An Application of Neural Networks for Sentiment Classification

Calley Ramcharan

James Kriel

Yashkir Ramsamy

Abstract

The development of a Neural Network (NN) to predict the sentiment of user reviews on Steam, a video game digital distribution service, is described in this report. Sentiment analysis is a type of natural language processing that aims to categorize the sentiment of textual data. User reviews are subjective and will express a feeling about a game, which can be either negative or favourable. Steam can employ automatic sentiment analysis of user reviews to assign tags and organize reviews into categories for customers to peruse and gain a better knowledge of the games they're considering buying. We implement a simple baseline solution in the form of a Feed-Forward Neural Network that classified observations correctly more than half of the time. We then extend this by using an existing implementation of a Recurrent Neural Network, which we then tuned to produce a better classification rate. The resulting network produced a 3% better result with a significant improvement in sensitivity in comparison to the benchmark implementation we used. In comparison to the baseline model, the resulting network produced a 16% better classification rate with significant improvements in miss-rate and sensitivity. Uncontrollable factors imposed limitations on tuning, such as Google Colab's GPU usage limits.

1. Introduction

This report describes the implementation of a Neural Network (NN) that attempts to predict the sentiment of a given user review on Steam, a video game digital distribution service.

1.1. Background

Steam is a video game distribution platform that has over 120 million monthly active users [1], with over 50 000 video games hosted and available for purchase on the platform [2]. It is the largest video game distribution for PCs in the world [1,2]. Steam has allowed its users to publicly write and post reviews about the games that it offers for purchase on their store [3].

Sentiment analysis is a natural language processing technique that attempts to classify the sentiment of textual information [4]. “Sentiment” refers to a view or opinion that is expressed. Therefore, sentiment analysis can be applied in various scenarios, such as aiding companies in making decisions about their operations, or in this case, Steam using users’ sentiment to classify and better rate games.

1.2. Problem & Usefulness

User reviews are subjective and will contain a sentiment towards a particular game, generally being either negative or positive. Seeing that these reviews are based on user input, it can also lead to abusive or false negative reviews about a particular game. Considering the number of users that use the service, the number of reviews per game will eventually surpass the developers’ ability to read the mass of reviews to gain an understanding of how their games performed. Steam does provide users with the option to recommend or reject a game to others, however, this isn't accomplished automatically, such as by means of sentiment analysis of published reviews.

Automatic analysis of the reviews could allow Steam to place tags and group the reviews into categories for users to browse and garner a better understanding of the games they consider purchasing.

1.3. Difficulty

The chosen data set described in section 1.5 contains a comparatively limited amount of data being approximately 17k unique reviews. This is a relatively low number of unique data points in the context of machine learning. This data set was chosen to specifically look at how tuning hyperparameters can be used to improve machine learning in the context of limited data.

Since sentiment analysis attempts to analyze colloquial language and subjective texts, many of these reviews could be written with sarcasm as its basis. Sarcasm in the reviews would encourage the model to misclassify the sentiment of the review, as it is typically used to portray a negative emotion through the use of positive terms.

Detecting tone via written text poses great difficulty in determining a user’s true emotions and thoughts towards a product or service. It is in this way that classifying large amounts of both objective and subjective pieces of text may be difficult, allowing room for a large number of misclassifications if not handled correctly.

The introduction of the emoji keyboard to many cellular and desktop keyboard configurations worldwide provides sentiment analysis algorithms with yet another degree of freedom. These emoticons convey emotions that extends beyond the usual tone of the written language. Removing them from the text for analysis thus removes

vital pieces of information to aid in sentiment classification, so it is important that these characters are also accounted for.

Many sentences provided in the written text may include comparisons that aid in conveying the user's thoughts and opinions on a product or service. That being said, the issue of decoding the underlying thoughts of the user becomes a difficult situation in sentiment analysis. Careful consideration needs to be given to these types of sentences, ensuring that the algorithm can decipher a positive review or negative review from the relative ordering provided in written text.

The English language, already as difficult as it is to manually interpret, makes use of many negations, such as "not", "never" etc. Hence, the use of double negative becomes prevalent in speech and consequently, the written text. It is up to the sophistication of the sentiment analysis algorithm to know that the use of a double negative may sometimes imply a positive sentence, while at other times may intensify the negativity conveyed.

These difficulties are important in understanding the complexity of a classifier such as this. For a classification algorithm to perform well, it is important that all difficulties are accounted for.

1.4. Ethics

There are minimal ethical implications for this use of sentiment analysis. However, data collection in this regard imparts the most significant ethical implication.

Sourcing this data from the Steam platform may come without the consent of the reviewing users. Considering that Steam caters to both adults and minors, game review collection could infringe upon minors' rights. This, too, is very dependent on the country in which the data is being collected. For example, South Africa's POPI Act states: "Collection directly from data subject" in regard to data collection, compliance is not necessary when "the information is contained in or derived from a public record or has deliberately been made public by the data subject" [5].

As a result, users may not necessarily be aware that their reviews are being used for automatic analysis, and its realistic application could allow Steam, or any other relevant company, to use their sentiment scores to serve users with more personalized video game suggestions to increase their sales. Some may find this approach to be intrusive.

1.5. Data Set and Tech Stack

1.5.1. Dataset

The dataset for this network can be found below.

```
Game Review Dataset
https://www.kaggle.com/arashnic/game-review-dataset
```

The dataset used is a compilation of Game reviews along with Game Metadata found on the Steam platform.

This data set is made up of 3 subsets:

- A training set: *17494 observations*
- A test set: *8045 observations*
- A game overview set: *64 Games*

Tables 1. and 2. below describe the subsets mentioned above.

Table 1. Game Review Dataset Feature Description (Training Set)

Training Set	
Attribute Name	**Description**
review_id	The unique identifier of the observation
title	The name of the game within the observation
year	The year of the game within the observation
user_review	The textual review of the game within the observation given by a user
user_suggestion	The recommendation of the game within the observation given by a user

It should be noted that the test subset contains the same features as the training subset with “user_suggestion” omitted.

Table 2. Game Review Dataset Feature Description (Game Overview Set)

Game Overview	
Attribute Name	**Description**
title	The name of the Game
developer	The developer of the Game
publisher	The publisher of the Game
tags	Categories that the Game falls under.
overview	A given synopsis or brief introduction of the Game.

From these subsets, only the training set will be used as it contains the necessary feature “user_review” and “user_suggestion”. The test set does not contain the user suggestion of the given review, making it unusable for testing. Instead, the training set will be split further into a training set, validation set and a test set to account for the missing feature in the given test set.

