Sentiment Analysis and Sarcasm Detection in Twitter Conversations using Deep Neural Networks

Felix Delos Santos III

Supervisor: Josephine Griffith

March 2017

Abstract

Sarcasm Detection has proven to be an important and necessary task for social media systems. The ability to detect sarcasm in text requires great knowledge in Natural Language Processing and feature engineering. This project considers the problem of automatic sarcasm detection in Twitter data. Deep Neural Networks and Convolutional Neural Networks are designed and developed to combat the difficulty of feature engineering by automatically learning features to aid in the sarcasm detection task. These trained Neural Networks are tested on their ability to predict the underlying sarcasm in two Twitter datasets.

Contents

List of Figures

List of Tables

1 Introduction

Natural Language Processing is the field of computing which focuses on the analysis of text. One of the more difficult tasks in Natural Language Processing is the ability to detect sarcasm in text. The presence of sarcasm in a text can completely change the polarity of the text and therefore the ability of detecting sarcasm becomes incredibly useful in understanding the correct sentiment being expressed. Given the various ways in which sarcasm can be expressed, it is a difficult task to classify whether text is sarcastic or not. To understand the presence of sarcasm, one must analyse not only the literal meaning of a text but also the other various meanings it may have. Detecting sarcasm can even be a difficult task for humans. This is due to the various ways that it can be expressed.

Sarcasm detection is a field in Natural Language Processing that is still relatively new. Previous approaches have tackled the problem with manual textual feature engineering. This is the process of manually creating features of the textual data such as n-grams, brown clusters and part-of-speech features. The previous approaches have then used these manually created features to train machine learning models to classify sarcasm in text. Although, some have achieved great accuracy in sarcasm detection, they have done so with difficult feature engineering which is not only time-consuming but is also very computationally expensive. This project approaches the problem of sarcasm detection in a different manner. The models proposed are Neural Network models which automatically learn patterns and cues in the data which suggest Sarcasm. Neural Network models have achieved great success in various fields such as image recognition with their ability to perform automatic learning on almost raw data. Their ability to automatically learn patterns in data have motivated their use in various text classification problems, one of which is sarcasm detection.

1.1 Project Goal

The overall task is the creation of Neural Network models which are capable of successfully classifying a tweet as a sarcastic tweet or a non-sarcastic tweet.
Previous work in sarcasm detection has focused on the use of features which were basically hand chosen, such as word uni-grams and bi-grams (Liebrecht et al. [2013]). The main goal for this project is to research and design a basic Deep Neural Network and see it's capabalities in Sarcasm Detection as well as the design and development of a Convolutional Neural Network architecture which can be used for automatic feature extraction and classification. With these features, which are automatically extracted, I investigate the effectiveness of using a learning framework such as a Convolutional Neural Network and automatically extracted features in the sarcasm detection problem.

This project consists of three main objectives:

1. Acquiring and cleaning of data
The first objective of this project will be accessing the Twitter API for the retrieval of tweets. Once the tweets have been retrieved from twitter, the data must be cleaned. From Section 4 of this document, it can be seen that the two datasets of tweets being used have already been labelled and prepared. However, for the prediction task further work will have to be done for the cleaning of this data, such as the removal of text that may not be useful e.g. links and hashtags.

2. Designing the DNN and CNN
The design of the neural networks for automated feature extraction and classification of sarcasm will be the main objective of this project.
When designing the Neural Networks, there are multiple aspects that need to be taken into account:

- How many hidden layers are required?
  In general, one hidden layer is sufficient for solving a large number of problems. However, given the nature of this problem, proper testing may indicate that more hidden layers will be required for this classification problem.

- How to represent the textual data for the Neural Networks?
  There are multiple ways to represent textual data in a manner that Neural Networks can interpret. This project will also explore the types of textual representations that the Neural Networks can best learn from. The two main focuses

for textual representations will be a simple Bag-Of-Words model and Word Embedding representations.

- How many convolutional layer(s) are required?
  Since convolutional layers reduce the number of input features being passed into a layer, using multiple convolutional layers can greatly reduce the amount of features being passed once connected to the fully connected layers. Convolutional Layers will produce much more simple features for the layer following it and so important information may be lost depending on the amount of convolutional layers that are used and so an increase in the number of convolutional layers might reduce the accuracy in the classification due to the generalized features. So to know the appropriate amount of convolutional layers to use, multiple tests will have to be performed.

3. Evaluation

To measure the quality of the Sarcasm Detection models created by both Neural Networks, the final models must be evaluated on a partition of data that they have not been trained on or tested on. This dataset will be called the held-out dataset, as it is held-out before any training or testing has been performed.

1.2 Related Work

Contextualized Sarcasm Detection:
Various approaches have relied on indicators in the text such as non-veridicality or hyperbole to aid in the classification problem. However, it's easy to tell that gaining an understanding of the context can provide clues that aid in the problem of sarcasm detection. Bamman and Smith (2015) explore this concept of contextualized sarcasm detection with twitter data. They make use of the context of the twitter data, such as information about the author, as features for sarcasm detection. It was found that the single most informative feature of all those evaluated was the Author historical salient terms. But the largest performance gain is achieved when the tweet features are combined with features describing the context of the environment of the tweet, context of the audience and context of the author of the sarcastic tweet. Therefore, having a context as well as tweet features such as word unigrams, can aid incredibly in the performance of a sarcasm detector. However, all of these features which were used by Bamman and Smith are hand-crafted. The features are designed in a complex manner which includes the context and content of a tweet. This is an expensive approach which requires a lot of manual effort.

Deep Learning For Automatic Extraction of Contextual Features:
Although previous works such as that by Bamman and Smith (2015) have achieved great accuracy in the modelling of content with relevant contexts such as Historical Tweets, they have done so with very difficult and complex feature engineering. Silvio Amir et.al (2016) explore a different approach to the sarcasm detection problem which avoids the effort of manually designing features. Just as previous works with contextual sarcasm detection, they have made use of not only the sarcastic tweet in question but also any historical tweets by the user in the detection of sarcasm. However, they make use of a convolutional neural network which automatically learns relevant features in the data and creates a model which learns to represent the content with the relevant context. With the use of user embeddings, Silvio Amir et.al (2016) model relevant context on the author of the tweet in question. So when defining whether a tweet is sarcastic or not sarcastic, the neural network considers both the content of the tweet and relevant context of the author. Word embeddings were used to model the content and User Embeddings were then used to model the context. The sarcasm detection problem is then modelled by the content and it's relevant user contexts. Using these user embeddings they were able to model the relationship between the user and his/her audience, allowing them to make use of knowledge regarding the users prior sentiments to their audience. A tweet is then said to be somewhat sarcastic if the sentiment in the text is incongruous to the sentiments in prior tweets of the user. For Example, if a users history of tweets mostly bear positive sentiments

and the tweet under consideration bears negative sentiment, then sarcasm/irony may be present in the text. Their model learns traits in users and allows them to map users close to one another if they are similar and quite far if they are not very similar at all. Using such representations, not only can they understand whether the user bears mostly positive/negative sentiment in their tweets but also whether they have a history of being ironic or sarcastic. If a user has a prior history of being sarcastic then any tweets that come from this user have a higher probability of being sarcastic than a user that is dissimilar to this user and is rarely sarcastic. Making use of content and context they outperform the state-of-the-art approach and although this approach is similar to the approach of Bamman and Smith (2015), the features describing context of the tweet are not manually designed.

Word-Embeddings:
Previous approaches in sarcasm detection have relied on the presence of sentiment bearing words. However, without the presence of sentiment bearing words, it may be difficult to correctly classify sarcasm. The features generated may not be sufficient in the classification problem of sarcasm. Using previous approaches to detect sarcasm which pay attention mostly to sentiment bearing words in a text such as in Example 1, will fail in the detection of sarcasm in this tweet as there are no significant sentiment bearing words, making this a difficult text to classify.

Example 1. Sarcastic Tweet:
"With a sense of humor like that, you could make a living as a garbage man anywhere in the country"

Joshi et.al (2016) have considered the notion of capturing "context incongruity in the absence of sentiment words". With the use of word embedding-based features Joshi et.al (2016) show that "word vector-based similarity/discordance is indicative of semantic similarity which in turn is a handle for context incongruity"[3]. For classifying whether a text is sarcastic or not sarcastic, they consider the semantic similarity of word-embeddings. Silvio Amir et.al (2016) discussed that a major trait of sarcasm is the context incongruity in a sentence and that irony may be present when words which are not semantically similar are used in a sentence. Word Embeddings provide multidimensional representations of words and so a deeper understanding of the meaning of a word can be formed. Word Embeddings allow for the mapping of words to an N-Dimensional space in which words that are similar to each other are quite close to one another, but words that are dissimilar from one another are quite far from one another. Using this notion, we can form features for learning that can understand and make use of the similarity of words being used in a sentence. So, if words that are defined as not very similar by the trained word-embeddings, are used in a sen-

tence together, then we can assume that irony/sarcasm may be present. Silvio Amir et.al (2016) made use of pre-trained word embeddings from different sources such as Word2Vec, LSA, GloVe and Dependency Weights. They found that Word2Vec word embeddings outperformed three other types of word embeddings, (LSA, GloVe and Dependency Weights) and that the use of word embeddings helped in the sarcasm detection task. Silvio Amir et.al (2016) have bypassed this manual effort of creating textual features by inputting pre-trained word embeddings into a convolutional layer of their neural network architecture. Features are then automatically extracted from these word-embeddings and the corresponding input texts.

2 Technical Review

Python: With regards to the programming language I will mostly be using for this project, I have chosen Python. Python will be used in the development of the Neural Network which will be used in the training of a model in the sarcasm detection task. Python will also be used in retrieving the twitter data and performing any prerequisite text pre-processing. As well as this, Python will allow the use and access of various libraries which will be incredibly useful in the development process of this project. I have chosen Python as the main programming language of this project as I've had no prior experience in python programming. I decided to take this opportunity to properly submerge myself in a technology that I have no major experience in. Python has also been an easy language to dive into due to the fact that python code is easy to read and understand. Finding solutions to problems, has been greatly aided due to the substantial amount of documentation for Python.

Tweepy: A Python library is required for easy access to the Twitter API. For this project I have chosen the Tweepy Python library for use in the retrieval of tweets from Twitter. The Tweepy library allows for easy data retrieval from Twitter. One can retrieve large amounts of tweets daily for analysis. One of the major benefits of Tweepy is the retrieval of tweets that make use of specific hashtags. Previous works in sarcasm detection have made use of streaming tweets from the hashtags, "#sarcasm", "#sarcastic" and "not", to retrieve user labelled sarcastic tweets. Data retrieval for the sarcasm detection task has been greatly aided because of the endless amounts of pre-labelled tweets under these hashtags. Tweepy also allow for users to retrieve tweets given the IDs of the tweets, making it easier for researchers to share their twitter datasets. Due to all of these features, I decided that the library that would allow for the easiest retrieval of tweets would be Tweepy.

Tensorflow: For the design and development of Deep Neural Networks, I wanted to choose a machine learning system/library that would be easy to learn and understand but would also meet all the requirements to make a predictive model for this project. For this project, I have chosen TensorFlow, an open source library for machine learning. Tensorflow provides us with functionality that will aid in the design and development of the Neural Networks. It represents the computations of a program in a graph which is incredibly applicable to Neural Networks and the manner in which they perform computations in a feed-forward manner and adjusting weights in a reverse back-propagation, like moving up and down a graph. So learning the work flow of a Tensorflow program is relatively easy. Tensorflow allows the addition of different computational nodes that one might require in a Neural Network allowing for an easy design and implementation. It also makes use of "ndarray" objects from

the NumPy Library, which are also used in this Neural Network implementation. With the large amounts of tutorials and documentation on the TensorFlow library, I believe TensorFlow would be perfect for the creation of a Neural Network model in a text classification task.

NumPy: A scientific and mathematical computing library was also required in the developing of a machine learning model for text classification. NumPy provides a large amount of functionality for performing computations on multidimensional NumPy arrays called "ndarray". With ease, NumPy allows one to perform manipulations on this data that is large in scale. NumPy makes it easy to create multidimensional matrices and vectors with which one can perform element-by-element operations, exactly like you would matrices. Computations on these large-scale NumPy arrays are also very fast, making this a perfect mathematical computation library for large amounts of textual data that will be used in the training of a Neural Network Model.

TensorBoard: Visualizing the learning process of a Neural Network model can be very difficult. Tensorboard is a visualization tool for TensorFlow that makes it easy to view the changes in the graph over time. As can be seen in Figure 1, Tensorboard makes it easy to visualize learning curves for the Neural Network, which is essential in analyzing the fit of a model for machine learning problems. Tensorboard also allows developers to view their network in a visual manner in which they can see the connections between the computational nodes in their graph on multiple layers.

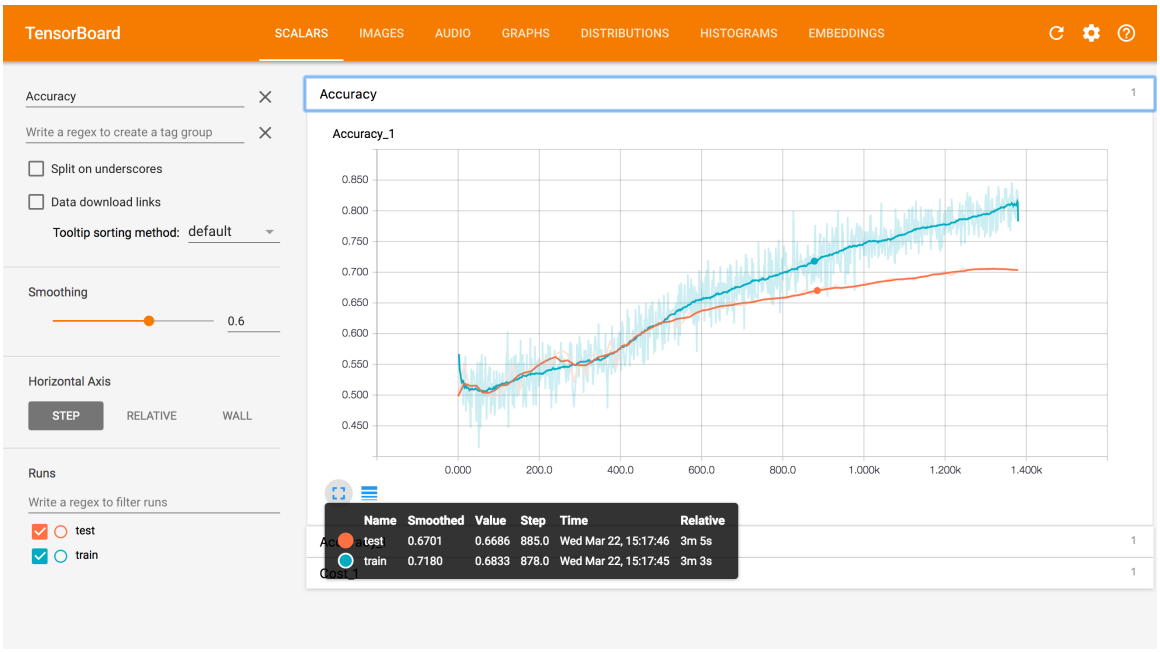

Figure 1: Tensorboard

NLTK: Natural Language Toolkit(NLTK) is a python package that allows python programs to work with Natural language. For this project some preprocessing of textual data was required, such as string tokenization and word lemmatisation(changing words to their most basic form e.g. dogs to dog). NLTK provides functionality that makes it easy to convert tweets to features that can be processed by machine learning algorithms.

Pandas: Pandas is a data analysis library for python and allows users to make use of data structures that are heterogeneous, that is can contain different types of data, in comparison to NumPy which is homogeneous. Pandas Dataframes are easy to manipulate, allowing for the ease of insertions, deletions etc. Pandas can be used with NumPy for mathematical computations on DataFrames such as is provided in the statistical programming language "R".

Atom: Choosing an appropriate text editor for the development of this project was not an easy task. For this project I have decided to perform all my programming with Atom text editor. Atom is an open source text editor and has a large following of developers which continuously provide addons for the text editor to improve the

programming experience. Atom provides a nice view of the projects, which allows for an enjoyable programming experience. Atom makes it easy to read code with it's syntax highlighting which supports multiple languages. Atom is also fully integrated with Git, making it easy to see the lines of code that have been added and changed. Due to the extensive functionality in Atom, I have chosen it as my primary and only text editor for any code.

