COMP5318 Assignment 2 - Group 149

Manan Choksi (mcho6881) and Lewis Krishnamurti (lkri3129)

May 2022

1 Abstract

Twitter is one of the most popular social media services on the planet. Every day users take to the site to tweet their thoughts and opinions on a wide variety of matters. These tweets can range from positive to negative in tone. Researchers and hobbyists alike have analyzed the sentiment of tweets in a variety of contexts using a variety of machine learning techniques focused on natural language processing. These analyses can be used for a range of purposes such as predicting popularity trends of people and products alike. In this report, we outline our methodology for completing this classification task. Using an open-source dataset from Kaggle containing 1.6 million tweets, we have completed preprocessing steps such as stemming, tokenization and regex to prepare these tweets for use with machine learning models. In particular, we have created models using a basic bidirectional layer recurrent neural network, a stacked recurrent neural network using long short-term memory cells and a composite model using a convoluted neural network and long short-term memory-based neural network. We found the composite model to be the most accurate at predicting sentiment of tweets, with an accuracy of 81%.

2 Introduction

The aim of the assignment is to build a machine learning algorithm which can classify the sentiment of certain tweets as either positive or negative. The dataset we have chosen to work with is the Sentiment140 dataset which has extracted a total of 1.6 million tweets using the twitter API. Kaggle provides the entire dataset as a .csv file with attributes like the target of the tweet which indicates if a tweet is positive (4) or negative(0). Another important attribute is the text which contains the actual words and content of the tweet. The dataset also has lesser important attributes like tweet id, date of the tweet, a query flag and the user who tweeted it.

This study is important because sentiment analysis has a plethora of applications in the real world. [1]Sentiment analysis can be a valuable tool when it comes to marketing a brand and understanding customer psychology. [2][3]Discerning how customers think is valuable information that can be used to leverage sales for many companies. Furthermore, its value in political campaigns has allowed administrations to gauge the wider public's thoughts on their policies and re-strategize. With the advent of social media, companies have an endless pool of data to sift through which can be used to predict trends as well. [4]Netbasequid, a popular leader in market intelligence, was able to identify and predict that "DIY" activities like baking were going to be trending in 2021 due to the overwhelming positive sentiment detected in 2020. Given how ubiquitous data is in the form of written text on all platforms, learning even a facet of any text based ML classification like sentiment analysis can unlock a new dimension of insights we can draw on.

In order to build a classifier model we began with a few preprocessing steps like removing unnecessary attributes like the date and user of the tweet. We also used stemming, tokenization, regex commands and removal of stopwords in order to remove unnecessary words and symbols tweets traditionally consist of. After this we performed hyper-parameter tuning like random search and grid search to find the best parameters for each of our algorithms. The three algorithms we created models for were LSTM, RNN and CNN+LSTM. Finally we finished off with analyzing different parameters and evaluating our results on a test set. We found that all models had an accuracy in the high 70's and low 80's, but our best model was our CNN+LSTM which yielded an 81% accuracy.

3 Previous Work

There are a few research papers who use the following hybrid model on text based classification problems. For example [5] tries to experiment on different datasets and combines CNN + LSTM + SVM and they try to vary the ordering of CNN + LSTM which gives the highest accuracy. The paper has similar goals to ours in the sense that they wish to build a model to identify sentiment in text as accurately as possible. They did find that one of the hybrid models had an accuracy of 84.1 percent on the Sentiment140 dataset, which was the highest out of all. It differs from our work in the sense that the models they construct have significantly more layers than our model does. They have 3 convolutional layers, 1 LSTM layer and 3 hidden layers. Furthermore, for word embeddings, the paper uses BERT whereas we use Glove and Word2Vec. Their final model incorporates SVM after CNN and LSTM whereas ours doesn't. This is because we wish to keep training times shorter due to large dataset sizes. [6] creates CNN-LSTM model on a tweet dataset. Whilst this is similar to our research, it's different to our body of work because they experiment with different orderings off CNN and LSTM networks as opposed to our work focussing strictly on a CNN-LSTM network. Interestingly an LSTM-CNN model had almost a 6 percent better accuracy than a CNN-LSTM model.

[7] incorporates some interesting variance to our LSTM model where they use

a bidirectional LSTM instead. They also use an ensemble method of CNN and LSTM to predict the sentiment of movie reviews. With the LSTM model they obtain an 89 percent accuracy and a 90 percent accuracy with the ensemble method.[8] compares the accuracy of different models like LSTM, Bert and Bi-LSTM. It operates on the same data set as our research topic but the actual architecture of the LSTM model is different in terms of activation functions and the batch sizes used for training. On top of this the paper compares it to the BERT pretrained model which was found to be the most accurate model for sentiment analysis at around 90 percent. For LSTM, the accuracy was 82.8 percent which was very similar to our outcome. One major thing that we could try to experiment with is only using a subset of the data to train the model like this paper did.

[9] is quite similar to our RNN implementation since it uses Word2Vec. The paper tries to create several models which will be able to analyse the sentiment behind different online reviews. Interestingly enough, they found RNN to be their best performing individual model with an accuracy of 87.5%. Our implementation differs because we try to use a bidirectional RNN instead of a regular RNN.

4 Methodology

Preprocessing Techniques- To begin with we cleaned the table of data by removing unnecessary columns like the time of tweet and user.

Removing Stop Words - This preprocessing technique involves removing unnecessary words which add no extra value or meaning to the tweets. This is an important step since we end up tokenizing and creating a dictionary of all the vocabulary we've come across in the tweets. By removing these words we not only remove potential noise but we also speed up the training time of our machine learning algorithms since they have to parse through fewer words.

Lowercase Words - This is another important technique which simply converts all uppercase letters to lowercase. Whilst some sentiment could be lost in situations where someone is trying to express exasperation and anger with upper case text, we were willing to accept the tradeoffs if this would mean avoiding tokenizing the same word multiple times due to different cases.

Tweet Tokenize - A special python library which is designed specifically for tokenzing tweets. Tokenization is the process of breaking sentences up into individual words. This differs from traditional methods of tokenization by preserving the information stored in hashtags. In tweets the information stored in hashtags can also be extremely valuable. On top of this, regular tokenizers split emoticons like ":¡)" into their individual characters. Emoticons preserve some valuable sentiment information.