Figure 1. describes the balance of the target variable within the distribution, we can observe that the distribution among positive and negative reviews are closely aligned. Thus, making it suitable for use for training.

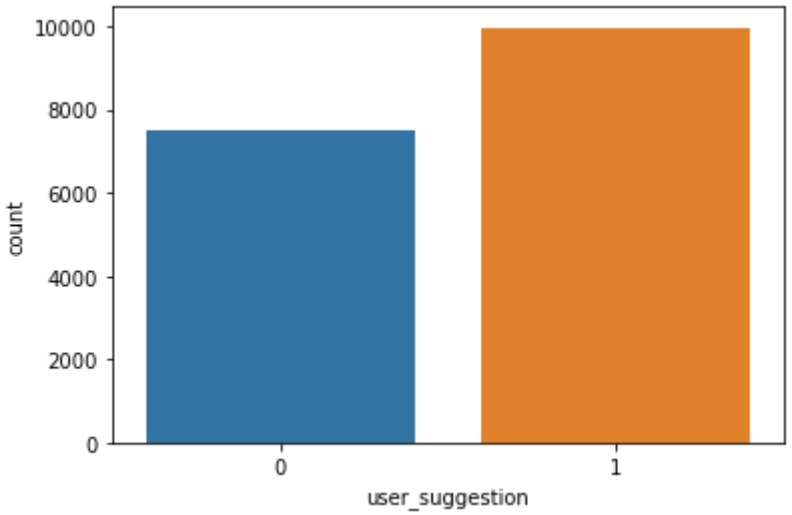

Figure 1. Distribution of User Suggestions over the data set

1.5.2. Tech Stack

For the model implementations, we used TensorFlow with Keras on Google Colab.

2. Baseline Model

2.1. Pre-Processing

2.1.1. Special Characters

User reviews within the dataset are not bound to only numbers and letters. For example, some special characters appear as shown in the review in Figure 2 below.

```
Just played this game for about an hour and it has given me some pretty good
scares... T_T Few silly jumps starting but as the game progresses it's harder to
avoid the paranoia and tension XDIt's fun indeed :)
```

Figure 2. User Review with special characters.

We can see that this review contains special characters, such as ":)", "T_T" to express emotion through textual "emoticons". For the purposes of this model, we remove these special characters, which includes punctuation marks.

We can also observe that this review makes use of characters "X" and "D" to express tone through a textual emoticon, this will impact the classification of sentiment as these characters will not be removed, as we are unable to manually remove it without confirming its context within a piece of text.

We make use of the regex shown in Figure 3. to remove special characters.

```
"[^a-zA-Z0-9]"
```

Figure 3. Regex to exclude special characters.

2.1.2. Tokenization

We translate the text into tokens before putting it into vectors. This aids in text classification. In this case, the reviews are tokenized into words.

We first create a dictionary of a set length of 6500 items within it. We do this so that we limit the number of irrelevant words used in classification. Initially, in the existing implementation in the codebase used, this number was set as 5000.

We use the Keras Tokenizer for this task, it receives the cleaned text generated in Section 2.1.1 and creates a dictionary in order of frequency within the texts. Every dictionary item will have a unique index, which will replace the tokenized word so that it is represented as an integer.

We can demonstrate this by way of example:
If we take a cleaned subset of the words in Figure 1. and categorize them into sentences as shown in Figure 4. The resulting dictionary, "dict", contains an index number and the word. This index number will become the new representation of the tokenized word.

```
sentences = ['Just played this game for about an hour and it has given me
some pretty good scares', 'Few silly jumps starting but as the game
progresses']
```

```
tokenizer = Tokenizer(number_of_words = 5) #create a dictionary of 10 words
tokens = tokenizer.tokenize(sentences) #tokenize the sentences

 > dict:
 > {1: game}
 > {2: just}
 > {3: played}
 > {4: this}
 > {5: for}
```

Figure 4. Pseudocode of Tokenizer Method and Output

Additionally, we altered the code to acquire the MAX_LEN variable which determined the maximum length for a review within the dataset. The original implementation made use of the 75th percentile to acquire this value, we changed this to the 80th to retain additional information in potentially longer reviews. The use of this method is important to prevent wasting space in the data with an unnecessary amount of padding as the maximum review length is 130 within this dataset but 99% of reviews fall below 20 words in length. By using the 80th percentile we set reviews to have a maximum of 15 words. Thus any review array longer than that was truncated and any review shorter was padded with 0s to fill up the remaining space.

2.2. Simple Feed-Forward Neural Network

2.2.1. Model Design

The baseline model is a simple Feed-Forward Neural Network consisting of all Dense layers. Figure 5. shows a minified structure below. It consists of **2 hidden layers** and an **output layer consisting of 1 neuron**. The number of neurons within the hidden layers was calculated by [6] :

$$\text{Number of Hidden Layer Neurons} = \frac{Ninputs + Noutputs}{2}$$

By Layer Activation Function:

- Layer 0 (input): ReLu - 16 Neurons
- Layer 1 (hidden) : ReLu - 8 Neurons
- Layer 2 (hidden) : ReLu - 8 Neurons
- Layer 3 (output): Sigmoid - 1 Neuron

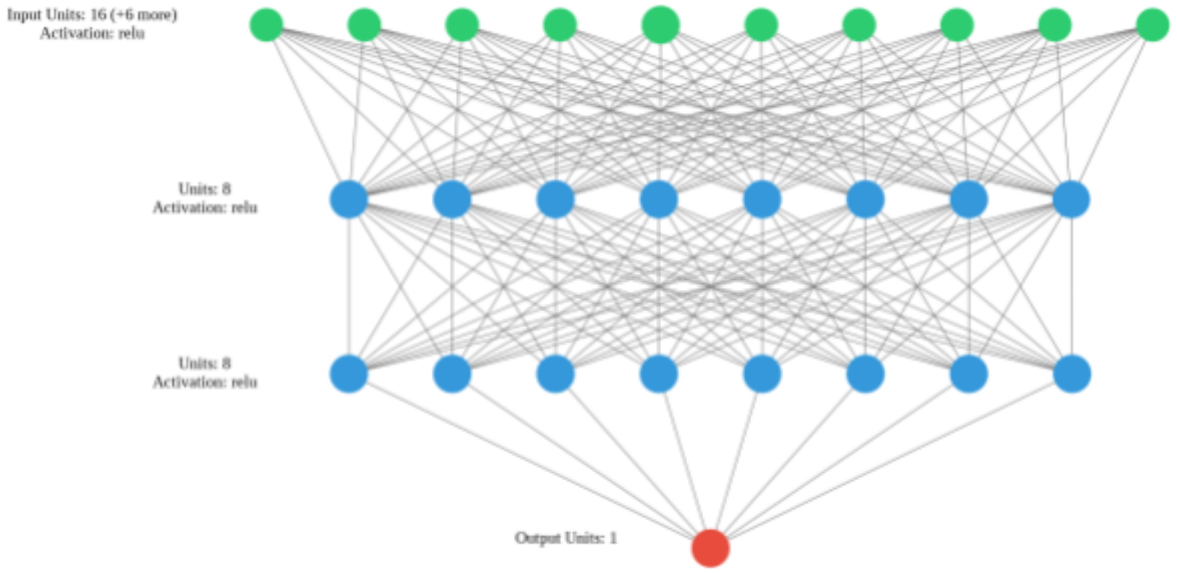

Figure 5. Baseline Model Visualization

The loss function used to evaluate the model was binary cross-entropy as our classification targets were either 1 or 0. **The learning rate was set to 0.001** as we used Keras' optimizer "Adam".