GitHub: Github uses git, which is a distributed version control system. For this project implementing a version control system is absolutely essential so as to keep track of the history of the project, push changes and revert any changes that may have introduced bugs in the project. Github stores a copy of the project repository which I can continuously update as I make changes to my project. There are many services like Github that use git such as BitBucket, GitLab etc. However, I've decided to choose Github as it provides a nice readable view of the project structure and it provides functionality for managing tasks in the project as well as reporting any issues on the project, such as bugs. Figure 2 shows the use of Github for project management. Github made the management of this project relatively easy with its extensive functionality in version control.

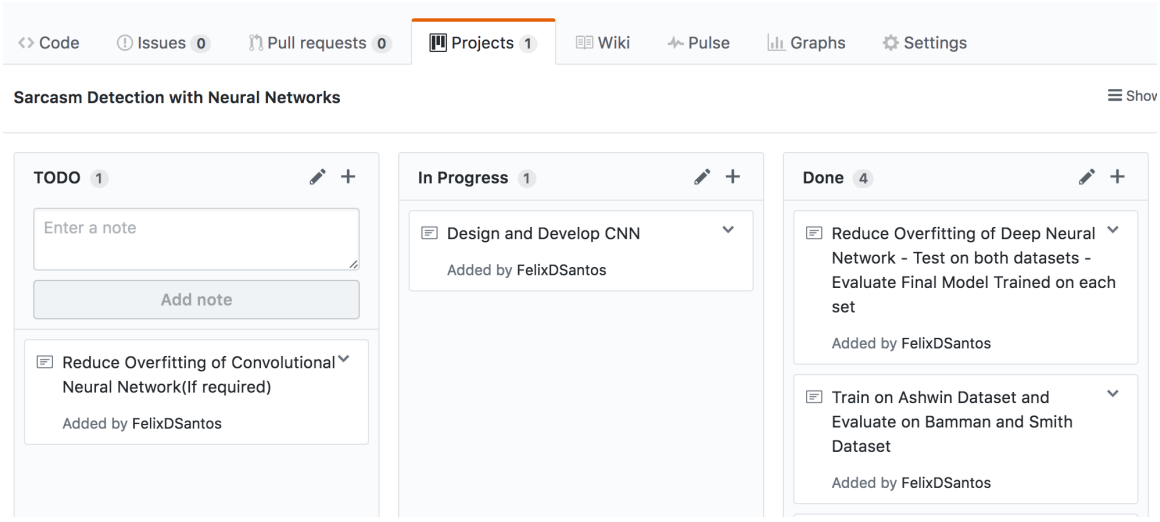

Figure 2: Handling tasks in Github

JSON: When training the neural networks, we do not want to create a vocabulary and also convert the text tweets to features every time we run the network as this

would make the training process slower than it needs to be. JSON allows us to save variables to files for later use. JSON is fast at loading files as well as writing variables to files such as a large vocabulary of words as well as thousands of text features making it optimal for this project.

Draw.io: Draw.io is a web-based application used for creating diagrams. Draw.io was used to create almost every diagram in this project. It's easy to create diagrams with this application as there are hundreds of templates and pre-designed diagrams.

3 Machine Learning

Machine Learning is the field in Computer Science which allows computers to learn and perform tasks without "explicitly" being programmed. Given a problem that a computer program must solve, a programmer might just create a set of logical rules which guide the program to a reasonable solution to a problem. This might work for very simple problems but real-world problems have become more difficult and complex for a program to solve with explicit sets of rules or guidance by a programmer. Machine Learning gives computers the ability to take some problem and learn how to solve it without being explicitly programmed. Computer Systems which are explicitly programmed cannot adapt and learn from examples and use the knowledge it has learned from experience to improve its performance for future problems. Machine Learning explores the idea of training a computer program to learn from its mistakes as it is trained and tested. Given a machine learning model that has been trained for a certain problem such as text classification, the model should be able to use the knowledge it has acquired from training and perform as well on similar problems it has not been trained on. If a machine learning algorithm has been taught to classify the difference between nouns and verbs using a collection of sentences, then given a collection of sentences it has not seen, it should be able to identify the nouns and verbs in this collection too.

3.1 Classification

Classification problems in machine learning focus on categorizing samples of data to a certain class/category e.g. classifying the disease a patient may have. The main task of this project is sarcasm classification in text and so it is a text classification problem tailored to a specific kind of sentiment. For this problem, a machine learning model must be trained to detect sarcasm in text. There are multiple examples of machine learning classifiers such as Logistic Regression, Support Vector Machines and Decision Trees. The major problem with using any of these classifiers, or others like them, is that they will not be able to learn from raw text. That means that if raw text is inputted into these classifiers, they won't be able to understand the important features of the text that lead to specific types of sentiment. Feature extraction is the task of finding relevant features of data that machine learning algorithms can use to learn. However, this task requires experts in the field and is quite difficult and tedious. It is because of the difficulty of feature extraction that Deep Learning has come into prominence. Deep-Learning Neural Networks can extract features from raw data without explicit programming. They are able to accomplish this by continuously trying to model the class of a training example. In the process of this, Deep-Learning

Neural Networks can learn to distinguish between the relevant features in the raw data without the difficult and tedious process of feature engineering.

3.2 Deep Learning with Neural Networks

As mentioned above, Neural Networks are extremely skilled in automatic feature extraction making them a perfect machine learning algorithm for text classification. Text classification and more specifically sarcasm detection requires very carefully crafted features that represent the text. Many approaches to sarcasm detection have used engineered features such as Bamman et al., (2015). This project focuses on the capabilities of Deep Neural Networks for automatic feature extraction.
As suggested by the name, the idea of Neural Networks is inspired by the human brain which is able to recognize patterns with continuous exposure to different inputs. Neural Networks recognize patterns in numbers. With multiple training iterations using a set of inputs and known outputs, a Neural Network can learn to focus on the patterns of the input data that are most significant on the outcome.

Neural Networks at their most simple form consists of just one neuron, that is one computational node that takes some inputs which are weighted and creates some output. The neuron will take all the weighted inputs and sum them up to create a result which is then passed into an activation function which defines the type of output by the network.
Let σ be the output of a neuron, after summing the inputs x_i weighted by their respective weighting w_i

$$\sigma = \sum_{i=0}^{n} w_i x_i$$

The activation function then takes σ and adjusts it to produce an optimal output for the function. Let's say for a given problem the output values required are probabilistic and so they are required to be between 0 and 1. The optimal activation function for such a problem would be a Sigmoid Function which will map the output of a neuron to a value between 0 and 1. Neural Networks are just multiple neurons organized in layers, where neurons in a layer connect to neurons in the layer after them. The first layer is just the input layer and the final layer is usually the output layer which produces the final result. Any layers in-between these two layers are hidden layers. This is where all computations are performed and where the network models an input to an output. Hidden layers can have hundreds of neurons, where each neuron takes the weighted inputs $w_i x_i$ and sums them and passes them to activation functions before passing this output to the next layer. Figure 3 shows a sample Neural Network with

3 layers, an input layer, 1 hidden layer and an output layer.

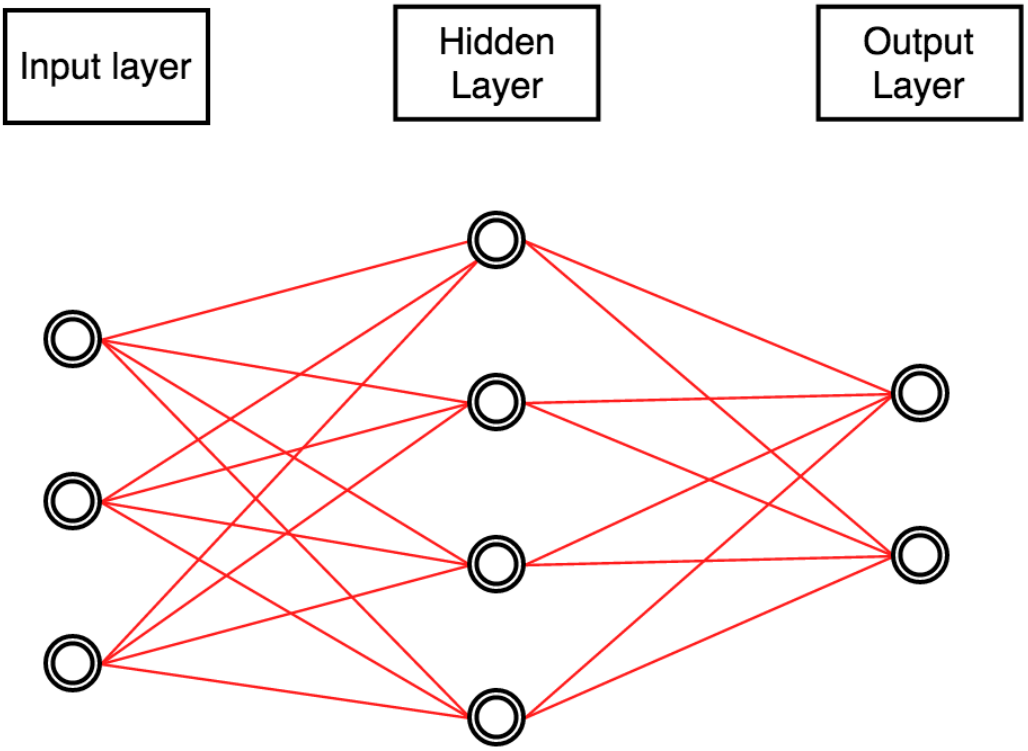

Figure 3: Neural Network with 3 layers. Layer 1 takes all input nodes and connects them with all neurons in the Layer 2. These links are the weights. The value that enters the first neuron in Layer 2 would be $\sigma = \sum_{i=0}^{n} w_i x_i$ from Layer 1 and similarily for all other neurons in Layer 2.

For text classification, text data will be encoded as some numerical representation and is then fed through the network. Each input is weighted and passed through the layers and the output layer then outputs values that can be interpreted as the output class of a given text e.g. sarcastic or non-sarcastic.
Essentially the process of training a neural network is finding the optimal weights that links nodes in the network. However, guessing the optimal weights for each link would be inefficient given the large scale of Neural Networks. The training process works by feeding data through the network to compute some output prediction $\hat{y}$ for

a given set of training example X. Once the network produces a prediction, it can then check how far off it was from the actual answer y using some error function e.g. Root-Mean-Squared-Error. By calculating the error E with respect to the weights in the network, the network can calculate the direction the error goes with respect to the weights. By knowing the direction at which the error goes with respect to the weights, the network can optimize the weights in a way that's guaranteed (to some extent) to get a global minimum for error. The network will look for the point at which the weights provide the least error for the problem and adjust the weights closer to those values until it eventually converges to some minimum error. This is called gradient descent. The network then back-propagates this error derivative to know how the error changes with respect to each layer in the network. One training iteration is finished when every layer is adjusted with respect to the error in the layer following them. This continues for a set amount of training iterations until it's evident the network cannot improve anymore.
For the sarcasm detection problem, these same principles will be used to create an optimal Neural Network to classify the sarcasm in tweets.

3.3 Word Embeddings

Typical word representations such as one-hot encodings are high dimensional and are vulnerable to overfitting. As well as this, they do not properly describe the semantic meaning of words. In a one-hot encoding of words each word is represented as a vector of the same length as the vocabulary. Let v be a vocabulary of words of length n. In a one-hot encoding, we encode a word u as a vector w. The value at w_i is 0 but is 1 at w_k when

$$v_k = u$$

Example: Let

$$v = [' Hello', ' there', ' friend']$$

and let $u =' Hello'$, then

$$w = [1, 0, 0]$$

as "Hello" appears on the 1st index of the vocabulary.

This means if a collection of words W is represented as:

$$W = \{w_1, w_2, ..., w_s\}$$

where each $w_j \in W$ is a word that is represented as one-hot vectors, then a document with s words is represented as an $s \times n$ matrix which is mostly zeros.

This can lead to a lot of problems when training a textual classifier as the majority of the data is represented as zeros making the data very sparse. It is difficult for any classifier to learn from such sparse data. As well as the problem of sparsity, the one-hot encodings aren't meaningful representations of words and do not take into account the similarity of words. Word embeddings can combat this problem. Word embeddings represent the relationships between words. Instead of a word being represented as a vector of length n, it is represented by a chosen dimensionality d. Not only is this less computationally expensive because of the reduced dimensionality but word embeddings allow us to learn the relationships of words in a vocabulary.
Let u and v be vectors of length d, with real values as entries. The goal of a word embedding is such that for a word u and a word v, we can calculate an inner product between the two words

$$u^T v = u_1 v_1 + u_2 v_2 + ... + u_d v_d$$

This inner product defines how similar the two words are. The inner product of two words like "cat" and "animal" will be quite large because they are such similar words. The inner product of two words like "chair" and "cat" would be smaller as these words may not be as commonly used as "cat" and "animal". Figure 4 shows the mapping of these words in a 2 dimensional space. It can be seen that "Cat" and "Animal" are quite close to one another as they are very similar words and are far away from "Chair" as "Chair" is dissimilar form the other words.

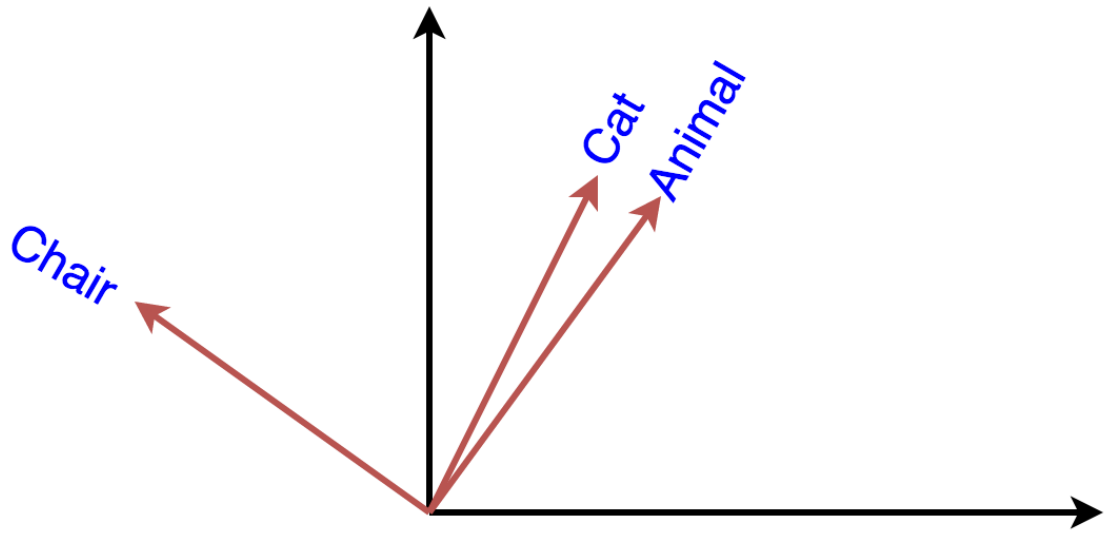

Figure 4: 2 Dimensional Word Embeddings

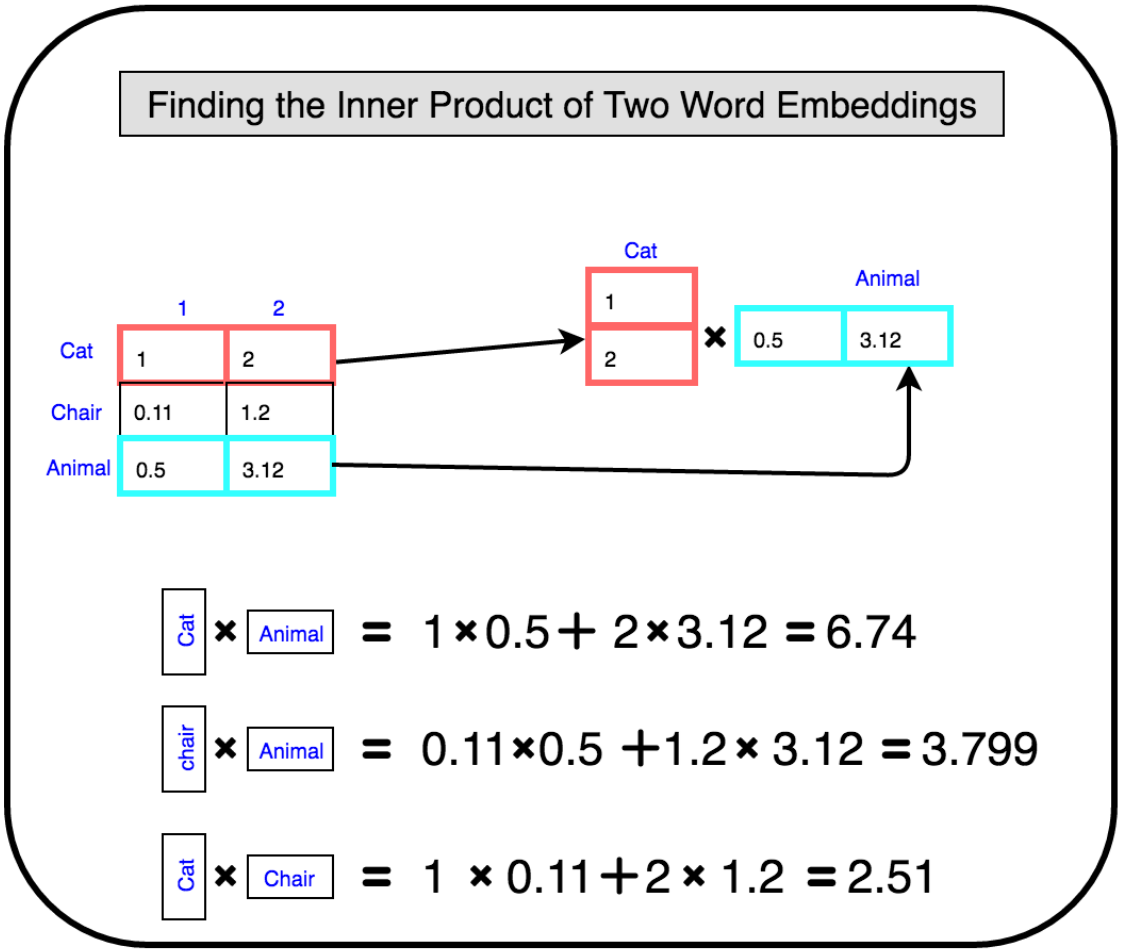

Figure 5: Finding the inner product between two word embeddings "Cat" and "Animal" of dimensionality=2