Stemming - We used the SnowballStemmer python class to get the root of every word. This way multiple words of the same meaning don't get added to our corpus of words. For example words like 'fast' and 'faster' will not be considered separate. This not only reduces training time but it also improves the overall accuracy of our models.

Regex - We used regex to remove links that were present in the tweet too. Again this can add words of no value to our corpus and make our training process unnecessarily long.

4.1 Hyperparameter Tuning

In order to obtain the best parameters for our neural network we decided to employ random search. Random search is very similar to grid search in the sense that we wish to create a list of different hyperparameters and sequentially test which combination will yield the highest accuracy. The benefit of using random search is that these values are picked randomly for a certain number of iterations we specify. We chose this over grid search because it would be much more efficient. Neural networks have so many different parameters that are tuneable like number of layers, number of neurons and drop out layers. As a result, random search seemed like the most appropriate way of hyperparameter tuning. Among some of the variables we tested were learning rate and number of neurons in different layers. We also tested different optimizers outside of the random search environment like the type of optimizer used. Results for this can be seen in the experiments and results section. We used rates like 0.1, 0.01 and 0.001 for the learning rate and for the layers, numbers between 20 to 500.

4.2 Encoding and Embedding Layer

After generating a sequence of tweets that have been cleaned and preprocessed we can then proceed to create a corpus of all the words in the tweets. To do this we used Keras tokenizer in order to generate a dictionary of words that we come across in all the tweets. The output of the object also enables us to access information about word frequency in all the tweets and the number of characters present in tweets. This is an important step as it enables us to substitute words for number encodings which will help our neural networks process data much more efficiently. After this we actually fit each encoding to each tweet, transforming them into arrays of encoded words. Finally, we stipulate a 'maxlength' threshold for each tweet and anything beyond we pad with 0's. We noticed this step was extremely important in speeding up our training time. Processing every word in every tweet was giving us 4 hour training times for a single epoch

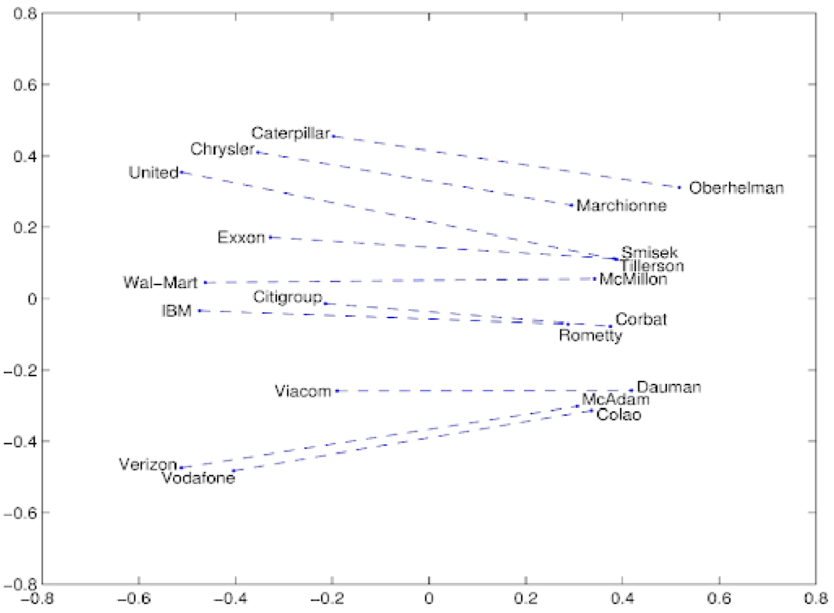


Figure 1: [12] Example of how words can be represented by vectors

which was not only too long, but shortening tweet lengths helped prevent overfitting when we trained our models since they became more generalized. We derived our maxlen value of 15 by finding the average tweet length which was about 7 and then allowing some extra leeway to capture extra information from each tweet.

Following this, we use Glove vectors and Word2Vec in order to create an embedding matrix of all the words in our tweets. The special aspect about Glove is that it's a pretrained word embedding which represents different words in a multi-dimensional vector space. Since the version of Glove we are using is 100D each word is represented using a 100 dimensional vector. If we map all the words in Glove we would notice that words with similar semantic meaning would be located closer together in the vector space. A similar vectorization can be created by Word2Vec which trains on using your corpus of words.[10] By creating this matrix, we are able to feed our neural networks with a set of pre-initialised weights through an embedding layer which makes the training process a lot more efficient and accurate and gives our network almost a 'head start' in understanding our vocabulary. [11] If we were to use random initialisers then our training would be slow and less accurate. Figure 1 shows an example of how words can be represented by vectors.

4.3 Algorithm Theory Comparison

4.3.1 RNN

Recurrent neural networks differ from the traditional feed forward neural networks in the sense that they consist of some component of memory [13]. Usually most feed forward neural networks have no ability to work well on sequential data due to lack of memory, but RNN's incorporate this through a hidden state. In an RNN each process of a hidden layer outputs some value called the hidden state. The hidden state is calculated by passing the current input and the hidden state from the previous input being multiplied together before feeding it into a tanh function which compresses the values between 1 and -1. This value is then fed back into the network again to form the next hidden state. This way the network is able to retain some memory of previous inputs. A rough diagram of an RNN can be seen in figure 2

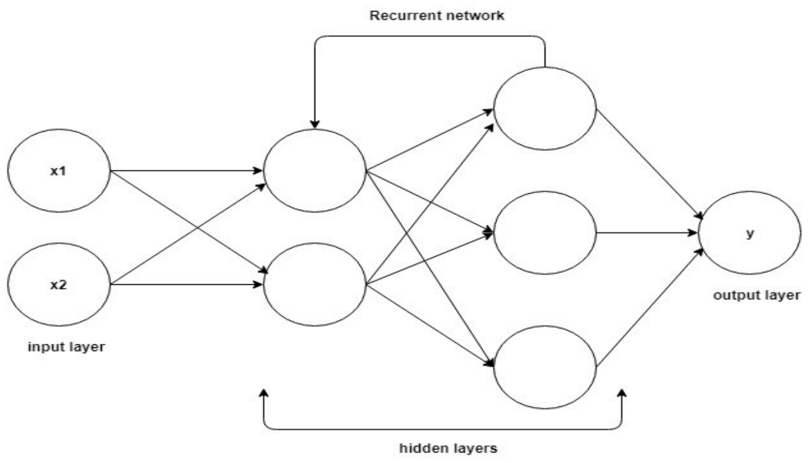


Figure 2: [14] Example of what an RNN looks like

4.3.2 LSTM

LSTMs are a special type of RNN. They are similar to RNN's except each LSTM cell contains a few more special states [15]. There are three main gates in an LSTM - a forget gate, input gate and an output gate. The LSTM also has 3 different inputs - input data, previous cell state and previous hidden state. The input data and previous hidden state are fed into a sigmoid activation function which outputs a value between 0 or 1, where 0 denotes useless information and 1 denotes very useful information. This value is then multiplied by the previous cell state to determine how much of the new input should be 'remembered'. An example can be seen in the highlighted section in figure 3.