Of the training set described in section 1.5, **20% is used as the test set and 20% is used as a validation set.** The model was fit with these parameters and ran for **15 epochs.**

2.2.2. Baseline Performance

Training and Evaluation
This baseline model's performance without any tuning is described in Figures 5,6 and 7.

Figure 6. shows the classification of the accuracy of the model over 15 epochs for both training and validation. As the number of epochs from 0 to 3 increases, the model's accuracy increases. From 3 onwards, we can observe that for both training and validation the accuracy begins to plateau.

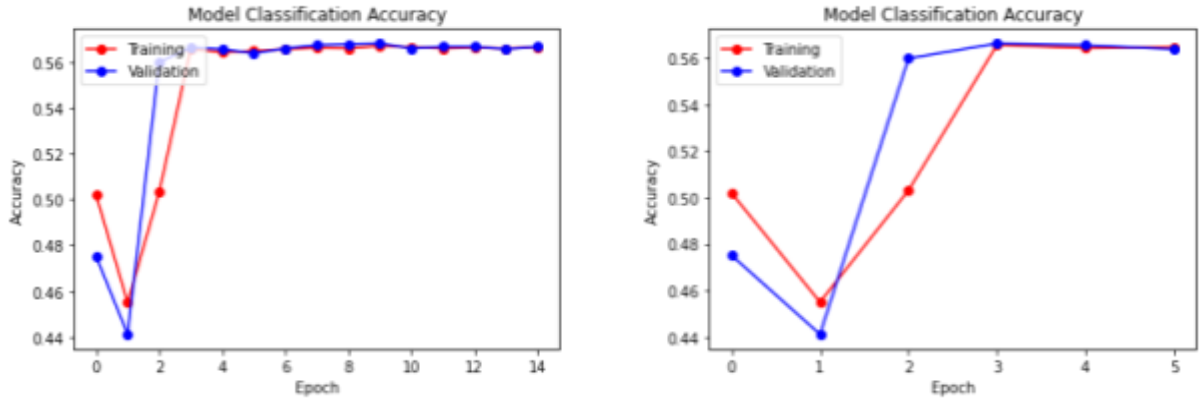

Figure 6. Baseline Model Accuracy

Figure 7. shows the binary cross-entropy loss results for both training and validation over 15 epochs. We can observe that as the number of epochs increases from 0 to 3, the resulting loss decreases rapidly. From 3 onwards, we can observe that for both the accuracy and validation the loss plateaus.

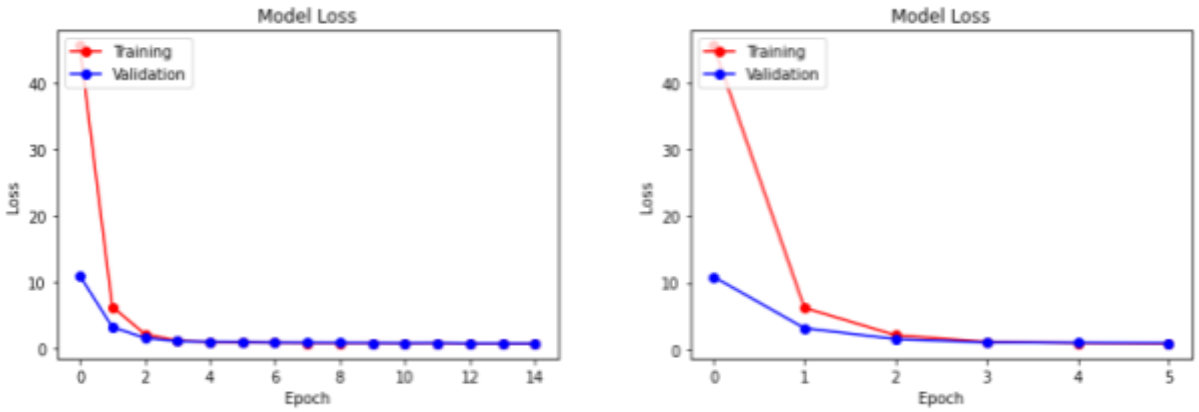

Figure 7. Baseline Model Loss

With the results shown in Figures 6 and 7, we can infer the model trained on the data adequately as the accuracy and loss curves were stable. The validation loss is at most equal to the training loss, inferring that the model capacity is adequate and the training and validation split is likely to be statistically equivalent. However, on the training and validation set, the model performs subjectively poorly. At roughly 55% accuracy, the simple baseline model would not be robust enough to deploy in a realistic setting.

Testing and Prediction

We have set the classification decision boundary at 0.5. Meaning, that if an observation's prediction is greater than 0.5, then it becomes classified as a Positive review, and Negative if otherwise.

Within this testing set, there are 3499 observations, of this 2026 are Positive (1) and 1473 are Negative (0) values. Figure 8. describes the actual versus predicted values. A description of this is as follows below:

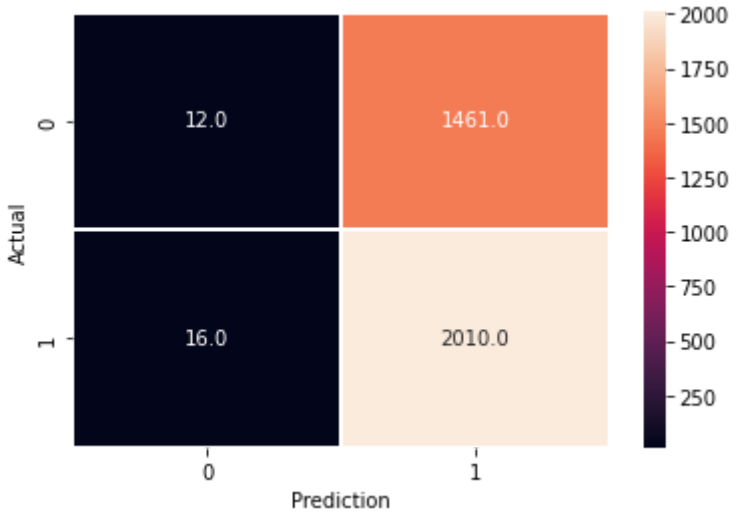

Figure 8. Prediction Confusion Matrix of Baseline

- 1461 negative observations were incorrectly predicted to be positive observations.
- 2010 positive observations were correctly predicted to be positive observations.
- 12 negative observations were correctly predicted to be negative observations.
- 16 positive observations were incorrectly predicted to be negative observations.