In figure 4, we have 3 words "cat", "animal" and "chair" mapped in a 2 dimensional space, for simplicity. "Cat" and "Animal" are both quite close to each other in this space as they share similarities but far away from "chair" as they are dissimilar from "chair". The process of calculating the inner product between two word embeddings (in this case 2-dimensional word embeddings) is the element-wise vector multiplication and summation of the transpose of the first word embedding with the second word embedding. This is a very powerful tool and the advantage of word embeddings is that they can be trained just like any other parameters in a Neural Network. By knowing how often words co-occur, a Neural Network can take the word embeddings

and adjust them so that the values in the word embeddings represent the words relationships with other words in a d dimensional space, where words that are similar are near to each other and words that are dissimilar are far. From Figure 5 we can see the inner product of "cat" and "animal" is greater than the inner product of any other word combination in this word embedding matrix as "cat" and "animal" co-occur more frequently. It is then possible to infer that "cat" and "animal" are more semantically similar than "cat and "chair" or "chair" and "animal".

3.4 Convolutional Neural Networks

Normally when Convolutional Neural Networks are mentioned it's with visual learning with Neural Networks. This is the task of learning patterns and features in visuals such as images. However, textual learning with Convolutional Neural Networks has also become popular as of late. This is due to the automatic textual feature construction that convolutional neural networks have achieved as can be seen in "Convolutional Neural Networks for Sentence Classification"[4]. Yoon Kim shows the capabilities of Convolutional Neural Networks with word embeddings in any sentence classification problem.

Convolutions: A Convolution is a sliding window function that is applied to a matrix. This sliding window is called a convolutional filter. These filters are trained during the training of a Convolutional Neural Network to detect important features of an arbitrary input, whether it be images or in our case text. A filter is also a matrix and has h rows and w columns containing real-valued elements. A convolutional filter will slide across an input matrix covering h rows and w columns at a time. In the process of sliding across the matrix, the filter will perform an element-wise matrix multiplication between the values in the filter window and the corresponding values in the filter. These values are then summed to get an entry of the new feature map before the filter slides across to the next window. To get the whole new feature map, this filter slides over the whole matrix.

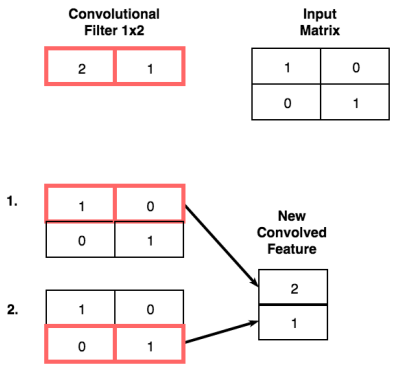

Figure 6: 1x2 Convolutional Filter creating a new feature out of the Input Matrix

Figure 6 shows the creation of a convolutional feature after sliding a 1×2 filter down a 2×2 matrix creating a 1×2 feature. A convolutional layer is made up of multiple convolutional filters that create multiple convolutional features which are then fed into an activation function and then to a hidden layer. The convolutional layer can have filters of different sizes to get different feature representations of the input data.

Convolutional Neural Networks in NLP: Assume we have some 6 word sentence using 128-dimensional word embedding, the matrix to represent the sentence would be 6×128.
When convolving over a word embedding matrix, the width of the filters are usually the same width as the embedding matrix and so the width of the matrix is the same as d, the amount of dimensions for our word embedding, the height h of the filter is then the amount of words we want the filter to consider at a time, so the filter shape is $h \times d$. There can be multiple filters of different heights h to get multiple features with different size context windows for words.
Each filter slides over the sentence matrix making a new feature map. 1-D Max Pooling then occurs over each feature map to get a univariate feature from each feature map. 1-D Max Pooling takes a feature map vector and outputs the maximum value in this vector. The effect of this being that we take the most significant feature. Each of these features are then concatenated into a single final feature vector. This feature vector is then inputted to a hidden fully connected layer to output the prediction just like for any other regular Neural network.

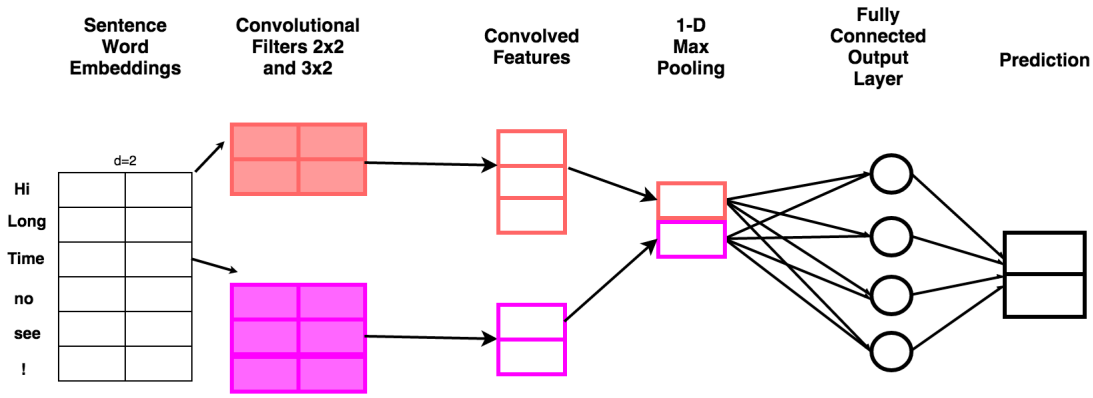

Figure 7: Convolutional Neural Network with 2 filter sizes for a classification task.

The convolutional neural network is trained just like any other Neural Network with the filters for automatic feature extraction also adjusted in the training process.

Max Pooling: The main objective of a pooling layer is to reduce the dimensionality of the output of convolutions. If we were to concatenate the outputs of hundreds of filters then the final vector after convolution will be very high dimensional which can lead to an overfitted Neural Network Model. A pooling layer takes each feature map and takes the max value and adds this value to a final single vector. A filter represents a specific feature in the data. There could be three filters, one that detects "positive sentiment", one that detects "negative sentiment" and one that detects "neutral sentiment". Applying a "positive sentiment" filter over a sentence matrix will result in regions with a "positive sentiment" word giving large values, while other areas will yield lower values. Similarly for the "negative sentiment" filter and the "neutral sentiment" filter. The Max pooling layer will take the large value from the feature map created, to show that there was a "positive sentiment" word in the sentence. The problem of course with this is that the context of where the "positive sentiment" word is lost but the knowledge of the occurrence of a feature in this sentence is still very useful information for our model.

Stride Size: The stride size of a convolutional filter is the step size of the filter as it slides over an input matrix. The above examples show convolutional filters with a stride size of 1 meaning the filter slides down one row of the matrix at a time, a stride size of 2 slides the filter down two rows and a stride size of n slides the filter

down n rows at a time. The greater a stride the smaller the feature map outputted by a filter and so the filter may not learn the patterns in an input matrix as easily as if it considered every row of the matrix.

3.5 Overfitting and Underfitting Models

Once a Neural Network Model has been trained, there must be multiple evaluations performed to ensure that the model is not overfitting or underfitting to the training set. Overfitting is when the model has a high variance, meaning it focuses too much on learning every little pattern in the training data. An overfitted model will underperform on any data that is not the data it was trained on making it essentially a useless Machine Learning model. Underfitting is the case when the model can not even fit to the training data as it may be too simple. This model is also useless in the real-world. The optimum model learns to generalize to the training data as training iterations occur. It does not learn off the training data. It does not focus on outliers but learns the general shape of the data. The optimum model will perform just as well with data it hasn't seen as it does for data it was trained on because it has learned to generalize to the training data and it truly understands the underlying patterns in the data.
Consider using a Univariate Linear Regression trying to fit to a dataset that is overly complex. This machine learning model would be overly simple and would not represent the shape and patterns in the data. Consider a high order polynomial regression algorithm for a simple problem. It would try to fit for training data point. Figure 8 shows this problem. The linear regression model can barely fit the data and the polynomial regression model fits for every data point. Neither of these models will perform well in unseen data.

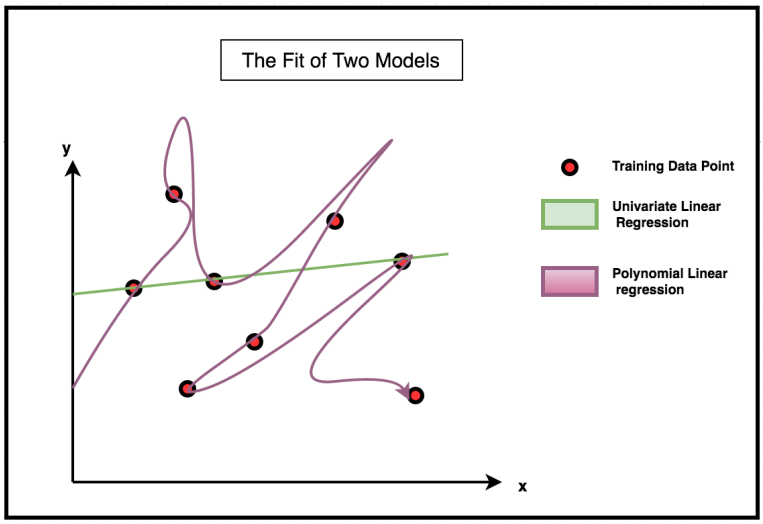

Figure 8: Two models trying to fit to a set of training data points.

To improve the generalization of a machine learning model there are multiple techniques such as L2 Regularization and Dropout Layers. All of these have been used in the optimization of the Neural Network models which have been trained for sarcasm detection for this project.

4 Data Retrieval and Cleansing

The data for this project was acquired from Twitter. The original approach was to retrieve sarcastic tweets by streaming tweets on twitter with specific hashtags such as "sarcasm", "not", "sarcastic" etc. However, it would have been difficult to ensure thousands of "sarcastic" tweets collected were truly sarcastic. Retrieving my own sarcasm dataset and cleaning out noise would be quite a tedious task and because of this, I decided to contact other researchers in sarcasm detection.
My supervisor, Josephine Griffith, was able to retrieve the tweets that Bamman and Smith [1] downloaded from twitter for their research in sarcasm detection. As Twitter restricts the sharing of datasets, I was only able to retrieve the IDs of tweets. The dataset had two columns which were the tweet ID and the corresponding classification label. To download the corresponding tweets, I acquired access tokens to the Twitter API and used these with the Tweepy library. These access tokens were used to authenticate my Twitter account to allow me to make calls to the Twitter API. The twitter ID's were read in one at a time and added to a new text file which contained the tweet and the classification label. The tweet and label were tab delimited and each row was separated by a new line character. Twitter limits the amount of GET and POST requests to their API and so to ensure that the script would not fail once this limit was reached, I allowed for the suspending of the script execution for 60 seconds. This script would sleep for 60 seconds when the Rate limit error was received and then would try again. This allowed the data retrieval script to continuously run until all tweets were retrieved. In total 17,105 tweets were retrieved. The split of sarcastic and non-sarcastic data can be seen in Table 1.
It would be counter intuitive to keep the hashtags in the tweet as they are explicit in declaring the underlying sarcasm in the tweet. "http" links are also removed as these would be too expensive to process. The final result is then a csv file containing the tweet text and label, where the label is 1 or 0 when the tweet is sarcastic or non-sarcastic respectively. For the duration of this document, I will refer to this dataset as Dataset 1 for simplicity.

Upon viewing this dataset, it was evident that this would be a very difficult dataset to learn from. This is due to the fact that all the tweets are conversational and so sarcasm in the tweets is in context of the previous tweets in the conversation.

Example of a Sarcastic Tweet from Dataset 1:
"@TedOfficialPage agreed ! i do it all the time"

The above tweet is classified as sarcastic by the user. However, to any person reading such a tweet without a notion of the context relating to this conversation, this is

actually quite difficult to classify as sarcastic or not sarcastic. If the text is difficult for a human, who has a greater understanding of irony, then for a machine learning algorithm this would be near impossible to classify.
Due to the difficulty that may arise for a Neural Network in learning the patterns in the sarcastic and non-sarcastic tweets in this dataset, I have acquired a different dataset which was collected and prepared for the paper "Sarcasm Detection on Twitter: A Behavioral Modeling Approach"[6]. By contacting Rajadesingan Ashwin, I was able to retrieve the ID's for the all the tweets used in their Sarcasm Detection paper. The methodology for retrieving the tweets for the corresponding ID's was the same as was done for Dataset 1. Using Tweepy, the data retrieval script would read a tweet ID and then retrieve the tweet that corresponds to this ID and append this tweet text to a new text file with it's classification label. To clean the dataset, all hashtags, "http" links and unnecessary new lines were removed. The final dataset had 13,006 labelled tweets. The split of sarcastic and non-sarcastic data for this dataset can be seen in Table 2. For the duration of this document, I will refer to this dataset as Dataset 2. The advantage of this dataset over Dataset 1 is that the tweets are not conversational and so no conversational context is required to understand the underlying the sarcasm in the text.

Example of a Sarcastic Tweet from Dataset 2:
"Boy ive missed these early morning starts..."

In comparison to the example tweet from Dataset 1, the sarcasm in the above example tweet from Dataset 2 is much easier to recognize. One does not require context or an understanding of the user's personal traits to recognize the underlying sarcasm in this tweet. Due to the lack of context needed, the neural network should find it easier to model the sarcastic patterns in this sentence.

Label	**No. Of Tweets**
Sarcastic	8370
Not-Sarcastic	8735

Table 1: Sarcastic/Not Sarcastic split in Dataset 1

Label	**No. Of Tweets**
Sarcastic	6,296
Not-Sarcastic	6,710

Table 2: Sarcastic/Not Sarcastic split in Dataset 2

5 Implementation

The following section describes two experiments which entail the design and implementation of two different Neural Networks, a Shallow Neural Network and a Convolutional Neural Network. This section describes any pre-processing performed on the textual data to represent them in manners that the Neural Networks could interpret e.g. Bag of Words and Word Embeddings. As well as this, I discuss the techniques used to finalize and the Neural Network models for the sarcasm detection problem such as the methods to reduce overfitting of the models to training data.