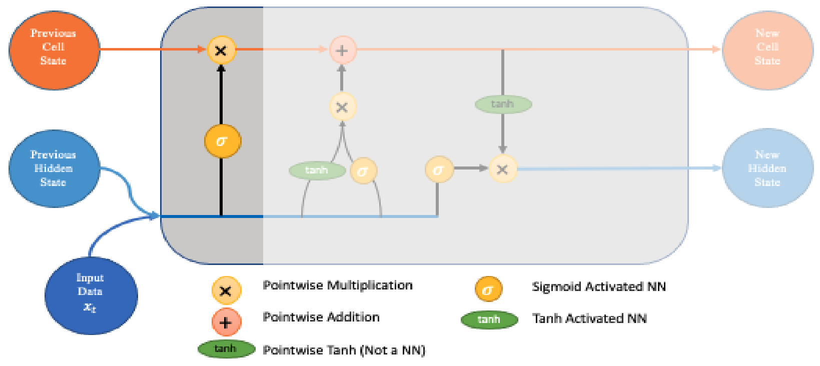


Figure 3: [15] First LSTM Part

The next part of the LSTM is used to judge how relevant the new information being fed into the new network is. The freshly input data and previous hidden state pass through a tanh function to determine how much to alter the pre-existing cell state. The sigmoid function right next to the tanh function is used to determine what component of the new input is of any use. This projects the value between 0 and 1 where 0 represents useless and 1 represents high importance. The two values outputted from each of the functions are then multiplied and used to alter the current cell state. This can be seen in this figure 4.

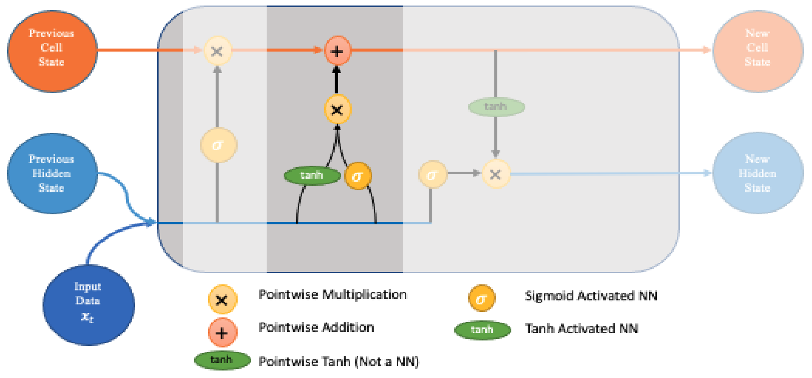


Figure 4: [15] Second LSTM Part

The final step in the process is to pass the previous hidden state and input data through another sigmoid function and the newly created cell state through another tanh function and multiply the output together. The two values are combined together to ensure that the output of the new cell state is filtered in some way before passing it through as the new hidden state. This can be seen in figure 5.

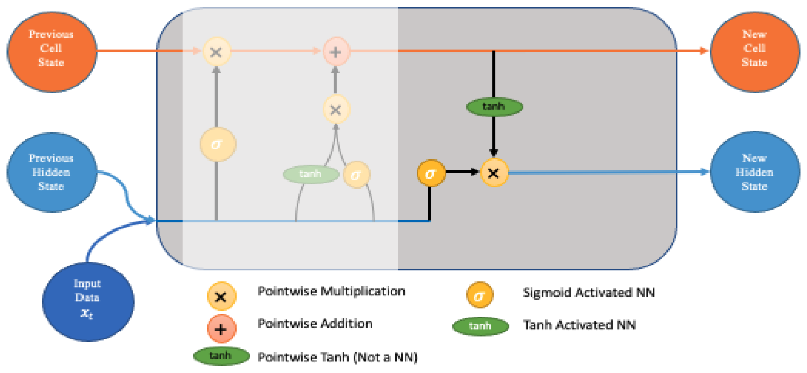


Figure 5: [15] Third LSTM Part

4.3.3 CNN-LSTM

For CNN-LSTM models we have already established the theory behind LSTMs, CNNs however operate differently in the sense that their predominant strength is to extract features from whatever data is fed into it. For text based CNN this begins from an initialized embedded layer. After this a few filters are passed over the input data and they convolve extracting certain features. This ultimately creates multiple feature maps usually of a smaller dimension. After this the feature maps are passed through non linear activation functions to change the format of the data. Finally we use a pooling layer to extract major features and reduce the dimensionality of the data we have. Essentially in most CNN's these processes may be repeated depending on the complexity of the data. Traditionally CNN's are used for images but they work well on text based data because they can extract recurring features from text. With pre-trained word embeddings in the mix CNN's have the ability to flourish. Finally, to finish off the model we feed the output of the CNN into an LSTM which we have already explained previously. Figure 6 represents a generic image of how CNN's can work with text based data.