The model's accuracy was 57.79%.

2.2.3. Baseline Conclusion

Considering that the model is a simple generalized implementation of a neural network, it has produced sufficient results to support that this is a feasible task to conduct. The model itself is not robust enough to produce a usable output, however, with extensions such as the use of more specialized layers. It has the capacity to produce better results. This will be discussed in Section 3, where we change layers to build a Recurrent Neural Network.

3. Model Design

The model described in this section is an extension of the Feed-Forward baseline model. The final model is a Recurrent Neural Network. The model will also make use of the same preprocessing methods described above for the baseline model, string cleaning for special characters and tokenization.

3.1. Model Architecture

The base model is sourced from an existing project that made use of this data but was altered for use in this project. The original model made use of an Embedding Layer with an output dimension of 100, three GRU layers and a final Dense layer with a single neuron for output.

The original model from which this was adapted can be found below.

```
Game Review Classification | Deep Learning Based
https://www.kaggle.com/mehmetlaudatekman/game-review-classification-deep-learning-based
```

The model used for this project consists of an initial Embedding layer, four Long short-term Memory layers and a final Dense layer containing the activation function. A dropout layer will be added at the beginning of the model to be used as a hyperparameter.

```
model = keras.Sequential()
  model.add(layers.Embedding(input_dim=VOCAB_SIZE,
                             output_dim=VECTOR_SIZE,
                             input_length=MAX_LEN
                            ))
  model.add(layers.Dropout(dropoutval))
  model.add(layers.LSTM(num_neurons1, return_sequences=True)) #was 256
  model.add(layers.LSTM(num_neurons2, return_sequences=True)) #was 512
  model.add(layers.LSTM(num_neurons2, return_sequences=True)) #was 512
  model.add(layers.LSTM(num_neurons2, return_sequences=False)) #was 512
  model.add(layers.Dense(1,activation=activation_func)) #was sigmoid

  opt = keras.optimizers.Adam(learning_rate=learningrate) #was 0.001
  model.compile(optimizer="Adam",loss="binary_crossentropy",metrics=["accuracy"])
```

Figure 9. Python Model Design

3.1.1. Embedding Layer

The Embedding Layer is used to allow the model to the reviews array as a dense vector allowing words that possess a similar meaning to have similar vector representations. This allows for a more natural method of learning language classification.
Within the model the **input_dim** of the layer is the size of our vocabulary which is 51929, the **output_dim** was chosen to be 50 and the **input_length** used the same maximum word length of 15 acquired from the 80th percentile.
The **output_dim** of 50 was chosen due to a large amount of complexity added by an Embedding Layer and on the basis of reviews not containing too many unique words that will likely convey a similar meaning. When initially running the model to determine the basic construction it was noted that a larger output dimension increased the validation loss while not changing the rate at which the model would initially overfit the training data before tuning. Therefore a smaller output dimension was chosen to increase the efficiency of the model and decrease the validation loss.

Additionally, the original model made use of pre-trained embedding weights while we decided against using them to prefer to let the model learn from only the data provided. This way we could properly see the effects of learning from a smaller set of training data as all weights would start from a neutral state.

3.1.2. Long short-term Memory (LSTM) Layer

The Long short-term Memory layer is the most commonly used layer for a Recurrent Neural Network. For this reason, it was chosen as the basis on which to build the network. Four LSTM layers were created to allow the model to learn complex features of the input data. To vary the rate at which the complexity of the data is learnt the hyperparameter tuning sets the number of neurons for the first LSTM layer independently of the remaining three allowing combinations of different numbers of neurons across the layers.

3.1.3. Dropout Layer

The Dropout layer was added to assist in reducing overfitting. It was positioned at the beginning of the network as it cannot be used within the recurrent connections. The values of the Dropout layer are also used as part of the hyperparameter tuning.

3.1.4. Dense Layer

The dense layer consists of a single neuron that facilitates the binary output of the model, being either a positive or negative classification. The Dense layer additionally holds the Activation function which will be used as one of the hyperparameters for tuning during the optimization stage.

3.1.5. Optimizer

Based on research into the best optimizer for a sentiment analysis problem it was decided to use the Adam optimizer with the learning rate being tuned as a hyperparameter.

3.1.6. Loss Function

Based on research into loss functions for use with binary classification it was decided to use the Binary Crossentropy function as this was cited most commonly as the best loss function for use in this scenario.

3.1.7. Unused Layer Types

The Bidirectional Layer using an LSTM was also considered for the model but based on the length of each array being on 15 items (words from the review) it was decided to add unnecessary complexity when the added value would be comparatively little given the short length.

3.2. Input Features

The input features consist of 15 array items representing the textual component of a review. The 80th percentile of the lengths of all the reviews was used to get the value of 15. This ensured that our data would not be skewed towards outlying large reviews and thus make use of an abundance of padding. As 85% of the reviews would have a length less than or equal to this we found it was acceptable to reduce the length of the remaining 20%.

These input features would be represented by their respective encodings as described in the Tokenization section above.

4. Model Validation

The initial model used, before fine-tuning hyperparameters, had Accuracy and Loss curves that looked as follows:

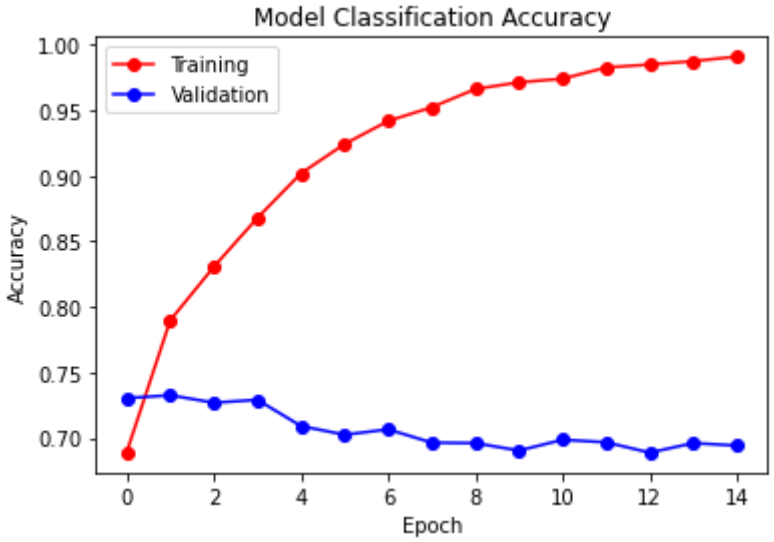

Figure 10. Accuracy curve for initial network model - prior to hyperparameter optimisation.

