Recurrent LSTM Network Design for Natural Language Processing

Project report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

Submitted by

Debanjali Saha (1810110059)

Under supervision

of

Dr. Madan Gopal
Department of Electrical Engineering

SHIV NADAR
UNIVERSITY
DELHI NCR

DEPARTMENT OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING
SHIV NADAR UNIVERSITY
(December 2021)

Candidate Declaration

I hereby declare that the thesis entitled "Recurrent LSTM Network Design for Natural Language Processing" submitted for the B. Tech. degree program has been written in my own words. I have adequately cited and referenced the original sources.

Debanjali

Debanjali Saha

(1810110059)

Date: 27/11/2021

CERTIFICATE

It is certified that the work contained in the project report titled “Recurrent LSTM Network Design for Natural Language Processing” by “Ms. Debanjali Saha” has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Signature)
Dr. Madan Gopal
Dept. of Electrical Engineering
School of Engineering
Shiv Nadar University
Date: dd/mm/yyyy

Abstract

The Recurrent Long-Short Term Memory or LSTM is an artificial Recurrent Neural Network architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can process not only single data points such as images, but also entire sequences of data such as speech or video. It has an advantage over simple Recurrent Neural Networks as it can perform as it can be difficult to train standard RNNs to solve problems that require learning long-term temporal dependencies. This is because the gradient of the loss function decays exponentially with time- the vanishing gradient problem.

LSTM networks are a type of RNN that uses special units in addition to standard units. LSTM units include a 'memory cell' that can maintain information in memory for long periods of time. A set of gates is used to control when information enters the memory, when it's output, and when it's forgotten. This architecture lets them learn longer-term dependencies.

In this project, extensive data preprocessing has been performed and multiple variations of Deep Neural Network models for Natural Language Processing on Online Social Network data have been experimented with. A sentiment classification task on human generated tweets mined from the popular Social Networking site called Twitter is the objective and achieving the highest performance metric is the aim.

A novel hybridised Recurrent LSTM model has been proposed at the end of the experimentation giving the highest performance achieved on this dataset as compared to other research papers and code blogs.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Overview

This project aims to employ Deep Learning methods in order to perform Sentiment Analysis on textual data with the use of Natural Language Processing Techniques. Sentiment Analysis is a technique to identify and categorize opinions underlying within data in discrete categories. By definition, Sentiment Analysis means to determine whether the sentiment towards any topic or product etc. is positive, negative, or neutral from a text corpus.

Deep Learning, a sub-field of machine learning that mimics the mechanism of the human brain to develop algorithms that are neural implementations of “neural networks” have been used to exploit the full potential of sentiment analysis tools. Harnessing the power of deep learning, sentiment analysis models can be trained to understand text beyond simple definitions, read for context, sarcasm, etc., and understand the actual mood of the writer.

This is a binary class classification problem with two output classes, positive and negative sentiments. The task achieved is to clean the text corpus and train the best model possible that gives highest accuracy in predicting sentiment of a tweet text, as compared to the already labelled sentiment of each tweet in the dataset.

For the purpose of NLP on textual data, the most relevant type of Neural Networks is the Recurrent Neural Networks, which come into picture when there’s a need for predictions using sequential data. Sequential data can be a sequence of words, images, etc. The layers in an RNN receive a time-delayed input of the previous instance prediction which is stored in the RNN cell that is a second input for every prediction. However, there is a problem with RNNs which makes it very difficult to remember earlier layer weights- the Vanishing Gradient problem arising due to the short term memory of RNNs. Herein comes the need for Long Short-Term Memory Networks. LSTM neural networks overcome the issue of Vanishing Gradient in RNNs by adding a special memory cell that can store information for long periods of time.

1.2 Motivation

The Deep Learning Market is expected to register a CAGR (Compound Annual Growth Rate) of 42.56% over the forecast period from 2020 to 2025. A subfield of machine learning (ML), it has led to breakthroughs in several artificial intelligence tasks, including speech recognition and image recognition. Furthermore, the ability to automate predictive analytics is leading to the hype for ML. Factors, such as enhanced support in product development and improvement, process optimization and functional workflows, and sales optimization, among others, have been driving enterprises across industries to invest in deep learning applications.

Nevertheless, the latest machine learning approaches have significantly improved the accuracy of models, and new classes of neural networks have been developed for applications, like image classification and text translation. [36]

Over the last decade, there has been extensive research going on in the field of Natural Language Processing. There have been tremendous advancements and breakthroughs as reported in the theory part. Sentiment analysis is very famous due to its trivial nature. We, as humans, have our feelings, expressions or sentiment towards everything. If those sentiments could be used for user specific recommendations and applications, it would be really great both for companies and consumers.

Applications include social media analytics, customer feedback analysis, brand monitoring and reputation management, product analysis, market and competitive research, etc. Considering the wide range of applications in business, it seemed to be an apt research topic to pick up.

I wish to build a career in the field of Analytics, particularly Text Analytics for insights in various purposes. This requires me to develop an in-depth understanding of these Text Analytics techniques and tools in Natural Language Processing which inevitably includes Deep Learning. Having done traditional Machine Learning in the past semesters, moving one step ahead and taking up advanced Machine Learning seemed a good idea as I find these technical fields extremely interesting. In order to develop experience in the field and get familiar with past and ongoing research, I decided to take up this project so that by the time I am down on the fields to perform these as an employee, I am well ahead of new learners aside me and already have an in-depth understanding and application experience of the work.

1.3 Problem Statement

For this project, the use of NLP is made to perform ‘Sentiment Analysis’ on online social network data. The most widely available and ample data being from the social networking site called Twitter, this project is going to be on Twitter Sentiment Analysis.

The problem at hand is to analyse the sentiment of any given sentence which does not belong to a particular topic and is generic in meaning. The aim is to be able to classify the sentiment that sentence/tweet holds without any particular context. This requires the tweets in the chosen data to be completely random in topic. This will however drastically reduce accuracy of any NLP model because of such a wide variety of contexts in the tweets and hence the problem becomes more complicated and harder to solve. Since the data must cover all the topics instead of say for example just sentiment of users about cars or about a movie, etc. Hence, a Tweet text based generic dataset was chosen for this project called the sentiment140-twitter microblogging dataset uploaded on kaggle [2].

1.4 Dataset

The dataset being used for this project is originally available on Kaggle as 'Sentiment140 dataset with 1.6 million tweets', but the student version of the same dataset available from STanford with all emoticons removed has been used. It contains 1.6 million tweets extracted using the twitter api. The tweets have been annotated with 2 classes of sentiments (0 = negative, 4 = positive). It contains the following 6 fields (Examples in brackets):

1. target: polarity of the tweet text (0 = negative, 4 = positive)
2. ids: The id of the tweet (2093)
3. date: the date on which the tweet was tweeted (Sat May 16 13:24:03 UTC 2019)
4. flag: The query. This value is NO_QUERY if there is no query.
5. user: the user that tweeted (debsa2000)
6. text: the text of the tweet

An impressive feature of this dataset is that it is perfectly balanced. Out of the 1.6 million tweets, 0.8 million tweets belong to each of the two classes (positive and negative sentiment labels). This is another reason for choosing this dataset particularly over any other generic tweet dataset. A visualization of the same has been reported in the Data Skewness Visualization section.

Since the data consists of raw tweets extracted from twitter directly, the tweets are not clean and require preprocessing to be done in order to use them for our machine learning models. Hence, extensive preprocessing has been done before feeding the data to the deep learning model.

Chapter 2
Literature Survey

Most of the reviewed literature proposed several types of hybrid LSTM models. These combine conventional LSTMs with other models. Most of them use Bi-directional LSTMs or otherwise stack Uni-directional LSTMs to form hidden layers.

Most commonly, Convolution Neural Networks are combined with LSTMs to create an optimized model such as the tree-structured regional CNN-LSTM model for Dimensional Sentiment Analysis[1], Co-LSTM: Convolutional LSTM model, hybrid CNN-LSTM model [5], A Hybrid CNN-LSTM Model with dorput and normalization rectified linear unit for accuracy improvement [6], ensemble of CNN and Bi-LSTM models [7], lexicon integrated two-channel CNN-LSTM family models [10], deep CNN-LSTM with combined kernels from multiple branches [14]. Stacking GRUs with LSTMs is proposed [18] and an 8-layer SR-LSTM (Stacked Residual LSTM) model consisting of several advanced regression techniques is also proposed [20].

Newer types of LSTMs with an attention mechanism such as the Attention-based LSTM for Aspect-level Sentiment Classification [3], Sentic LSTM: a Hybrid Network for Targeted Aspect-Based Sentiment Analysis [2], Targeted Aspect-Based Sentiment Analysis via Embedding Commonsense Knowledge into an Attentive LSTM [4], Lexicon-Enhanced LSTM With Attention for General Sentiment Analysis [8], Twitter Sentiment Analysis via Bi-sense Emoji Embedding and Attention-based LSTM [9], Bidirectional LSTM with self-attention mechanism and multi-channel features for sentiment classification [21] have been introduced.

In the paper [11], comparison of 3 LSTM models, namely Conventional LSTM, Deep LSTM and Bidirectional Deep LSTM is done in terms of Validation Accuracy and Validation Loss. In another paper [12], Dynamic Routing is investigated in Tree-Structured LSTM for Sentiment Analysis. Moreover, for conversational sentiment analysis, learning interaction dynamics is done with an interactive LSTM in another paper.

Optimizations of word embeddings like the WWE(Weighted Word Embeddings) [15] and Sentiment Specific Word Embeddings [16] are also proposed.

Apart from Deep learning, many papers also provide good results with an ensemble of traditional Machine Learning models like SVM and Logistic Regression models [17].

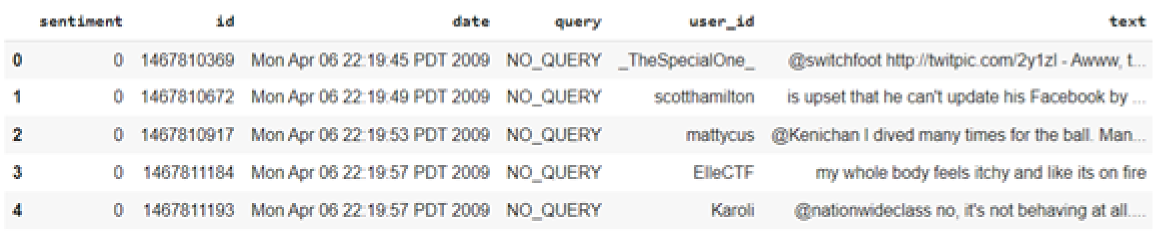
Text classification is seen to be improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling [20].

Chapter 3
Work Done

3.1 Exploratory Data Analysis

3.1.1 Discovering the dataset

a. Import the dataset: The first step is importing the original raw csv dataset.
b. Naming Columns (since unnamed): The dataframe is without column heading names. Hence there is a need to name the columns as described in the Data section.



	sentiment	id	date	query	user_id	text
0	0	1467810369	Mon Apr 06 22:19:45 PDT 2009	NO_QUERY	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by ...
2	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Man...
3	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
4	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all....

TABLE 3.1 SAMPLE DATA FROM DATASET

c. Dealing with missing values (if any): Many times the datasets are imperfect and contain many empty cells in between. Hence it is absolutely necessary to check for Null values and eliminate such rows if any. It is found that there are no Null values in the entire dataset.
d. Removing redundant columns (not required for analysis): The dataset contains 6 columns (out of which only 2 are useful for the purpose of this project which is sentiment analysis on the tweet text, hence the other columns are dropped.
e. Labelling classes: The sentiment classes are labelled from ‘0’ and ‘4’ to ‘Negative’ and ‘Positive’.

3.1.2 Data skewness Visualization

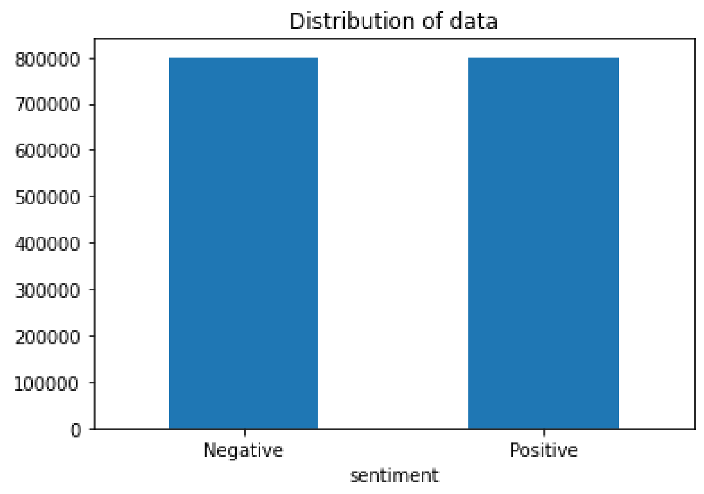
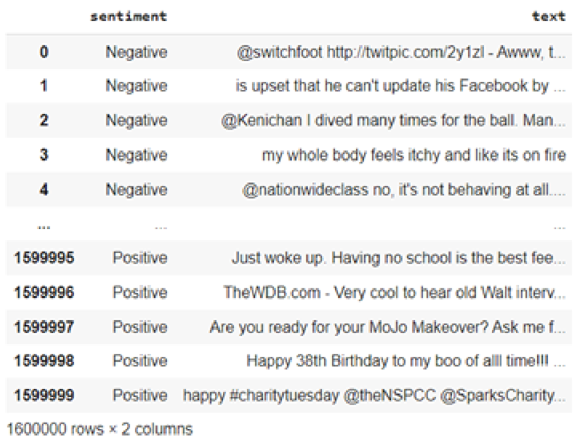


FIGURE 3.1 DATA SKEWNESS BAR PLOT

It is evident from the plot that the data is perfectly balanced as mentioned in the data description section earlier. This proves that the two output classes in the dataset sum up to the entire dataset.



	sentiment	text
0	Negative	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	Negative	is upset that he can't update his Facebook by ...
2	Negative	@Kenichan I dived many times for the ball. Man...
3	Negative	my whole body feels itchy and like its on fire
4	Negative	@nationwideclass no, it's not behaving at all....
...	...	...
1599995	Positive	Just woke up. Having no school is the best fee...
1599996	Positive	TheWDB.com - Very cool to hear old Walt interv...
1599997	Positive	Are you ready for your MoJo Makeover? Ask me f...
1599998	Positive	Happy 38th Birthday to my boo of alll time!!! ...
1599999	Positive	happy #charitytuesday @theNSPCC @SparksCharity...

1600000 rows × 2 columns

TABLE 3.2 SAMPLE DATA WITH REDUNDANT COLUMNS REMOVED AND CLASSES LABELLED

3.2 Data Cleaning

STEP-WISE DATA CLEANING AND PREPROCESSING:

The order in which these steps are done is of utmost importance as many of the steps depend on previous steps. The following are the steps most appropriate for this dataset:

a. **Removing URLs, hashtags, mentions and old style retweet text**

URLs (Uniform Resource Locators) in a text are references to another webpage on the web, but do not provide any additional information and serve no purpose for our classification of sentiment polarity. Hence, strings starting with “http” or “https” are removed since these only create junk in the data given their wide presence and we do not want them to be a part of our corpus.

A hashtag is a metadata tag which is prefaced by the hash symbol '#'. These are used on microblogging and photo-sharing services such as social networking websites as a form of user-generated tagging that enables cross-referencing across content, that is, sharing specifying a topic or theme. Hashtags are used abundantly in tweets and do not serve any purpose for sentiment analysis as they usually are a set of joined words, all in small cases due to which they cannot even be split up into separate words to be used up for analysis. Hence we remove them.

A mention is a tag to another person's username anywhere in the body of the Tweet using the '@' symbol. This is especially done to send notifications to the ones mentioned in the tweet so they can know that they have been addressed and do not miss out on replying or reacting.

These are of the form '@userprofilename' and clearly are of no use to us, hence we get rid of them.

During earlier days of Twitter, the retweets used to contain a text- 'RT'. Since the repetition would give it importance in the model, but actually it is of no significance, we remove these.

Thus, the above 4 things are removed using the library named "re" in python, which provides regular expression matching operations.

b. Split attached words

In row 1599996 that 'TheWDB.com' has been split to 'The WDB .com'

c. Lowercasing all words

If the text is in the same case, it is easy for the machine to interpret the words because the lowercase and uppercase are treated differently by the machine. For example, words like Ball and ball are treated differently. So, we need to make the text in the same case, preferably lowercase (because stopwords in a later step are all in lowercase by default in the used library).

d. Expand contracted words

Contracted words or contractions are shorter words made by putting two words together wherein few letters are omitted in the contraction and replaced by an apostrophe. The apostrophe shows the position of the omitted letters. Examples of contracted words :

do not -> don't | is not -> isn't | he is -> he's

Converting contractions into their natural form will help in simplifying our data for computation.

e. Removing punctuation

Punctuation is basically the set of symbols [!"#$%&'()*+,-./:;?@[\]^_`{|}~]:

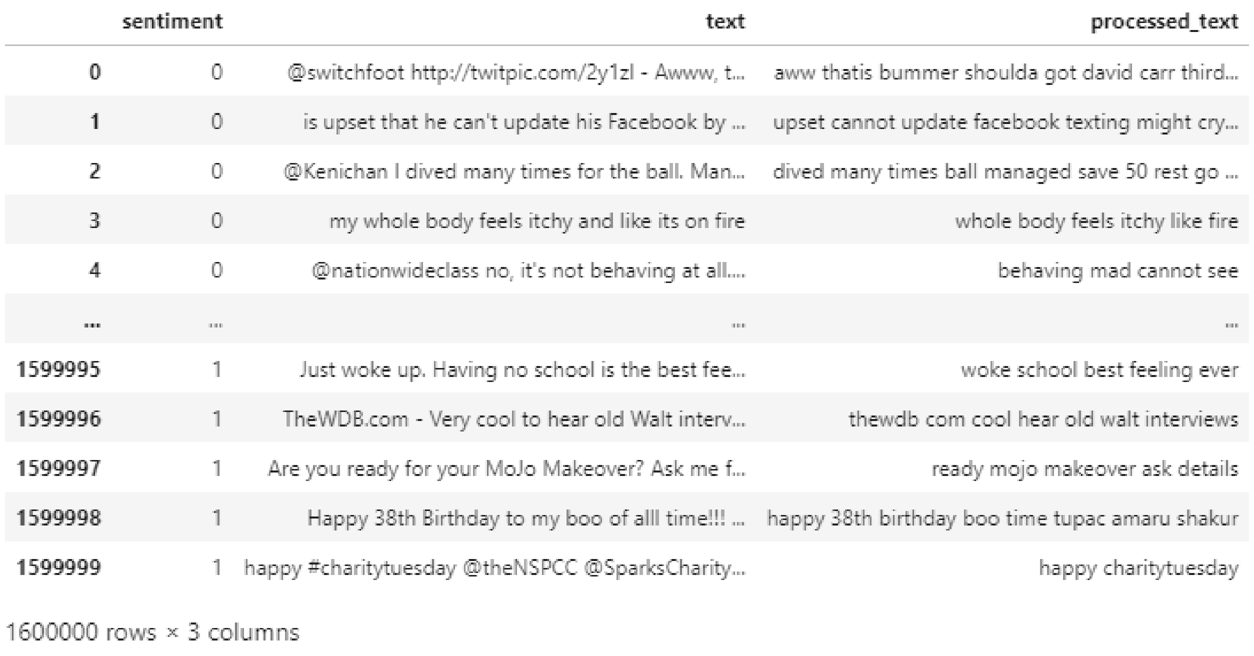
There are a total of 32 main punctuations that need to be taken care of. The string module can be directly used with a regular expression to replace any punctuation in text with an empty string. After removal of punctuation, it can be seen that none of the tweets contain any of the special characters.

f. Removing Stopwords

“Stopwords” is the name used to describe the most commonly occurring words in a language like “the”, “a”, “me”, “is”, “to”, “all”,. Words like I, me, you, he and others increase the size of text data but don’t help our model to understand the context better and thus can be removed.