5.1 Preprocessing Data

One of the biggest advantages that Neural Networks have is their ability to learn any function and compute any function for a given tailored problem. In the case of this project, that problem is Sarcasm Detection in text. In order for a Neural Network to create a function that properly describes this text classification problem of sarcasm, the text needs to be represented in a way that the Neural Network can learn from. One of the simplest ways of representing textual data is the Bag of Words model or Vector-Space Model.

5.1.1 Bag of Words

Bag of words is a model which represents a sentence or document as a vector of words. The bag of words model takes words from documents, ignoring the order in which the words come, and adds these words to a vector of representing a vocabulary of words. The bag of words model can then represent a sentence or even a whole document as a feature which describes the number of times specific words in the vocabulary appear in the sentence or document.
Let

$$D = \{d_0, d_1, ..., d_n\}$$

be a collection of documents. We wish to create a vocabulary of words from all these documents. The bag of words model takes each document d_i and adds each word w_j in this document to the Bag of Words. This is done until we have a bag of words/vocabulary of all the words in all n documents.

How can we use these Bags of Words to represent a tweet as a feature for a machine learning algorithm?
Let t be some text tweet and v be the vocabulary with m words which was created from n documents. To create a feature from t, create a vector of zeros f of length m.

Check the occurence of every word t_j in the vocabulary v. Each i^{th} value in f then represents the occurrence of the i^{th} word in t.

5.1.2 Creating a lexicon from the datasets

Since I will be creating a neural network model for each dataset, all pre-processing for the datasets are done independently of one another. This means that the vocabulary of words retrieved for Dataset 1 is not created using any of the data in Dataset 2 and likewise. The lexicon for Dataset 1 will be called lexicon 1 and the lexicon for Dataset 2 will be called lexicon 2. Using the bag of words model described above, we create a list from all the words in the tweets. Using NLTK's lemmatize function these words in the list are then lemmatized to return them to their canonical/dictionary form. This eliminates repetitions of words e.g. eating, eats, ate all have the same lemma which is eat. The list of words is then just a list of all the words in the tweet dataset reduced to their canonical form. It is not optimal to have a vocabulary of words in which words are repeated. It is also not optimal to have a vocabulary of words containing words that are very often used such as "the" or "I". To solve this, count the occurrence of each unique word in the list of words and put a bound on the occurrence of the words. I have limited the max occurrence of words to 1500 and the minimum occurrence of words to 50, to eliminate words that occur nearly all the time in text and words that very rarely appear. The final lexicon is then a list of unique words within this boundary of occurrence. Table 3 summarises the size of lexicon 1 and 2.

Dataset	**Size of Lexicon(In Words)**
1	420
2	386

Table 3: Vocabulary size from Datasets 1 and 2

5.1.3 Creating features from tweets in the datasets

To create a feature out of the tweets in the dataset, we follow the approach described above for a bag of words model and creating textual features. For a given tweet in the dataset, create a NumPy array of zeros of the same length of the lexicon. For each word in the tweet, check it's occurrence in the lexicon and if it occurs in the lexicon at an index i, then increment index i in the NumPy array created for the tweet.

Example:

Take two tweets:

t_1 : "Hello My Name is Joe"
t_2 : "Joe has a good name"

Using these two documents the bag of words model would create lexicon:

$$L = [\prime Hello\prime, \prime My\prime, \prime Name\prime, \prime is\prime, \prime Joe\prime, \prime has\prime, \prime a\prime, \prime good\prime]$$

To convert t_1 and t_2 to features using this lexicon, check the occurrence of each word in the tweet in the lexicon L. The tweet features f_1 and f_2 are then created from the tweets t_1 and t_2 respectively.

$$f_1 = [1, 1, 1, 1, 1, 0, 0, 0]$$
$$f_2 = [0, 0, 1, 0, 1, 1, 1, 1]$$

Features are created for every tweet in Dataset 1 using Lexicon 1 and similarily for Features in Dataset 2. For each tweet feature we also append the class of the tweet, which is either sarcastic or not sarcastic. We encode the sarcastic class as a '[0,1]' and the non-sarcastic class as '[1,0]'. Assuming the both tweets are non-sarcastic, the feature set F containing all features f would then be represented as:

$$F = [[1, 1, 1, 1, 1, 0, 0, 0], [1, 0]$$
$$[0, 0, 1, 0, 1, 1, 1, 1], [1, 0]]$$

We create feature sets from both datasets and get 17105 features from Dataset 1 and 13006 features from Dataset 2. Now all tweet textual data has been represented in a manner in which Neural Networks can automatically map the patterns of sarcasm.

5.2 Experimental Setup

5.2.1 Three-Way Split

In order to evaluate the trained model of the Neural Network, I performed a three way split on both datasets. It would not make sense to train the model on 100% of the data and evaluate on that same data as this would be counter-intuitive. This would be similar to learning off answers for an exam when you know every question to be asked in the exam. In this case, you would get nearly 100% in the exam due to the fact you have just learned off the answers to questions that you knew would come up for the exam.
What if the test had questions that you have not studied for?
Given the fact that you only learned off answers to questions you knew were going to come in the test, then you would fail the exam. A model that can predict sarcasm in text would be useless if it could only perform well on data it was trained on. We want to create a Sarcasm Detection model that can generalize to patterns in the text and use the knowledge it learns from training to help it in understanding and identifying the underlying sarcasm in text it's never seen before and to do this a test partition of the dataset must be taken out. This test partition can simulate data the model might see in the future and the models performance on this test set can describe the possible accuracies it may achieve in sarcasm detection in the future. Of course, we need to ensure that the data is split in a way that maintains the statistical distribution of sarcastic and non-sarcastic tweets in the test and training dataset. That is, if the whole dataset contains somewhat of a 50/50 split of sarcastic/non-sarcastic tweets then it is optimal to maintain this split and that the partitions are done so randomly. If the partitions are not done at random, then a portion of the dataset which have all very similar patterns may be chosen for training and so this model would not learn the wider set of patterns that can represent sarcasm in text.
80% of the data is then used for training the model and 10% to test the model.The Neural network model is trained on 80% of data and is evaluated at each training step. Every 50 steps the model is then tested on the 10% of testing data. This is data the model has not been trained on and will show the performance of the model as the training process occurs. However before any training is done, 10% of the data is held out ensuring that this data is completely unseen by any trained model, this is the partition of data that we use finalize the results of the model. The accuracy of the trained model on this dataset represents the possible accuracies of this model on sarcastic data it may see in the future.

Dataset Partition	No. of Tweets
Train	13684
Test	1711
Validation	1710

Table 4: Train/Test/Validation Split for Dataset 1

Dataset Partition	No. of Tweets
Train	10404
Test	1302
Validation	1300

Table 5: Train/Test/Validation Split for Dataset 2

5.2.2 Batches

When a Tensorflow program is run, the computational graph is run and any data required for the graph is passed in through the placeholder variables. The graph then performs all computations specified and returns some result. The Tensorflow graph must be fed data in order for the graph to run. It would be horribly inefficient if the network was trained on one training example at a time. Tensorflow allows the input of data in batches and since Tensorflow is very efficient at handling array operations we can input the training examples in large batches for training. Training by inputting batches of certain sizes can improve the training time by an incredible amount. However, without sufficient computational power larger batch sizes may result in an extended running time. Choosing too large of a batch size may cause the Neural Network model to overfit to the batch inputted as Network variables are only optimized and adjusted after a batch is inputted. For all training, I have chosen a batch size of 300 training examples. At each training step, 300 training examples will be fed to the network as one batch to train the network. The trained model is then tested on the testing set after every 15 training steps.

5.3 Deep Neural Network

5.3.1 Design and Development Of Shallow Neural Network

Once all the tweets had been represented in some numerical vector form, the Shallow Neural Network was designed and developed using Tensorflow to train for the sarcasm detection task. I call this a Shallow Neural Network as the Neural Network was only 3 layers. TensorFlow made the design and development of a Neural Network very easy. Data in a TensorFlow graph is respresented as tensors. Tensors are essentially just a multi-dimensional array which can describe the relationship of vectors in a multi-dimensional space.

Design: A Tensorflow graph can take in an arbitrary amount of samples but the size of each input data must be known. The input layer of the Neural Network passes in the data X which is fixed as the length of the lexicon of which the text features were constructed from. Each value in the text feature then connects to every neuron in the next layer with a corresponding weight. The variables of the Neural Network are the weights and biases at each each layer. The weight vector at a layer maintain a shape of $[i_l, o_l]$, where i_l and o_l are the amount of inputs and outputs for a given layer l. A bias is also added at each layer to help improve the fit of the model. Each layer of the neural network takes the inputs and performs a summation of the weighted inputs from the previous layer and produces an output. The outputs for a neuron on a of a given layer l can be calculated as such:

$$\sum_{i=0}^{n} w_i^l \cdot x_i^l + b^l$$

where x_i^l, w_i^l and b^l are the i^{th} inputs, weights and bias for layer l.
This is computed for all neurons on each layer with respect to the inputs and weights of a given layer. These are then passed into an activation function I've chosen which is RELU(Rectified Linear Unit).

RELU: The state-of-the-art for non-linear activation functions in Neural Networks for text classification and sentiment analysis has been RELU. A non-linear activation function is required as the relationship between text and its corresponding class is non-linear. The patterns for such a relationship cannot be expressed in a linear manner. In the construction of their Deep Neural Network for Sarcasm Detection Amir et al.(2016) made use of this non-linear activation function. Essentially, RELU takes an output value of a neuron and turns it into a non negative number. RELU can simply be represented as a function that chooses the max between 0 and a

$$r = max(0, a)$$

where a is the output after summing the weighted inputs to a neuron.
One of the biggest reasons that RELU is such a popular choice for a non-linear activation function is the boost in computational speed it gives Neural Networks as the function is a simple max function between two numbers. As well as this, RELU allows the Neural Network to evade the "vanishing gradient" problem which makes it difficult for the front layers of a Neural Network to learn. Usually this problem occurs when the outputs are activated by a Sigmoid Function and with very deep neural networks, but it is still a problem that could make a Neural Network very difficult to train.
The outputs of the hidden layer are passed through a RELU function to map non-linear relationships between the inputs and the outputs. The hidden layer then connects to an output layer with only two outputs. The final output layer takes the outputs from the hidden layer and performs the same sum of the weighted inputs. The final output layer only has 2 outputs, whether the input was a sarcastic text or a non-sarcastic text and we call this final vector the vector of class scores z, where the value at the 0 index is the non-sarcastic score of the input to the network and the 1 index is the sarcastic score. But the final outputs will be of the form $[\phi, \Upsilon]$, and ϕ and Υ are some values that may be hard to interpret. To transform the values of this final output vector to values that are easily interpretable, I've taken the outputs of the output layer and passed them into a SoftMax Function.

SoftMax: Softmax essentially converts this output vector to values between 0 and 1 and ensures that they add to 1. The advantage of this of course is that the values in this vector now represent probabilities of the classes. This is done by normalizing the values in this vector.
Let z be an n-dimensional real valued vector of class scores. The softmax output of z can be calculated as such:

$$soft(z)_j = \frac{e^{z_j}}{\sum_{d=1}^{n} e^{z_d}}$$

for all $j = 1, ..n$
This output is more easily interpretable and shows the probability of a class j given some input to the Neural Network. A prediction is then the max value of this class score vector as it has the highest probability of being the corresponding class for an input. This can be calculated as the $argmax$ of the output of the softmax function given z.

Calculating Error: As an error function, I have chosen Cross-Entropy error. Classification error is a simple way to determine the amount of the inputs the classifier

predicted wrong classes for. However, classification can be naive error mechanism in a way as it does not consider how wrong the classifier was. For a set of n training inputs of which r were correctly classified and $\hat{r}$ were incorrectly classified, the error would simply be

$$Error = \frac{\hat{r}}{n}$$

This is just the percentage of the samples that were classified incorrectly. For the first incorrectly classified sample this does not take into account how close the softmax outputs were to the actual expected outputs. Cross Entropy error takes into account how close the predictions actually were and is a much more reasonable measure of the inaccuracies of the networks predictions. The cost/loss/error is just the average cross-entropy error for all samples.

Optimizing the Weights: The most basic choices for back propagating gradients and optimizing network variables is Gradient Descent but the network optimizer I have chosen is Adam Optimizer[5]. Choosing gradient descent might cause problems in converging to a minimum error. With Gradient descent optimizers, more hyper-parameter tuning is required to guide the network to the weights which lead to the least amount of error. The hyper-paramaters of a network are of course just the parameters that will guide the networks learning such as learning rate, weight decay etc.
As mentioned in "Practical Recommendations for Gradient-Based Training of Deep Architectures"[2], some of the most important hyper-parameters of the Network are the initial learning rate and the "Learning rate Schedule" which allows the learning rate to decay as we get closer to the minimum result. But choosing the optimum hyper-parameters to allow for a faster convergence of the network is actually a very difficult task and is proven to be quite tedious. By using Gradient Descent, we know that we either have to increase or decrease the weights but it can be difficult to access how much we'd like to adjust the weights given the error. Adam Optimizer allows the network to converge faster to a point of minimum error by using momentum which essentially takes a portion of the previous update and adds it to the current update allowing the slope to move faster in a direction of least error. The Adam Optimizer can also alleviate the "Vanishing Gradient" problem in networks that are increasingly deep by giving larger updates to parameters of the network that frequently get little to no updates. Adam Optimizer reduces the manual effort of tuning hyper parameters which requires a lot of time and evaluation with other Gradient Descent Optimizers.

Training The Network:

Note: Before training is performed, the 90% of data remaining after 10% was held out is partitioned so that an additional 10% is taken out for testing.

50 Training Epochs occurred meaning the training dataset was fed into the network 50 times. However, feeding the whole dataset into the network to train would cause the network to overfit and may be computationally expensive so as mentioned before the dataset is split into batches of 300 examples. The dataset is shuffled and split into batches of at most 300 examples, and at each training step of an epoch this batch of 300 examples and their corresponding classification labels are fed into the network for training. Once a batch has been fed-forward through the network, the average cross-entropy error is calculated using the predictions by the network and the correct classification labels.
The Adam Optimizer then computes the gradients of this error with respect to the weights in the network and applies the gradients to all the trainable variables in the network. The training iteration is attached to a TensorBoard summary file for graphing.
To know how the network performs on unseen data as it is trained, the network is also tested on a testing data of 10% at around every 50 training steps. The performance of the network on this data can show the accuracies that this Neural Network model may have on sarcasm detection. These results are also saved to a TensorBoard summary file for comparison with the training results.

5.3.2 DNN: Finalising The Models

A Neural Network model would be trained on each of the respective datasets, dataset 1 and 2. However, it is important to ensure that the models are well tuned to maximize not only the accuracy on the training dataset but also the testing dataset. When choosing a final Neural Network Model, one must be aware of how well the model actually fits the problem. A good "fit" of a model does not mean 100% accuracy on data that it's training on but rather a model that has a similar output on unseen data as with the data it is training on as this represents the Neural Network's understanding of the underlying patterns that lead to answers to a given problem. A model that performs as well on unseen data as it does on training data is a model that has generalized from learning and has not just "learned-off" the answers that provides the highest accuracies on training data. A type of model that is optimal and ready to be exposed to the real world will not underfit/overfit to the training data. Neural Networks have a tendency to overfit to training data as training occurs and this can be seen in the learning curves for both dataset 1 and 2 in figure 9 and figure 10. It is evident that the Neural Network model has overfitted to the training data as the there is a large difference in the training and test accuracy.

