result

This Model acts Better Than Previous One, because we Added Time Distributed with CNN Layer Which Capture More Continuous Speech, also it is Trained on Same Data.

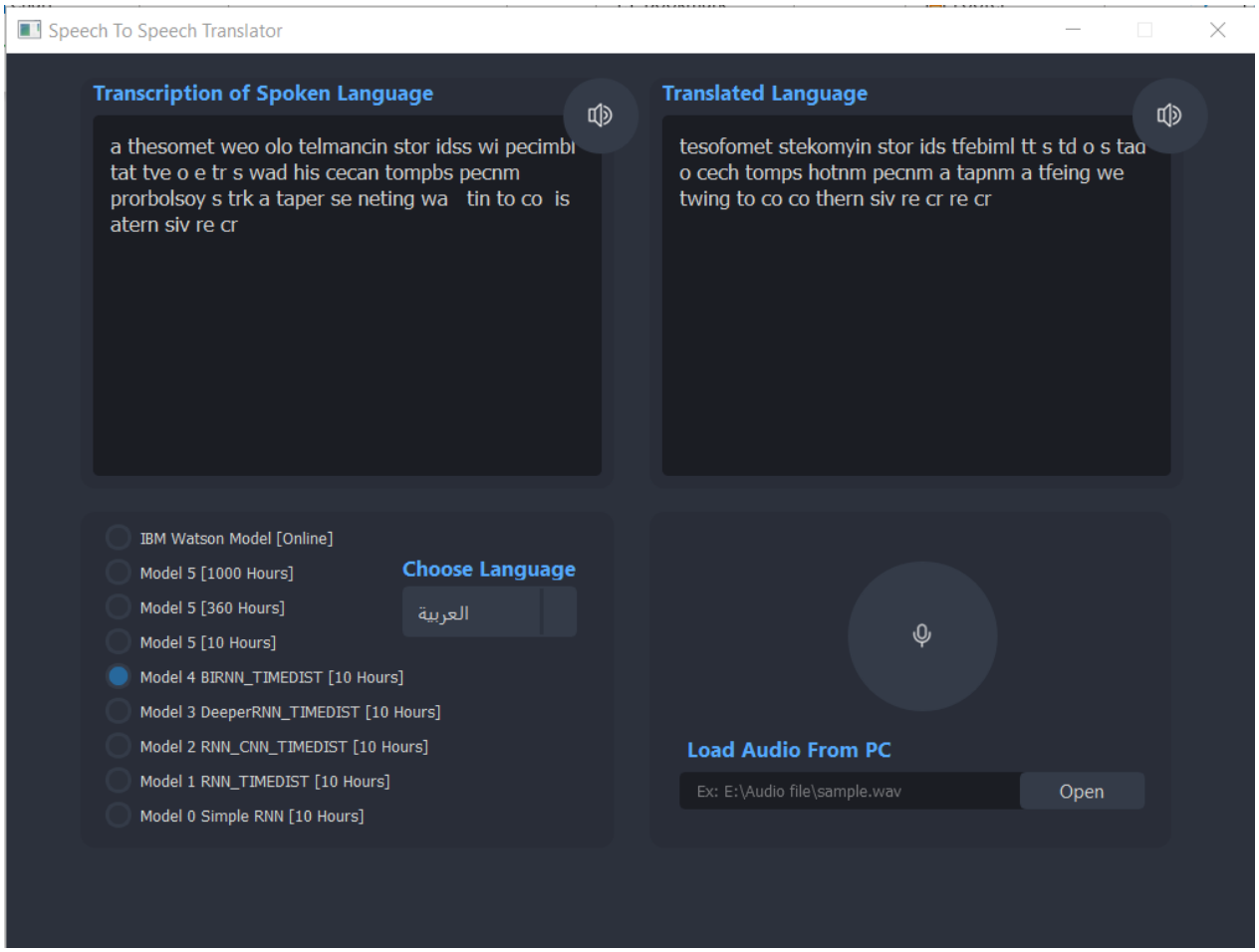
5.11.4 Result of 4th Model Deeper RNN



CH5 4 Model 4 Result

We introduced Deep Arch So, we Saw Better Performance on This Model.