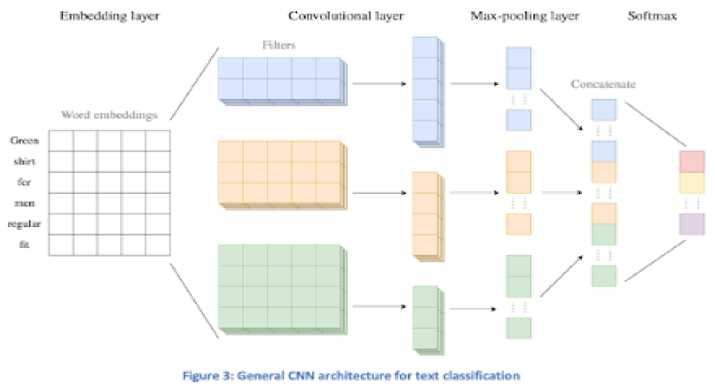


Figure 6: [16]Example of what CNN looks like on Text data

4.3.4 Comparing Models

Each of the methods chosen are different in the aspect that they all are some form of neural network. Perhaps the two most similar of the models we have chosen were LSTM and the basic RNN. In principle both models work well on sequential forms of data and they both consist of a ‘memory’ component in their neural networks through the idea of hidden states. The major difference comes in the design between both cell types of an RNN and an LSTM. An LSTM has [17] 4 times as many parameters as an RNN does. On top of this an LSTM has significantly more gates, like an output gate, input gate and forget gate. While an RNN only passes its hidden state and inputs through a tanh function, some gates require an LSTM to pass through a tanh and sigmoid function in order to compute its memory component. LSTM’s are more robust to the exploding and vanishing gradient issues because of the passage through these different functions. LSTM’s are able to decide which pieces of information to ignore and which to value when calculating hidden states and outputs. Our final hybrid model - the CNN-LSTM - consists of an LSTM but also has a CNN component to it. CNN is unique because it is widely considered a neural network suited for images. It consists of a convolutional layer which applies a filter to help extract local features, as well as apply some pooling features which RNN and LSTMs don’t possess. While CNN’s specialize on local and spatial data, LSTM’s and RNN’s specialize on temporal and sequential data. The combination of an LSTM and CNN brings those extra capabilities over the other models.

4.4 RNN Design and Reasoning

RNNs are a type of neural network that have a basic form of memory by using information from the previous state in a hidden layer. This makes them a good choice for dealing with sequential data as they can consider previous data in the sequence rather than just individual points. As a result, we decided creating a model using a vanilla RNN would be a good starting point for this classification task.

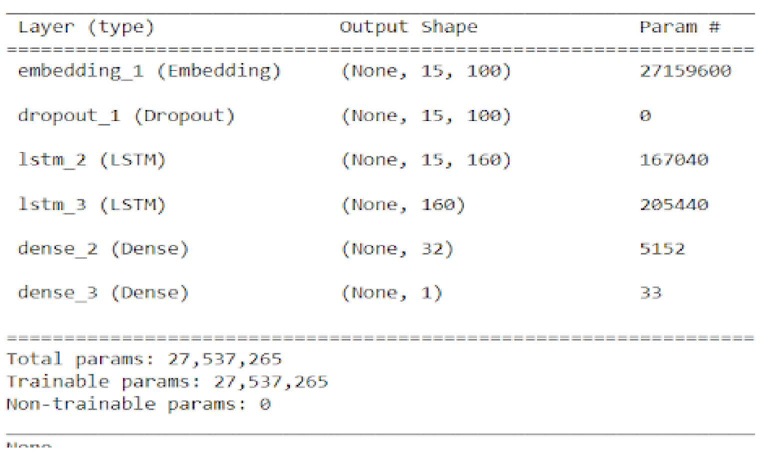
In order to build upon a basic RNN model, we decided to extend the RNN layer by using Keras' bidirectional wrapper. This works by first running the sequence over the layer in the forward direction (i.e. from the start of the tweet to the end) and then by running it over the layer backwards (i.e. from the end of the tweet to the start), before finally concatenating the results two for the final output [18]. The main advantage of this is that it aids the model with taking into account the context surrounding each word on both sides, rather than just the words that precede it, proving a more robust evaluation of an entire sentence.

For the RNN layer used in our model, we used Kera's SimpleRNN as this represents regular RNN cells (as opposed to LSTM or GRU). In terms of the architecture of the model, we followed a similar structure as outlined in the Keras documentation for ease of prototyping purposes (as we expected later models to outperform this one). This consists of the embedding layer (for which we used Word2Vec for reasons mentioned earlier), a dropout layer (which we added ourselves to the API example) to help prevent overfitting, a single bidirectional regular RNN layer (we only used one to save training time as we wanted to use this layer bidirectionally), and finally two dense layers to perform processing and transformation of vectors into a single result (as this is a classification task).

4.5 LSTM Design and Reasoning

LSTM is a special type of RNN which is widely regarded due to its ability to have a 'memory' almost. Upon reading articles and understanding the fundamentals of LSTM, it's widely known to perform well with text related data. This relates back to its 'memory' capabilities because the model has the capability to remember useful information and discard anything irrelevant to its purpose. Since text data, despite all preprocessing efforts, can still contain words which have no value, LSTM's are able to shine in their ability to forget these words.

We attempted to extend this architecture by using a double LSTM layer. According to research papers the use of a stacked LSTM can be used to enhance the memory capabilities of a single LSTM [19]. By creating a deeper network of LSTM we are able to memorize finer details and sentiment within the text, so that when we feed the information into the fully connected layer, more important information is retained. The hierarchical nature of the model just enables more details to be learned. When it came to layers and neuron numbers we decided to keep it to a maximum of 3 or 4 layers with no more than 248 neurons in any one layer. Again, this is because we wanted to keep the training times in reasonable ranges but we also wanted to strike a balance with being able to fit well on more complicated data. Attempting to stack too many layers without reducing the dimensionality of the data can lead to overfitting as well as large training times. Intermittently we add dropouts to the layers which will remove a fraction of the nodes from the network. This helps to prevent overfitting as it causes the network to generalize a bit more. According to random search our best parameters were 200 nodes for LSTM and 48 nodes for our hidden dense layer. Finally, since this is a classification task our output layer consists of one node. A summary of our model can be seen in figure 7.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 15, 100)           27159600