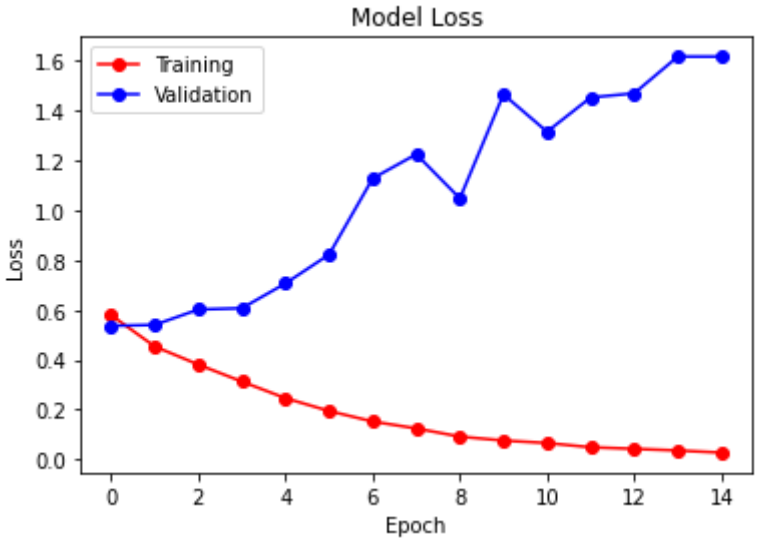

Figure 11. Accuracy curve for initial network model - prior to hyperparameter optimisation.

The model was trained on 15 epochs, with a training-set:validation-set split of 80:20 and a training-data:test-data split of 90:10.

As can be seen from the above plots, the model is quite clearly overfitting the training data provided. This is noted from the perfect exponential decay seen by the training loss coupled with the large rise in the validation loss. Due to this overfitting, it can be seen that the validation accuracy decreases with increasing epoch, implying that the model will not fare well in classifying unseen data. The model, therefore, requires fine-tuning of the hyperparameters that form part of the model structure and learning process in order to eliminate this issue of overfitting.

4.1. Hyperparameter Choices

Various optimisations were employed to handle the issue of low validation accuracy and high validation loss, which caused the overfitting of the training set in the network. The approach that was taken to test for the optimal sets of parameters that minimise loss involved iteratively varying single hyperparameters (based on a decided ranking of parameters). Once the optimal value for that hyperparameter was found, it was used as a constant parameter to vary the next hyperparameter and so on. This process was carried out until all hyperparameters had been optimised, allowing for a final model with optimal hyperparameters.

A function definition was made to aid in the varying of hyperparameters. Its definition is shown below:

```
def buildModel_varyparams(MAX_LEN,dropoutval,activation_func,learningrate,num_neurons1,num_neurons2):

    model = keras.Sequential()
    model.add(layers.Embedding(input_dim=VOCAB_SIZE,
                               output_dim=VECTOR_SIZE,
                               input_length=MAX_LEN
                              ))
    model.add(layers.Dropout(dropoutval))
    model.add(layers.LSTM(num_neurons1, return_sequences=True)) #was 256
    model.add(layers.LSTM(num_neurons2, return_sequences=True)) #was 512
    model.add(layers.LSTM(num_neurons2, return_sequences=True)) #was 512
    model.add(layers.LSTM(num_neurons2, return_sequences=False)) #was 512
    model.add(layers.Dense(1,activation=activation_func)) #was sigmoid

    opt = keras.optimizers.Adam(learning_rate=learningrate) #was 0.001
    model.compile(optimizer=opt,loss="binary_crossentropy",metrics=["accuracy"])
    return model
```

Figure 12. Python implementation of parameter varying

The function takes in the maximum review array length as described from tokenization, which was set to be a constant value throughout testing, a dropout rate value, an activation function, a learning rate, the incoming LSTM layer number of neurons, and finally the other LSTM Layers' number of neurons. Coupled with the train:validation split specified when calling model.fit() (i.e. after the model is built) these form the hyperparameters to be varied and optimised in the training process.

The optimisation algorithm described as well as all other source code used for data preprocessing and model building can be found and run on Google Colaboratory (Colab) at the following link:

```
**REMOVED CONTACT YASHKIRRAMSAMY@GMAIL.COM FOR LINK**
```

To run the notebook, ensure that the runtime type is set to "GPU", then navigate to Runtime > Run all. This should produce similar results to the results displayed below, with only minor variations in the plots of validation loss owing to the random nature used in sampling the validation set from the training set.

4.1.1. Varying the Number of Neurons in the first LSTM Layer

It is well known that possibly reducing the number of neurons in a layer would reduce the complexity of the network, and consequently reduce the overfitting of training data. As a result, the number of neurons in the incoming LSTM layer was varied, with values of 32, 64, 128, 256, and 512 being tested. While varying these parameters, the other hyperparameters were set as follows:

Table 3. Hyperparameter values set as constant during varying of number of neurons in the first LSTM layer

Train:validation split	70:30
Dropout rate	0
Activation function	"sigmoid"
Learning rate	0.001
Number of neurons in other LSTM layers	512

The number of epochs used in the training process was also kept to a value of 15 epochs, however, this was not a hyperparameter. A dropout rate of 0 was chosen to mimic the absence of a dropout layer completely, while the other hyperparameters were simply set to those that were used in the initial structure of this model.

Varying the number of neurons in the first layer of the model and training the model for each of these values yielded a final validation loss value at the 15th epoch of each training process. These final validation loss values were the metric for determining the optimal number of neurons, and thus could be plotted as a function of the number of neurons. This plot is displayed below.