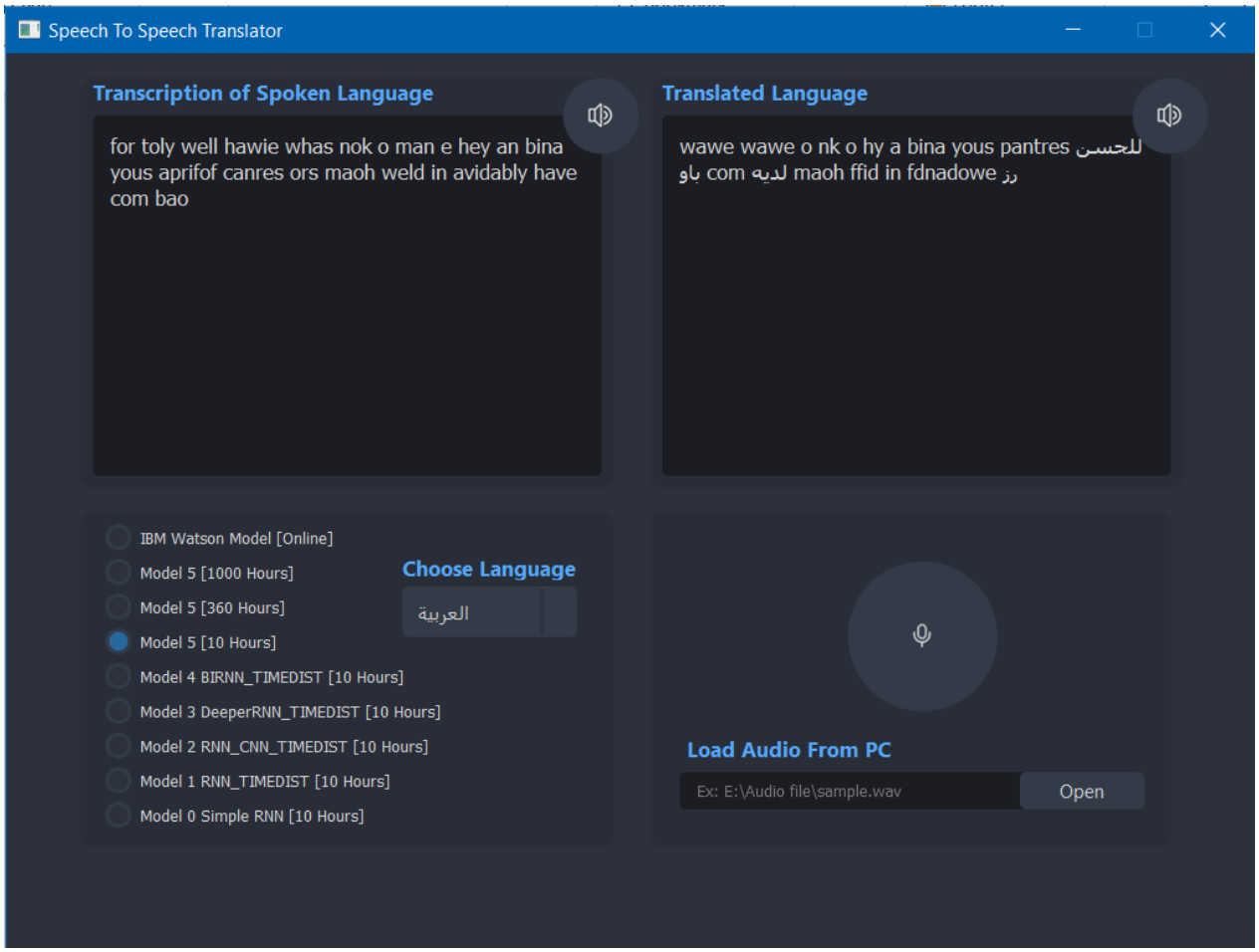
5.11.5 Result of 5th BIRNN with Time Distributed



CH5 5 Model 5 Result

This Model Acts Better Than RNN-TimeDist according to it we use BIRNN in Final Model.