 dropout_1 (Dropout)         (None, 15, 100)           0

 lstm_2 (LSTM)               (None, 15, 160)           167040

 lstm_3 (LSTM)               (None, 160)               205440

 dense_2 (Dense)             (None, 32)                5152

 dense_3 (Dense)             (None, 1)                 33

=================================================================
Total params: 27,537,265
Trainable params: 27,537,265
Non-trainable params: 0
_________________________________________________________________
```

Figure 7: What our LSTM model looks like

4.6 CNN-LSTM Design and Reasoning

[20] [21] The motivation behind using a hybrid model stems from the success it's had from previous works. The benefit of using a hybrid model is its ability to perform well on both spatial and temporal aspects of the data. CNN performs classification tasks well on the spatial component of the data, which for text classification refers to local features like word placement. LSTM performs well for sequential data and remembers context better than any other network model due to its 'memory' properties. As a result they form the perfect combination where a CNN can extract local text features effectively and LSTM can remember the entire context of the tweet or sentence.

5 Experiments and Results

Hardware and Software Specifications - The google colab VM uses 13 GB RAM 33 GB HDD and uses a 2-core Xeon 2.2GHZ. There was no NVIDIA driver installed with the google colab notebook.

5.1 RNN

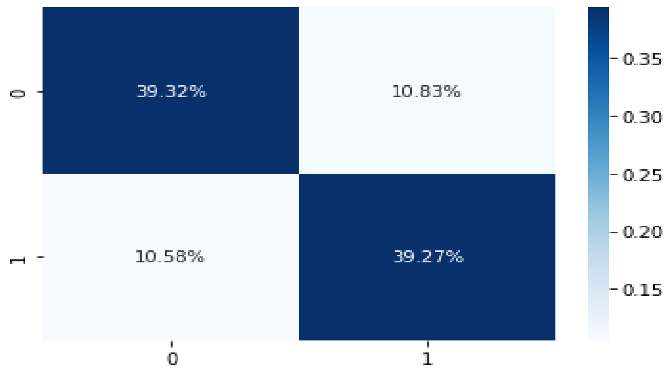


Figure 8: Confusion matrix 1 = Positive and 0 = negative

Our simple bidirectional RNN model performed reasonably well on tweet data. As can be seen in figure 8, of all tweets 39.32% were classified as positive that were positive and 39.27% that were negative as negative, giving an accuracy of 78.59%. Precision and recall also have a similar value. Given that this was a simple model, we did not expect it to perform to such a degree. Initially we thought this performance in part may be due to our hyperparameter optimization using random search, however we mostly found the difference in parameters that we used for the search to only give a difference in accuracy of less than a percent, suggesting that either our tested parameters were too similar or that this model may not be able to be much more accurate. Upon further reflection, it may have been interesting to compare a uni-directional RNN layer architecture to the bdirectional layer we used, however this would have involved changing the entire architecture of the model rather than tuning a parameter. We suspect that the accuracy of this model may be due to the fact that tweets are short after our preprocessing, with the sentiment not requiring a lot of context to determine, meaning that the basic memory aspect of using the previous state as input may have been enough to produce an adequately accurate model.

5.2 LSTM

Overall our algorithm performed relatively well on the tweet data. From figure 9 we can see approximately 39 percent of each positive and negative tweet was classified correctly. According to figure 10 our accuracy, precision and recall were all around 79 percent. Interestingly, this model did not perform significantly better than the vanilla RNN model. This could be due to the fact our LSTM model was only uni-directional, but we also believe that the length of

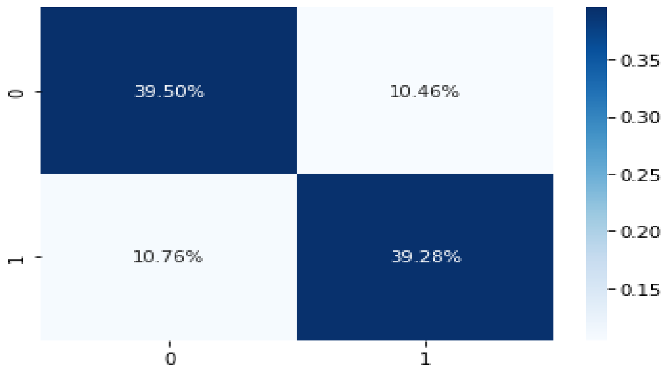
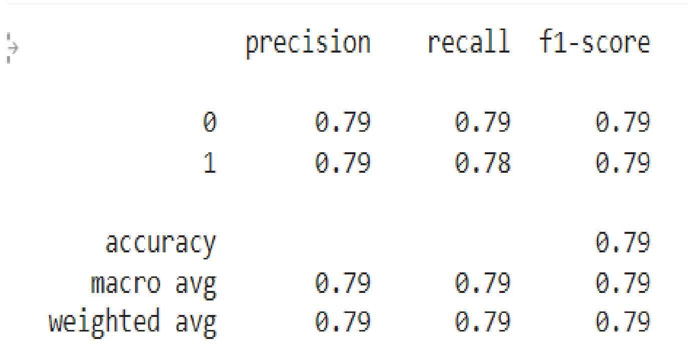


Figure 9: Confusion matrix 1 = Positive and 0 = negative