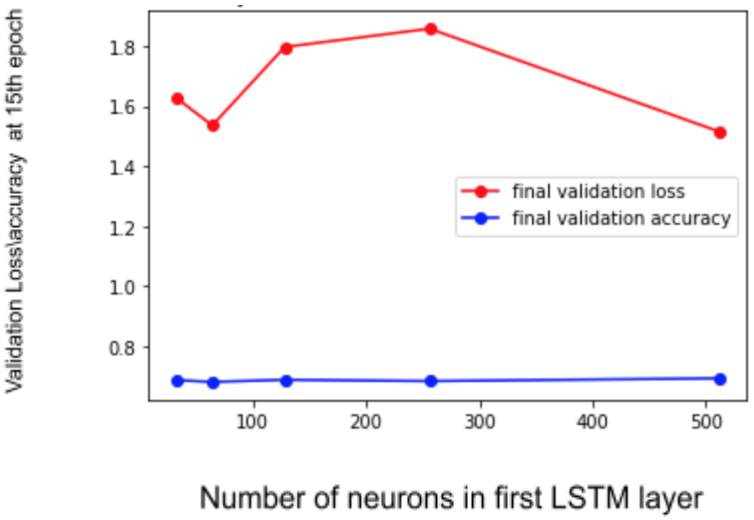

Figure 13. Validation loss at 15th epoch as a function of number of neurons in first LSTM Layer

As can be seen from the above plot (and as determined in code by finding the number of neurons corresponding to the lowest validation accuracy), it is evident that 512 neurons at the first LSTM layer minimise the 15th epoch validation loss the most and does not decrease the validation accuracy at all. Hence, 512 neurons were chosen as the optimal number of neurons for the first LSTM layer.

4.1.2. Varying the Number of Neurons in the latter LSTM Layers

To reduce complexity, not every LSTM layer's number of neurons were optimised. The initial model included a first LSTM layer with the number of neurons being different to the subsequent LSTM layers' numbers of neurons. These subsequent LSTM layers had the same number of neurons, and thus to reduce complexity, the number of neurons were varied consistently for all these subsequent LSTM layers.

The hyperparameters kept constant during this process are listed below.

Table 4. Hyperparameter values set as constant during varying of number of neurons in the subsequent LSTM layers

Train:validation split	70:30
Dropout rate	0
Activation function	"sigmoid"
Learning rate	0.001
Number of neurons in first LSTM layer	Optimal number of neurons found: 512

The values being iterated over were 32, 64, 128, 256, and 512. The results of this optimization are depicted below, in a similar fashion to figure 13.

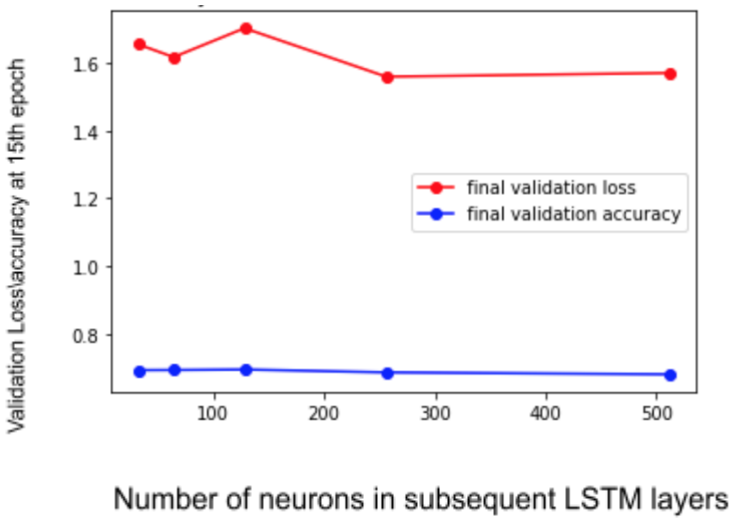

Figure 14. Validation loss at 15th epoch as a function of the number of neurons in subsequent LSTM Layers

It is seen that the 15th epoch validation loss is minimised with 256 neurons in the subsequent LSTM layers while not affecting the validation accuracy at all. As a result, the optimal number of neurons for subsequent LSTM layers was found to be 256. However, it is seen that the 15th epoch validation loss for 512 neurons is very close to that of 256 neurons, so as a result, the answer tends to sway between 512 and 256 neurons over various runs, indicating that either one is optimal. For continuity however, a value of 256 neurons is carried through for the rest of the model optimization.

4.1.3. Varying train:validation ratio

The network's exposure to training data is important in the training process. The amount of data used to train versus to validate the correctness of the network is thus very important in the training process and may have the capability of reducing overfitting if the correct ratio of training data to validation data is found. As a result, the fraction of training data being used to validate the model was varied from 0.1 to 0.5 in increments of 0.1. The hyperparameters kept constant during this process were as follows:

Table 5. Hyperparameter values set as constant during varying of train:validation ratio split.

Number of neurons in subsequent LSTM layers	Optimal number of neurons found: 256
Dropout rate	0
Activation function	“sigmoid”
Learning rate	0.001
Number of neurons in first LSTM layer	Optimal number of neurons found: 512

The validation loss and accuracy at the 15th epoch for each of these splits were tracked and is plotted in figure 15.

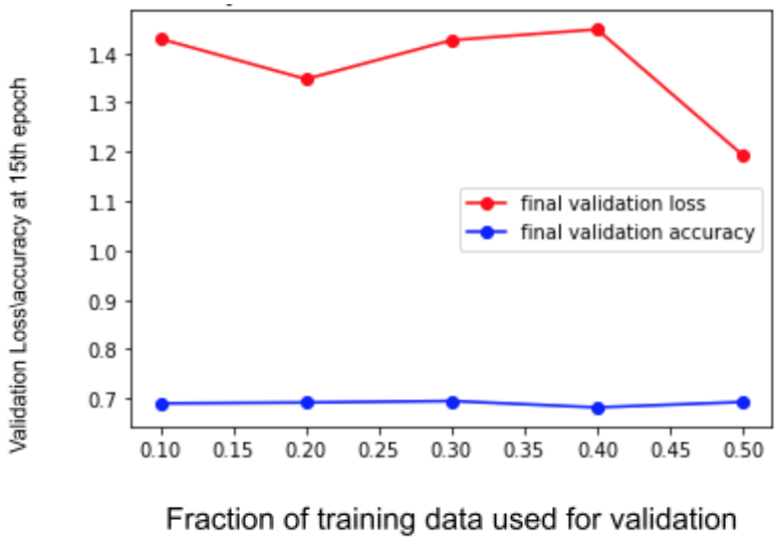

Figure 15. Validation loss/accuracy plotted as a function of train:validation split hyperparameter

Analysing the results, it is evident that the 50:50 split (corresponding to a fraction of training data used for validation of 0.5) results in the lowest 15th epoch validation loss and highest 15th epoch validation accuracy. As a result, the 50:50 split is the most ideal to reduce overfitting.