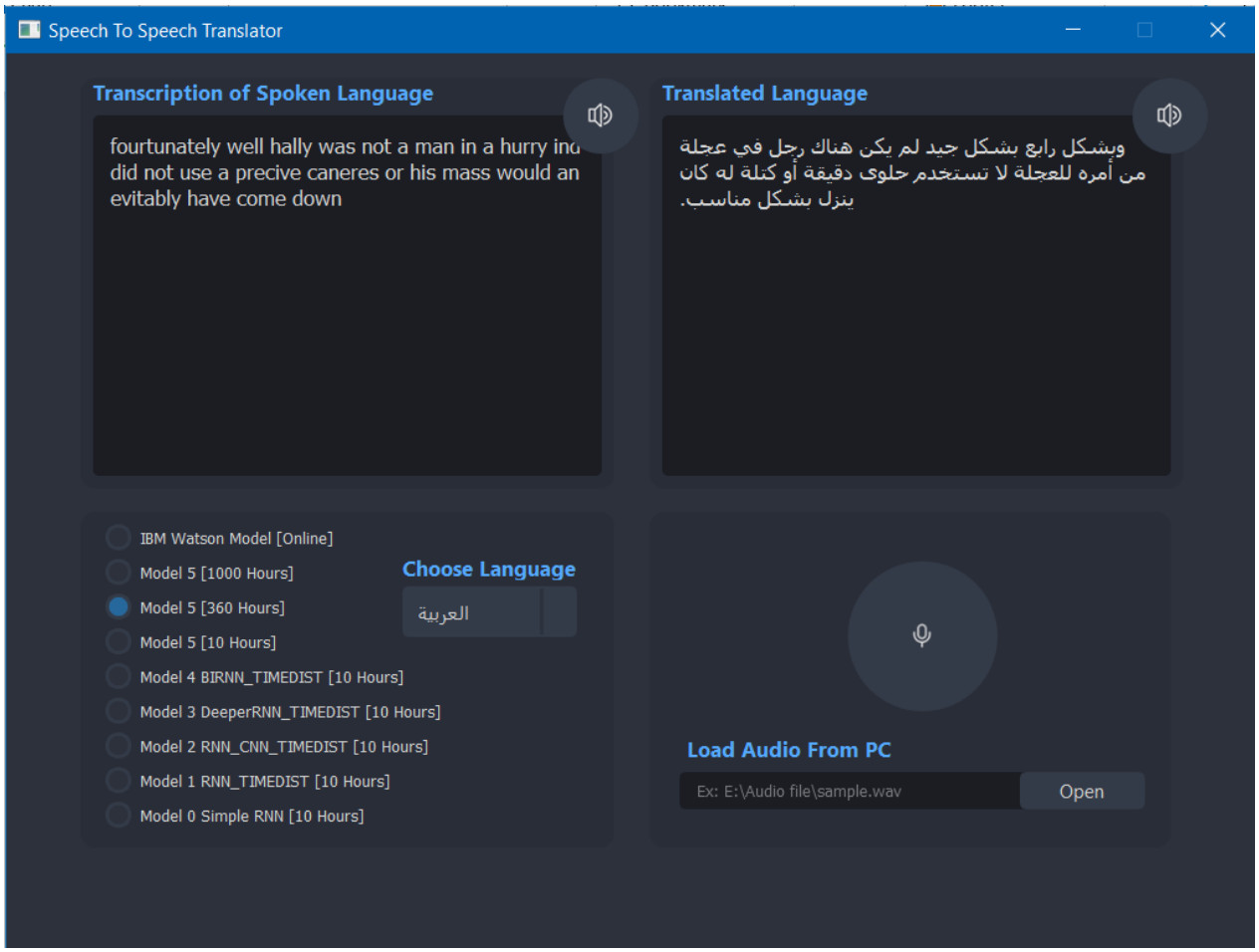
5.11.6 Result of Final Model With 10 Hours of Training



CH5 6 Final Model 10 Hours of Training

Now we can capture some correct words from Continuous Speech.