For the task, a predefined stopwords collection (e.g. from NLTK or any other NLP library) can be used. We can create a customized list of stop words for different problems or add to this list. I added a few words to the list.

{'a', 'about', 'above', 'after', 'again', 'against', 'ain', 'all', 'am', 'an', 'and', 'any', 'are', 'aren', "aren't", 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between', 'both', 'but', 'by', 'can', 'couldn', "couldn't", 'd', 'did', 'didn', "didn't", 'do', 'does', 'doesn', "doesn't", 'doing', 'don', "don't", 'down', 'during', 'each', 'few', for', 'from', 'further', 'had', 'hadn', "hadn't", 'has', 'hasn', "hasn't", 'have', 'haven', "haven't", 'having', 'he', 'her', 'here', 'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in', 'into', 'is', isn', "isn't", 'it', 'its', 'itself', 'just', 'll', 'm', 'ma', 'me', mightn', "mightn't", 'more', 'most', 'mustn', "mustn't", 'my', 'myself', 'needn', "needn't", 'no', 'nor', 'not', 'now', 'o', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'our', 'ours', 'ourselves', 'out', 'over', 'own', 're', 's', 'same', 'shan', "shan't", 'she', "she's", 'should', "should've", 'shouldn', "shouldn't", 'so', 'some', 'such', 't', 'than', 'that', "that'll", 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', 'these', 'they', 'this', 'those', 'through', 'to', 'too', 'under', 'until', 'up', 've', 'very', 'was', 'wasn', "wasn't", 'we', 'were', 'weren', "weren't", 'what', 'when', 'where', 'which', 'while', 'who', 'whom', 'why', 'will', 'with', 'won', "won't", 'wouldn', wouldn't", 'y', 'you', "you'd", "you'll", "you're", "you've", 'your', 'yours', 'yourself', 'yourselves'}



	sentiment	text	processed_text
0	0	@switchfoot http://twitpic.com/2y1zl - Awww, t...	aww thatis bummer shoulda got david carr third...
1	0	is upset that he can't update his Facebook by ...	upset cannot update facebook texting might cry...
2	0	@Kenichan I dived many times for the ball. Man...	dived many times ball managed save 50 rest go ...
3	0	my whole body feels itchy and like its on fire	whole body feels itchy like fire
4	0	@nationwideclass no, it's not behaving at all....	behaving mad cannot see
...	...	...	...
1599995	1	Just woke up. Having no school is the best fee...	woke school best feeling ever
1599996	1	TheWDB.com - Very cool to hear old Walt interv...	thewdb com cool hear old walt interviews
1599997	1	Are you ready for your MoJo Makeover? Ask me f...	ready mojo makeover ask details
1599998	1	Happy 38th Birthday to my boo of alll time!!! ...	happy 38th birthday boo time tupac amaru shakur
1599999	1	happy #charitytuesday @theNSPCC @SparksCharity...	happy charitytuesday

1600000 rows × 3 columns

TABLE 3.3 DATA AFTER CLEANING

3.3 Text Visualization

Data visualizations is an effective way to communicate important information present in data, at a glance. When the data is text-based, using a word cloud can convey crucial information. It is a visualization format to highlight important textual data points by sizing. Hence, the best way to visualize what text each sentiment is tagged with is plotting a word cloud.

Word Clouds, also known as wordle, word collage or tag cloud, are visual representations of words by giving greater prominence to words that appear more frequently. A word cloud is a collection or cluster of words depicted in different sizes, colours and boldness. The bigger and bolder the word appears, the more often has it been mentioned within a given text and hence the more important it is.

This type of visualization can help to quickly and most efficiently highlight the most important data points in text and present the data in a way that everyone can understand.

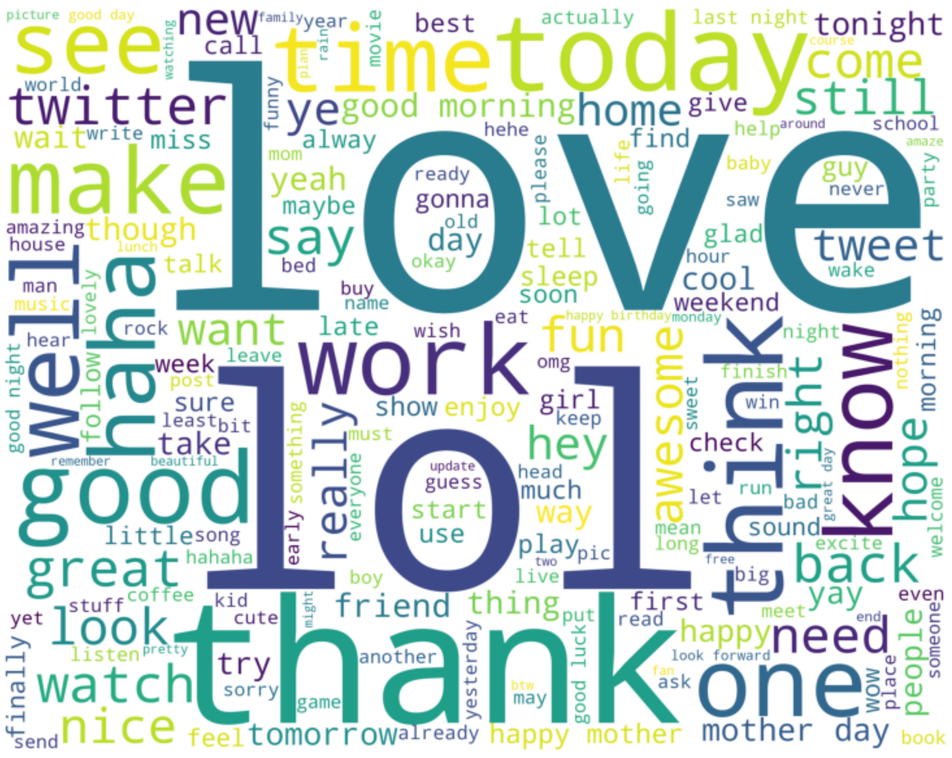


FIGURE 3.2 POSITIVE WORDCLOUD

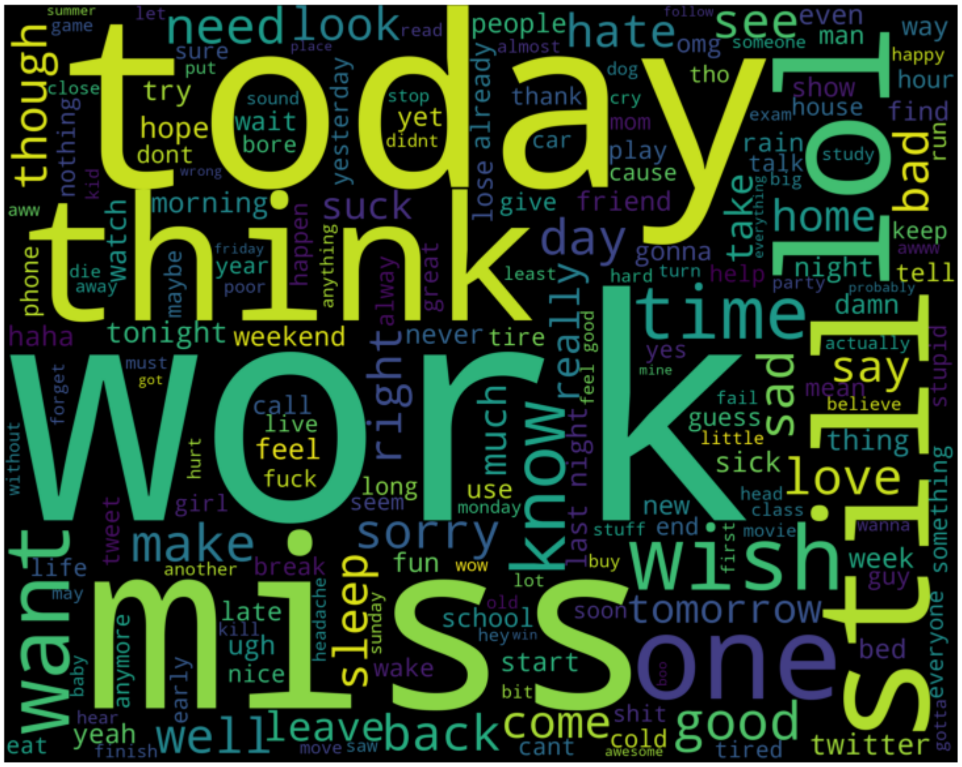


FIGURE 3.3 NEGATIVE WORDCLOUD

3.4 Data Preprocessing

a. Splitting data

Machine Learning models are trained and tested on different sets of data. This is done so to reduce the chance of the model overfitting to the training data, i.e it fits well on the training dataset but has poor fit with new ones.

The library sklearn.model_selection.train_test_split shuffles the dataset and splits it into train and test dataset.

The Pre-processed Data is divided into 2 sets of data:

1. Training Data: The dataset upon which the model would be trained on. Contains 95% data.

2. Test Data: The dataset upon which the model would be tested against. Contains 5% data.

The training data is 95% of the data and the testing data is 5% of the data. Validation will be done on 10% of training data during training.

b. Tokenization, Unigram Sequence modelling and Zero Padding

Tokenization means to separate a piece of text into smaller units called tokens which can be either words, characters, or subwords. Consequently, tokenization can be classified into 3 types – Word Tokenization, Character Tokenization, and Subword (n-gram characters) Tokenization.

In computational linguistics and probability, an ‘n-gram’ is a contiguous sequence of n items from a given sample of text or speech. In NLP, the concept of N-gram is widely used for text analysis. An N-gram sequence of size 1 is referred to as a “unigram“, of size 2 is a “bigram” and of size 3 is called a “trigram”.

N-gram plays an important role in text analysis in Machine Learning because sometimes a single word alone isn’t sufficient to observe the context of a text.

Example: For sentiment prediction of the following text into positive or negative,

text= “The pizza is not bad in taste”

1-gram	**2-gram**	**3-gram**
The	The pizza	The pizza is
pizza	pizza is	pizza is not
is	is not	is not bad
not	not bad	not bad in
bad	bad in	bad in taste
in	in taste	
taste		

TABLE 3.4 N-GRAM SEQUENCING EXAMPLE

If we consider unigram or a single word for text analysis, the negative word “bad” leads to the wrong prediction of the text. But if we use bigram, the bigram word “not bad” helps to predict the text as a positive sentiment.

Converting the entire corpus into a unigram model would essentially mean tokenizing the entire corps into single word tokens. So, in this case, the tokenization of 1-gram characters i.e.

word tokenization of the entire corpus into a single sequence would be unigram sequence modelling.

This will help us in creating an embedding matrix which will store all the word embeddings obtained from Word2vec, word by word so that when a sentence is fed as an input, each word's representation can be looked at in this embedding matrix.

All neural networks require inputs that have the same shape and size. But, when we pre-process text data as inputs for our model e.g. LSTM, not all the sentences have the same length. Since the inputs must be of the same size, this is where zero padding is utilised.

Padding is the process by which we can add padding tokens at the start or end of a sentence to increase its length upto the required size. If required, we can also drop some words to reduce to the specified length.

Tokenizer: Tokenizes the dataset into a list of tokens.

pad_sequences: It is a function that zero-pads the tokenized data to a certain length.

The input_length has been set to 60 which is the maximum length each tweet text will be padded to and which will be the length after the data is tokenized and padded. 60 has been chosen because the longest tweet has been found to be 57 words.

3.5 Word Embedding

A Long Short-Term Memory, or LSTM, is a type of machine learning neural network. More specifically, it belongs to the family of Recurrent Neural Networks (RNN) in Deep Learning, which are specifically conceived in order to process temporal data. Temporal data is defined as data that is highly influenced by the order that it is presented in. This means that data coming before or after a given datum (singular for data) can greatly affect this datum. Text data is an example of temporal data.

For example:

1) In the sentence "*Maya is not very happy. She's still mad at you!"*, the word "not" greatly influences the meaning of the upcoming words "very happy".
2) Conveying the influence of words' positions directly influencing a sentence's meaning:
   We eat to live.
   We live to eat.
   The minute nuances of such differences are smoothly handled by a Recurrent Neural Network, specifically LSTM in this case.

a. Word Embedding using Word2Vec:

Word embedding is one of the most popular representations of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. Loosely speaking, word embeddings are vector representations of a particular word.
Word2Vec was developed by Google and is one of the most popular techniques to learn word embeddings using shallow neural networks. It is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned through Word2Vec have proven to be successful on a variety of downstream natural language processing tasks.

Word2Vec can create word embeddings using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW).

- Continuous Bag-of-Words Model which predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

- Continuous Skip-gram Model which predicts words within a certain range before and after the current word in the same sentence. A worked example of this is given below.

Choosing Training Parameters:

- **vector size=100** : The number of dimensions (N) that the Word2Vec maps the words onto. Bigger size values require more training data, but can lead to better (more accurate) models. The vector size is obviously equal to the embedding dimensions. Generally, the exact number of embedding dimensions does not affect task performance. The number of dimensions can affect training time. A common heuristic is to pick a power of 2 to speed up training time.

- **workers=4**: Specifies the number of worker threads for training parallelization, to speed up training. It controls the number of independent threads doing simultaneous training. In general, more workers than the number of CPU cores is never used. If your system has 2 cores, and if you specify workers=2, then data will be trained in two parallel ways. By default , worker = 1 i.e, no parallelization. Since my CPU has 4 cores, I use 4 here.

The vocab size after building the gensim Word2Vec model on the dataset is found to be 49476.

The word2vec model is trained for 32 epochs which takes about 4 minutes.

After training, I checked the model by trying to find words most similar to the word “love” which is the biggest word in the positive wordcloud. The output is:

```
[('adore', 0.640842080116272),
 ('loved', 0.6098442673683167),
 ('luv', 0.60610032081604),
 ('lt3', 0.5747049450874329),
 ('loove', 0.5223230123519897),
 ('amazing', 0.5129105448722839),
 ('loooove', 0.5113006830215454),
 ('looove', 0.5109059810638428),
 ('miss', 0.5001137852668762),
 ('loving', 0.4910278022289276)]
```

Hence, the Word2Vec model is justified to be trained properly.

b. Creating Embedding Matrix and generating Embedding Layer

Embedding Matrix is a matrix of all words and their corresponding embeddings. We use embedding matrix in an Embedding layer in our model to embed a token into it's vector representation that contains information regarding that token or word.

Final step before modelling our LSTM network is to construct an embedding layer. Embedding layer is the first and foremost layer of our LSTM model. As our data is actually in the form of a sentence, the size of each input will vary and can never be known beforehand. The embedding layer will make sure that all the inputs that are being fed into the LSTM network are of same length by padding zeros if the length is shorter than a predefined length and will cut down if the
length is larger than a predefined length. Embedding layer takes the words from our sentence and looks it up in our embedding matrix. Then from the embedding matrix, it picks up the vector representation(word2vec representation) of that word and stacks the word vectors on top of each other making the data in the shape required by LSTM.

Keras offers an Embedding layer that can be used for neural networks on text data. It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the Tokenizer API also provided with Keras.

The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

- **input_dim**: This is the size of the vocabulary in the text data which is 60000 after training the word2vec model.
- **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. In our case this is 100.
- **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. Since padding was done for a maximum length of 60, input_length is 60.

The Embedding layer has weights that are learned. These weights are assigned as equal to the embedding matrix created.

The output of the Embedding layer is a 2D vector with one embedding for each word in the input sequence of words (input document).

Hence the shape of our embedding matrix should be and is found to be:
(total number of words after tokenizing, max_length of padding) = (60000,60)

3.6 Theory

3.6.1 Introduction to RNNs

Recurrent Neural Networks or RNN were introduced in 1982 by John Hopfield as special Hopfield Networks but computational RNN came into existence by David Rumelhart's work in 1986 [4]. RNNs were introduced to tackle problems where the input has some dynamic temporal nature. RNN is specifically designed for data that has dependencies within itself. Feed Forward Neural Nets were unable to grasp sequential aspects of the data whereas Recurrent Neural Networks take previous outputs into account to handle the sequential nature of inputs.

3.6.1.1 Architecture of RNNs

RNNs are also known as feedback networks in which connections between units form a directed cycle. RNNs can use their internal memory to process arbitrary sequences of inputs. In RNN, the signals travel both forward and backward by introducing loops in the network as shown in FIGURE 3.4:

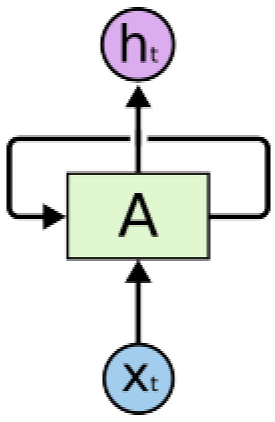


FIGURE 3.4 BASIC RNN FEEDBACK STRUCTURE

In a way, RNNs can be seen as wrapped up FFNNs where each cell has a connection with the previous cell and a new input coming in. This is shown in FIGURE 3.5 where every cell's output is connected to the next cell and is shown in unrolled fashion. The fact that RNNs can use their internal state to process sequences of input makes them applicable to numerous tasks including but not limited to speech recognition, summary generation and word prediction.