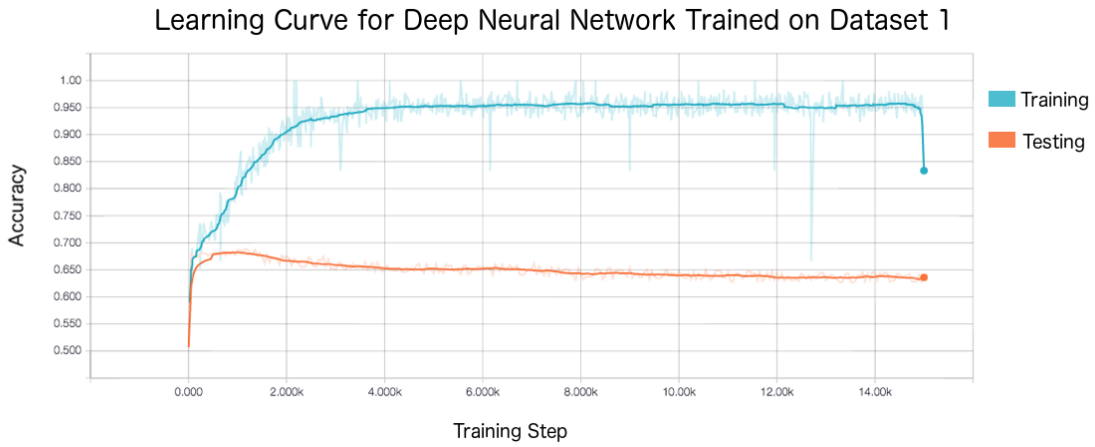

Figure 9: Learning curve when Training model on Dataset 1(Smoothed)

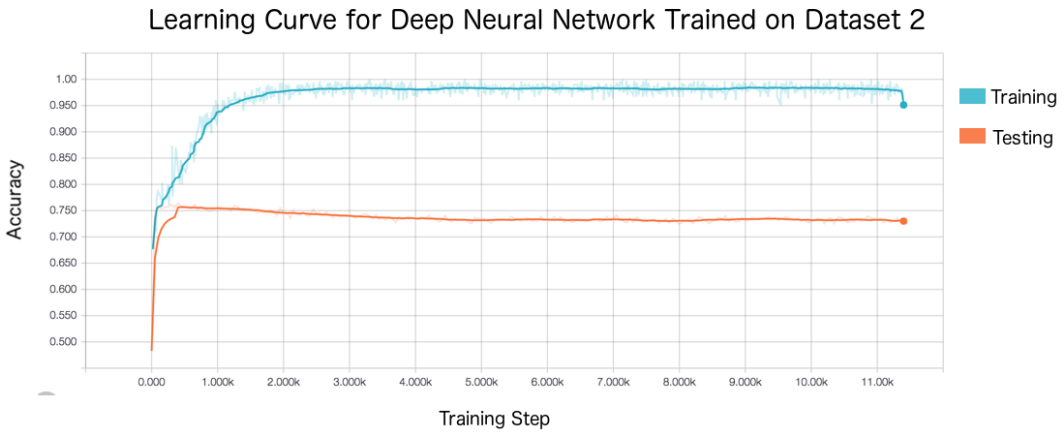

Figure 10: Learning curve when Training model on Dataset 2(Smoothed)

The next sections show the ideas and techniques which were used to create a best fit for Neural Network Models trained on datasets 1 and 2 to avoid overfitting.

L2 Regularization: One of the most common ways to reduce overfitting is using L2 Regularization. Overfitting occurs when there is too much variance in the trained model, that is it learns every little pattern in the training data and does not generalize. There is such thing as a variance/bias trade-off and this is what Regularization may control. By adding a regularization penalty, it's possible to control the variance of the model. By adding a bias to the model when optimizing the weights, the large network parameters are penalized and are shrunk towards 0. The regularization lambda λ is a hyperparameter for a Neural Network that is adjusted to determine the scale of a penalty to each weight in each layer.

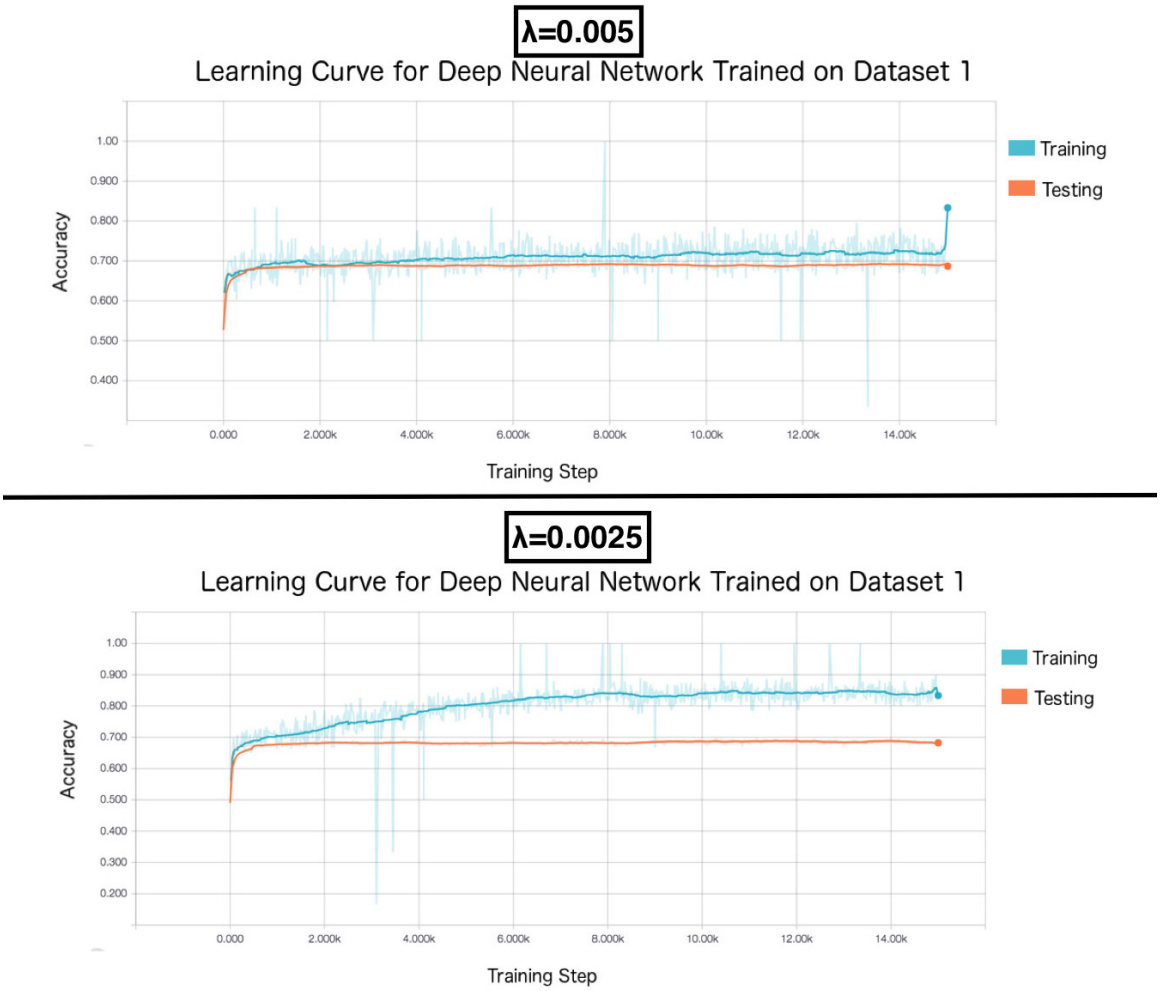

Figure 11: Learning curve when Training model on Dataset 1 for 300 epochs with L2 λ penalty set to 0.005, 0.0025(Smoothed)

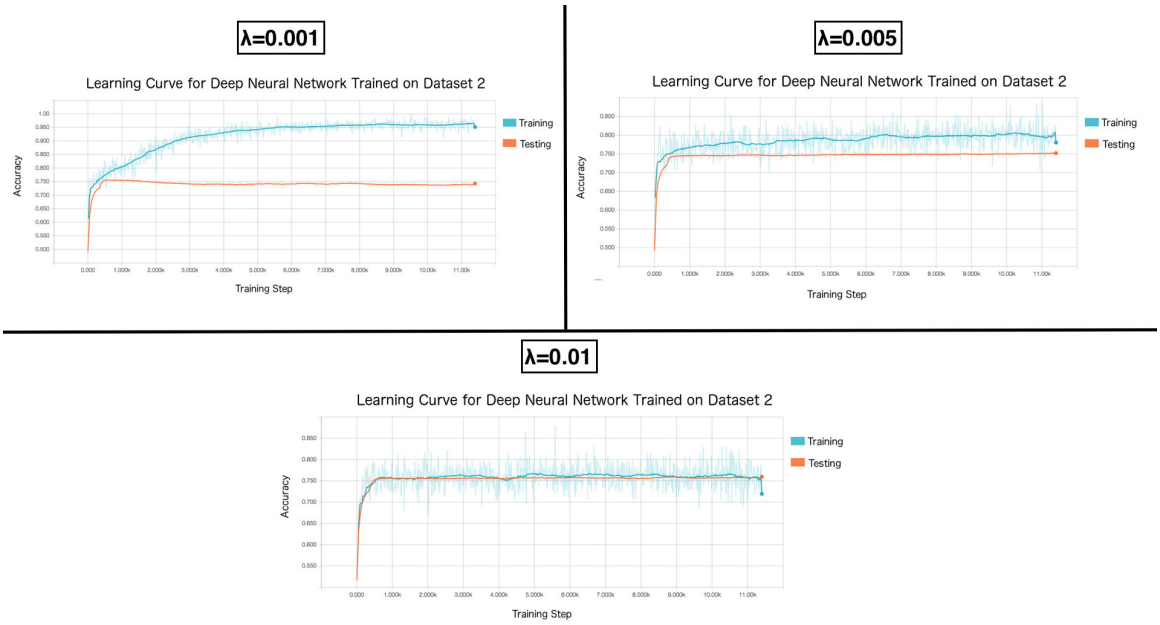

Figure 12: Learning curve when Training model on Dataset 2 for 300 epochs with L2 λ penalty set to 0.001, 0.005, 0.01(Smoothed)

It's evident that adding L2 penalties helps reduce the overfitting of the Neural Network model on both datasets. This can be seen in the reduced difference in the training accuracy and testing accuracies. The L2 λ that is chosen for the each model is the model that provides the best fit. For the case of dataset 1, I have chosen an l2 λ of 0.005 and for dataset 2, I have also chosen 0.005 as the λ.

Dropout: Another technique to improve the learning of a Neural Network is adding a dropout layer, which turns off a percentage of the neurons on a layer to reduce the dependency neurons will have with one another in learning. These neurons which are ignored for training are chosen at random by a dropout probability. It teaches neurons to form reasonable results on their own and to not rely on other neurons. The weights linking to neurons settle to their purpose in the network and don't change to independently learn. This means that specific neurons will rely on their neighbouring neurons. Dropout "prevents units from co-adapting too much". When neurons are turned off, the other neurons must try harder to try and learn from the data.[7]

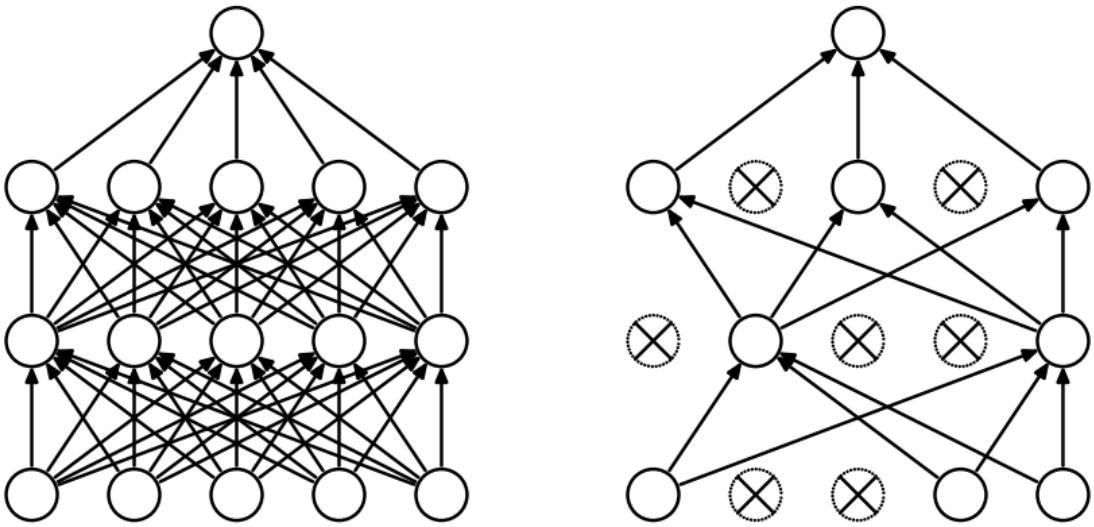

Figure 13: Dropout Neural Network Model
Source: "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" [7]

The result of adding a dropout layer to both neural networks can be seen in figure 14 and 15, with a dropout probability of 0.5 meaning only half of the neurons in the hidden layer are turned on for training. Of course, when testing the model on the testing data all neurons are turned on which means the dropout probability is set to 1.0. Adding a dropout to both Neural Network models haven't made a massive impact on the learning of the model but it is still good practice to maintain a dropout layer to improve the learning of the individual neurons in the network.

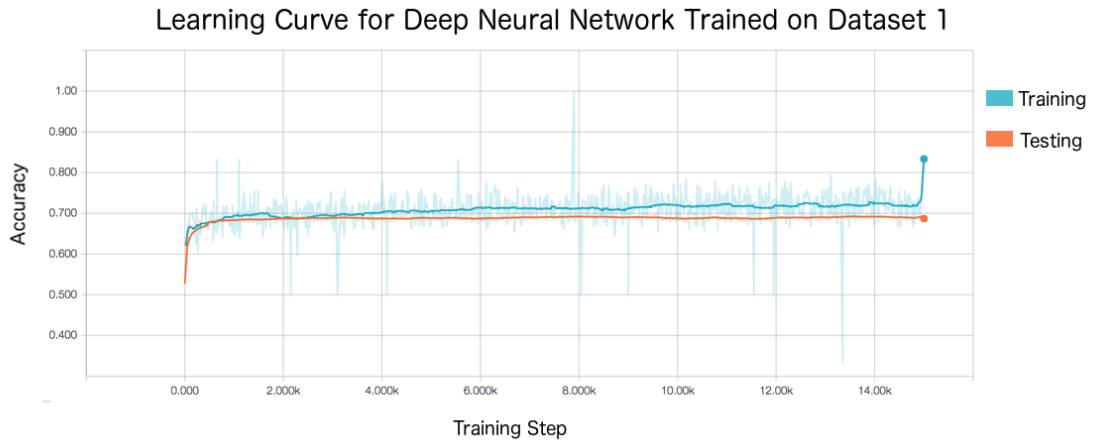

Figure 14: Learning curve when Training model on Dataset 1 for 300 epochs with L2 λ penalty set to 0.005 and Dropout probability of 0.5(Smoothed)

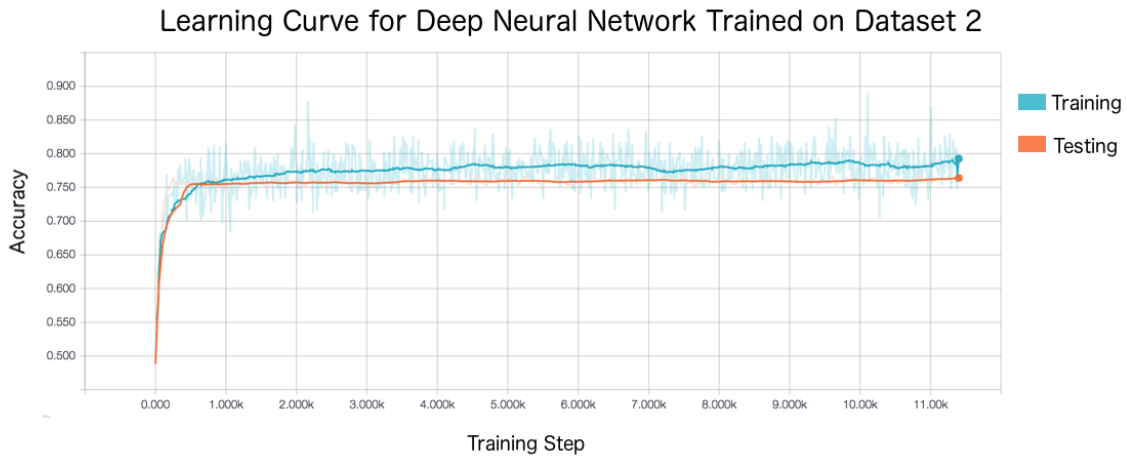

Figure 15: Learning curve when Training model on Dataset 2 for 300 epochs with L2 λ penalty set to 0.005 and Dropout probability of 0.5(Smoothed)

Early Stopping: Evidently, the training process does not need to continue after a certain amount of training steps. The point at which we can decide to stop the training process is generally the point at which the testing accuracy and training accuracy is no longer improving, or the point before the training process starts to overfit the model. For the model trained on Dataset 1, the training process can be stopped anytime after 1500 training steps and the model trained on Dataset 2, the training process can be stopped after the same amount of training steps.