4.1.4. Varying dropout rate

To further reduce overfitting, a dropout layer was included in the structure of the model. Up until this point of optimization, the dropout layer was set with a dropout rate of zero to mimic the training of the model with the absence of the dropout layer. In this step of the optimization, the dropout rate was varied from 0.1 to 1.0 in increments of 0.1 to determine the most optimal dropout rate for the model.

The hyperparameters kept constant during this step in the optimisation process were as follows:

Table 6. Hyperparameter values set as constant during iterations over dropout rates.

Number of neurons in subsequent LSTM layers	Optimal number of neurons found: 256

Train:validation split	Optimal split found: 50:50
Activation function	“sigmoid”
Learning rate	0.001
Number of neurons in first LSTM layer	Optimal number of neurons found: 512

The validation loss and accuracy at the 15th epoch during training were again measured for differing dropout rates and the results of this are depicted in figure 16.

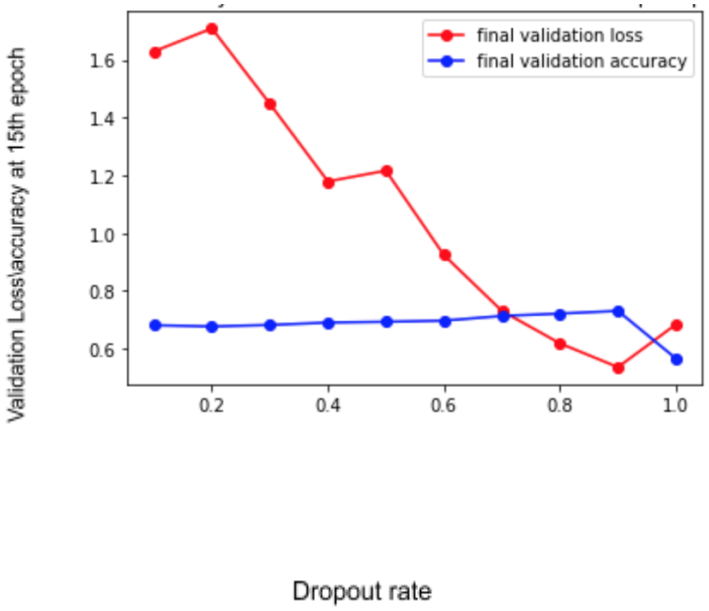

Figure 16. 15th epoch validation loss/accuracy as a function of dropout rate

As seen from the above plot, it is clear that the optimal dropout rate which reduces validation loss significantly is a dropout rate of 0.9. This dropout rate also in turn maximises the validation accuracy as seen by the local maximum of validation accuracy at a dropout rate of 0.9. As a result, the optimal value for the dropout rate for the dropout layer used was found to be 0.9.

4.1.5. Varying activation function

The choice of activation function used in the Dense Layer of the model structure is generally between one of two of the most popular activation functions used in neural network implementations: the sigmoid activation and the rectified linear unit (ReLu) activation. As a result, these two functions were put to the test to determine which function may reduce overfitting to a larger extent. The original model made use of the sigmoid activation but ensuring the best choice of activation function was important in covering all bases of model optimisation.

The hyperparameters kept constant during this step of the optimisation process were as follows:

Table 7. Hyperparameter values set as constant during testing of sigmoid versus ReLu activations.

Number of neurons in subsequent LSTM layers	Optimal number of neurons found: 256

Train:validation split	Optimal split found: 50:50
Dropout rate	Optimal rate found: 0.9
Learning rate	0.001
Number of neurons in first LSTM layer	Optimal number of neurons found: 512

The validation loss and accuracy curves for each of the two model training processes (using sigmoid activation and ReLu activation respectively) were plotted for analysis of model performance.

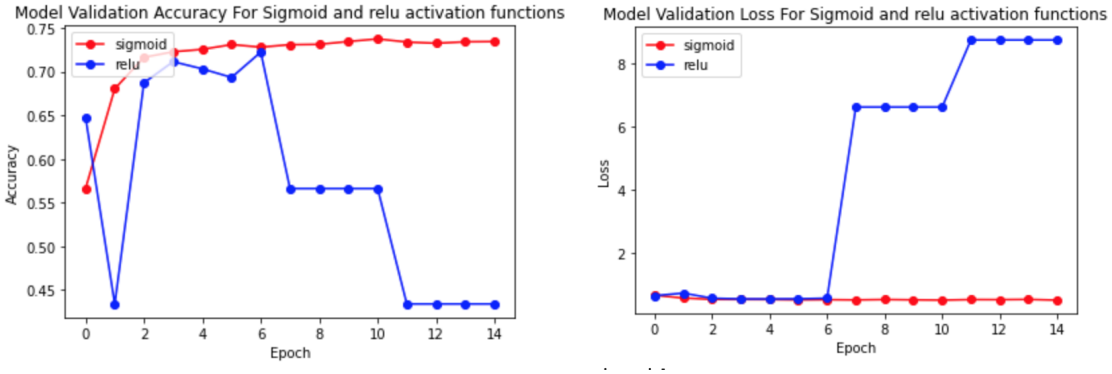

Figure 17.: model validation accuracy (left) and model validation loss (right) for both sigmoid and ReLu activation functions

Looking at the validation accuracy, it is evident that the model produces more accurate predictions using the sigmoid activation compared to the ReLu activation. Additionally, looking at the validation loss curves, it is prevalent that training with the ReLu activation produces a shocking validation loss (and zero improvements in validation accuracy) as the number of epochs progresses, thus the sigmoid version is confirmed to be, as expected, the best activation function for this model.

4.1.6. Varying learning rate

A final hyperparameter optimisation to be considered is the learning rate of the Adam optimiser used in the model. The learning rate determines how quickly the model is adapted to a problem and a smaller learning rate corresponds to more epochs required during the training process. The learning rate has an effect on how quickly the model can converge to a local minimum in the loss, and therefore a local maximum in the accuracy of the model. As a result, the learning rate was varied over multiple orders of magnitude in the range of [0,1) to determine the most optimal learning rate for validation loss minimisation, given that 15 epochs are consistently being used to train the model.

The hyperparameters kept constant during this final phase of optimisation were as follows:

Table 8. Hyperparameter values set as constant during iterations across various learning rates

Number of neurons in subsequent LSTM layers	Optimal number of neurons found: 256
Train:validation split	Optimal split found: 50:50
Dropout rate	Optimal rate found: 0.9

Activation function	Optimal function found: sigmoid
Number of neurons in first LSTM layer	Optimal number of neurons found: 512

The following plot depicts the 15th epoch validation loss as a function of the base-10 logarithm of the learning rate (logarithm taken to aid in visualisation).