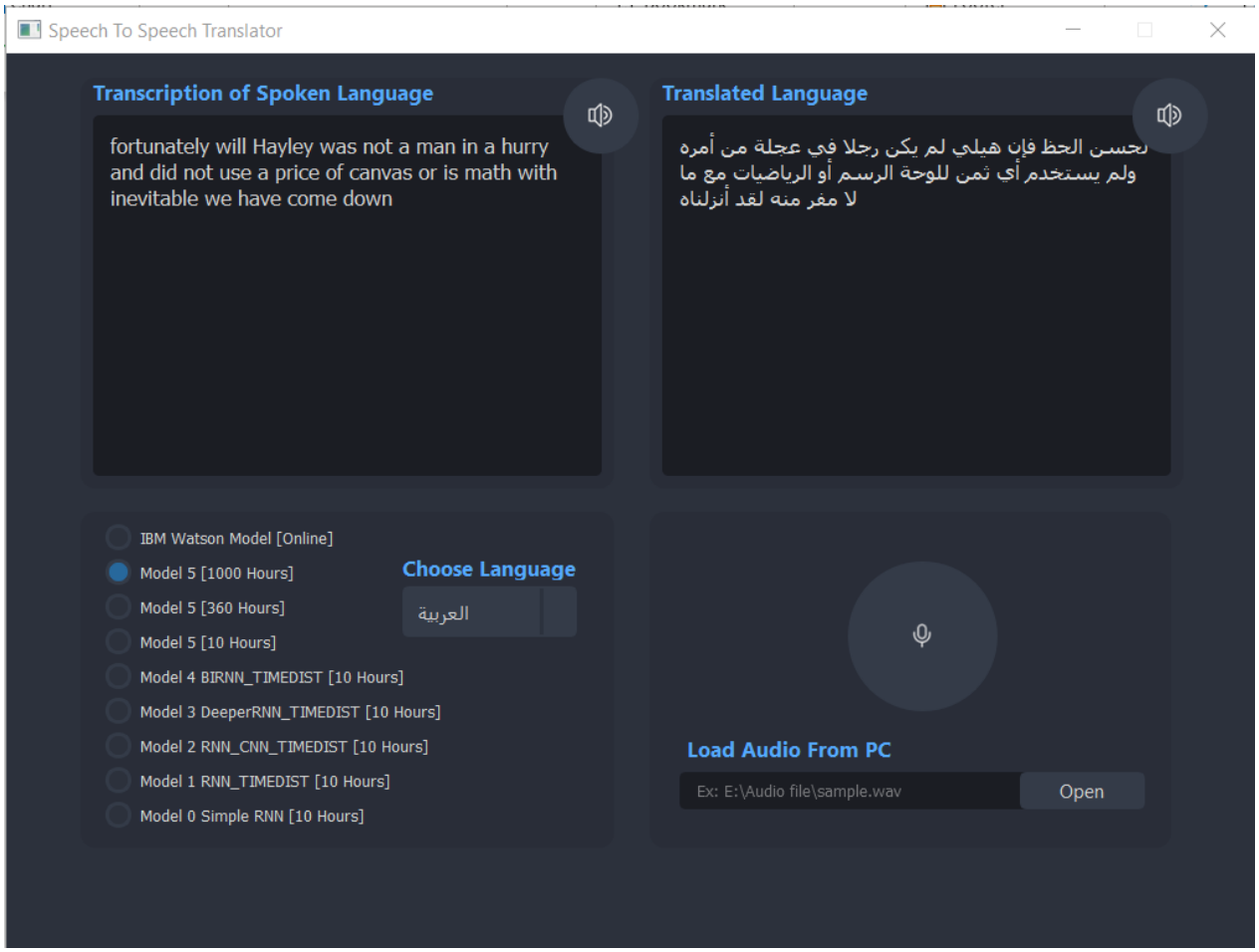
5.11.7 Result of Final Model With 360 Hours of Training



CH5 7 Final Model 360 Hours of Training

After we Trained Previous Model with Bigger Dataset, we have got better Accuracy.