After all steps to improve the fit of the Neural Network models, the final model trained on Dataset 1 peaks at around 69% accuracy on the testing data and the final model trained on Dataset 2 peaks at around 76% accuracy on testing data. These models are further evaluated on the hold-out datasets taken out before any training occurred. The results of these can be seen in the section 6.

5.4 Convolutional Neural Network

5.4.1 Design and Development Of Convolutional Neural Network

Although some of the Network components from the Shallow Neural Network are used in this Convolutional Neural Network, the general design and structure is quite different. With this Convolutional Neural Network we want to show the capabilities of a Convolutional Neural Network in automatic feature extraction as well as classification from tweets. The structure of the features inputted to the Shallow Neural Network were of the same length as the lexicon. Each feature was of the length of the lexicon. The problem with this is that this textual representation is quite sparse. Given that most tweets are under 40 words and each feature is represented by a vocabulary of over 350 words, we have features that are mostly zeros. A Convolutional Neural Network will not learn from such sparse data. As well as this, the CNN will also not be able to learn the true patterns in the textual data. Imagine a filter that covers 3 words at a time, the outputs would be quite small given the fact that each word feature is almost 90% zeros. Automatic feature extraction will be quite difficult with such a sparse set of input features to a Convolutional Neural Network.
For the construction of the inputs to a CNN model for Sarcasm Detection, I propose an embedding layer which takes the textual data and learns word embedding representations for each word in the vocabulary.

Preprocessing for CNN: Just as has been already done for the Shallow Neural Network, the tweets from datasets 1 and 2 are prepared so that in one list we have the tweets and in another separate list we have each corresponding label. Both lists are of the same length as there must be exactly the same amount of labels as there are tweets as there must be a label for each tweet. Using TensorFlows vocabulary processor, a vocabulary is created equal to all the unique words in the dataset.

Dataset	Size of Lexicon(In Words)
1	37863
2	13537

Table 6: Size of Lexicons Created by Tensorflow Vocabulary Processor for each Dataset

The above table represents the size of the vocabulary created for each dataset. These lexicons are much larger than the ones used in the SNN(Shallow Neural Network). I chose to not remove the stop words from the lexicon to see the capabilities of a CNN

in feature extraction given a minimal amount of preprocessing. Each tweet is then represented as a bag of words model as was done for the SNN but we set a fixed size for each tweet feature as the maximum length across all tweets. This differs from our SNN input features as they are not of the same size as the lexicon. Given the sizes of the lexicons created by the vocabulary processor, creating features as was done for the SNN would make the learning process very difficult. As well as this, the model would be very prone to overfitting.
Let the maximum tweet length be m, then each tweet feature f will be of length m, where each entry of this vector f_i represents a word by it's corresponding index in the lexicon.

Example:
Let a lexicon l be

$$l = [' Hello', ' My', ' Friend']$$

and let tweets $T = \{t_1, t_2\}$, where

$$t_1 =' Hello\ Friend'$$

and

$$t_2 =' Hello'$$

To represent these tweets t_1 and t_2 as features, let f_1 and f_2 be of the same length as the longest tweet in T. In this example, the longest tweet has 2 words. Transform the tweets to their feature representation by taking each word in the respective tweet and checking it's index position in the lexicon. The i^{th} word in the tweet is then represented by the i^{th} position in the feature representation. The value at the i^{th} position of the feature representation of a tweet is the index position of that word in the lexicon l. Once the transformation is completed, the features are then:

$$f_1 = [1, 3]$$

and

$$f_2 = [1, 0]$$

These transformations are performed for each tweet in each dataset. For dataset 1, the maximum tweet length was 33 and so each tweet feature from dataset 1 is represented as a list of length 33. For dataset 2, the maximum tweet length was 36 and so each tweet feature from dataset 2 is represented as a list of length 36. The values for each word in each tweet feature from Dataset 1 is from the vocabulary created from Dataset 1 and the values for each word in each tweet feature from Dataset 2 is from the vocabulary created from Dataset 2. These features have a much lower

dimensionality than the features inputted to the SNN and there are very few values in the features which are zeros. The only zero values will be the values at the end which are padded on to reach the maximum tweet length. If a tweet is shorter than the maximum tweet length then the remaining values in its feature representation are zeroes which are padded at the end of the list.
Once the tweets had been transformed to their bag of words representation the TensorFlow Convolutional Neural Network graph was designed.

Embedding Layer: To learn the relationships of the words in the lexicon, instead of inputting the textual features directly to a convolutional neural network, we add an embedding layer before the convolutional layer. This embedding layer will then represent each word inputted as a word embedding. A simple bag-of-words model will not show the deep patterns present in textual features. Every word in the textual feature is represented by a d dimensional vector. The word embeddings are trained so that they best represent the inner product between words and so in this d dimensional space the words in our lexicon should be represented such that similar words are close to one another and dissimilar words are farther apart. Words that are nearer to one another will have a large inner product and words far apart will have a smaller inner product with one another.
In the case of the CNN's implemented for this project, the embedding dimensionality was set to 128 dimensions. Choosing a higher dimensionality showed little improvements and made training the word embeddings harder and choosing a lower dimensionality gave poor results so 128 dimensions was chosen for word embeddings in the CNN's in this project.
A single word in the tweet feature will be represented by a word embedding representation with d dimensions. This is done for all words in the tweet feature. These word embeddings are then concatenated together to form a word embedding matrix representation of the tweet feature which has m rows and d columns. The values in the embedding of each word are optimized in the process of training the Neural Network just like any other Network variable with the aim of representing the words as accurately as possible in d-dimensional space. Figure 16 describes the process of taking a tweet feature of length m and transforming it to a word embedding matrix with d columns and m rows.

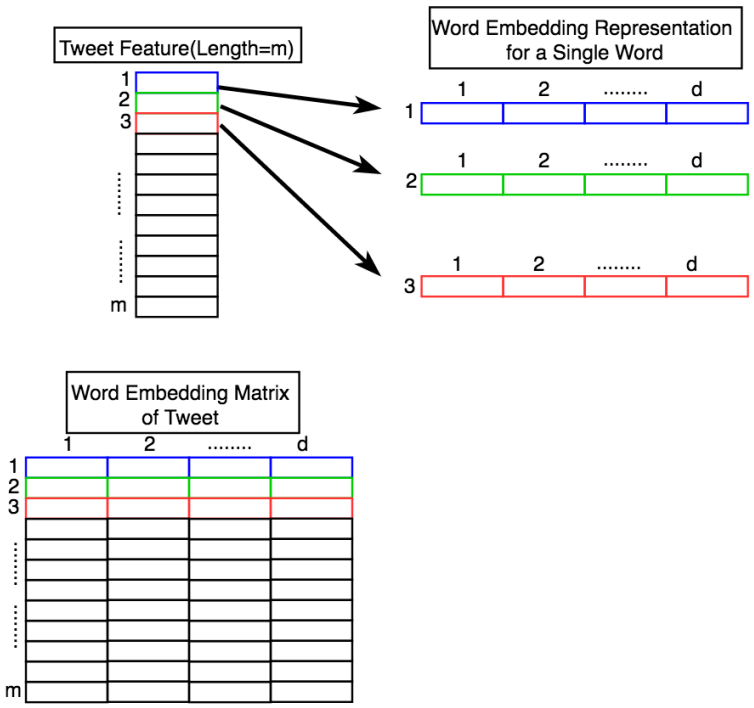

Figure 16: Tweet feature of length m converted to a word embedding matrix of shape $m \times d$

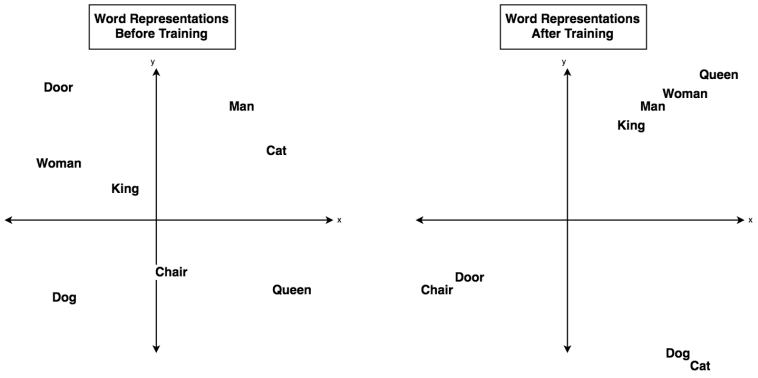

Figure 17: Words in a 2-Dimensional Space before and after the training of their word embeddings. Words that are similar are grouped together

Figure 17 shows the possible results of training word embedding representations of

words. This example is in a 2-Dimensional space. It can be seen that after training the word embeddings, words that are similar to one another are in the same region such as "Man" and "Woman", "Woman" and "Queen" etc. The words that are dissimilar from one another are then moved into regions farther from one another such as "Chair" and "Dog".

Convolutional Layer: For the CNNs developed for this Sarcasm Detection, I have only added one layer of convolutions. However, there are three types of convolutional filters of three different heights $[3, 4, 5]$. The width of the filters is the same as the dimensionality of the word embedding matrix. One of the filters cover 3 words at a time, another covers 4 words at a time and another filter covers 5 words at a time. This is to get multiple different representations of the textual patterns. As well as this there are 64 of each filter. By having multiple filters, we hope to learn as many textual features that suggest sarcasm as possible. A convolutional layer has these 3 filters of which there's 64 of each. The values in the filters are randomly generated when the Network is run just like any other weights in a Neural Network. Each filter will convolve over the input word embedding matrix from the embedding layer to create multiple convolved features. The convolutional filters have a stride of 1, and so as the filters slide down the input matrix performing convolutions they will move down one row at a time. These filters are automatically learned as part of the training process. So after each training iteration, a filter can more easily find features in the input word-embedding matrix than before when the filters were just randomly generated.
After a filter convolves over the input matrix, the output feature vector of the convolution is passed into a non-linear activation. The following description of the design of the CNN for Sarcasm can be seen in Figure 18. In this case, just as with the SNN I have chosen a RELU activation function which changes all negative values in the vector to a zero. 1-Dimensional Max pooling then occurs over this vector to choose the largest value. This value is then concatenated as an entry to the single final vector to be inputted to the next layer. The final output of the convolutional layer is a vector of all the concatenated maxpooled results from all the filters.

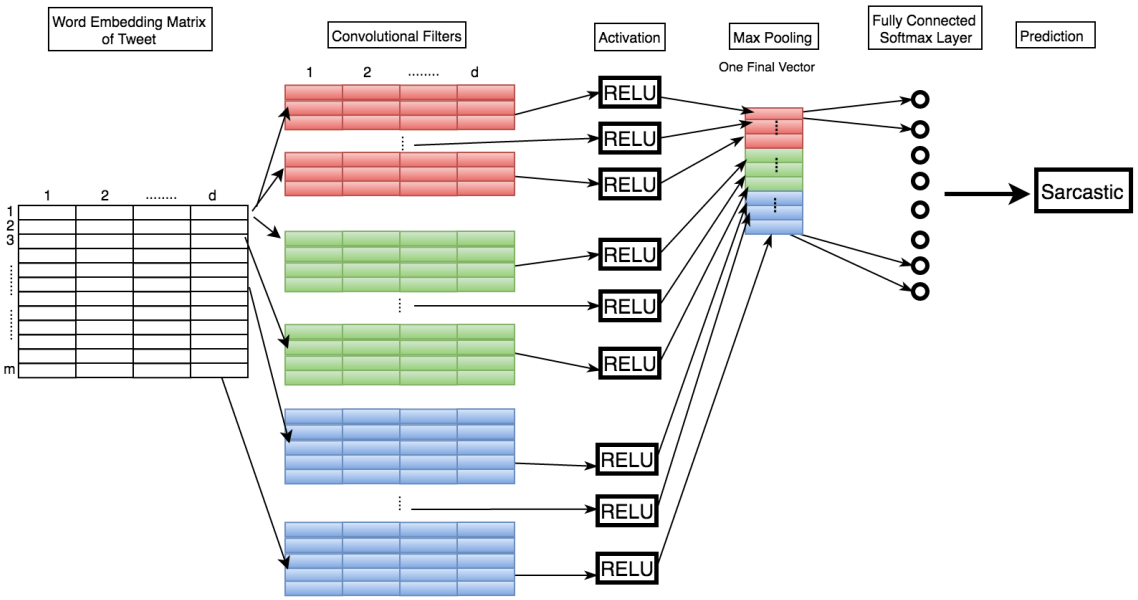

Figure 18: CNN Structure. A convolutional layer with 3 different shaped of filters. A RELU Activation function and a 1-D Max Pool to create a feature vector for the Fully Connected Layer which then produces a classification prediction.

Fully-Connected Layer: The fully connected layer takes in a vector which is the concatenated maxpooled outputs of the convolutions. Every value from this vector is weighted as it enters a Neuron in the fully connected layer. The Neuron takes the weighted inputs and sums them together. The output of the Fully-Connected layer is a 1×2 vector, which represents the prediction of the Neural Network. This output is passed into a Soft-Max function to normalize the values so that each value in this vector is between 0 and 1 and sums to 1. This output can then be interpreted as probabilities of classes. If the output of the soft-max fully connected layer is [0.1,0.9], then the Network is 90% sure that this input is Sarcastic and only 10% sure that the input was Not Sarcastic. The predicted class is then the array entry which has the highest value.

Cost Calculation and Optimization: Just as for the SNN, the cost function for this CNN is Cross-Entropy. The same reasons apply as to why this was chosen as the cost function for this network too. To summarize, Cross-Entropy calculates the closeness of a prediction, not only the correctness of a prediction. Cross-Entropy

error will take into account how wrong a classification actually was. Consider the cost function chosen was simply Classification error, and let $y_1 = [0, 1]$ be the true classification and $\hat{y_1} = [0.51, 0.49]$ be the prediction for input 1.
Classification error would simply say that $\hat{y_1}$ is incorrect and the error for this instance is then 1. However, it's evident that the classification was actually quite close to the true classification. Cross-Entropy Error will measure how wrong this classification actually was and produce some real-valued output. The cross entropy error of a networks predictions can be calculated as

$$C(Y, \hat{Y}) = -\frac{1}{n} \sum_{i=1}^{n} y^i ln(\hat{y}^i) + (1 - y^i) ln(1 - \hat{y}^i)$$

where $Y = \{y_1, y_2, ..., y_n\}$ are the true labels for the inputted examples to the network and $\hat{Y} = \{\hat{y_1}, \hat{y_2}, ..., \hat{y_n}\}$ are the predicted classes by the network for these inputted examples. Cross-Entropy Error leads to more granular optimizations on the Network variables as it is aware of the closeness of predictions. With much more granular optimizations, the network can converge to a point of minimum error much easier.

To Optimize the network paramaters such as the weights in the Fully-Connected Layer, the convolutional filters and the Word Embeddings in the first layer the network needs some way of calculating gradients and backpropagating them in the network. For the CNNs designed in this project, the optimizer chosen was an Adam Optimizer. I have chosen the Adam Optimizer for the CNNs as I did for the SNNs for the same reasons. The Adam Optimizer utilizes various techniques to reduce the amount of hyper-parameter tuning required for the Networks convergence to a point of minimum error. In general, Adam Optimizer makes the optimization process of a Neural Network easier as tuning the extensive amount of hyperparameters of a Neural Network, such as "learning rate" and "rate of decay", to ensure the convergence to minimum error can be quite tedious and difficult.

Training The Network:

Note: Before training is performed, the 90% of data remaining after 10% was held out is partitioned so that an additional 10% is taken out for testing.

The training process is the same as was executed for the SNN, utilizing batches of 300 examples to train the CNN at every iteration then calculating the average cross-entropy error for the iteration. The Adam Optimizer then computes the gradients of the error with respect to the Network variables. The Adam Optimizer then adjusts the trainable network variables with respect to the amount of error they caused. The

Cross-Entropy Error and Accuracy at each training step is saved to a TensorBoard summary file for viewing.
To validate the network on unseen data as it's training, the testing data of 10% is fed into the network to evaluate the networks progress around every 50 training steps. These results are also saved to TensorBoard summary files for viewing.