```
              precision    recall  f1-score

           0       0.79      0.79      0.79
           1       0.79      0.78      0.79

    accuracy                           0.79
   macro avg       0.79      0.79      0.79
weighted avg       0.79      0.79      0.79
```

Figure 10: Classification Report

tweets means that the ability for LSTM to 'forget' irrelevant information may not be too useful in this scenario. Reflecting, we possibly could have used a bidirectional LSTM layer somwhere in the model to see if it made a difference or not. A point of interest we wanted to look deeper into was how the optimizer type affected accuracy. With Adam and Nadam we approached an accuracy of 78 percent. Looking at figure 12 we saw a significant drop off with SGD and Adadelta optimizers. The easiest way to explain this discrepancy could be the fact that Adam is a significantly more optimized version of SGD and Adadelta. Adam combines the best of two properties that optimizers should account for: learning rate and gradient. While SGD generalizes better according to [7] it is also described as a very unstable algorithm which can converge and arrive to local minima prematurely which could be an explanation for its poor performance. Furthermore, we also wanted to assess how the number of LSTM cells would affect the accuracy. As seen in figure 13, it's hard to tell what the optimum number of cells are because there is slight fluctuation as the number of cells

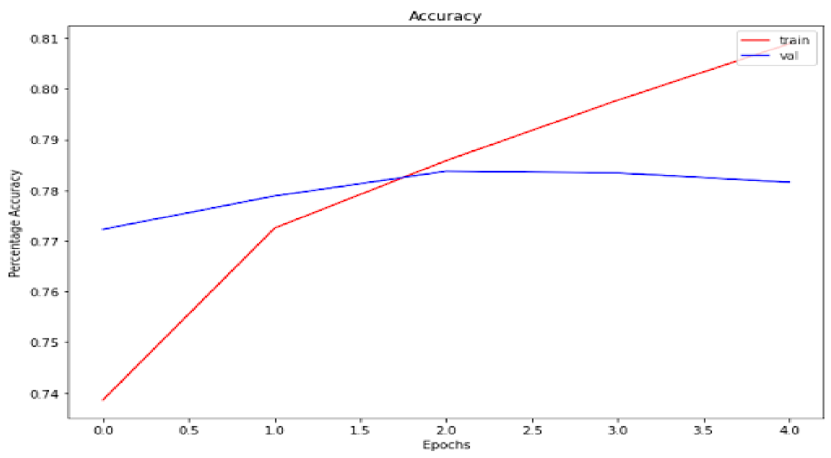


Figure 11: Training Accuracy Graph

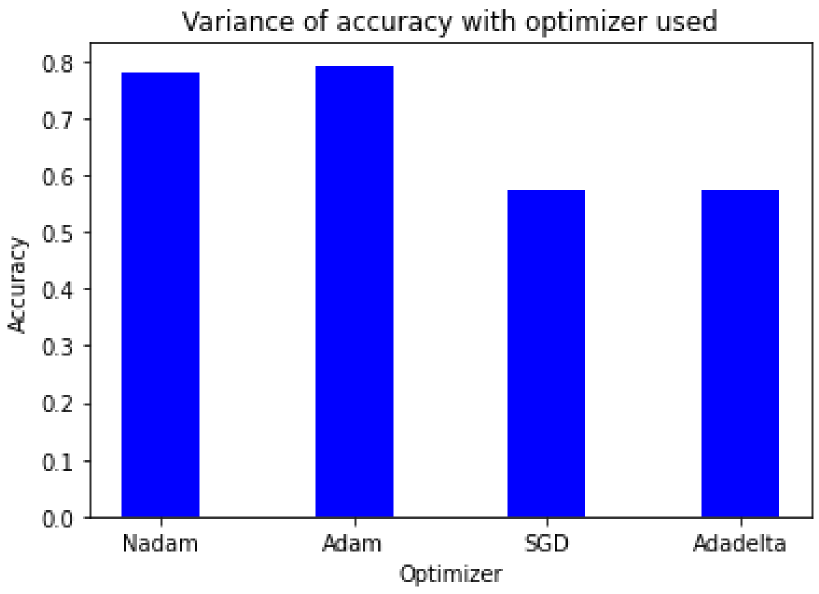


Figure 12: Optimizer Bar Graph

increases. This could be attributed to the fact that there are other important factors that influence the overall accuracy more than the actual number of cells in the network. We didn't want to extend the number of LSTM cells to 500 because this would lead to large training times.

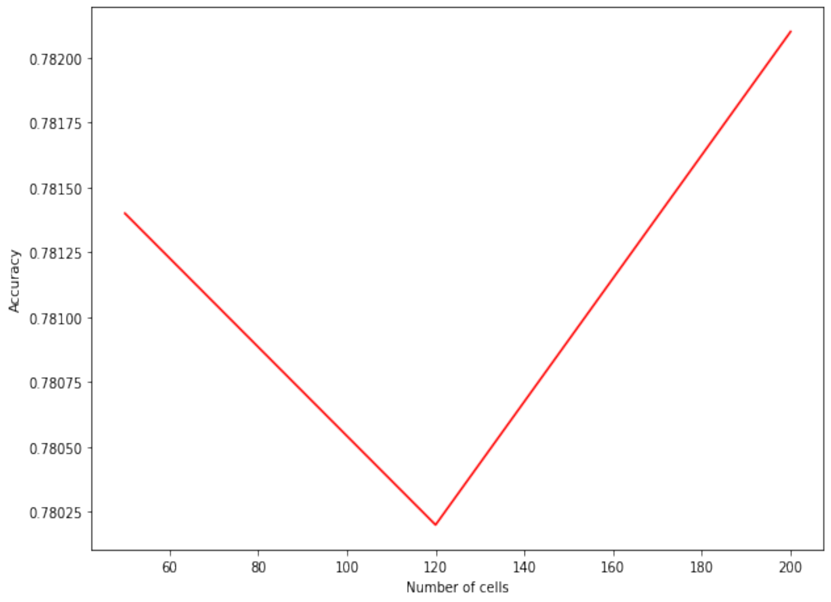


Figure 13: As LSTM number of cells changes how does the accuracy change

5.3 CNN-LSTM

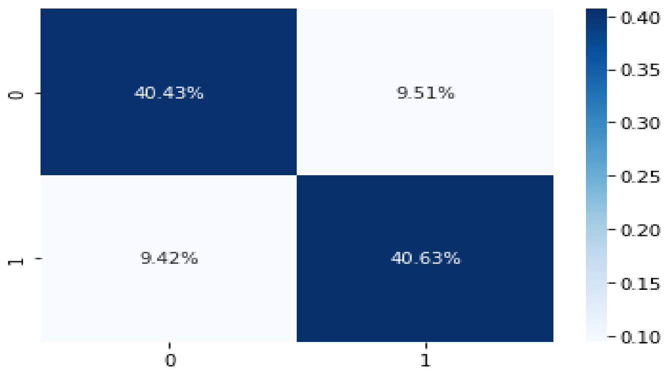
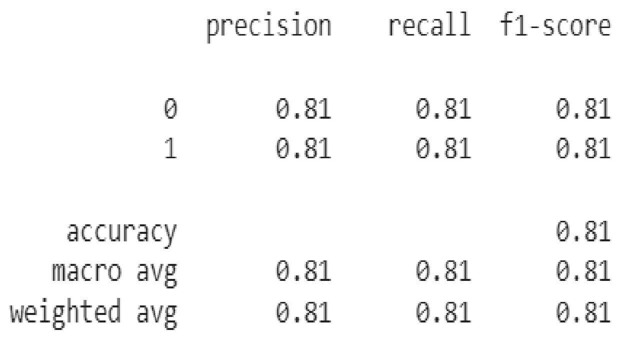


Figure 14: Confusion matrix 1 = Positive and 0 = negative

Overall this algorithm performed well obtaining an 81 percent accuracy on our test dataset with precision and recall scores also being 81 percent. This can be seen from figures 14 and 15. We noticed that with high learning rates like 0.1, the accuracy with which the neural network was able to classify the tweet sentiment was significantly lower from figure 18 you can notice that the accuracy was almost 50 percent as opposed to close to 80 percent with a learning rate