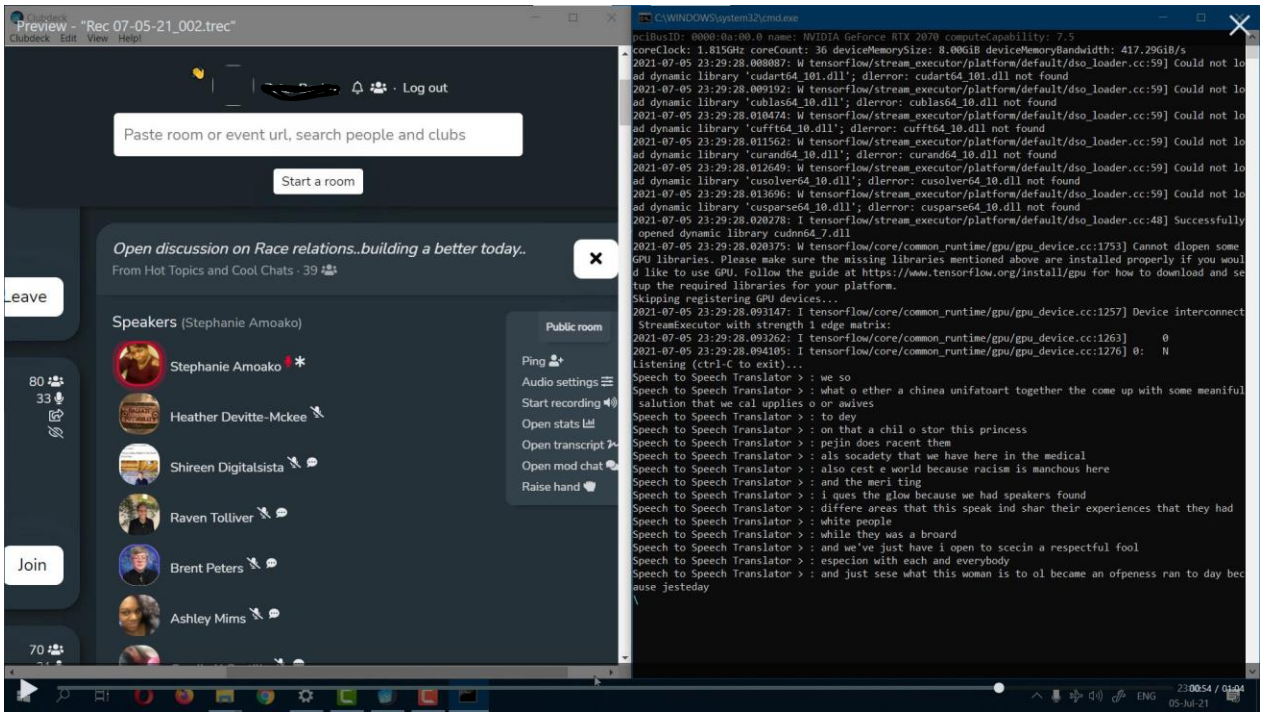
5.11.8 Result of Final Model With 1000 Hours of Training



CH5 8 CH5 7 Final Model 1000 Hours of Training

Finally, We Get Satisfied Result and Accuracy When Using Model 5 And 1000 Training Hours of Speech.

5.12 Live Transcription Sentence by Sentence



CH5 9 Live Transcription

- At the left is Audio Streaming from Clubhouse.
- At the Right we saw a live Transcription for Streamed Audio, it is working as Sentence by Sentence.

6. Conclusions and Future Work

6.1 Conclusion

Speech to Speech Translator Provide the ability to Conversion With all People around World with Different Languages and Remove Barriers of Conversion so will Open Closed Roads.

At this Research we have Introduced Designs of Different Architectures and analysis of all Components in Each Design also we've Talked about the Advantage and Disadvantage for Each Components.

6.2 Future Tasks

6.2.1 Future Tasks: Voice Characteristics

When We Transcribe Voice Then Translated then synthesize it we lost the Characteristics of Voice, in Future it should improve to Transfer This Characteristics like: Frequency, Pitch, Tempo …etc.

6.2.2 Future Tasks: High Precision All-Purpose Translation System

The current technology relies heavily on the corpus. The performance improves as the amount of corpus increases. Conversely, high performance cannot be expected with an insufficient amount of corpus. A race between organizations that pursue research and development has begun to see who can collect more data, but the inconsistent accuracies of many translation services—due to limitations in language or domain—clearly state that the collection method needs to be reexamined. Thus, a high precision all-purpose translation system has yet to be realized.

6.2.3 Future Tasks: Simultaneous Interpretation

he current speech translation system begins the translation process after one has finished speaking. The completion of the process will be largely delayed when the input is long. In order to avoid such delay, the system needs to start processing before the utterance reaches its end.

In such way, simultaneous interpretation is difficult even for humans, and to realize simultaneous interpretation using a computer, it requires a great deal of time and a huge breakthrough in technology.

6.2.4 Future Tasks: Context-Based Translation

The current speech-to-speech translation system and its elemental technologies only process translation sentence by sentence, without any consideration of the preceding sentences. Studies on context have also been carried out for many years, however, the applicability and processing speed are some of the issues that remain.
Translation is often performed by determining the surrounding words. If translating sentence by sentence would be practical enough, context may not have to be taken into account.