5.4.2 CNN: Finalising The Models

Just as was done for the Shallow Neural Network in experiment 1, there are various techniques that need to be utilized to ensure an optimum fit of the CNN model to the Sarcasm Detection problem. This section describes the techniques used to reduce the overfitting of the model to the training data from Dataset 1 and Dataset 2. As this network is much deeper than the shallow network implemented for the first experiment, it is more prone to overfitting. The SNN only had 3 layers while this network has an input layer, an embedding layer, a convolutional layer with multiple types of filters then finally a fully connected layer and output layer. With the scale of this network, overfitting can be a massive problem. By examining the learning curves in Figure 19 and 20, when training the CNN model on dataset 1 and 2, it's evident the model has overfitted on the training partition of both datasets. Controlling the overfitting of the CNN proved to be a more difficult task than controlling the overfitting of the SNN. The approaches described in this section to control overfitting are L2 Regularization and Dropout layers.

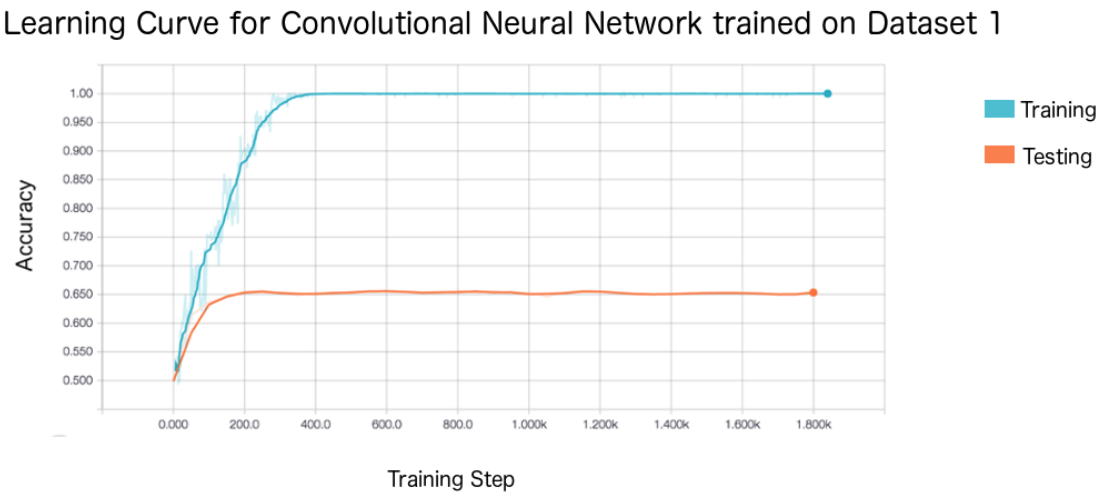

Figure 19: Learning curve when Training model on Dataset 1 for 200 epochs. This model is clearly overfitting due to the large difference between training and testing accuracy.

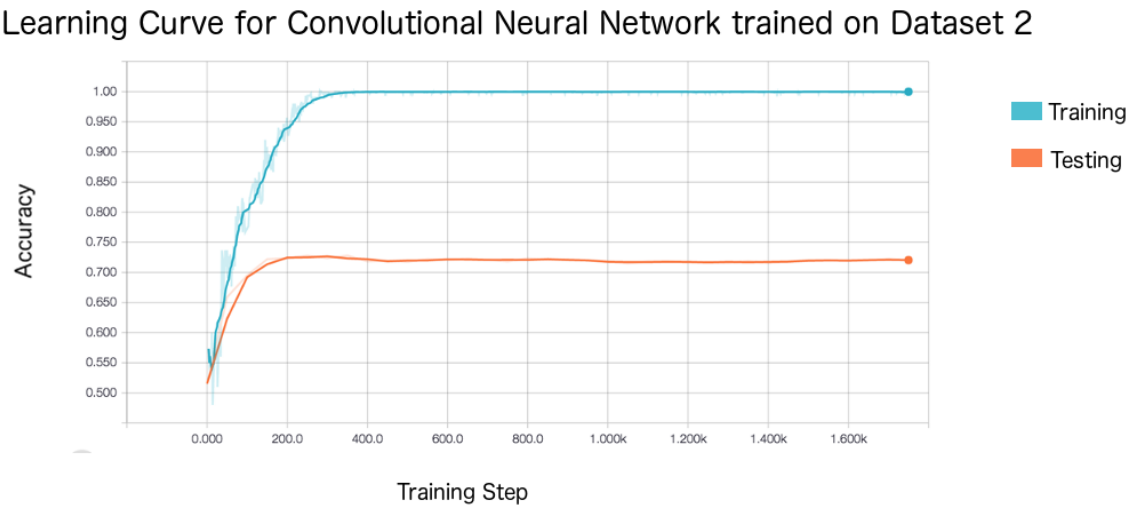

Figure 20: Learning curve when Training model on Dataset 2 for 200 epochs. This model is also clearly overfitting due to the large difference between training and testing accuracy.

L2 Regularization: The process of choosing an L2 λ to reduce the overfitting of the CNN models while ensuring that the model still converged to a point of minimum error was much harder for the CNN than for the SNN. The L2 penalties chosen were much larger than the ones applied to the SNN model. This is because the CNN is a much deeper network and thus requires heavier weight penalties so that they actually affect the variables at the front of the Network such as the embedding weights. Figures 21 and 22 show the learning curve when training on dataset 1 and 2 with different L2 penalties. The final L2 penalties chosen for the models trained on dataset 1 and 2 is 0.7. I have chosen $\lambda = 0.7$ as it reduces the overfitting while still maintaining the accuracy of the model. With a higher L2, the models have problems in converging to points of minimum error. From observing figures 21 and 22, it's clear the models are still overfitting on the training data. To reduce this overfitting even more, I have implemented dropout layers.

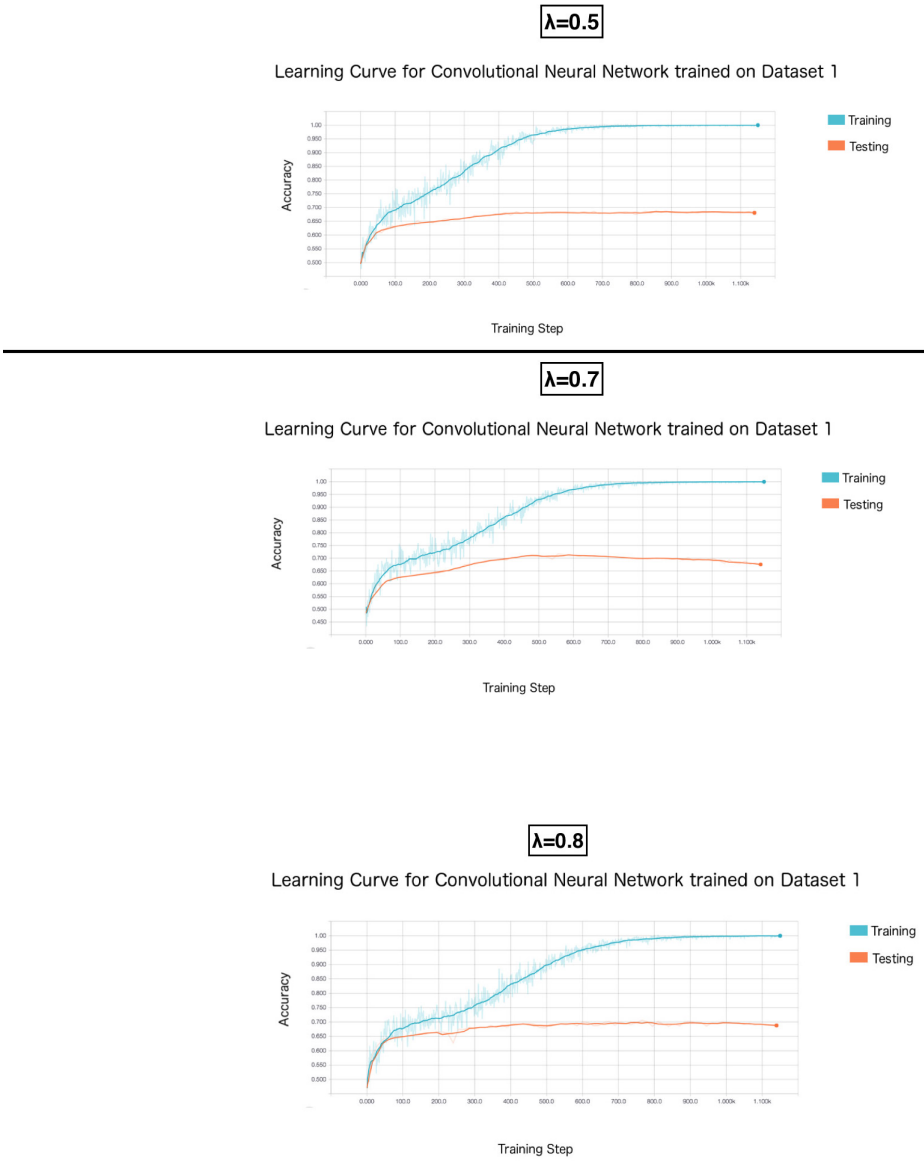

Figure 21: Learning curve when Training model on Dataset 1 with L2 λ penalty set to 0.5, 0.7, 0.8(Smoothed)

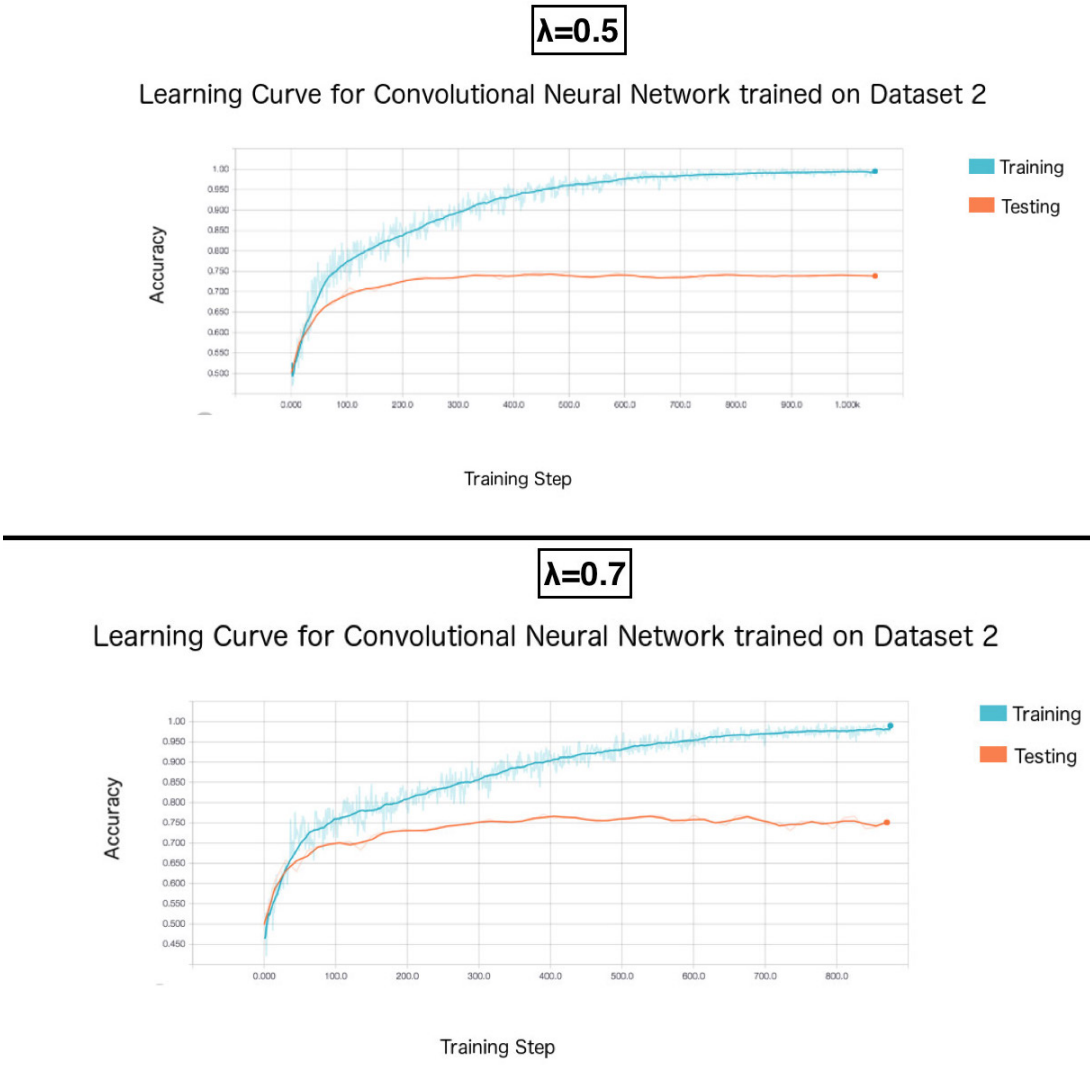

Figure 22: Learning curve when Training model on Dataset with L2 λ penalty set to 0.5, 0.7(Smoothed)

Dropout: To reduce co-adapting of variables in the CNN, I introduce a dropout layer after the convolutional layer and after the fully connected layer. With a given dropout probability, only this amount of neurons are used. This puts pressure on the other set of variables in that layer to learn as much as possible.
The effect of introducing a dropout layer to a convolutional layer is like turning off a portion of the filters. Turning off filters in the convolutional layer will force the remaining filters to try and learn their own features. This means there won't be hundreds of filters working together to search for one feature in the data. There could now be hundreds of filters looking for different features in the data. Essentially, a dropout layer can improve the diversity and accuracy of the features.
Initially the cross-entropy error will be high, but as training occurs the individual network variables will improve and hopefully they won't overfit on the training data. Figures 23 and 24 show the experimentation of different dropout probabilities when training the models on dataset 1 and 2. These models are also regularized by an L2 penalty of 0.7. For the model trained on dataset 1, I have chosen a dropout probability of 0.5 and for the model trained on dataset 2 I have chosen a dropout probability of 0.475. These chosen dropout probablities were chosen as they provide the best fit.

Note: Dropout Probabilities of lower than 0.475 are not shown as the CNNs were unable to learn with such a low dropout probabilities. It's evident from the learning curves shown that dropout layers have greatly reduced the overfitting of the models and has improved the generalization of the models by an incredible amount.