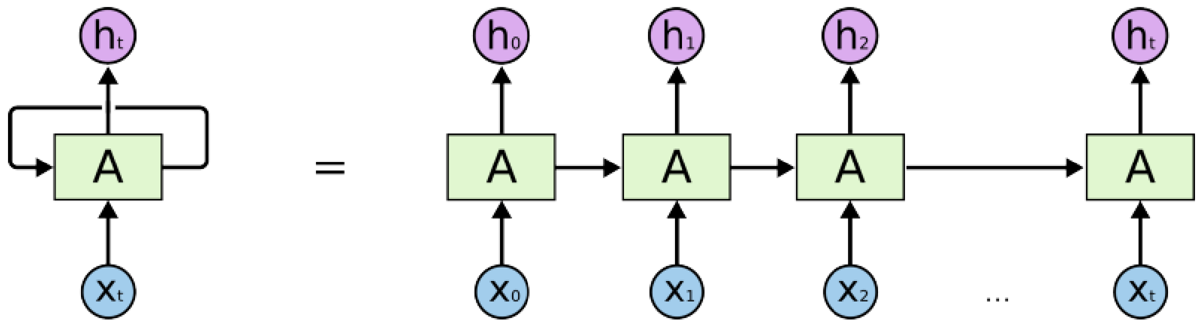


FIGURE 3.5 UNFOLDED RNN W.R.T. TIME

3.6.1.2 Working of RNNs

RNN takes input as a form of tensor or a 3-D matrix in which all the instances are stored row wise. Instances are fed to the network(layer of neurons) and errors are calculated using predicted output and actual output. This error is then back propagated to update the weights by computing gradients similar to FFNNs.

Back propagation in time

To reduce the errors by backpropagation method, RNN uses BPTT to compute gradients for updation of weights. This means that the gradient needs to be calculated over time. Since there is no direct connection of output to the very first input, BPTT follows a chain-like sequence of partial derivatives to observe the change. The whole point of introducing RNN was to take into account the sequential data and learn from the past to apply in the future. In fact, Kilian and Siegelmann [5] proved that from a theoretical perspective, the RNN is a universal functional representation. The theory is promising, but it does not necessarily translate to practice. While it is nice to know that it is possible for an RNN to represent any arbitrary function, it is more useful to know whether it is practical to teach the RNN a realistic functional mapping from scratch by applying gradient descent algorithms. Turns out that in practicality, RNN faces problems of vanishing and exploding gradients in handling long term dependencies and tends to forget things that need to utilize memory from long back.

Short Term Dependencies

Sometimes, we only need to look at recent information to perform the present task. Consider a word predicting problem which has very few sentences and length of sentences is also very less. This problem does not require a lot to be remembered that the problem of vanishing or exploding gradients could arise. In such cases, where the gap between the relevant information and the place where it's needed is small as shown in FIGURE 3.6, RNNs can learn to use the past information. It is shown that the output h3 is at max. dependent on inputs x0 and x1 which are not very far away to cause vanishing and exploding gradient problems.

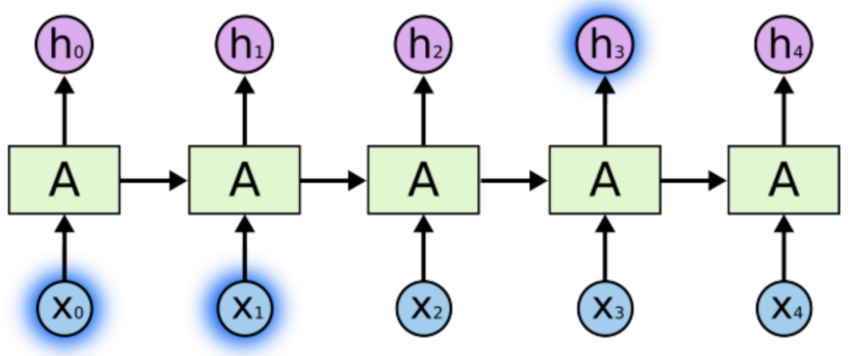


FIGURE 3.6 NETWORK WITH SHORT-TERM DEPENDENCIES

Long Term Dependencies

Sometimes, we need to look at some information very far away from our current point. An intuitive example would be to suppose that there is a million word long text file in which the last word depends on the information provided in the very first line. This type of dependency is called long term dependency and RNNs practically face vanishing and exploding gradient problems. A network model depicting this type of dependency is shown in FIGURE 3.7 in which the output ht+1 is dependent on the inputs x0 and x1 far away. This could cause our model the mentioned gradient problems.

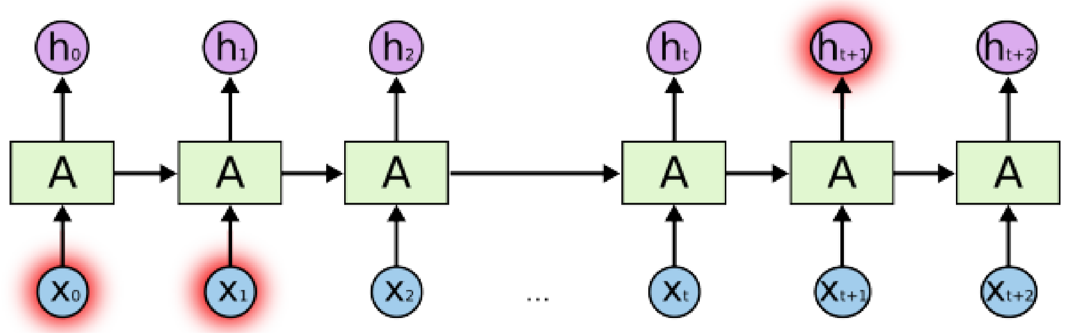


FIGURE 3.7 NETWORK WITH LONG-TERM DEPENDENCIES

3.6.1.3 Limitations of simple RNNs

Simple Recurrent Neural Networks work fine when dealing with short-term dependencies. For example: *"The sky is __________"*. A simple RNN will work fine in this case because this problem has nothing to do with the context of the statement. The RNN does not need to remember what was said before this, or what was its meaning, all it needs to know is that in most cases, the sky is blue. Thus the prediction would be: *"The sky is blue"*.

However, a simple RNN will fail to understand the context behind an input. If something was said long before, it cannot be recalled when making predictions in the present by a simple RNN.

For instance: *"I lived in France for 20 years and then moved to Australia.*

.

.

I can speak fluent ______".

It can be understood that since the author has lived in France for 20 years, it is very likely that he speaks French fluently. To make a proper prediction, the RNN needs to remember this context. The relevant information may be separated from the point where it is needed, by a huge load of irrelevant data. This is where a Recurrent Neural Network fails.

The reason behind this is the problem of Vanishing Gradient. Since for a conventional feed-forward neural network, the weight updating that is applied on a particular layer is a multiple of the learning rate, the error term from the previous layer and the input to that layer. Thus, the error term for a particular layer is somewhere a product of errors of all previous layers. When dealing with activation functions like the sigmoid function, the small values of its derivatives (occurring in the error function) get multiplied multiple times as we move towards the starting layers. As a result of this, the gradient almost vanishes as we move towards the starting layers, and it becomes difficult to train these layers.

A similar case is observed in Recurrent Neural Networks. RNN remembers things for just small durations of time, i.e. if we need the information after a small time it may be reproducible, but once a lot of words are fed in, this information gets lost somewhere. This issue can be resolved by applying a slightly tweaked version of RNNs – the Long Short-Term Memory Networks.[39]

3.6.2 Introduction to LSTM Networks

As explained earlier, it is very difficult for RNN models to recollect the information from the initial states for long term time dependencies due to vanishing gradients explained in chapter 2 subsection 2.2.2. To tackle these problems, Hochreiter and Schmidhuber proposed the Long Short Term Memory (LSTM) approach in 1997. This is a modified version of Vanilla Recurrent Neural Networks but the basic principle has remained the same. LSTM is basically a part of RNN with extra control of the memory through internal mechanisms of several gates inside each LSTM cell. We can now control the amount of information relevant to us which can be retained and the amount that we can forget.

3.6.2.1 Architecture of LSTM networks

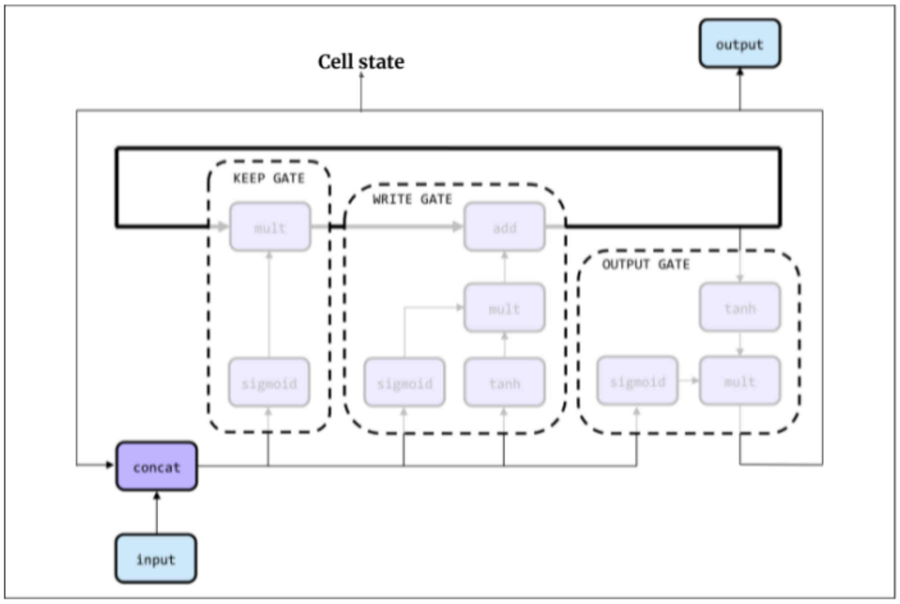


FIGURE 3.8 LSTM ARCHITECTURE

The input and output blocks are tensors and concat is concatenation operation which will be discussed later.

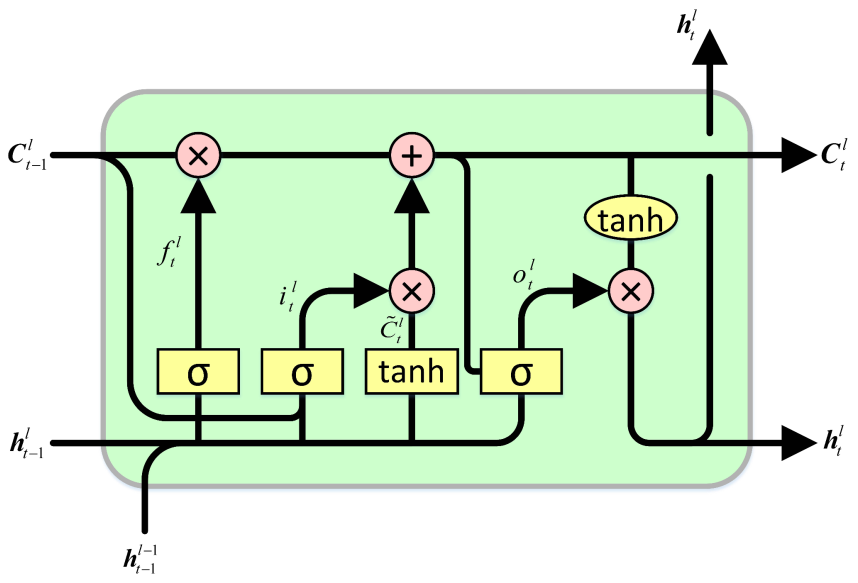


FIGURE 3.9 MATHEMATICAL STRUCTURE

3.6.2.2 Components of LSTM networks

Cell State

Cell state or Memory state is the thing that is responsible for solving gradient problems. It is one of the major components of the LSTM structure which acts as a transport highway and is shown at the top in figure 3.1. Basically, it holds critical information that it has learned over time. The cell state carries information throughout the sequence processing. Information through earlier time steps could be carried all the way to the last time step. The network is designed to effectively maintain useful information in the memory cell over many time steps. Thus, eradicating the problems of memory loss due to vanishing gradients. The information gets added or removed from the cell state via gates.

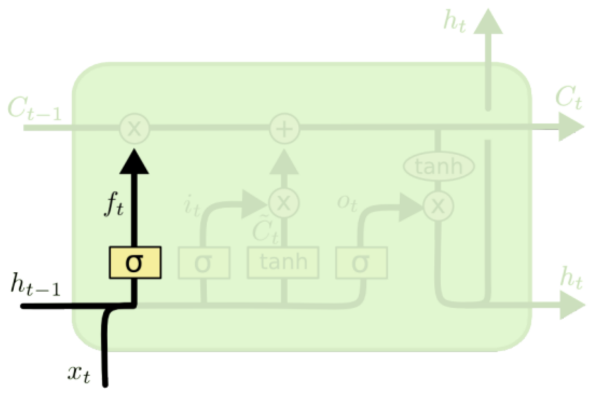
Hidden State

The output from the previous cell is called the hidden state. Similar to the output that was being fed to the RNN's next layer as input along with the actual current input as shown in figure 2.2. The only difference is that the output for RNN was based on the memory that it had recollected from just previous output while here the hidden state contains the output based on the memory collected by the cell state. The LSTM units are engineered to provide more flexibility by emitting an output that is an "interpretation" or "external communication" of what that cell state represents.

<u>Gates</u>

Gates are a way to optionally let information through the cell state. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. A value of zero means “let nothing through” while a value of one means “let everything through”. LSTM has 3 gates namely input gate(write gate), output gate, and a forget gate(keep gate) to control the information content of the cell state.

Forget/Keep Gate: First, the unit must determine how much of the previous memory to keep. The cell state from the previous time step h (t−1) is rich with information, but some of that information might be stale and therefore might need to be erased. So, the forget gate decides which information to keep and which to throw away.

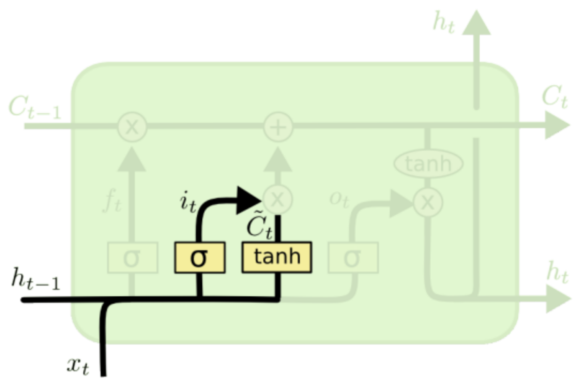


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

FIGURE 3.10 FORGET GATE

The gate shown in FIGURE 3.10 consists of a neural net with sigmoid function as the activation function, and current input xt and hidden state of previous time step ht−1 after concatenation as the input with Wf as weight of the NN and bf as bias. The neural net then creates a bit tensor mask of values close to 0s and 1s due to sigmoid activation. The sigmoid output of this NN is then multiplied with the previous cell state. If a particular location in that tensor holds a value close to 1, it means that location in the memory cell ought to be kept while if that particular location instead held a 0, it means that location in the memory cell is no longer relevant and ought to be erased.

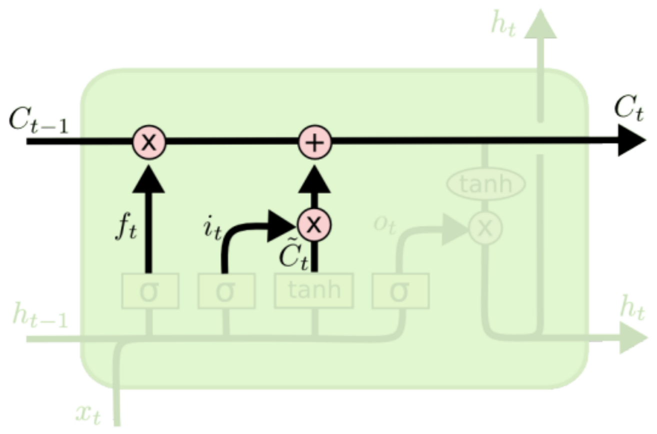
Input/Write Gate: This gate decides what new information are we going to update in our cell state. This gate actually consists of two parts. One is a Tanh layer that creates a vector of new candidate values C (an intermediate tensor) which could be the possible values to be added to the cell state and decides what information we’d like to write in the cell state. Another is a sigmoid layer that decides on the importance of the candidate values to be added to the cell state and figures out which components of the intermediate tensor we actually want to include into the cell state and which components we want to toss by creating a bit tensor mask using the same strategy as done in the forget gate.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

FIGURE 3.11 INPUT/WRITE GATE WITH CANDIDATE VALUE

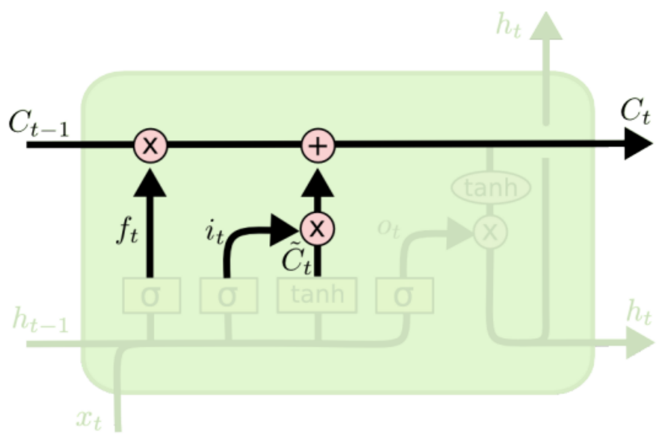
In order to update the previous cell state which we had multiplied with the forget gate, we add the candidate values that we had just obtained post deciding their importance with the previous cell state. The equation of the input gate's NNs is given by FIGURE 3.11 and FIGURE 3.12



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

FIGURE 3.12 UPDATION OF PREVIOUS CELL STATE WITH LATEST INPUTS

Output Gate: The output gate decides what our output is going to be which is also our next hidden state. This output is used for prediction if required else is passed on to the next LSTM cell as the hidden state. First, the sigmoid layer decides the importance of the output values then the current cell state is passed through a tanh layer (to keep the values between -1 and 1) and then both the outputs are multiplied to get the final output or next hidden state. The new obtained hidden state and the current cell state are passed to the next LSTM layer as the input along with the actual input to predict the output. The equations of the output state are given by FIGURE 3.13



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

FIGURE 3.13 OUTPUT GATE

3.7 Experimentation with Models

Every model that has been experimented and trained to be evaluated has its unique model architecture, model evaluation and model testing sections. However, the sequential nature of the architecture, model compilation parameters such as optimizers, losses and metrics and fitting parameters such as batch size and callbacks are consistent across all models. Hence, the details about these common characteristics and different components used in the models are explained in the following sections.

3.7.1 Methodology

3.7.1.1 Building a model

Keras models represent the actual neural network model. Keras provides two modes to build a model:

1. a simple and easy to use *Sequential API*
2. a more flexible and advanced *Functional API*.

For the purpose of our task, a Sequential model is good enough.

Every model trained in this project is a Keras Sequential model. Sequential models are appropriate for a plain stack of layers where each layer has exactly one input tensor and exactly one output tensor.