6.2.5 Future Tasks: Spoken Arabic Language

The current speech-to-speech translation system does not have great performance on spoken Arabic Language because we don’t have big dataset of spoken speech of Arabic language, in future we should collect big data set at least 1000 hours as we have seen.

References

1. Arora, K., Arora, S. & Roy, M.K. Speech to speech translation: a communication boon. *CSIT* **1,** 207–213 (2013). https://doi.org/10.1007/s40012-013-0014-4

2. Do, Q., Toda, T., Neubig, G., Sakti, S., & Nakamura, S. (2017). Preserving Word-Level Emphasis in Speech-to-Speech Translation. IEEE/ACM Transactions On Audio, Speech, And Language Processing, 25(3), 544-556. doi: 10.1109/taslp.2016.2643280

3. Brownlee, J. (2020). How to Implement a Beam Search Decoder for Natural Language Processing. Retrieved 23 December 2020, from https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/

4. Illustrated Guide to LSTM's and GRU's: A step by step explanation. (2020). Retrieved 23 December 2020, from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

5. Gated Recurrent Units explained using Matrices: Part 1. (2020). Retrieved 23 December 2020, from https://towardsdatascience.com/gate-recurrent-units-explained-using-matrices-part-1-3c781469fc18

6. Dive into Deep Learning — Dive into Deep Learning 0.15.1 documentation. (2020). Retrieved 23 December 2020, from https://d2l.ai/index.html

7. Goodfellow, I., Bengio, Y., & Courville, A. *Deep learning*.

8. M. (2020). Tutorial - Sequence Modelling | Attention Models. Retrieved 24 December 2020, from https://www.analyticsvidhya.com/blog/2019/01/sequence-models-deeplearning/

9. Word Level English to Marathi Neural Machine Translation using Seq2Seq Encoder-Decoder LSTM Model. (2020). Retrieved 24 December 2020, from https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7

10. Alammar, J. (2020). Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention). Retrieved 24 December 2020, from https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

11. Neural Machine Translation using word level seq2seq model. (2020). Retrieved 24 December 2020, from https://medium.com/@dev.elect.iitd/neural-machine-translation-using-word-level-seq2seq-model-47538cba8cd7

12. A ten-minute introduction to sequence-to-sequence learning in Keras. (2020). Retrieved 24 December 2020, from https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

13. Introduction to Word Embedding and Word2Vec. (2020). Retrieved 24 December 2020, from https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

14. NLP—Word Embedding & GloVe. (2020). Retrieved 24 December 2020, from https://jonathan-hui.medium.com/nlp-word-embedding-glove-5e7f523999f6

15. Common Loss functions in machine learning. (2020). Retrieved 26 December 2020, from https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23

16. What is Mean Squared Error, Mean Absolute Error, Root Mean Squared Error and R Squared? - Studytonight. (2020). Retrieved 26 December 2020, from https://www.studytonight.com/post/what-is-mean-squared-error-mean-absolute-error-root-mean-squared-error-and-r-squared#

17. Evaluating a machine learning model. (2020). Retrieved 26 December 2020, from https://www.jeremyjordan.me/evaluating-a-machine-learning-model/

18. Kulbear/deep-learning-coursera. (2021). Retrieved 22 January 2021, from https://github.com/Kulbear/deep-learning-coursera

Appendices

Appendix 1: Mathematics for Deep Learning

1. Geometry and Linear Algebraic Operations
2. Eigen decompositions
3. Single Variable Calculus
4. Multivariable Calculus
5. Random Variables
6. Maximum Likelihood
7. Distributions
8. Naive Bayes
9. Statistics

Contact Us

Peter Boshra Aziz
01032470065
Peter_boshraaa@yahoo.com

Salma Hesham Othman
01155284729
Salmahesham471@gmail.com

Mariam Lotfy Mohamed
01017784300
Mariam.laer@gmail.com

Abdelrahman Sanad Abdelrahman
01033812665
abdulrahman.snad@gmail.com

Mahmoud Ahmed Manaa
01090786551
mahmoudamanaa98@gmail.com

Abdullah Ahmed Abdulkarem
01112730321
aretg1@hotmail.com