```
              precision    recall  f1-score

           0       0.81      0.81      0.81
           1       0.81      0.81      0.81

    accuracy                           0.81
   macro avg       0.81      0.81      0.81
weighted avg       0.81      0.81      0.81
```

Figure 15: CNN-LSTM Classification Report

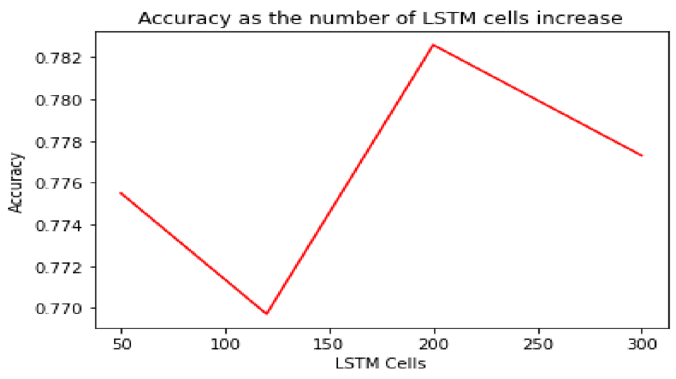


Figure 16: CNN-LSTM accuracy as number cells increases

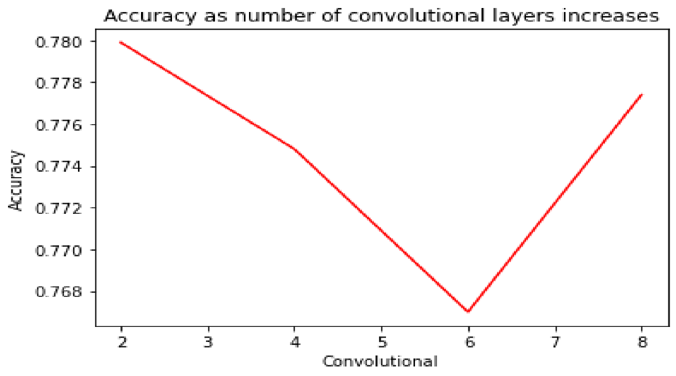


Figure 17: Accuracy as Convolution Layers Increase

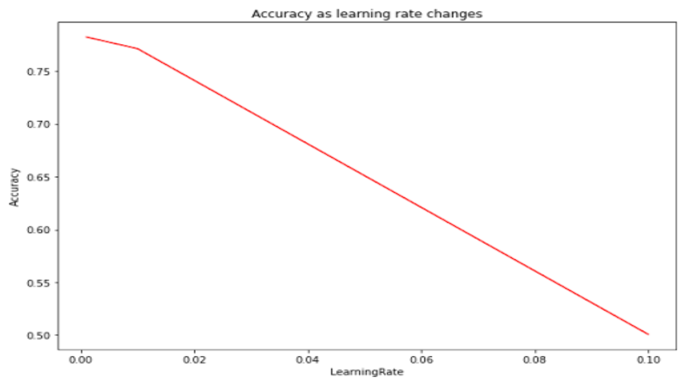


Figure 18: Accuracy as Learning Rate Changes

of 0.001. This can perhaps be explained by the fact that with high learning rates it's very common for the algorithm to converge to suboptimal solutions. Because of how much the learning rate can influence weightage in the network, a high learning rate causes the model to learn much faster than expected. Again we saw a lot of the similar trends we saw for LSTM with regards to optimizers used and how they affected the overall accuracy. We tried to observe how adding a different number of convolution layers would affect the overall accuracy of the classification. It's safe to say that as the number of convolutions increased there was no observable increase or decrease. The same was found to be true with the number of LSTM cells specified. Perhaps this is because the data complexity is such that it doesn't require too many hidden layers or neurons. If we were to add more neurons, this would lead to overfitting. What we did want to observe was the effects of two commonly used word embeddings. These being Glove and Word2Vec. It was clear that there was a slight improvement in performance with word2vec. A possible reason for the improvement is that word2vec is more alterable. You can adjust the way the model learns on your corpus, whereas glove is a preset text file that contains all the word vectors you need. As a result the values are less customisable. With word2vec the accuracy was only slightly better than the best result obtained with Glove by about 0.7 percent.
We believe that our CNN-LSTM performed the best out of our three models due to the aforementioned positives of using a CNN (being able to effectively extract local feature data such as word placement in a tweet). However, the accuracy of the CNN-LSTM model is not much higher than the LSTM model, which we suspect may be due to the short nature of tweets not allowing for local feature extraction to make a large difference in the final result.

Model Accuracy and Prediction Speed			
	RNN	LSTM	CNN+LSTM
Accuracy	0.7859	0.7898	0.8107
Speed	31s	1157s	551s

Table 1: Table showing Model Accuracies and Runtimes for Prediction on Test Data

6 Conclusion

Table 1 provides a summary of the accuracies and time to predict on the test data for each model. As we initially expected, RNN performs the worst, followed by LSTM then CNN+LSTM being the most accurate. Despite this, all models had reasonably similar accuracies. It is of particular note that the RNN using stacked LSTM layers did not substantially outperform the regular bidirectional RNN model, which as mentioned ealier we suspect may be due to the nature of the data not allowing for the extra gates in LSTM to provide more accurate results. In terms of time taken to test however, the regular RNN was substantially faster than the LSTM architecture, due to the LSTM model having stacked layers and more gates in each cell to process. In this sense, the RNN model is preferred to the LSTM model as it achieves almost identical accuracy with only a fraction of the runtime. With regards to CNN+LSTM, the dimentionality reduction caused by the feature extraction of the CNN results in a faster runtime than the base LSTM model, demonstrating the the LSTM layers are most likely the main bottleneck in these architectures. Although it is still quite slower than the base RNN model, it does have a couple more percentage points of accuracy, making it the preferred model in this case. Overall, the relative results we obtained are reasonably similar to what we expected based on our research of prior work.

6.1 Future Work

In the context of this classification task, it would be intersting to create a RNN using GRU cells as these are more contemporary (2014) and are suggested to have similar performance in NLP tasks to LSTM [22].
Further work involving tweets and NLP could be to attempt to classify tweets by predicting the user who wrote them, with an extension of this being creating a Generative Adversial Network to produce tweets in the style of certain users. Other potential tasks could involve predicting if a tweet was written by a bot, as this is commonplace on the platform. This would be of particular use to help prevent scams.

7 Appendix

If you want to use glove make sure you download this text file from this link: https://www.kaggle.com/datasets/danielwillgeorge/glove6b100dtxt and store it in your google drive somewhere.

Instructions for running different files

7.1 RNN.ipynb

The code files will have comments at the top of each code block like this: "/CODE BLOCK N" where N will denote the code block number. We will provide a list of numbers which will represent the code block numbers you will run to run our submission. The rest of the blocks can be run optionally.