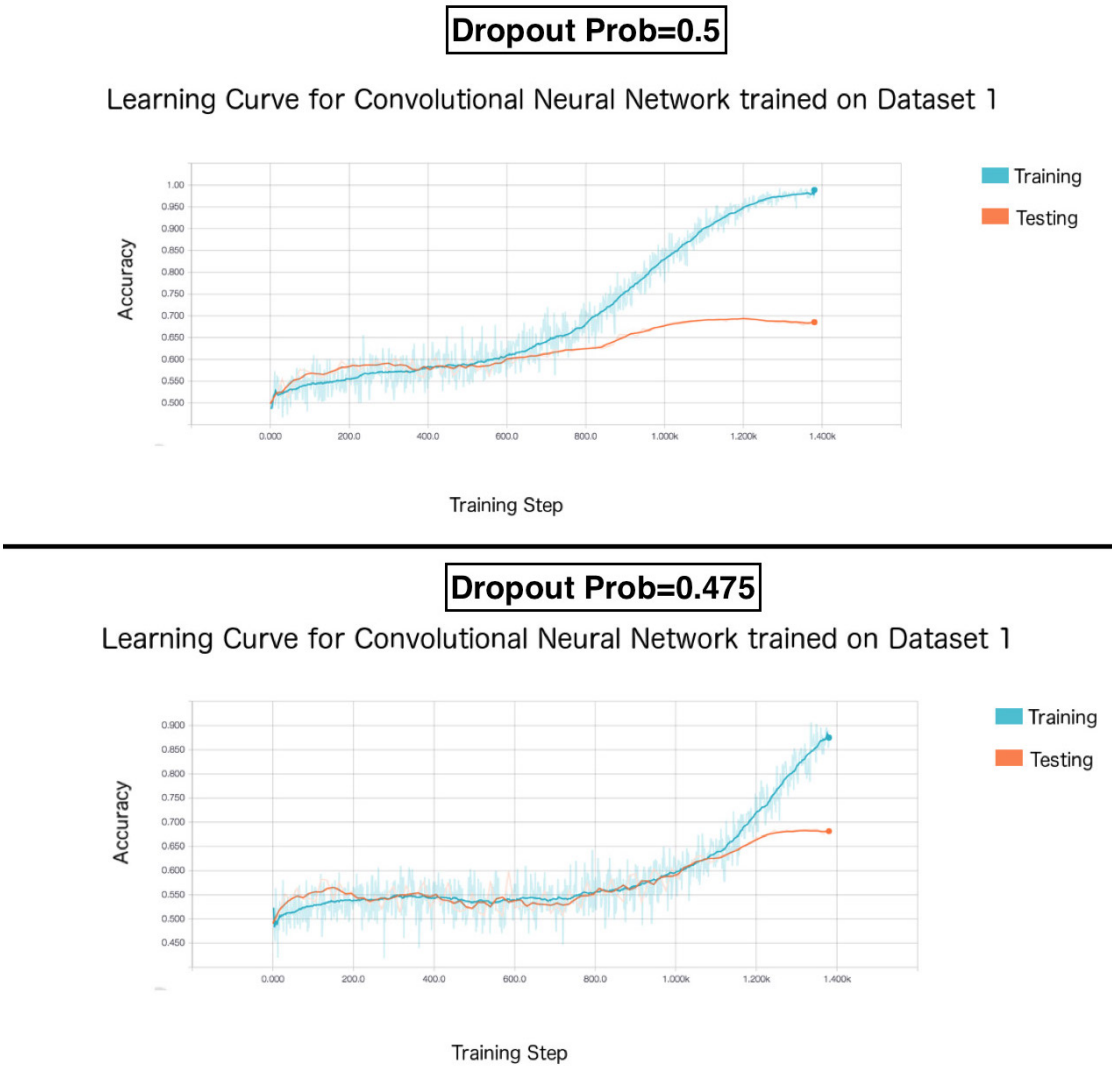

Figure 23: Learning curve when Training model on Dataset 1 with Dropout probabilities of 0.5 and 0.475 and L2 λ penalty set to 0.7(Smoothed)

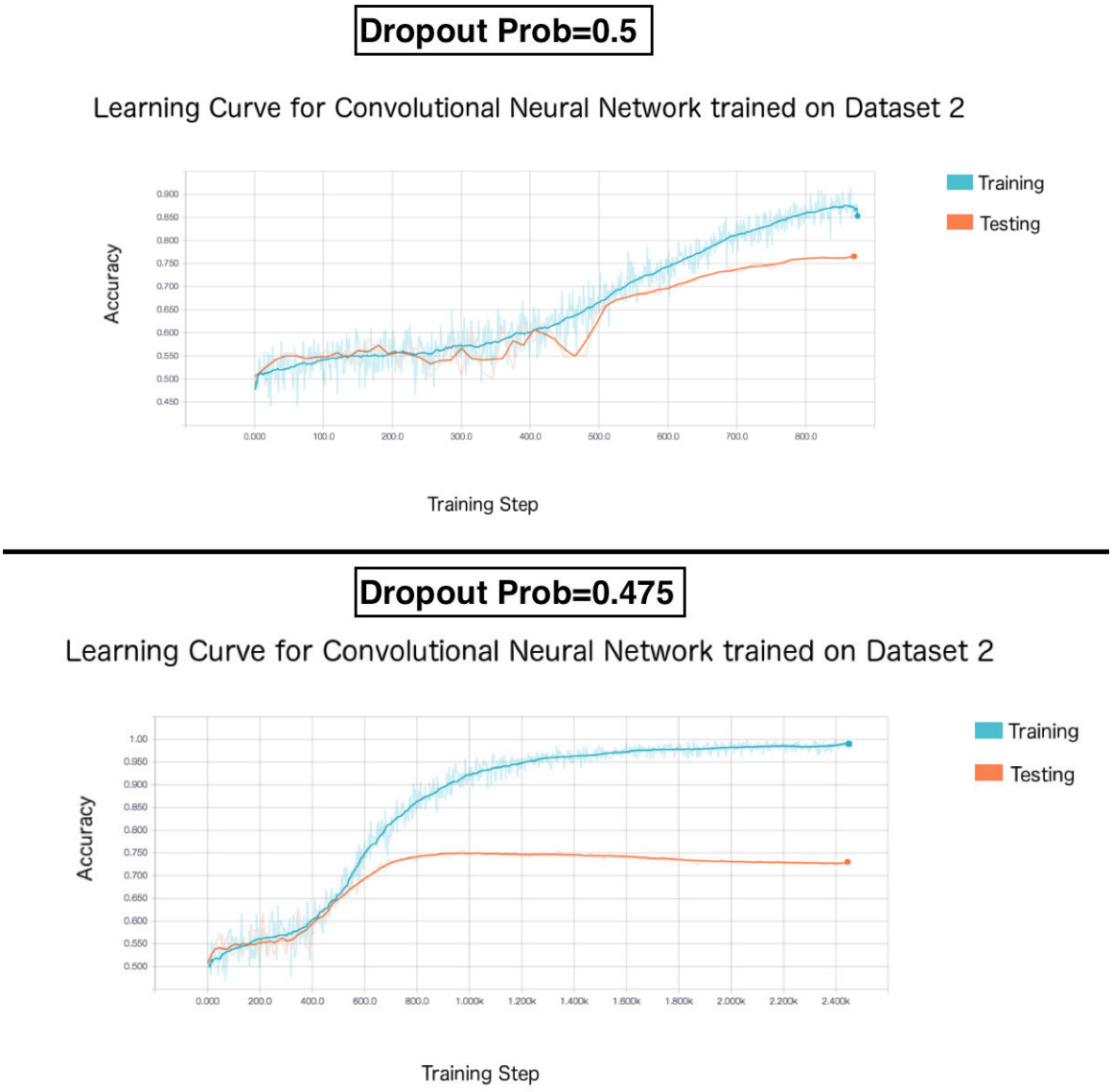

Figure 24: Learning curve when Training model on Dataset 2 with Dropout probabilities of 0.5 and 0.475 and L2 λ penalty set to 0.7(Smoothed)

Early Stopping: Just as with the SNN in the first experiment, the training process does not need to continue after a certain amount of training steps. It is a good idea to stop the training once the model stops improving on the unseen data. For the model trained on dataset 1, the training can be stopped after around 1200 training steps and the model trained on dataset 2, the training proecess can be stopped after around 700 training steps.

After all the steps to improve the fit of the CNN models, the final model trained on Dataset 1 peaks at around 69% on the testing data and the final model trained on dataset 2 peaks at around 76% on testing data. To further evaluate these models, they are tested on the 10% hold-out set that was taken out before nay training occured. The results of these can be seen in section 6.

Note: The model trained on dataset 2 learns faster as it is a much easier dataset to learn from than dataset 1.

6 Results

It's possible to assess a model on a partition of testing data as the model is training. The chosen model after training is essentially the model that does not underfit or overfit to the training data and scores the highest accuracy on the testing set. However, our understanding of the quality of the model then relies on this data that we constantly test on as we train. In theory, the model is being trained to do well for the testing set. When this final model has been chosen, it has to be assessed on data it may see in the future. A model is only really usable if it can predict on a constant stream of data, some of which it may never have seen. To simulate such a case, a portion of the dataset was taken out before the training of the neural network models. This is a dataset that the model has not seen at all. The results shown in this section show the performance of the 2 SNN's, one which was trained on dataset 1 and one trained on dataset 2, and the 2 CNN's, one which was trained on dataset 1 and one trained on dataset 2.

Performance Measures: To show the capability of a model in solving our sarcasm detection problem classification accuracy may not be enough. There are 500 million tweets per day of which there will be a smaller percentage of sarcastic tweets than non-sarcastic tweets. Consider the case that only 20% of these tweets were sarcastic, then a model that classifies all the data as sarcastic would achieve 100% classification accuracy for classifying the sarcastic tweets. The problem with this is that this model also classified all the other non-sarcastic tweets as sarcastic. So this model would also achieve 0% classification accuracy on non-sarcastic data. This is a terrible model as it is essentially guessing the majority class for all cases. To properly assess the sarcasm detection models results, confusion matrices are used. Confusion matrices show the distribution of the predictions of a classifier. The rows represent the true classes and the columns represent the predicted classes. Figure 25 shows a sample matrix, where TP represents the True Positives, FP represents the False Positives, TN represents the True Negatives and FN represents the False Negatives. Confusion matrices allow us to calculate Recall, Precision and F1 scores. These are the performance measures used to evaluate the Neural Network models.

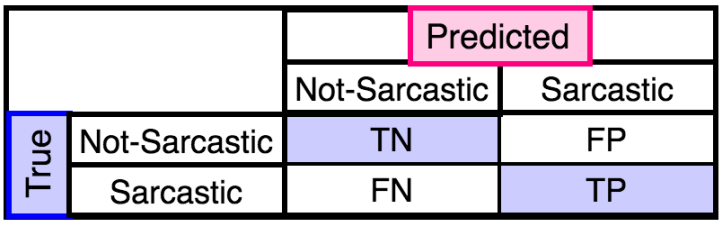

Figure 25: Sample Confusion Matrix

Recall allows us to calculate the completeness of classifications. For a given class, recall calculates the amount of the true classifications of that class given the amount of that class in the testing dataset. The recall for the sarcastic class will be the amount correctly classified as sarcastic out of all the actual sarcastic samples in the dataset. If the classifier was classifying every sample in the test set as sarcastic, then the negative recall will be 0%. Although this model achieves 100% recall on the sarcastic data, it completely underperforms on the non-sarcastic data allowing us to know that this model will not be applicable for our problem. The formula below calculates the positive recall.

$$Recall(P) = \frac{TP}{TP + FN}$$

where P represents the positive class.

Precision is the preciseness of classifications. For a given class, precision calculates the amount of the samples from the testing set correctly predicted as that class given the amount of the samples predicted as that class. If the classifier calculates everything as sarcastic, then it will have a low precision as only a small portion of the samples it classified as sarcastic were actually sarcastic. The formula below calculates the positive precision of a classifiers predictions.

$$Precision(P) = \frac{TP}{TP + FP}$$

where P represents the positive class.

Choosing between a model which has low precision and high recall and a model with high precision but low recall could be a very difficult task. An F1 score is the

balance between both precision and recall. A model with a high F1 score on held-out data has both a high recall score and a high precision score and should hopefully be able to handle the sarcasm detection problem with any data it encounters. The F1 score for positive classifications can be calculated as:

$$F1(P) = 2\left(\frac{Precision(P) \times Recall(P)}{Precision(P) + Recall(P)}\right)$$

where P represents the positive class.

Note: Before any training was performed on either datasets 1 and 2 by the SNN or the CNN, 10% of datasets 1 and 2 were held-out. These are called Hold-Out 1 and Hold-Out 2 respectively. For comparability the SNN and CNN trained on dataset 1 are evaluated on Hold-Out 1 and the SNN and CNN trained on dataset 2 are evaluated on Hold-out 2.

SNN Model Results on Hold-Out Datasets:

True/Predicted	**Not-Sarcastic**	**Sarcastic**	**All**
Not-Sarcastic	666	253	**919**
Sarcastic	238	553	**791**
All	**904**	**806**	**1710**

Table 7: Confusion Matrix for SNN evaluated on Hold-Out 1

True/Predicted	**Not-Sarcastic**	**Sarcastic**	**All**
Not-Sarcastic	482	185	**667**
Sarcastic	115	518	**663**
All	**597**	**703**	**1300**

Table 8: Confusion Matrix for SNN evaluated on Hold-Out 2

Hold-Out Dataset	**Sarcasm Recall(%)**	**Sarcasm Precision(%)**	**Sarcasm F1-Score(%)**
1	69.912	68.610	**69.255**
2	81.833	73.684	**77.545**

Table 9: Recall Scores, Precision Scores and F1-Scores for SNN models when trained on datasets 1 and 2 and evaluated on Hold-Out from datasets 1 and 2

CNN Model Results on Hold-Out Datasets:

True/Predicted	Not-Sarcastic	Sarcastic	All
Not-Sarcastic	731	188	**919**
Sarcastic	77	714	**791**
All	**808**	**902**	**1710**

Table 10: Confusion Matrix for CNN evaluated on Hold-Out 1

True/Predicted	Not-Sarcastic	Sarcastic	All
Not-Sarcastic	551	116	**667**
Sarcastic	72	561	**633**
All	**623**	**677**	**1300**

Table 11: Confusion Matrix for CNN evaluated on Hold-Out 2

Hold-Out Dataset	Sarcasm Recall(%)	Sarcasm Precision(%)	Sarcasm F1-Score(%)
1	90.265	79.157	**84.347**
2	88.626	82.866	**85.649**

Table 12: Recall Scores, Precision Scores and F1-Scores for CNN models evaluated on Hold-Out 1 and 2

With this project I have described the work in creating 2 Shallow Neural-Network models and 2 Convolutional Neural Network models, trained on 2 datasets. The models designed required little to no data pre-processing. As well as this, no manual feature engineering occurred. Although the SNN's designed for this project were quite simple, they were still able to achieve a **69.255%** F1-Score in classifying sarcasm on Hold-Out 1 and **77.545%** in classifying sarcasm on Hold-Out 2. Dataset 1 contained conversational tweets of which around 50% were sarcastic. Given that this data was conversational and no context data was provided for these tweets and the SNN automatically learned the features, I believe an F1-Score of **69.255%** by the SNN in classifying Sarcasm is sufficient.
It's evident that dataset 2 was a much easier dataset for sarcasm classification. This is evident as the SNN achieved an F1 score 10% greater in classifying sarcasm with dataset 2 than with dataset 1. Nevertheless, with little pre-processing and no feature

engineering such as creation of n-grams, brown clusters etc., a simple SNN has been able to achieve a relatively sufficient F1-Score in classifying sarcasm in tweets.

Convolutional Neural Networks are incredibly useful for automatic feature extraction and pattern recognition. This project utilizes Convolutional Neural Networks which take tweet data as word embeddings. Word embeddings gave the CNN's an advantage over the SNN's in their ability to represent relationships between words. The CNNs are able to automatically learn the patterns in the tweets to aid them in classifying sarcasm. From analysing Table 12, it's evident that the increased complexity in Neural Network structure by adding convolutional layers and using word embeddings has greatly aided the models in sarcasm detection with an F1-Score of **84.347%** on Hold-Out 1 and **85.649%** on Hold-Out 2. Changing the input data from a simple bag-of-words representation to a word-embedding representation has greatly improved the accuracy in sarcasm detection which shows that not only must the model know the occurrence of words in a tweet, it must also know the relationship between these words.

7 Conclusion

Social media analysis systems struggle with handling and performing analysis on the abundant textual data streamed every day. This is partially due to the inability of detecting sarcasm in text. The sentiment in a given tweet can be the complete opposite of what is interpreted if it's sarcastic. This is why the detection of sarcasm in text is such an important task. A social media analysis systems interpretation on the sentiment of large amounts of data it streams every day could be wrong as the text is actually sarcastic. Previous works have approached this problem but with difficulty as they have implemented various difficult manual feature engineering such as the creation of n-grams, profile unigrams etc.
Although some have achieved great accuracy in classfying sarcasm in tweets, they have done so with very time-consuming and difficult feature engineering. The models proposed in this project achieves great results in the Sarcasm Detection task with the use of neural network models. These models were able to learn the patterns in textual data without any manual feature engineering making this a relatively easy approach to the sarcasm detection task. Manual feature engineering is not only a very difficult task but it is also a very expensive task as various experts in the field of Natural Language Processing are required to manually create features that machine learning models can learn from. Not only can Neural Networks achieve great results in the sarcasm detection task, but they do so without any feature engineering.

References

[1] David Bamman and Noah A Smith. Contextualized sarcasm detection on twitter. In *ICWSM*, pages 574–577. Citeseer, 2015.

[2] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[3] Aditya Joshi, Vaibhav Tripathi, Kevin Patel, Pushpak Bhattacharyya, and Mark Carman. Are word embedding-based features useful for sarcasm detection? *arXiv preprint arXiv:1610.00883*, 2016.

[4] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[5] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Ashwin Rajadesingan, Reza Zafarani, and Huan Liu. Sarcasm detection on twitter: A behavioral modeling approach. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 97–106, New York, NY, USA, 2015. ACM.

[7] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.