Sequential is a class that groups a linear stack of layers into a Keras model. It provides useful training and inference features on this model.

The *add()* method is used to add a layer instance on top of the layer stack.

When building a new Sequential architecture model, it is useful to incrementally stack layers with *add()* function and frequently print model summaries to monitor how a stack of, for example, Conv2D and MaxPooling2D layers is downsampling image feature maps.

3.7.1.2 Summarizing and Plotting a model

It is crucially important to understand the model, the input and output tensors of each layer and the parameters in each layer. Keras provides a simple method called *summary()* to print details about the Output Shapes and Parameters of each layer.

Additionally, the generation of the model summary using the *model_name.summary()* function also reports the number of total parameters, trainable parameters and non-trainable parameters. The word-embedding parameters previously trained constitute the non-trainable parameters.

I have also plotted the model using the *plot_model()* method in tensorflow's keras utils class. It is an effective way to understand the inputs and outputs of each layer in a flow chart format.

3.7.1.3 Compiling a model

For compiling a Keras model, the first 2 are absolutely necessary arguments:

1) **Optimizer:** The Adam Optimizer is used in all the models trained.

   It implements the Adam algorithm. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

   The only argument is the learning rate. By looking at the loss curve, we can identify if the learning rate is too high or too low. The default value is 0.001

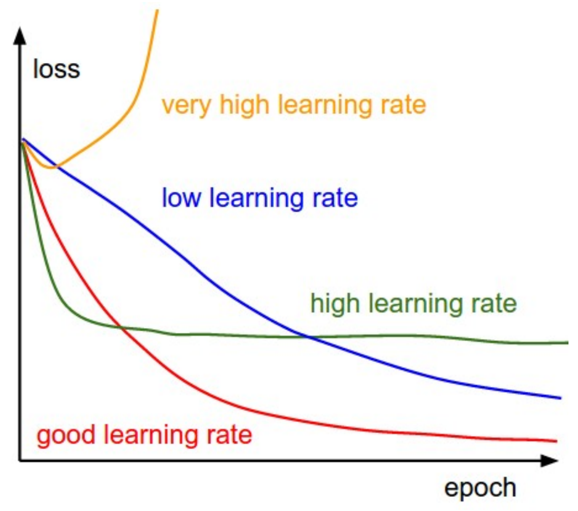


FIGURE 3.14 LEARNING RATE VARIATION WITH LOSS

2) **Loss:** The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

   The loss function used here is Binary cross Entropy (or Log Loss) as this is a Binary Classification Problem. Binary cross entropy compares each of the predicted probabilities to actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value. Binary Cross Entropy is basically the negative average of the log of corrected predicted probabilities. It essentially is a product of probabilities of an instance belonging to both the classes.

3) **Metrics:** A metric is a function that is used to judge the performance of your model.

   Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. The metric used here is accuracy.

3.7.1.4 Training a model and Hyperparameter tuning

Fitting the model requires 3 hyperparameters to be decided:

1) **batch_size**=1024
   The training data is divided into batches of the specified size and these batches are trained in every iteration. It is the number of training examples in one forward/backward pass. The batch size depends on the size of the data. Since this dataset is humungous with 1600000 tweets, the batch size should be fairly large. Batch size should be as much as the GPU RAM hardware being used can support. Since the

GPU I used is 16GB, I checked with a batch size of 2048 but the kernel stopped automatically as the GPU RAM reached its limit. So I decided to go with 1024.

2) **epochs=** set to 50 initially for all models
The number of epochs is the number of complete passes through the training dataset. The number of epochs can be set to an integer value between one and infinity. Initially, the number of epochs is set to 50 for all models so that the defined callbacks can come into play and stop the training at the appropriate epoch where saturation defined by either of the callbacks is reached.

3) **callbacks**=[ReduceLROnPlateau], [EarlyStopping] and [ModelCheckpoint]
A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc). Creating new callbacks is a simple and powerful way to customize a training loop. Callbacks can be used to:
    - Write TensorBoard logs after every batch of training to monitor your metrics
    - Periodically save your model to disk
    - Do early stopping
    - Get a view on internal states and statistics of a model during training

The ReduceLROnPlateau callback used here reduces learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

The EarlyStopping callback is set to stop the training if validation accuracy stops changing by 1e-4 for more than 5 epochs. Early stopping method to determine the number of epochs for a neural network. In this method we keep training the network for an arbitrary number of epochs and monitor the performance on a hold out validation dataset. When there is no sign of performance improvement on validation dataset, we stop training.

ModelCheckpoint saves the best model out of all the epochs which can be used later during model testing and evaluation for making predictions on testing data and finding accuracy score on testing set.

3.7.1.5 Plotting training results

According to Keras documentation, the *model_name.fit()* method returns a History callback, which has a history attribute containing the lists of successive losses and other metrics. It is used to plot the training and validation curves.

3.7.1.6 Evaluating the model

Finally, after training, the model is evaluated on the testing set and the testing accuracy (the evaluation metric) and loss are calculated using the *model_name.evaluate(X_test, y_test)* function.

3.7.2 Models

3.7.2.1 Model 1

The first model is the simplest possible LSTM model consisting of an embedding layer, a single LSTM layer and an output layer containing 1 neuron. The following is its architecture:

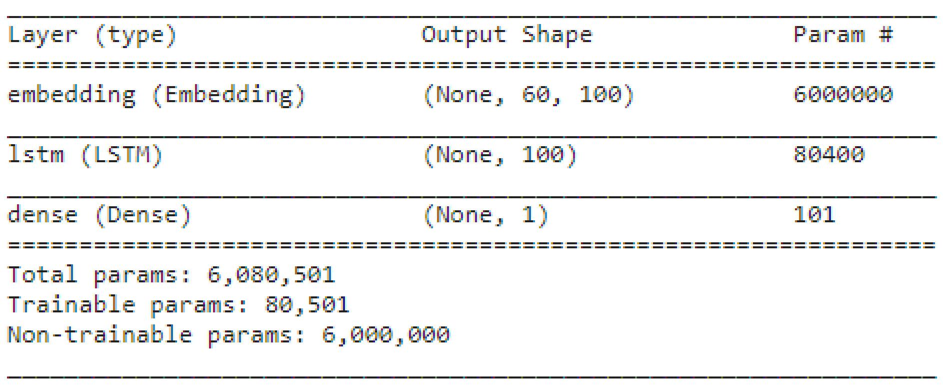
1) **Embedding Layer:** This is the layer responsible for converting the tokens into their vector representation that is generated by Word2Vec model. The predefined layer from Tensorflow is used in my model.
2) **Unidirectional LSTM Layer:** A variant of RNN which has a memory state cell to learn the context of words which are further along the text to carry contextual meaning rather than just neighbouring words as in case of RNN. Hyperparameters for this layer:
    - **recurrent units:** Positive integer, dimensionality of the output space. This was kept to be 100 because the timesteps/words in a sentence are limited to be 100.
    - **dropout:** Fraction of the units to drop for the linear transformation of the inputs. After first training, it is seen in the curves that there is some overfitting which is starting to emerge and hence 0.2 dropout is used in the 3rd training which means 20% of neurons of the LSTM layer are to be dropped before feeding the next layer.
3) **Dense Layer (Output Layer):** This layer has only 1 neuron. The activation function is experimented with both 'Sigmoid' and 'Softmax' functions. Performance is found to be significantly high for 'sigmoid' function because this is a binary classification problem. 'Softmax' activation gives a flat curve at 50%accu Sigmoid activations are generally used when we have 2 categories to output in. Since we have two classes here, namely Positive and Negative sentiments, sigmoid is clearly the better choice.

For this model, training has been done the following variations to decide best hyperparameters:

1. **Single unidirectional LSTM with 100 recurrent units as output, no dropout, and sigmoid activation function in the output layer containing 1 neuron**

2. **Single unidirectional LSTM with 100 recurrent units as output, no output, and softmax activation function in the output layer containing 1 neuron.**
3. **Single unidirectional LSTM with 100 recurrent units as output, dropout of 0.2 in LSTM layers and sigmoid activation function in output layer containing 1 neuron**

<u>Model 1 | Architecture</u>

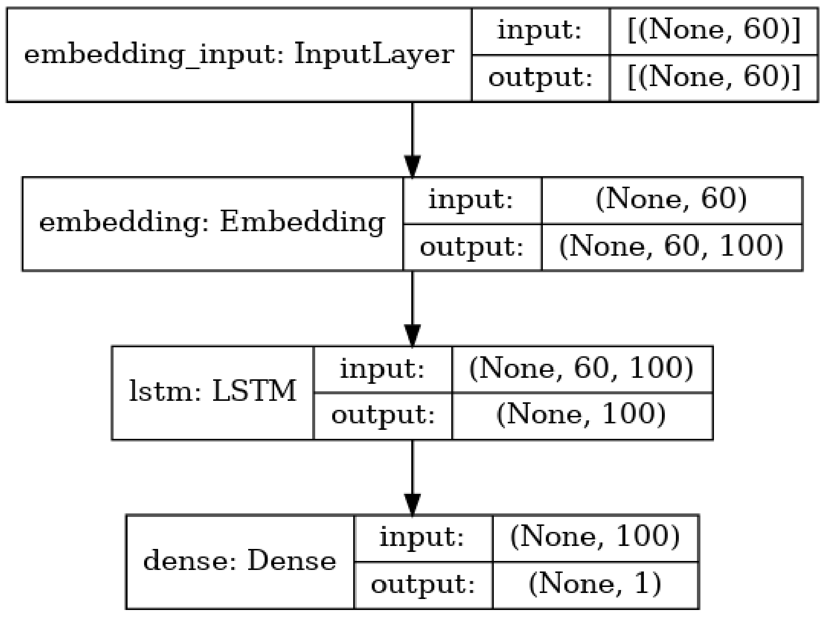


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000

_________________________________________________________________
lstm (LSTM)                  (None, 100)               80400

_________________________________________________________________
dense (Dense)                (None, 1)                 101
=================================================================
Total params: 6,080,501
Trainable params: 80,501
Non-trainable params: 6,000,000

_________________________________________________________________
```

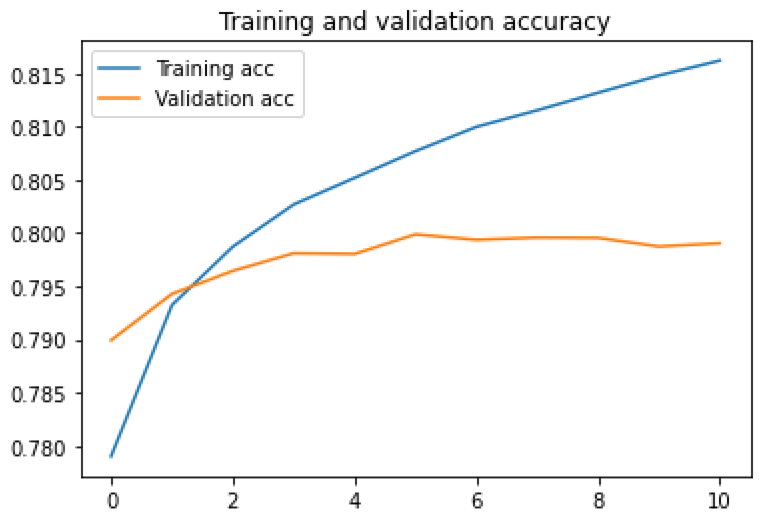


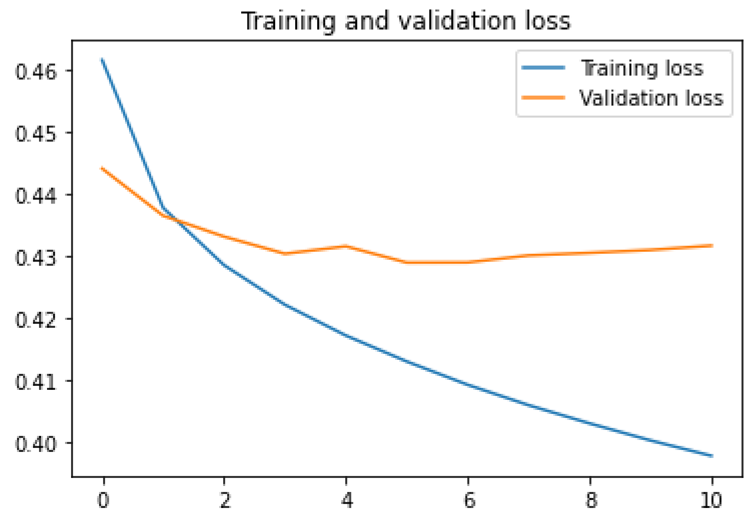
➢ **TRAINING 1 | SIGMOID ACTIVATION IN OUTPUT LAYER**

<u>Model 1 | Training 1 | Data</u>

```
Epoch 1/50   loss: 0.4615 - accuracy: 0.7790 - val_loss: 0.4440 - val_accuracy: 0.7899
Epoch 2/50   loss: 0.4377 - accuracy: 0.7932 - val_loss: 0.4364 - val_accuracy: 0.7943
Epoch 3/50   loss: 0.4285 - accuracy: 0.7987 - val_loss: 0.4331 - val_accuracy: 0.7964
Epoch 4/50   loss: 0.4221 - accuracy: 0.8027 - val_loss: 0.4303 - val_accuracy: 0.7981
Epoch 5/50   loss: 0.4172 - accuracy: 0.8052 - val_loss: 0.4315 - val_accuracy: 0.7980
Epoch 6/50   loss: 0.4130 - accuracy: 0.8077 - val_loss: 0.4289 - val_accuracy: 0.7999
Epoch 7/50   loss: 0.4092 - accuracy: 0.8100 - val_loss: 0.4289 - val_accuracy: 0.7994
Epoch 8/50   loss: 0.4059 - accuracy: 0.8115 - val_loss: 0.4300 - val_accuracy: 0.7996
Epoch 9/50   loss: 0.4030 - accuracy: 0.8132 - val_loss: 0.4305 - val_accuracy: 0.7995
Epoch 10/50  loss: 0.4003 - accuracy: 0.8148 - val_loss: 0.4309 - val_accuracy: 0.7987
Epoch 11/50  loss: 0.3978 - accuracy: 0.8162 - val_loss: 0.4316 - val_accuracy: 0.7990
```

Model 1 | Training 1 | Curves





Model 1| Training 1 | Evaluation

Output of model_1_1.evaluate(X_test, y_test) is:

```
loss: 0.4312 - accuracy: 0.7988

[0.4311657249927521, 0.7988250255584717]
```

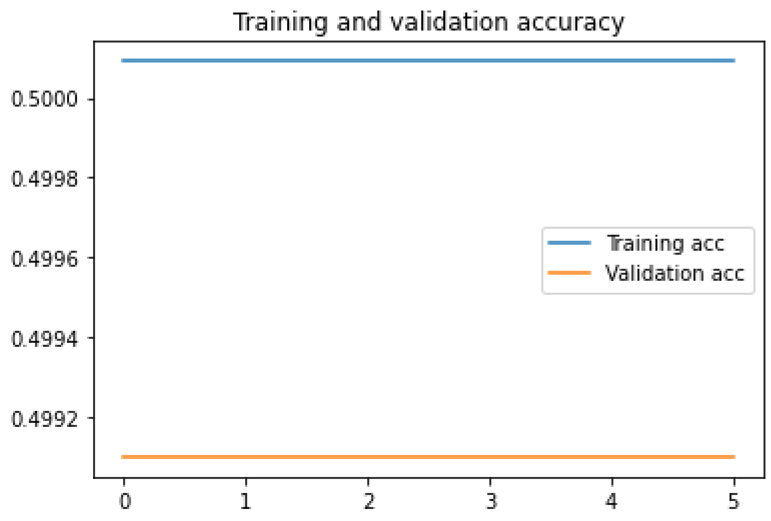
Testing accuracy= 79.88%

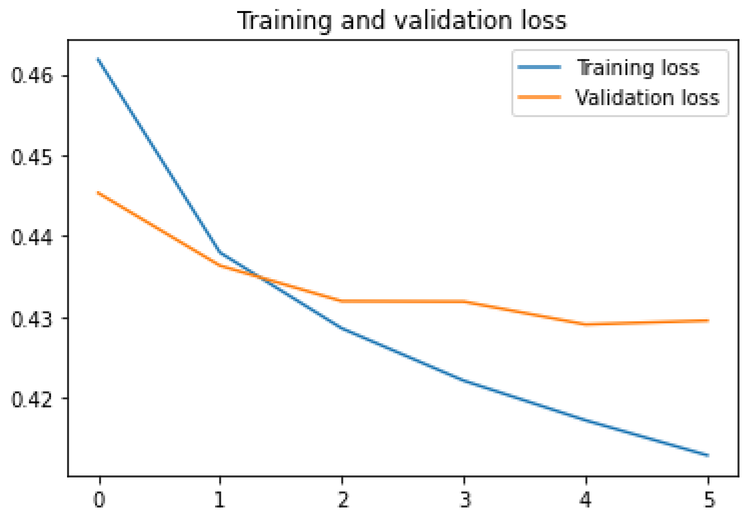
➢ TRAINING 2 | SOFTMAX ACTIVATION IN OUTPUT LAYER

Model 1 | Training 2 | Data

```
Epoch 1/50   loss: 0.4618 - accuracy: 0.5001 - val_loss: 0.4453 - val_accuracy: 0.4991
Epoch 2/50   loss: 0.4379 - accuracy: 0.5001 - val_loss: 0.4363 - val_accuracy: 0.4991
Epoch 3/50   loss: 0.4286 - accuracy: 0.5001 - val_loss: 0.4319 - val_accuracy: 0.4991
Epoch 4/50   loss: 0.4221 - accuracy: 0.5001 - val_loss: 0.4319 - val_accuracy: 0.4991
Epoch 5/50   loss: 0.4172 - accuracy: 0.5001 - val_loss: 0.4290 - val_accuracy: 0.4991
Epoch 6/50   loss: 0.4128 - accuracy: 0.5001 - val_loss: 0.4295 - val_accuracy: 0.4991
```

Model 1 | Training 2 | Curves





Model 1 | Training 2 | Evaluation

Output of model_1_2.evaluate(X_test, y_test) is:

```
loss: 0.4292 - accuracy: 0.5001
```

```
[0.42917442321777344, 0.5001375079154968]
```

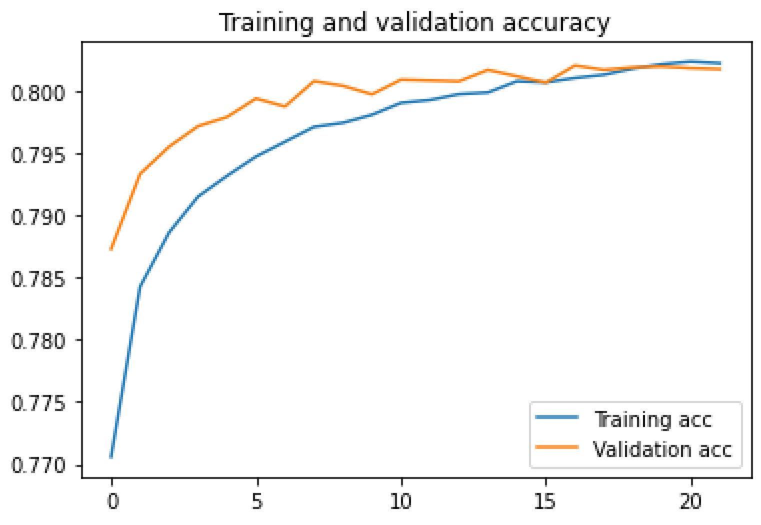
Testing accuracy= 50.01%

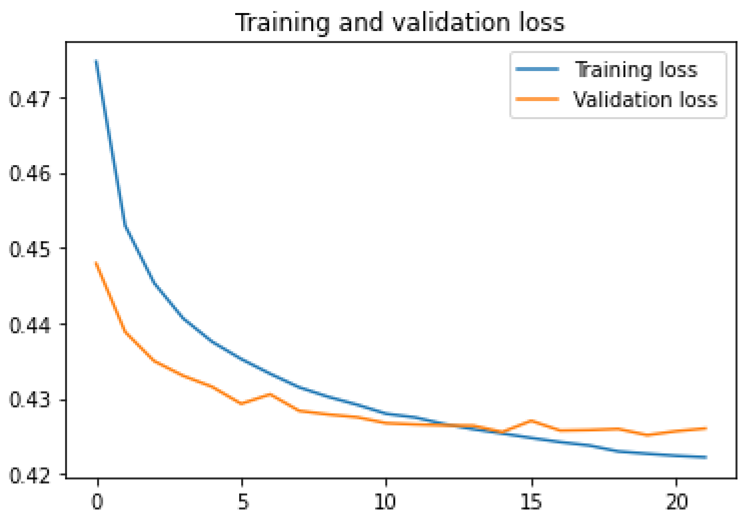
➢ TRAINING 3 | 20% DROPOUT IN LSTM LAYER WITH SIGMOID ACTIVATION IN OUTPUT LAYER

Model 1 | Training 3 | Data

```
Epoch 1/50   loss: 0.4747 - accuracy: 0.7706 - val_loss: 0.4479 - val_accuracy: 0.7873
Epoch 2/50   loss: 0.4529 - accuracy: 0.7843 - val_loss: 0.4388 - val_accuracy: 0.7933
Epoch 3/50   loss: 0.4453 - accuracy: 0.7886 - val_loss: 0.4350 - val_accuracy: 0.7955
Epoch 4/50   loss: 0.4406 - accuracy: 0.7915 - val_loss: 0.4330 - val_accuracy: 0.7972
Epoch 5/50   loss: 0.4375 - accuracy: 0.7932 - val_loss: 0.4316 - val_accuracy: 0.7979
Epoch 6/50   loss: 0.4352 - accuracy: 0.7947 - val_loss: 0.4293 - val_accuracy: 0.7994
Epoch 7/50   loss: 0.4333 - accuracy: 0.7959 - val_loss: 0.4306 - val_accuracy: 0.7987
Epoch 8/50   loss: 0.4315 - accuracy: 0.7971 - val_loss: 0.4284 - val_accuracy: 0.8008
Epoch 9/50   loss: 0.4302 - accuracy: 0.7974 - val_loss: 0.4279 - val_accuracy: 0.8004
Epoch 10/50  loss: 0.4292 - accuracy: 0.7981 - val_loss: 0.4275 - val_accuracy: 0.7997
Epoch 11/50  loss: 0.4280 - accuracy: 0.7990 - val_loss: 0.4267 - val_accuracy: 0.8009
Epoch 12/50  loss: 0.4275 - accuracy: 0.7993 - val_loss: 0.4266 - val_accuracy: 0.8008
Epoch 13/50  loss: 0.4266 - accuracy: 0.7997 - val_loss: 0.4264 - val_accuracy: 0.8008
Epoch 14/50  loss: 0.4259 - accuracy: 0.7999 - val_loss: 0.4264 - val_accuracy: 0.8017
Epoch 15/50  loss: 0.4254 - accuracy: 0.8008 - val_loss: 0.4256 - val_accuracy: 0.8012
Epoch 16/50  loss: 0.4248 - accuracy: 0.8007 - val_loss: 0.4271 - val_accuracy: 0.8007
Epoch 17/50  loss: 0.4242 - accuracy: 0.8010 - val_loss: 0.4258 - val_accuracy: 0.8020
Epoch 18/50  loss: 0.4238 - accuracy: 0.8013 - val_loss: 0.4258 - val_accuracy: 0.8017
Epoch 19/50  loss: 0.4230 - accuracy: 0.8018 - val_loss: 0.4260 - val_accuracy: 0.8019
Epoch 20/50  loss: 0.4227 - accuracy: 0.8021 - val_loss: 0.4252 - val_accuracy: 0.8020
Epoch 21/50  loss: 0.4224 - accuracy: 0.8024 - val_loss: 0.4257 - val_accuracy: 0.8018
Epoch 22/50  loss: 0.4222 - accuracy: 0.8022 - val_loss: 0.4260 - val_accuracy: 0.8018
```

Model 1 | Training 3 | Curves





<u>Model 1 | Training 3 | Evaluation</u>

Output of model_1_3.evaluate(X_test, y_test) is:

```
loss: 0.4278 - accuracy: 0.7999
```

```
[0.427838534116745, 0.799875020980835]
```

Testing accuracy= 79.99%

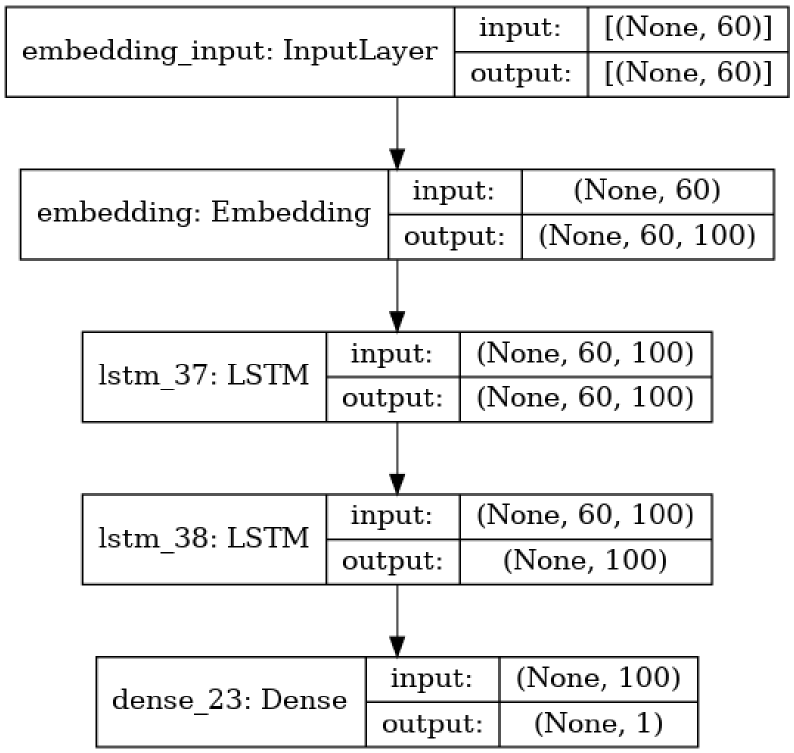
3.7.2.2 Model 2

In the second model, an additional unidirectional LSTM layer is introduced to make a Stacked Unidirectional LSTM Model. The following are the layers:

1) **Embedding Layer**
2) **Unidirectional LSTM Layer 1**
    - **recurrent units=** 100
    - **dropout=** 20%
    - **return_sequence:** Whether to return the last output in the output sequence, or the full sequence. This should be "True" for this layer as the output of this layer has to be feeded as input to the next LSTM layer.
3) **Unidirectional LSTM Layer 2:** Same parameter values as **Unidirectional LSTM Layer 1** except that **return_sequence** should be "False" for this layer as this is the last hidden layer. Since its default value is "False", it may not be specified.
4) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 2 | Architecture

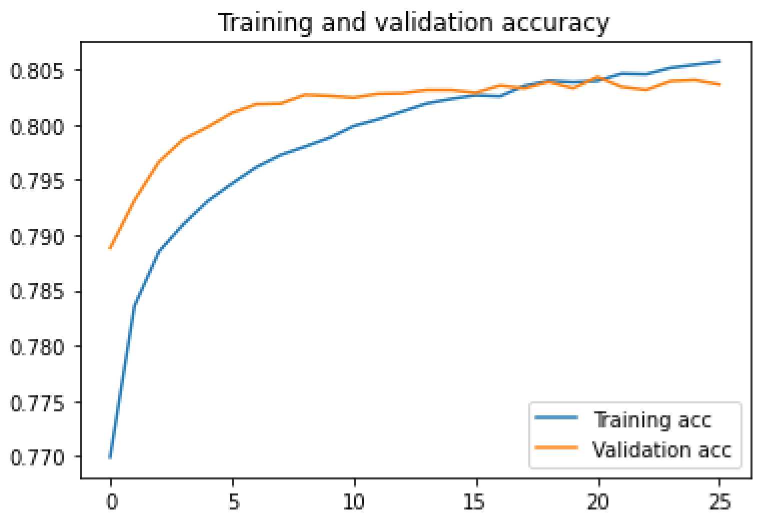
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000
_________________________________________________________________
lstm_37 (LSTM)               (None, 60, 100)           80400
_________________________________________________________________
lstm_38 (LSTM)               (None, 100)               80400
_________________________________________________________________
dense_23 (Dense)             (None, 1)                 101
=================================================================
Total params: 6,160,901
Trainable params: 160,901
Non-trainable params: 6,000,000
_________________________________________________________________
```

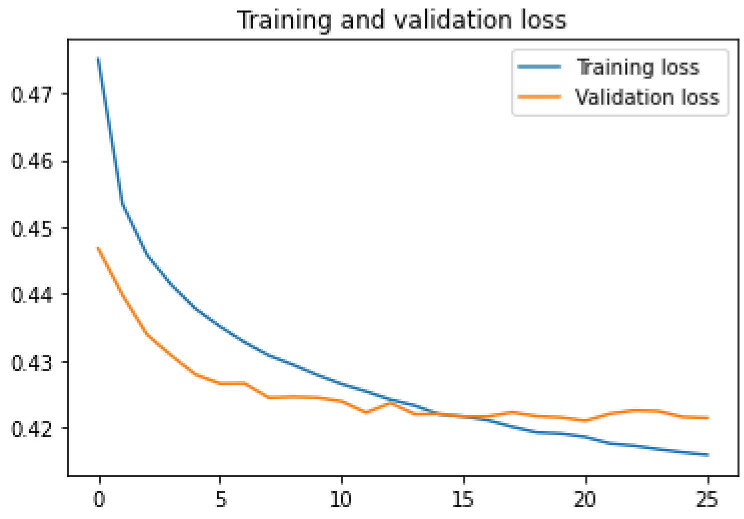


Model 2 | Training Data

```
Epoch 1/50   loss: 0.4751 - accuracy: 0.7699 - val_loss: 0.4467 - val_accuracy: 0.7888
Epoch 2/50   loss: 0.4534 - accuracy: 0.7836 - val_loss: 0.4398 - val_accuracy: 0.7931
Epoch 3/50   loss: 0.4459 - accuracy: 0.7885 - val_loss: 0.4339 - val_accuracy: 0.7966
Epoch 4/50   loss: 0.4414 - accuracy: 0.7909 - val_loss: 0.4307 - val_accuracy: 0.7986
Epoch 5/50   loss: 0.4377 - accuracy: 0.7931 - val_loss: 0.4279 - val_accuracy: 0.7998
Epoch 6/50   loss: 0.4351 - accuracy: 0.7946 - val_loss: 0.4265 - val_accuracy: 0.8010
Epoch 7/50   loss: 0.4328 - accuracy: 0.7961 - val_loss: 0.4266 - val_accuracy: 0.8018
Epoch 8/50   loss: 0.4308 - accuracy: 0.7972 - val_loss: 0.4244 - val_accuracy: 0.8019
Epoch 9/50   loss: 0.4294 - accuracy: 0.7980 - val_loss: 0.4245 - val_accuracy: 0.8027
Epoch 10/50  loss: 0.4278 - accuracy: 0.7988 - val_loss: 0.4244 - val_accuracy: 0.8026
Epoch 11/50  loss: 0.4264 - accuracy: 0.7999 - val_loss: 0.4239 - val_accuracy: 0.8024
Epoch 12/50  loss: 0.4254 - accuracy: 0.8005 - val_loss: 0.4222 - val_accuracy: 0.8028
Epoch 13/50  loss: 0.4241 - accuracy: 0.8012 - val_loss: 0.4236 - val_accuracy: 0.8028
Epoch 14/50  loss: 0.4232 - accuracy: 0.8019 - val_loss: 0.4219 - val_accuracy: 0.8031
Epoch 15/50  loss: 0.4219 - accuracy: 0.8023 - val_loss: 0.4220 - val_accuracy: 0.8031
Epoch 16/50  loss: 0.4216 - accuracy: 0.8026 - val_loss: 0.4216 - val_accuracy: 0.8029
Epoch 17/50  loss: 0.4210 - accuracy: 0.8025 - val_loss: 0.4216 - val_accuracy: 0.8035
Epoch 18/50  loss: 0.4200 - accuracy: 0.8035 - val_loss: 0.4222 - val_accuracy: 0.8033
Epoch 19/50  loss: 0.4192 - accuracy: 0.8040 - val_loss: 0.4217 - val_accuracy: 0.8039
Epoch 20/50  loss: 0.4190 - accuracy: 0.8038 - val_loss: 0.4215 - val_accuracy: 0.8033
Epoch 21/50  loss: 0.4185 - accuracy: 0.8039 - val_loss: 0.4210 - val_accuracy: 0.8043
Epoch 22/50  loss: 0.4175 - accuracy: 0.8046 - val_loss: 0.4220 - val_accuracy: 0.8034
Epoch 23/50  loss: 0.4172 - accuracy: 0.8045 - val_loss: 0.4225 - val_accuracy: 0.8031
Epoch 24/50  loss: 0.4167 - accuracy: 0.8051 - val_loss: 0.4224 - val_accuracy: 0.8039
Epoch 25/50  loss: 0.4162 - accuracy: 0.8054 - val_loss: 0.4215 - val_accuracy: 0.8040
Epoch 26/50  loss: 0.4158 - accuracy: 0.8057 - val_loss: 0.4214 - val_accuracy: 0.8036
```

Model 2 | Curves





Model 2 | Evaluation

Output of model_2.evaluate(X_test, y_test) :

```
loss: 0.4231 - accuracy: 0.8036

[0.4231144189834595, 0.803600013256073]
```

Testing accuracy= 80.36%

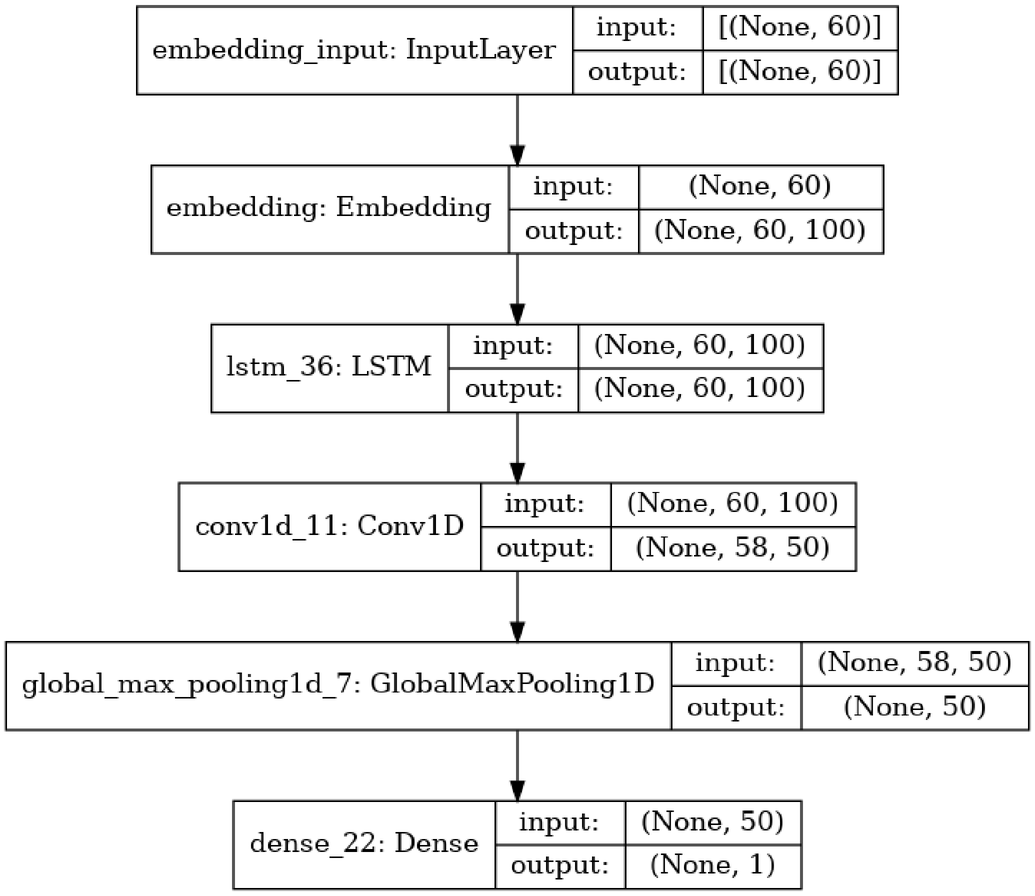
3.7.2.3 Model 3

In the third model, a CNN layer is added to the simple unidirectional LSTM layer first model to create a CNN-LSTM Model. Global max pooling is used to reduce tensor to a single layer. The following are the layers:

5) **Embedding Layer**
6) **Unidirectional LSTM Layer**
    - **recurrent units=** 100
    - **dropout=** 20%
    - **return_sequence:** Whether to return the last output in the output sequence, or the full sequence. This should be "True" for this layer as the output of this layer has to be feeded as input to the next LSTM layer.
7) **CNN Layer**
    - **filters:** Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution). It is set as 50 for this model as the input to this layer coming from the LSTM layer has size 100.
    - **kernel_size:** An integer or tuple/list of a single integer, specifying the length of the 1D convolution window. It is set to 3.
    - **activation:** Activation function to use. If you don't specify anything, no activation is applied. The best results are given by 'relu' activation here so it has been set as 'relu' function.
8) **GlobalMaxPooling 1D:** Global max pooling operation for 1D temporal data. Downsamples the input representation by taking the maximum value over the time dimension.
9) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 3 | Architecture