- 1
- 2
- 3
- 4
- 5
- 6
- 7 if you want W2V embedding / 8 if you want to use Glove (we used W2V)
- 9
- 10
- (11 only if you want to run RandomSearch)
- (12 only if you want to run RandomSearch)
- 13
- 14

7.2 LSTM.ipynb

The code files will have comments at the top of each code block like this: "/CODE BLOCK N" where N will denote the code block number. We will provide a list of numbers which will represent the code block numbers you will run to run our submission. The rest of the blocks can be run optionally.

- 1

- 2
- 3
- 4
- 5
- 6
- 7 if you want W2V embedding / 8 if you want to use Glove
- 9
- 10
- (11 only if you want to run RandomSearch)
- (12 only if you want to run RandomSearch)
- 13
- 14

7.3 CNN-LSTM.ipynb

The code files will have comments at the top of each code block like this: "/CODE BLOCK N" where N will denote the code block number. We will provide a list of numbers which will represent the code block numbers you will run to run our submission. The rest of the blocks can be run optionally.

- 1
- 2
- 3
- 4
- 5
- 6
- 7 if you want W2V embedding / 8 if you want to use Glove
- 9
- 10
- 11 if you want to run our model
- 12 and 13 if you want to run the random search

7.4 Dataset Link

https://www.kaggle.com/datasets/kazanova/sentiment140

7.5 Libraries

All versions the the most current versions installed on Google Colab as of 22/5/22

- NLTK
- Keras
- Numpy
- Tensorflow
- Sklearn
- Seaborn
- KerasTuner (!Not installed by default on Google Colab)

7.6 Team member contribution

Lewis did 50% and Manan did 50% of the assignment. Roughly even contribution no issues here.

References

[1] 8 applications of sentiment analysis; 2020. Available from: https://monkeylearn.com/blog/sentiment-analysis-applications/.

[2] Link S. The real reasons why sentiment analysis is important in 2021. LinkedIn; 2021. Available from: https://www.linkedin.com/pulse/real-reasons-why-sentiment-analysis-important-2021-stefan-link/?trk=public$_{p}rofile_{a}rticle_{v}iew$.

[3] Sentiment analysis: How does it work? why should we use it?;. Available from: https://www.brandwatch.com/blog/understanding-sentiment-analysis/.

[4] What is social sentiment analysis and why is it important?; 2021. Available from: https://netbasequid.com/blog/what-is-social-sentiment-analysis/.

[5] Dang CN, Moreno-García MN, De la Prieta F. Hybrid deep learning models for sentiment analysis. Complexity. 2021;2021:1–16.

[6] Wankhade M, Rao AC, Kulkarni C. A survey on sentiment analysis methods, applications, and challenges. Artificial Intelligence Review. 2022.

[7] Minaee S, Azimi E, Abdolrashidi A. Deep-Sentiment: Sentiment Analysis Using Ensemble of CNN and Bi-LSTM Models. CoRR. 2019;abs/1904.04206. Available from: http://arxiv.org/abs/1904.04206.

[8] Varshney A, Kapoor Y, Thukral A, Sharma R, Bedi P. Performing sentiment analysis on Twitter data using Deep Learning Models: A Comparative Study. Advances in Data and Information Sciences. 2022:371–381.

[9] Balakrishnan V, Shi Z, Law CL, Lim R, Teh LL, Fan Y. A deep learning approach in predicting products' sentiment ratings: A comparative analysis. The Journal of Supercomputing. 2021;78(5):7206–7226.

[10] Brownlee J. How to use word embedding layers for deep learning with keras; 2021. Available from: https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/.

[11] Deep Learning, NLP, and representations;. Available from: http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/.

[12] Pennington J;. Available from: https://nlp.stanford.edu/projects/glove/.

[13] Biswal A. Recurrent neural network (RNN) tutorial: Types and examples [updated]: Simplilearn. Simplilearn; 2022. Available from: https://www.simplilearn.com/tutorials/deep-learning-tutorial/rnn.

[14] DD. RNN or recurrent neural network for Noobs. HackerNoon.com; 2018. Available from: https://medium.com/hackernoon/rnn-or-recurrent-neural-network-for-noobs-a9afbb00e860.

[15] Dolphin R. LSTM networks: A detailed explanation. Towards Data Science; 2021. Available from: https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9: :text=LSTMs%20use%20a%20series%20of,each%20their%20own%20neural%20network.

[16] Convolutional neural network text classification with risk assessment; 2020. Available from: https://machine-learning-company.nl/en/technical/convolutional-neural-network-text-classification-with-risk-assessment-eng/.

[17] Week 8 Lecture Notes; 2022.

[18] Recurrent Neural Networks (RNN) with Keras. Google; 2022. Available from: https://www.tensorflow.org/guide/keras/rnn.

[19] Yu L, Qu J, Gao F, Tian Y. A novel hierarchical algorithm for Bearing Fault diagnosis based on stacked LSTM. Shock and Vibration. 2019;2019:1–10.

[20] Dang CN, Moreno-García MN, De la Prieta F. Hybrid deep learning models for sentiment analysis. Complexity. 2021;2021:1–16.

[21] Alotaibi FM, Asghar MZ, Ahmad S. A hybrid CNN-LSTM model for psychopathic class detection from tweeter users. Cognitive Computation. 2021.

[22] Ravanelli M, Brakel P, Omologo M, Bengio Y. Light Gated Recurrent Units for Speech Recognition. IEEE Transactions on Emerging Topics in Computational Intelligence. 2018;2:92-102.