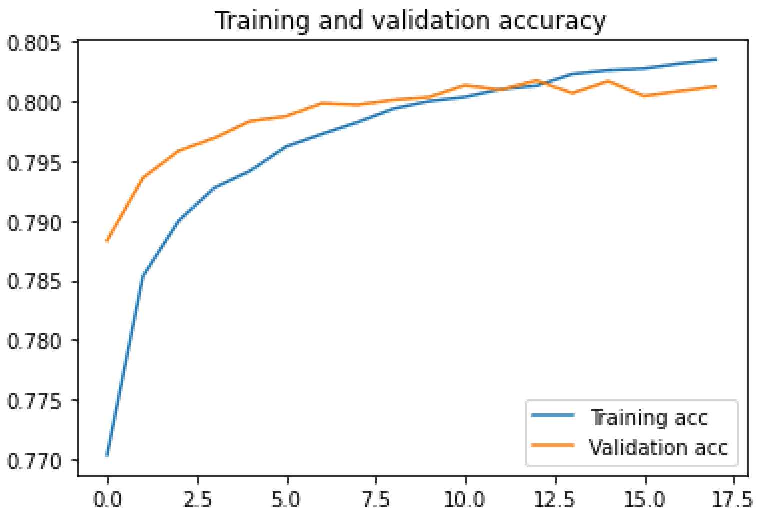
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000   
_________________________________________________________________
lstm_36 (LSTM)               (None, 60, 100)           80400     
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 58, 50)            15050     
_________________________________________________________________
global_max_pooling1d_7 (Glob (None, 50)                0         
_________________________________________________________________
dense_22 (Dense)             (None, 1)                 51        
=================================================================
Total params: 6,095,501
Trainable params: 95,501
Non-trainable params: 6,000,000
_________________________________________________________________
```

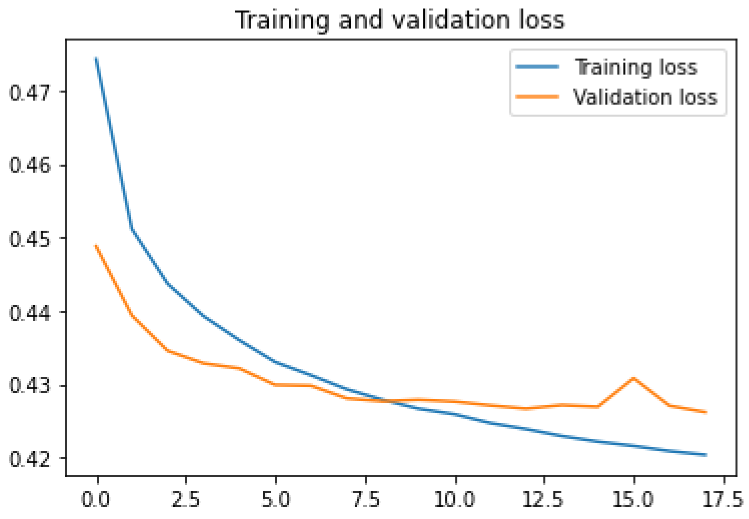


Model 3 | Training Data

```
Epoch 1/50   loss: 0.4742 - accuracy: 0.7704 - val_loss: 0.4488 - val_accuracy: 0.7883
Epoch 2/50   loss: 0.4511 - accuracy: 0.7853 - val_loss: 0.4394 - val_accuracy: 0.7936
Epoch 3/50   loss: 0.4437 - accuracy: 0.7900 - val_loss: 0.4346 - val_accuracy: 0.7958
Epoch 4/50   loss: 0.4393 - accuracy: 0.7927 - val_loss: 0.4329 - val_accuracy: 0.7969
Epoch 5/50   loss: 0.4360 - accuracy: 0.7942 - val_loss: 0.4322 - val_accuracy: 0.7983
Epoch 6/50   loss: 0.4330 - accuracy: 0.7962 - val_loss: 0.4299 - val_accuracy: 0.7987
Epoch 7/50   loss: 0.4312 - accuracy: 0.7972 - val_loss: 0.4298 - val_accuracy: 0.7998
Epoch 8/50   loss: 0.4293 - accuracy: 0.7982 - val_loss: 0.4281 - val_accuracy: 0.7997
Epoch 9/50   loss: 0.4279 - accuracy: 0.7994 - val_loss: 0.4277 - val_accuracy: 0.8001
Epoch 10/50  loss: 0.4267 - accuracy: 0.8000 - val_loss: 0.4279 - val_accuracy: 0.8003
Epoch 11/50  loss: 0.4259 - accuracy: 0.8004 - val_loss: 0.4276 - val_accuracy: 0.8013
Epoch 12/50  loss: 0.4247 - accuracy: 0.8010 - val_loss: 0.4271 - val_accuracy: 0.8010
Epoch 13/50  loss: 0.4239 - accuracy: 0.8013 - val_loss: 0.4267 - val_accuracy: 0.8018
Epoch 14/50  loss: 0.4229 - accuracy: 0.8023 - val_loss: 0.4272 - val_accuracy: 0.8007
Epoch 15/50  loss: 0.4222 - accuracy: 0.8026 - val_loss: 0.4269 - val_accuracy: 0.8017
Epoch 16/50  loss: 0.4216 - accuracy: 0.8027 - val_loss: 0.4308 - val_accuracy: 0.8004
Epoch 17/50  loss: 0.4209 - accuracy: 0.8031 - val_loss: 0.4271 - val_accuracy: 0.8008
Epoch 18/50  loss: 0.4204 - accuracy: 0.8035 - val_loss: 0.4262 - val_accuracy: 0.8012
```

<u>Model 3 | Curves</u>





Model 3 | Evaluation

Output of model_3.evaluate(X_test,y_test)

```
loss: 0.4260 - accuracy: 0.8019

[0.4260101616382599, 0.8018749952316284]
```

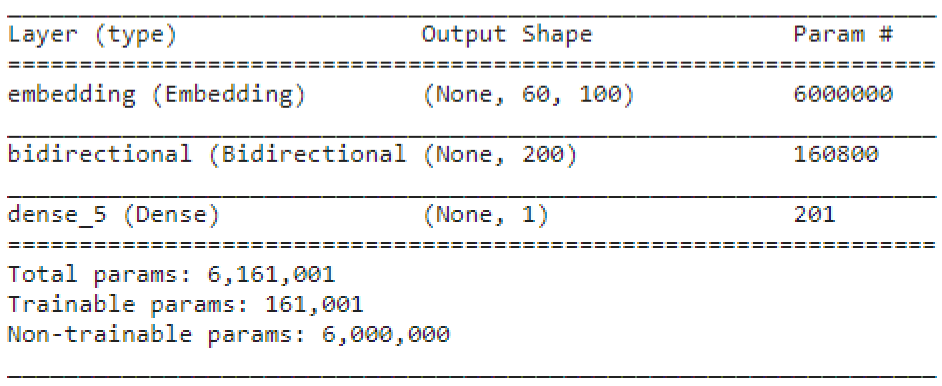
Testing accuracy=80.19%

3.7.2.4 Model 4

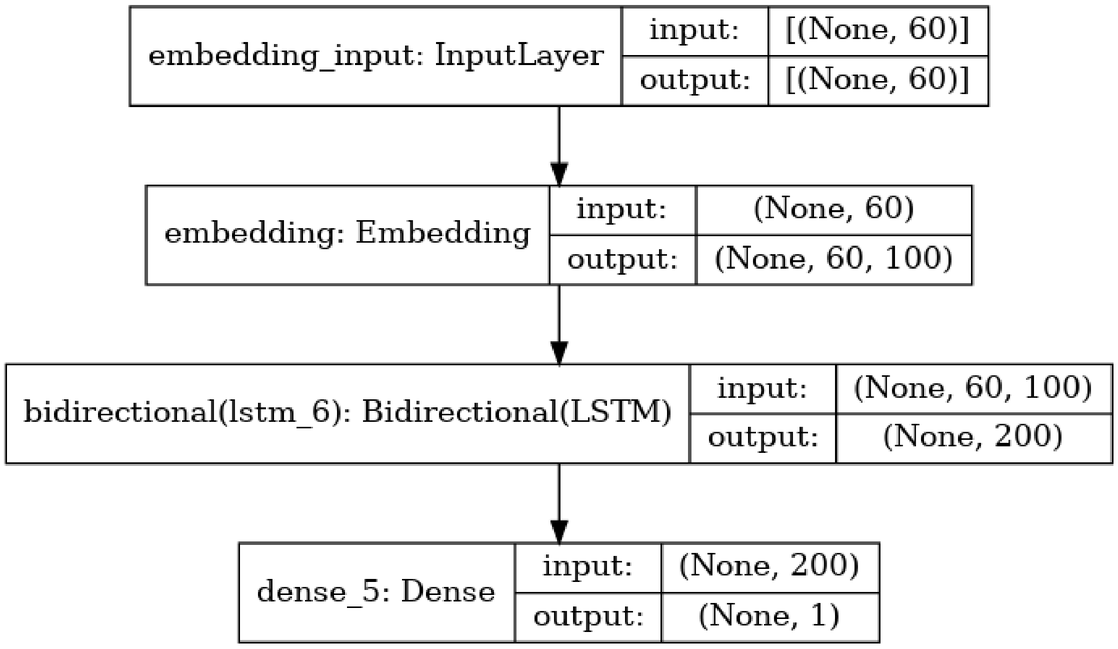
In the fourth model, the simplest bidirectional LSTM model possible is trained. The layer details are as follows:

10) **Embedding Layer**
11) **Bidirectional LSTM Layer:** This is a simple bidirectional LSTM layer wherein the unidirectional LSTM code is used inside the *Bidirectional()* method with the same hyperparameters as before:
    - **recurrent units=** 100
    - **dropout=** 20%
12) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 4 | Architecture



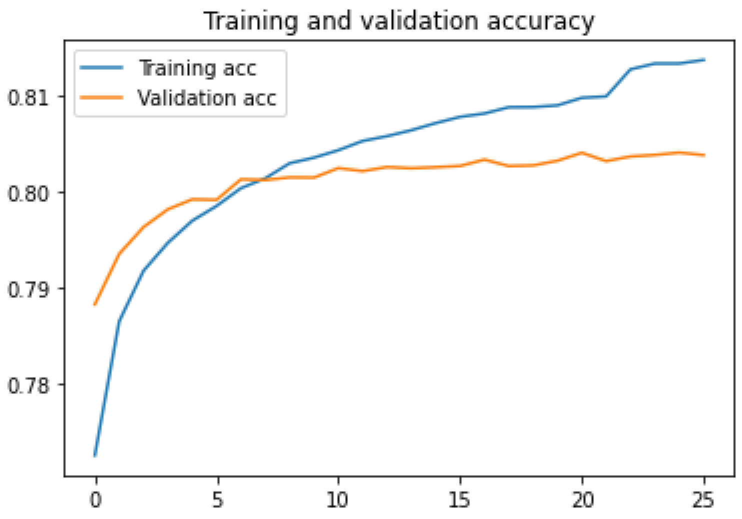
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000
_________________________________________________________________
bidirectional (Bidirectional (None, 200)               160800
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 201
=================================================================
Total params: 6,161,001
Trainable params: 161,001
Non-trainable params: 6,000,000
_________________________________________________________________
```

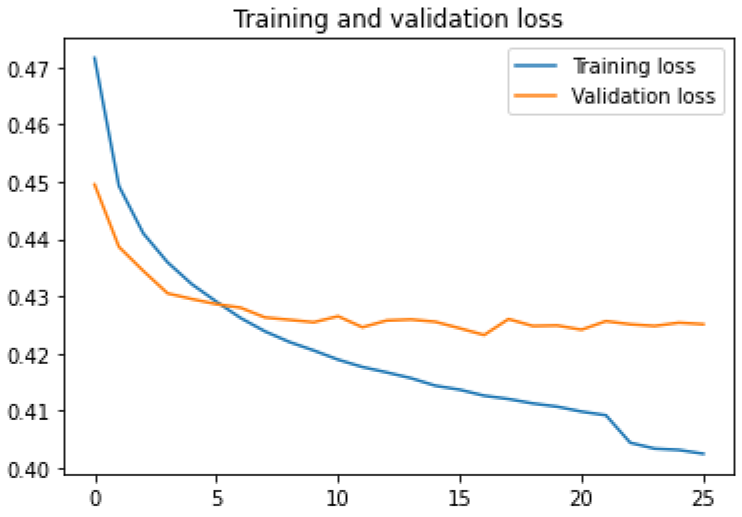


Model 4 | Training Data

```
Epoch 1/50   loss: 0.4716 - accuracy: 0.7726 - val_loss: 0.4495 - val_accuracy: 0.7883
Epoch 2/50   loss: 0.4492 - accuracy: 0.7865 - val_loss: 0.4386 - val_accuracy: 0.7935
Epoch 3/50   loss: 0.4409 - accuracy: 0.7917 - val_loss: 0.4344 - val_accuracy: 0.7963
Epoch 4/50   loss: 0.4359 - accuracy: 0.7947 - val_loss: 0.4305 - val_accuracy: 0.7981
Epoch 5/50   loss: 0.4320 - accuracy: 0.7969 - val_loss: 0.4295 - val_accuracy: 0.7992
Epoch 6/50   loss: 0.4290 - accuracy: 0.7985 - val_loss: 0.4286 - val_accuracy: 0.7991
Epoch 7/50   loss: 0.4261 - accuracy: 0.8003 - val_loss: 0.4279 - val_accuracy: 0.8012
Epoch 8/50   loss: 0.4238 - accuracy: 0.8013 - val_loss: 0.4262 - val_accuracy: 0.8012
Epoch 9/50   loss: 0.4220 - accuracy: 0.8029 - val_loss: 0.4258 - val_accuracy: 0.8014
Epoch 10/50  loss: 0.4205 - accuracy: 0.8035 - val_loss: 0.4254 - val_accuracy: 0.8014
Epoch 11/50  loss: 0.4189 - accuracy: 0.8043 - val_loss: 0.4265 - val_accuracy: 0.8024
Epoch 12/50  loss: 0.4176 - accuracy: 0.8052 - val_loss: 0.4245 - val_accuracy: 0.8021
Epoch 13/50  loss: 0.4166 - accuracy: 0.8057 - val_loss: 0.4257 - val_accuracy: 0.8025
Epoch 14/50  loss: 0.4156 - accuracy: 0.8063 - val_loss: 0.4259 - val_accuracy: 0.8024
Epoch 15/50  loss: 0.4143 - accuracy: 0.8071 - val_loss: 0.4255 - val_accuracy: 0.8025
Epoch 16/50  loss: 0.4136 - accuracy: 0.8077 - val_loss: 0.4244 - val_accuracy: 0.8026
Epoch 17/50  loss: 0.4126 - accuracy: 0.8081 - val_loss: 0.4232 - val_accuracy: 0.8033
Epoch 18/50  loss: 0.4120 - accuracy: 0.8087 - val_loss: 0.4260 - val_accuracy: 0.8026
Epoch 19/50  loss: 0.4112 - accuracy: 0.8087 - val_loss: 0.4248 - val_accuracy: 0.8027
Epoch 20/50  loss: 0.4107 - accuracy: 0.8089 - val_loss: 0.4248 - val_accuracy: 0.8032
Epoch 21/50  loss: 0.4098 - accuracy: 0.8097 - val_loss: 0.4241 - val_accuracy: 0.8040
Epoch 22/50  loss: 0.4091 - accuracy: 0.8099 - val_loss: 0.4256 - val_accuracy: 0.8031
Epoch 23/50  loss: 0.4043 - accuracy: 0.8127 - val_loss: 0.4251 - val_accuracy: 0.8036
Epoch 24/50  loss: 0.4033 - accuracy: 0.8133 - val_loss: 0.4248 - val_accuracy: 0.8038
Epoch 25/50  loss: 0.4031 - accuracy: 0.8133 - val_loss: 0.4253 - val_accuracy: 0.8040
Epoch 26/50  loss: 0.4024 - accuracy: 0.8136 - val_loss: 0.4251 - val_accuracy: 0.8038
```

Model 4 | Curves





Model 4 | Evaluation

Output of model_4.evaluate(X_test, y_test) :

```
loss: 0.4243 - accuracy: 0.8035

[0.42425552010536194, 0.8034999966621399]
```

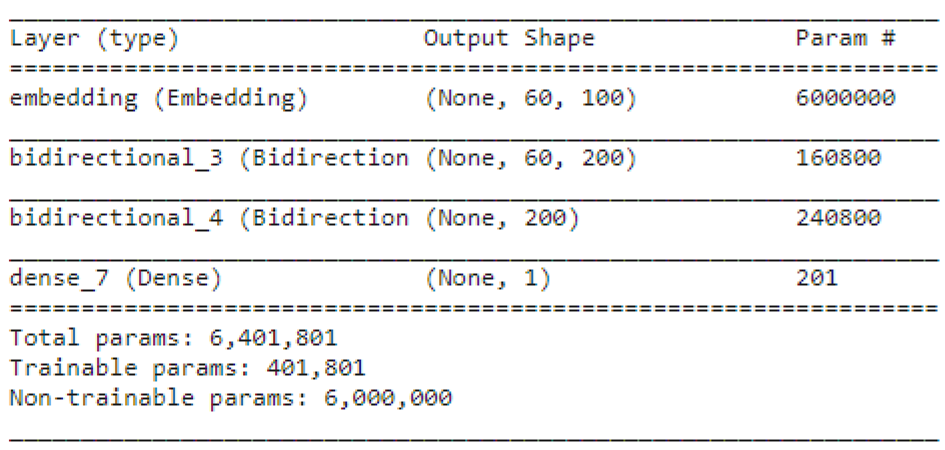
Testing accuracy= 80.35%

3.7.2.5 Model 5

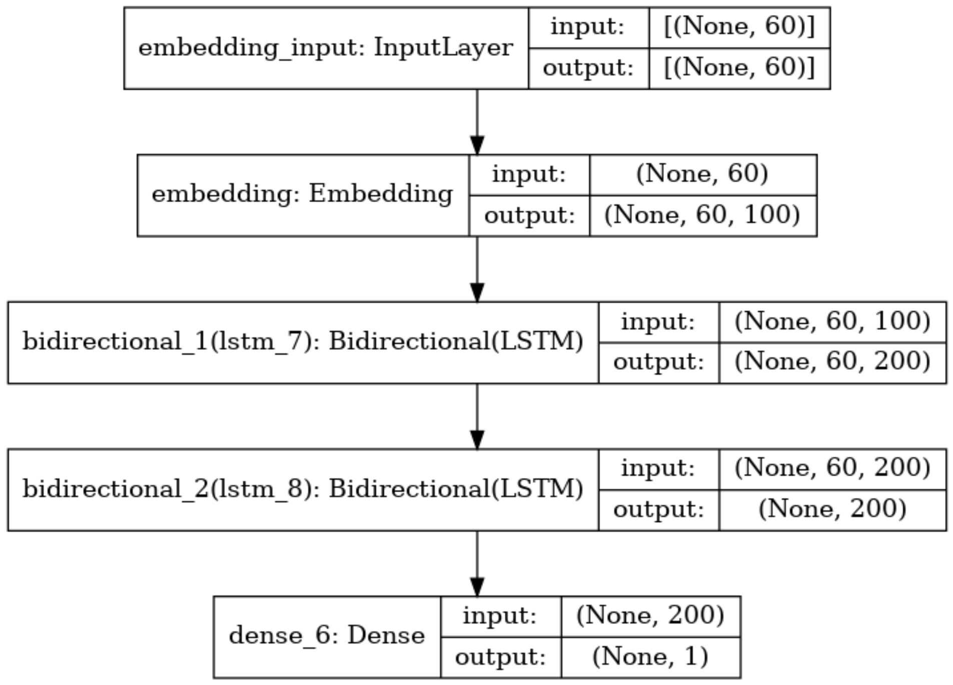
In the fifth model, a doubly stacked bidirectional LSTM model is trained. The layer details are as follows:

1) **Embedding Layer**
2) **Bidirectional LSTM Layer 1:** This is a simple bidirectional LSTM layer wherein the unidirectional LSTM code is used inside the *Bidirectional()* method with the same hyperparameters as before:
    - **recurrent units**= 100
    - **dropout**= 20%
3) **Bidirectional LSTM Layer 2:** Same parameters as previous bidirectional layer.
4) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 5 | Architecture



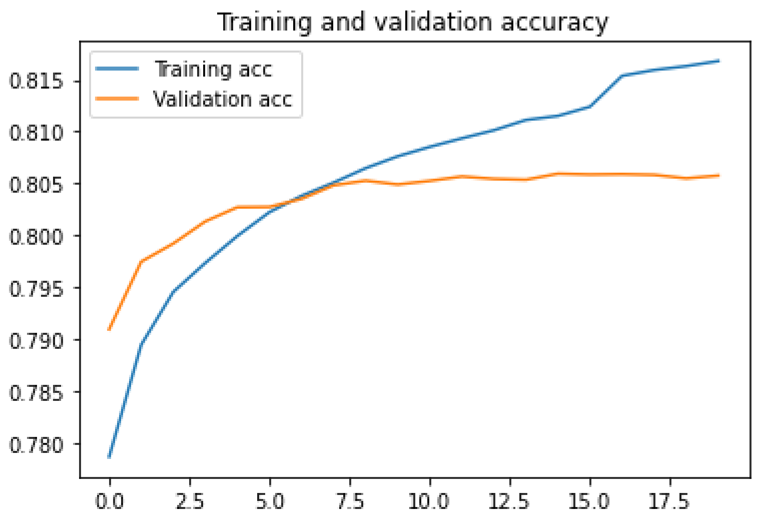
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000
_________________________________________________________________
bidirectional_3 (Bidirection (None, 60, 200)           160800
_________________________________________________________________
bidirectional_4 (Bidirection (None, 200)               240800
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 201
=================================================================
Total params: 6,401,801
Trainable params: 401,801
Non-trainable params: 6,000,000
_________________________________________________________________
```

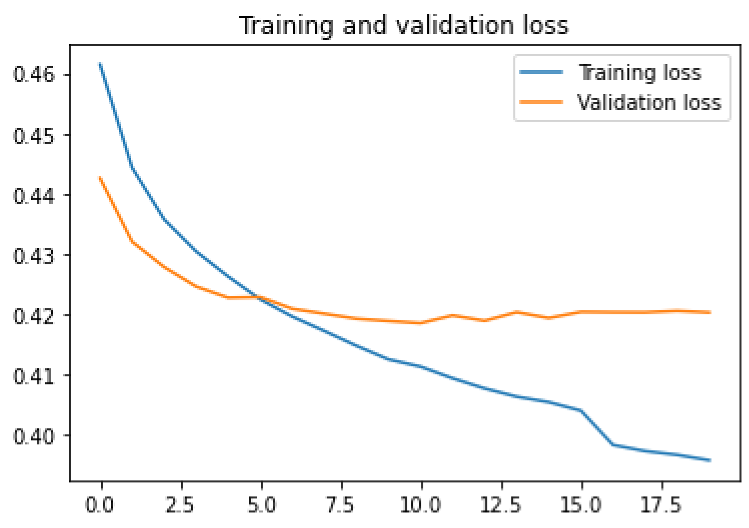


Model 5 | Training Data

```
Epoch 1/50   loss: 0.4615 - accuracy: 0.7787 - val_loss: 0.4425 - val_accuracy: 0.7910
Epoch 2/50   loss: 0.4443 - accuracy: 0.7895 - val_loss: 0.4320 - val_accuracy: 0.7975
Epoch 3/50   loss: 0.4357 - accuracy: 0.7946 - val_loss: 0.4278 - val_accuracy: 0.7992
Epoch 4/50   loss: 0.4304 - accuracy: 0.7974 - val_loss: 0.4246 - val_accuracy: 0.8014
Epoch 5/50   loss: 0.4262 - accuracy: 0.8000 - val_loss: 0.4227 - val_accuracy: 0.8027
Epoch 6/50   loss: 0.4224 - accuracy: 0.8023 - val_loss: 0.4228 - val_accuracy: 0.8028
Epoch 7/50   loss: 0.4196 - accuracy: 0.8038 - val_loss: 0.4209 - val_accuracy: 0.8035
Epoch 8/50   loss: 0.4172 - accuracy: 0.8051 - val_loss: 0.4201 - val_accuracy: 0.8048
Epoch 9/50   loss: 0.4147 - accuracy: 0.8065 - val_loss: 0.4192 - val_accuracy: 0.8053
Epoch 10/50  loss: 0.4125 - accuracy: 0.8076 - val_loss: 0.4188 - val_accuracy: 0.8049
Epoch 11/50  loss: 0.4113 - accuracy: 0.8085 - val_loss: 0.4185 - val_accuracy: 0.8053
Epoch 12/50  loss: 0.4094 - accuracy: 0.8094 - val_loss: 0.4198 - val_accuracy: 0.8057
Epoch 13/50  loss: 0.4077 - accuracy: 0.8101 - val_loss: 0.4189 - val_accuracy: 0.8055
Epoch 14/50  loss: 0.4063 - accuracy: 0.8111 - val_loss: 0.4203 - val_accuracy: 0.8054
Epoch 15/50  loss: 0.4054 - accuracy: 0.8115 - val_loss: 0.4193 - val_accuracy: 0.8060
Epoch 16/50  loss: 0.4040 - accuracy: 0.8124 - val_loss: 0.4204 - val_accuracy: 0.8059
Epoch 17/50  loss: 0.3983 - accuracy: 0.8154 - val_loss: 0.4203 - val_accuracy: 0.8059
Epoch 18/50  loss: 0.3973 - accuracy: 0.8159 - val_loss: 0.4203 - val_accuracy: 0.8058
Epoch 19/50  loss: 0.3967 - accuracy: 0.8163 - val_loss: 0.4205 - val_accuracy: 0.8055
Epoch 20/50  loss: 0.3958 - accuracy: 0.8168 - val_loss: 0.4203 - val_accuracy: 0.8058
```

Model 5 | Curves





<u>Model 5 | Evaluation</u>

Output of model_5.evaluate(X_test, y_test) :

```
loss: 0.4209 - accuracy: 0.8061
```

```
[0.4209284782409668, 0.80611252784729]
```

Testing accuracy= 80.61%

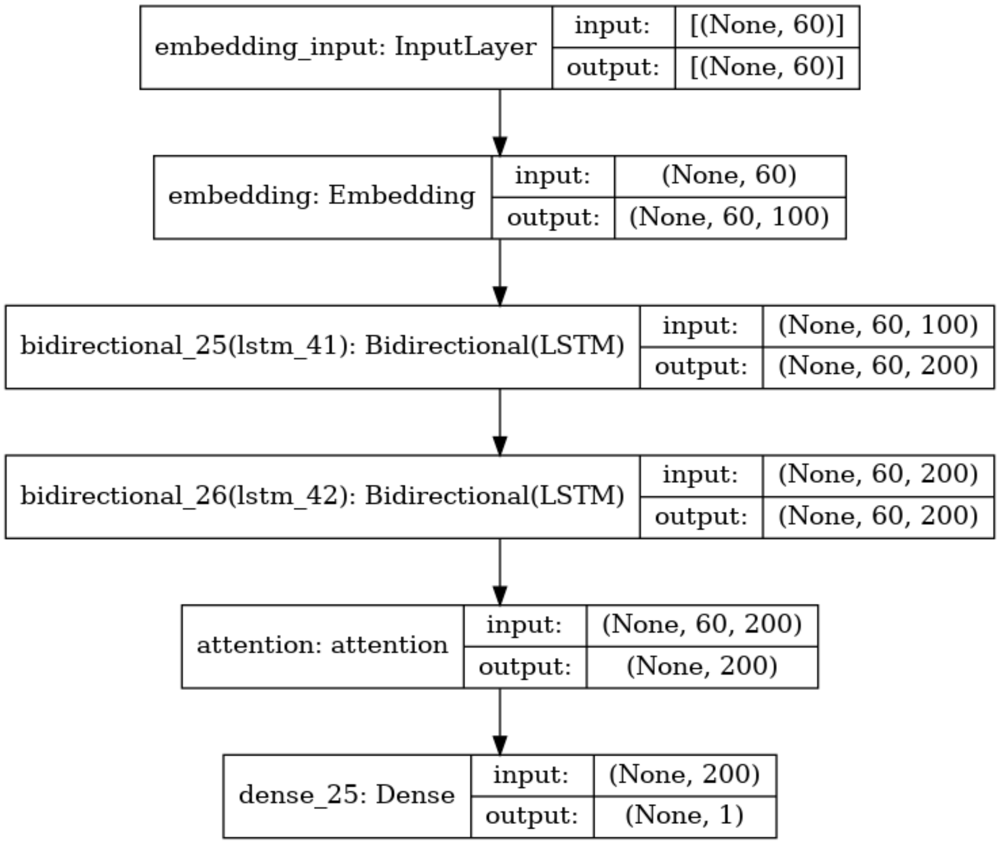
3.7.2.6 Model 6

In the 6th model, a customized attention mechanism is introduced to the bidirectional doubly stacked model. The details are as follows:

1) **Embedding Layer**
2) **Bidirectional LSTM Layer 1:** This is a simple bidirectional LSTM layer wherein the unidirectional LSTM code is used inside the *Bidirectional()* method with the same hyperparameters as before:
    - **recurrent units=** 100
    - **dropout=** 20%
3) **Bidirectional LSTM Layer 2:** Same parameters as previous bidirectional layer.
4) **Attention Layer**
5) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 6 | Architecture

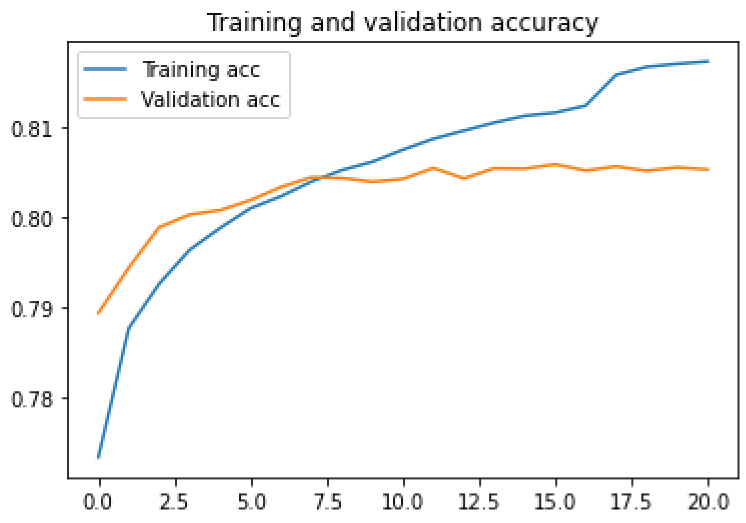
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           6000000
_________________________________________________________________
bidirectional_25 (Bidirectio (None, 60, 200)           160800
_________________________________________________________________
bidirectional_26 (Bidirectio (None, 60, 200)           240800
_________________________________________________________________
attention (attention)        (None, 200)               260
_________________________________________________________________
dense_25 (Dense)             (None, 1)                 201
=================================================================
Total params: 6,402,061
Trainable params: 402,061
Non-trainable params: 6,000,000
_________________________________________________________________
```

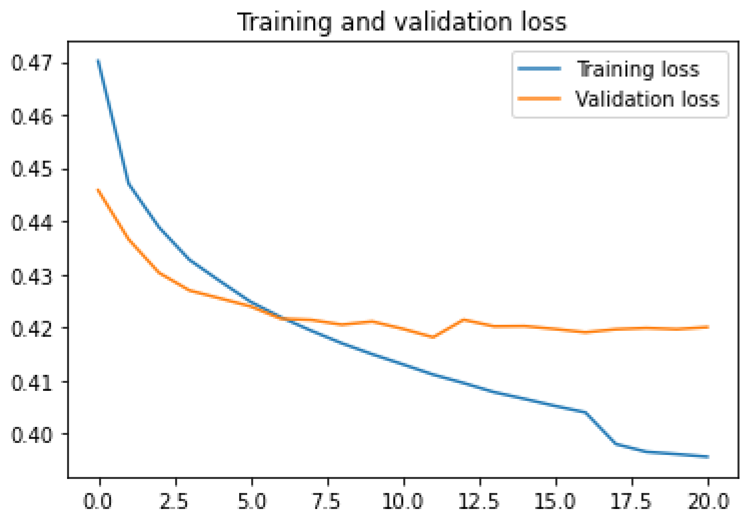


Model 6 | Training Data

```
Epoch 1/50   loss: 0.4701 - accuracy: 0.7735 - val_loss: 0.4458 - val_accuracy: 0.7894
Epoch 2/50   loss: 0.4470 - accuracy: 0.7878 - val_loss: 0.4366 - val_accuracy: 0.7945
Epoch 3/50   loss: 0.4388 - accuracy: 0.7927 - val_loss: 0.4303 - val_accuracy: 0.7989
Epoch 4/50   loss: 0.4327 - accuracy: 0.7964 - val_loss: 0.4269 - val_accuracy: 0.8003
Epoch 5/50   loss: 0.4287 - accuracy: 0.7988 - val_loss: 0.4255 - val_accuracy: 0.8008
Epoch 6/50   loss: 0.4248 - accuracy: 0.8010 - val_loss: 0.4240 - val_accuracy: 0.8019
Epoch 7/50   loss: 0.4219 - accuracy: 0.8024 - val_loss: 0.4216 - val_accuracy: 0.8034
Epoch 8/50   loss: 0.4194 - accuracy: 0.8040 - val_loss: 0.4214 - val_accuracy: 0.8045
Epoch 9/50   loss: 0.4170 - accuracy: 0.8053 - val_loss: 0.4205 - val_accuracy: 0.8044
Epoch 10/50  loss: 0.4149 - accuracy: 0.8062 - val_loss: 0.4211 - val_accuracy: 0.8040
Epoch 11/50  loss: 0.4131 - accuracy: 0.8075 - val_loss: 0.4197 - val_accuracy: 0.8043
Epoch 12/50  loss: 0.4111 - accuracy: 0.8087 - val_loss: 0.4181 - val_accuracy: 0.8055
Epoch 13/50  loss: 0.4095 - accuracy: 0.8096 - val_loss: 0.4214 - val_accuracy: 0.8043
Epoch 14/50  loss: 0.4078 - accuracy: 0.8105 - val_loss: 0.4202 - val_accuracy: 0.8055
Epoch 15/50  loss: 0.4065 - accuracy: 0.8113 - val_loss: 0.4202 - val_accuracy: 0.8054
Epoch 16/50  loss: 0.4052 - accuracy: 0.8116 - val_loss: 0.4197 - val_accuracy: 0.8059
Epoch 17/50  loss: 0.4040 - accuracy: 0.8124 - val_loss: 0.4191 - val_accuracy: 0.8052
Epoch 18/50  loss: 0.3980 - accuracy: 0.8158 - val_loss: 0.4196 - val_accuracy: 0.8057
Epoch 19/50  loss: 0.3966 - accuracy: 0.8167 - val_loss: 0.4199 - val_accuracy: 0.8052
Epoch 20/50  loss: 0.3961 - accuracy: 0.8170 - val_loss: 0.4197 - val_accuracy: 0.8056
Epoch 21/50  loss: 0.3957 - accuracy: 0.8173 - val_loss: 0.4200 - val_accuracy: 0.8053
```

Model 6 | Curves





Model 6 | Evaluation

Output of model_6.evaluate(X_test, y_test) : `loss: 0.4202 - accuracy: 0.8070`
```
[0.42016127705574036, 0.8070250153541565]
```

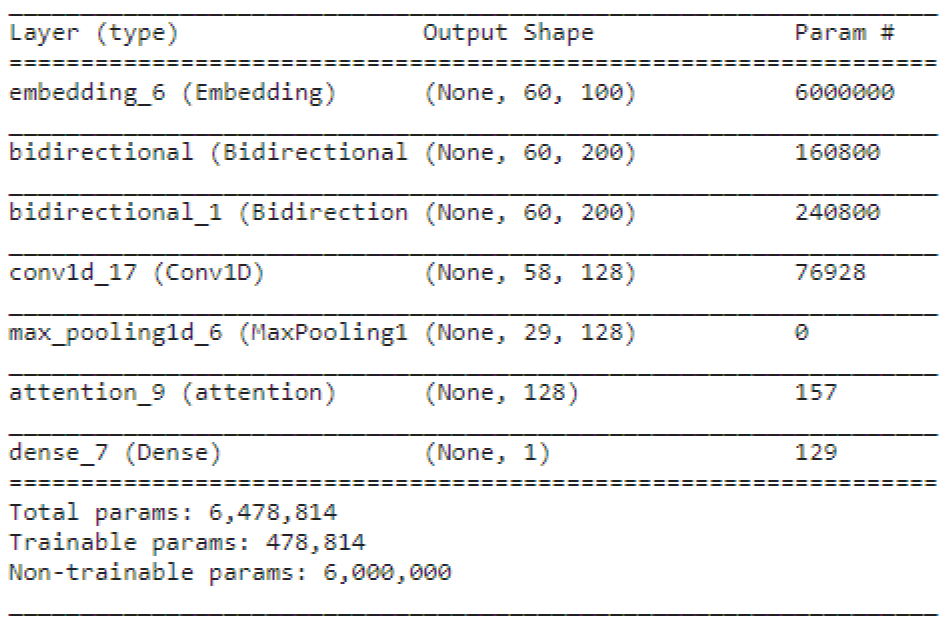
Testing accuracy= 80.70%

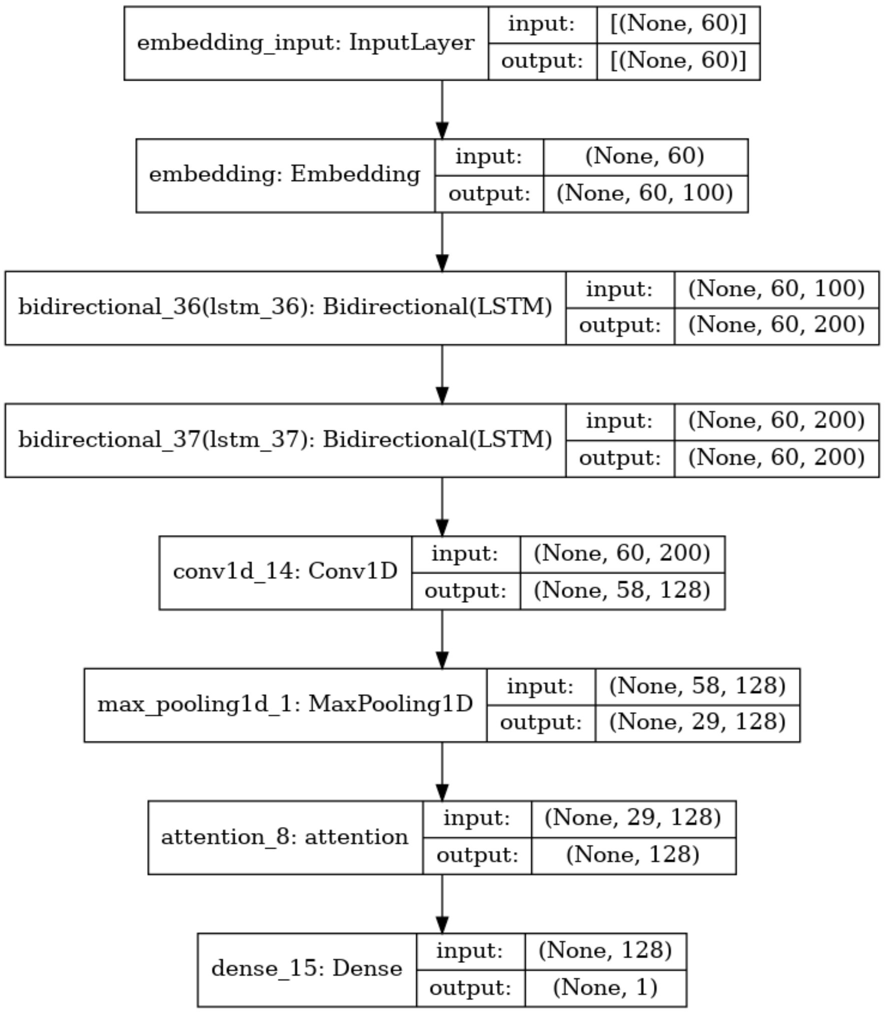
3.7.2.7 Model 7

In the final model, a customized attention mechanism along with the CNN-biLSTM stacked model is used. The model structure is as follows:

6) **Embedding Layer**
7) **Bidirectional LSTM Layer 1:** This is a simple bidirectional LSTM layer wherein the unidirectional LSTM code is used inside the *Bidirectional()* method with the same hyperparameters as before:
    - **recurrent units=** 100
    - **dropout=** 20%
8) **Bidirectional LSTM Layer 2:** Same parameters as previous bidirectional layer.
9) **CNN Layer:** All previous hyperparameters-
    - **filters=**128 because increasing the number of filters gives better performance(found by experimentation)
    - **kernel_size=**3 as before
    - **activation:** 'relu' function.
10) **Attention Layer**
11) **Dense Layer (Output Layer):** This has 1 neuron with **"Sigmoid"** activation function.

Model 7 | Architecture

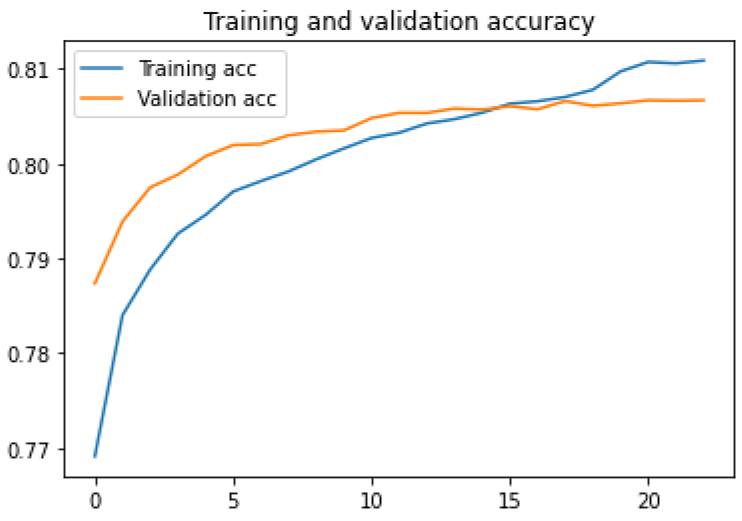

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_6 (Embedding)      (None, 60, 100)           6000000   
_________________________________________________________________
bidirectional (Bidirectional (None, 60, 200)           160800    
_________________________________________________________________
bidirectional_1 (Bidirection (None, 60, 200)           240800    
_________________________________________________________________
conv1d_17 (Conv1D)           (None, 58, 128)           76928     
_________________________________________________________________
max_pooling1d_6 (MaxPooling1 (None, 29, 128)           0         
_________________________________________________________________
attention_9 (attention)      (None, 128)               157       
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 129       
=================================================================
Total params: 6,478,814
Trainable params: 478,814
Non-trainable params: 6,000,000
_________________________________________________________________
```

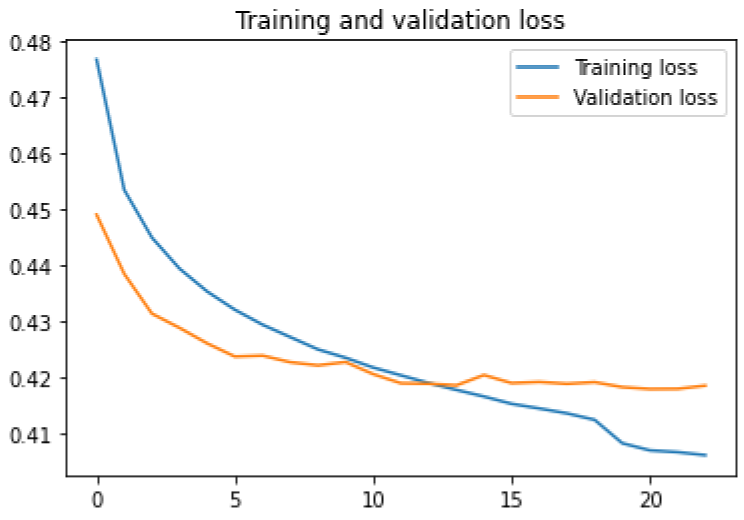


<u>Model 7 | Training Data</u>

```
Epoch 1/50   loss: 0.4768 - accuracy: 0.7691 - val_loss: 0.4490 - val_accuracy: 0.7874
Epoch 2/50   loss: 0.4534 - accuracy: 0.7840 - val_loss: 0.4384 - val_accuracy: 0.7939
Epoch 3/50   loss: 0.4449 - accuracy: 0.7888 - val_loss: 0.4313 - val_accuracy: 0.7975
Epoch 4/50   loss: 0.4394 - accuracy: 0.7926 - val_loss: 0.4288 - val_accuracy: 0.7988
Epoch 5/50   loss: 0.4353 - accuracy: 0.7946 - val_loss: 0.4260 - val_accuracy: 0.8008
Epoch 6/50   loss: 0.4320 - accuracy: 0.7971 - val_loss: 0.4237 - val_accuracy: 0.8020
Epoch 7/50   loss: 0.4294 - accuracy: 0.7981 - val_loss: 0.4238 - val_accuracy: 0.8021
Epoch 8/50   loss: 0.4271 - accuracy: 0.7992 - val_loss: 0.4227 - val_accuracy: 0.8030
Epoch 9/50   loss: 0.4249 - accuracy: 0.8004 - val_loss: 0.4221 - val_accuracy: 0.8034
Epoch 10/50  loss: 0.4235 - accuracy: 0.8016 - val_loss: 0.4227 - val_accuracy: 0.8035
Epoch 11/50  loss: 0.4217 - accuracy: 0.8027 - val_loss: 0.4205 - val_accuracy: 0.8048
Epoch 12/50  loss: 0.4203 - accuracy: 0.8033 - val_loss: 0.4189 - val_accuracy: 0.8053
Epoch 13/50  loss: 0.4189 - accuracy: 0.8042 - val_loss: 0.4188 - val_accuracy: 0.8053
Epoch 14/50  loss: 0.4177 - accuracy: 0.8047 - val_loss: 0.4185 - val_accuracy: 0.8058
Epoch 15/50  loss: 0.4165 - accuracy: 0.8054 - val_loss: 0.4204 - val_accuracy: 0.8057
Epoch 16/50  loss: 0.4152 - accuracy: 0.8063 - val_loss: 0.4189 - val_accuracy: 0.8061
Epoch 17/50  loss: 0.4144 - accuracy: 0.8066 - val_loss: 0.4191 - val_accuracy: 0.8057
Epoch 18/50  loss: 0.4135 - accuracy: 0.8070 - val_loss: 0.4188 - val_accuracy: 0.8066
Epoch 19/50  loss: 0.4124 - accuracy: 0.8078 - val_loss: 0.4191 - val_accuracy: 0.8061
Epoch 20/50  loss: 0.4082 - accuracy: 0.8097 - val_loss: 0.4182 - val_accuracy: 0.8064
Epoch 21/50  loss: 0.4069 - accuracy: 0.8107 - val_loss: 0.4179 - val_accuracy: 0.8067
Epoch 22/50  loss: 0.4066 - accuracy: 0.8106 - val_loss: 0.4179 - val_accuracy: 0.8066
Epoch 23/50  loss: 0.4061 - accuracy: 0.8109 - val_loss: 0.4185 - val_accuracy: 0.8067
```

Model 7 | Curves





Model 7 | Evaluation

Output of model_7.evaluate(X_test, y_test) :

```
loss: 0.4105 - accuracy: 0.80765

[0.41048712964246836, 0.807682638153541565]
```

Testing accuracy= 80.77%

3.7.3 Results

The summarized results are as follows:

Model No.	**Architecture**	**Testing Accuracy**	**Loss**
Model 1	Single unidirectional LSTM (with sigmoid activation and 20% dropout)	79.99%	0.4278
Model 2	Doubly stacked unidirectional LSTM	80.36%	0.4231
Model 3	Unidirectional LSTM+CNN	80.19%	0.4260
Model 4	Single Bidirectional LSTM	80.35%	0.4243
Model 5	Doubly stacked bidirectional LSTM	80.61%	0.4209
Model 6	Doubly stacked bidirectional LSTM+Attention	80.70%	0.4202
Model 7	Stacked Bidirectional LSTM+CNN+Attention	80.77%	0.4105

TABLE 3.5 RESULT TABLE

3.7.4 Observations and Inferences

It can be observed that stacking up of unidirectional LSTM layers gives improvement in performance from 79.99% to 80.36%

Similarly, stacking up of unidirectional LSTM layers also gives improvement in performance from 80.35% to 80.61%

A single bidirectional LSTM model gives nearly about the same performance as a doubly stacked unidirectional LSTM model.

The addition of a CNN layer improves performance as claimed by Liang et al. in their paper [21]. However, this improvement in performance is lesser than the one which is observed with staking.[10]

However, a single bidirectional LSTM layer gives results as good as a doubly stacked unidirectional LSTM layer.[7]

Chapter 4
Conclusion

In this project, a lot of experimentation has been done with LSTM models. The metric used to evaluate the performance of a model is the testing accuracy since the data is perfectly balanced. Several kinds of architectures with different types of layers are tested for performance. Hyperparameters are also tuned by observing training results for the same model. Firstly, a simplest LSTM model with a single unidirectional LSTM layer, an input embedding layer and an output dense layer consisting of a single neuron is trained. Thereafter, unidirectional LSTM layers are stacked up and it is found that there is an improvement in testing accuracy. A CNN layer is added to the single unidirectional LSTM layer and it is found that there is a slight improvement in performance. However, the improvement is more with the stacking than with addition of a CNN layer[5]. Introduction of a customized attention mechanism also gives benefit to the model [3][4]. Finally, an ensemble of Bi-LSTM, CNN and attention mechanism achieves the highest performance.

The key observation is how the information propagates through the network when we unroll the LSTM unit through time. We observe the propagation of the state vectors (cell state and hidden state), whose interactions are primarily linear through time [10]. The result is that the gradient that relates an input several time steps in the past to the current output does not attenuate as dramatically as in the simple RNN architecture. This means that the LSTM can learn long-term relationships much more effectively than our original formulation of the RNN.

The final model in this project did very well at almost 81% as compared to many kaggle kernels of the same dataset, almost all the kernels achieved an accuracy of 77% or less.

Chapter 5
Future prospects

This report aims at serving a basic layout and a generic approach to any task of input having sequential nature. An overview of the backbone of NLP i.e word representations is given in the project. With this, diving deeper into NLP will not be as intimidating as it sounds.

Future prospects can be but not limited to including sentiments other than positive and negative, using of pre trained model such as Bert, XLNet, etc., Opting for other NLP tasks such as document summarising, hate text detection, etc. and can also go beyond NLP such as in Recommendation tasks where Word2vec's approach is very much appreciated.

A lot of further experimentation is also possible, with multiple different hybridised models which can be checked for training performance.

References

[1] Wang et al.; Tree-Structured Regional CNN-LSTM Model for Dimensional Sentiment Analysis; IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING, VOL. 28, 2020

[2] Mal et al.; Sentic LSTM: a Hybrid Network for Targeted Aspect-Based Sentiment Analysis; Springer, Cognitive Computation volume 10, pages 639–650 (2018)

[3] Wang et al.; Attention-based LSTM for Aspect-level Sentiment Classification; Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, The ACL Anthology, pages 606–615

[4] Peng et al.; Targeted Aspect-Based Sentiment Analysis via Embedding Commonsense Knowledge into an Attentive LSTM; The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)

[5] Behera et al.; Co-LSTM: Convolutional LSTM model for sentiment analysis in social big data; Elsevier, Information Processing and Management 58 (2021) 102435

[6] Rehman et al.; A Hybrid CNN-LSTM Model for Improving Accuracy of Movie Reviews Sentiment Analysis; Springer Science+Business Media, LLC, part of Springer Nature 2019, Multimedia Tools and Applications

[7] Minaee et al.; Deep-Sentiment: Sentiment Analysis Using Ensemble of CNN and Bi-LSTM Models; arXiv, Cornell University : Computer Science > Computation and Language

[8] Fu et al.; Lexicon-Enhanced LSTM With Attention for General Sentiment Analysis; IEEE Access, SPECIAL SECTION ON ARTIFICIAL INTELLIGENCE AND COGNITIVE COMPUTING FOR COMMUNICATION AND NETWORK

[9] Chen et al.; Twitter Sentiment Analysis via Bi-sense Emoji Embedding and Attention-based LSTM; 2018 ACM Multimedia Conference (MM '18), October 22–26, 2018, Seoul, Republic of Korea. ACM, New York, NY, USA, 9 pages

[10] Li et al.; Sentiment analysis using lexicon integrated two-channel CNN-LSTM family models; Applied Soft Computing Journal (2020)

[11] Pal et al.; Sentiment Analysis in the Light of LSTM Recurrent Neural Networks; International Journal of Synthetic Emotions Volume 9, Issue 1, January-June 2018

[12] Wang et al.; Investigating Dynamic Routing in Tree-Structured LSTM for Sentiment Analysis; Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, The ACL Anthology, pages 3432-3437

[13] Zhang et al.; Learning interaction dynamics with an interactive LSTM for conversational sentiment analysis; Elsevier, Neural Networks 133 (2021) 40-56

[14] Yantar et Verma; Deep CNN-LSTM with Combined Kernels from Multiple Branches for IMDb Review Sentiment Analysis; 2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)

[15] Dai et Prout; Unlock big data emotions: Weighted word embeddings for sentiment classification; 2016 IEEE International Conference on Big Data (Big Data)

[16] Othman et al.; Improving Sentiment Analysis in Twitter Using Sentiment Specific Word Embeddings; The 10th IEEE International Conference on Intelligent Data

Acquisition and Advanced Computing Systems: Technology and Applications 18-21 September, 2019, France, Metz

[17] Edilson et al.; A Multi-view Ensemble for Twitter Sentiment Analysis; NILC-USP at SemEval-2017 Task 4 (THIS WAS A SHARED TASK)

[18] More et al.; Sentiment Analysis on Amazon Product Reviews with Stacked Neural Networks; ResearchGate

[19] Zhoul et al.; Text Classification Improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling; arXiv, Cornell University, Computer Science > Computation and Language

[20] Li et al.; Bidirectional LSTM with self-attention mechanism and multi-channel features for sentiment classification; Elsevier, Neurocomputing

[21] Liang et al.; A Double Channel CNN-LSTM Model for Text Classification; 2020 IEEE 22nd International Conference on High Performance Computing and Communications

[22] https://www.analyticsvidhya.com/blog/2021/06/text-preprocessing-in-nlp-with-python-codes/

[23] https://www.analyticsvidhya.com/blog/2021/06/must-known-techniques-for-text-preprocessing-in-nlp/

[24] https://www.analyticssteps.com/blogs/introduction-natural-language-processing-text-cleaning-preprocessing

[25] https://medium.com/analytics-vidhya/text-preprocessing-for-nlp-natural-language-processing-beginners-to-master-fd82dfecf95

[26] https://catriscode.com/2021/05/01/tweets-cleaning-with-python/

[27] https://towardsdatascience.com/basic-tweet-preprocessing-in-python-efd8360d529e

[28] https://towardsdatascience.com/nlp-text-preprocessing-a-practical-guide-and-template-d80874676e79

[29] https://www.einfochips.com/blog/nlp-text-preprocessing

[30] https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/

[31] https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

[32] https://machinelearningmastery.com/stacked-long-short-term-memory-networks/

[33] https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

[34] https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046

[35] https://keras.io/api/layers/recurrent_layers/lstm/

[36] https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/

[37] https://www.mordorintelligence.com/industry-reports/deep-learning

[38] https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/

[39] https://www.kaggle.com/kazanova/sentiment140

[40] https://www.analyticsvidhya.com/blog/2019/11/comprehensive-guide-attention-mechanism-deep-learning/