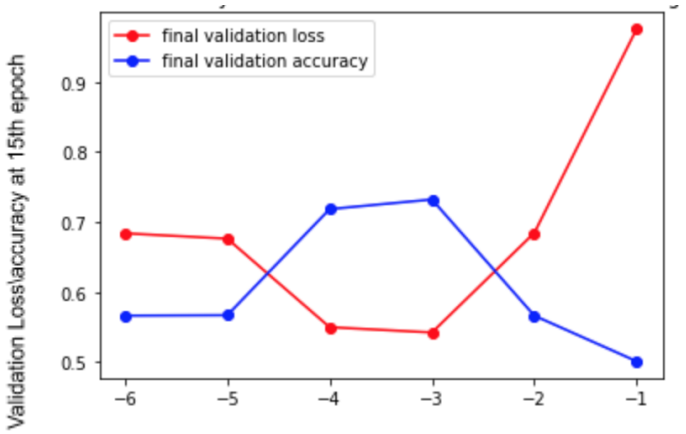

Figure 18: 15th epoch validation accuracy/loss as a function of the base-10 logarithm of learning rate

From the graph it can be seen that the optimal logarithm of the learning rate for the Adam optimiser corresponding to the lowest validation loss and highest validation accuracy is -3. This corresponds to a learning rate of 0.001, and is the default used by the Adam optimizer. As a result, the optimal learning rate for this model is 0.001.

4.2. Final Optimised Hyperparameters

Below displays the final results for the optimised hyperparameters found during one run of the optimisation algorithm.

Table 9. Final optimised parameters

Number of neurons in subsequent LSTM layers	Optimal number of neurons found: 256
Train:validation split	Optimal split found: 50:50
Dropout rate	Optimal rate found: 0.9
Activation function	Optimal function found: sigmoid
Number of neurons in first LSTM layer	Optimal number of neurons found: 512
Learning rate	Optimal rate found: 0.001

It must be noted that answers slightly vary from run to run even when using a seed value of 2021 when creating the test data set and training data set. This is due to the random fluctuations in how keras and tensorflow splits the training data set into a

validation set and training set. This was beyond control of the experimentation process but should be accounted for. In future, it would be ideal to run the algorithm multiple times and obtain the mode of each optimal hyperparameter found for final model evaluation. This would obtain the most accurate hyperparameters for evaluation in future, but due to google colaboratory GPU time constraints, this could not be done in this iteration of the algorithm.

4.3. Limitations and Alternate Methods

Initial optimization plans for the model involved making use of the Grid Search within the sklearn python library with Google Colab facilitating the execution. During initial training runs, Google Colab was either detecting inactivity or the accounts were running out of GPU time resulting in the session being terminated. To remedy this we attempted to run the Grid Search on a smaller parameter space but this still resulted in the same issues mentioned above.

An alternate plan was then implemented to make use of the sklearn library's Randomized Search as this would not test every combination as was done with Grid Search. Through researching best practices and advised values the initial test was done with 50 iterations on the original parameter space. This solution also resulted in session terminations before completion. Finally, a Randomized search using a reduced parameter space and a reduced number of iterations to 30 was attempted but this also ended with session termination.

Attempts were made to run the tuning on local hardware but this caused crashes due to the high amount of RAM it reported requiring during computation. As such we decided to change our validation strategy to the one presented above.

4.4. Model Evaluation

The final model was evaluated based on the hyperparameters listed in table 9. The model was fitted based on these parameters and the training and validation metrics for this fine-tuned model are displayed below.

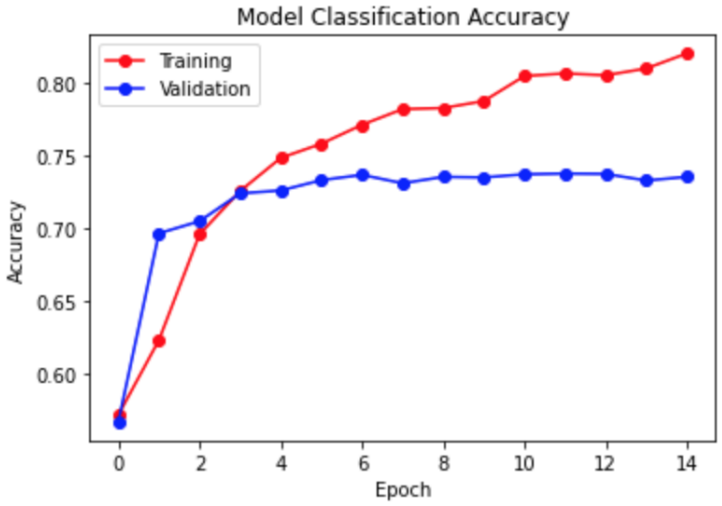

Figure 19: Fine-tuned model training and validation accuracy

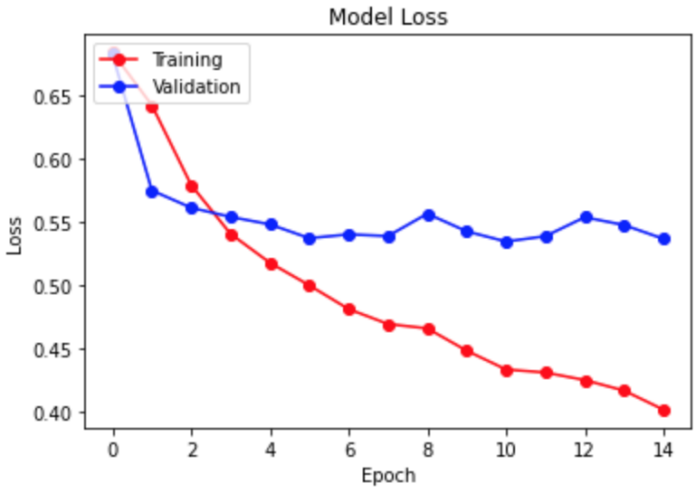

Figure 20: fine-tuned model training and validation loss

As can be seen by figures 19 and 20, the issue of overfitting seen in figures 10 and 11 have been reduced substantially. The model now has a more or less steady decrease in both training loss and validation loss. However, the validation loss depicted does seem to steady out at around 0.55. Ideally we wish to have a validation loss with a steady decrease. Looking at the accuracy curves it is also evident that has increased slightly, implying a better classification capability of this fine tuned model.

Testing the model on the 10% ratio test data generated, a confusion matrix as follows was obtained.

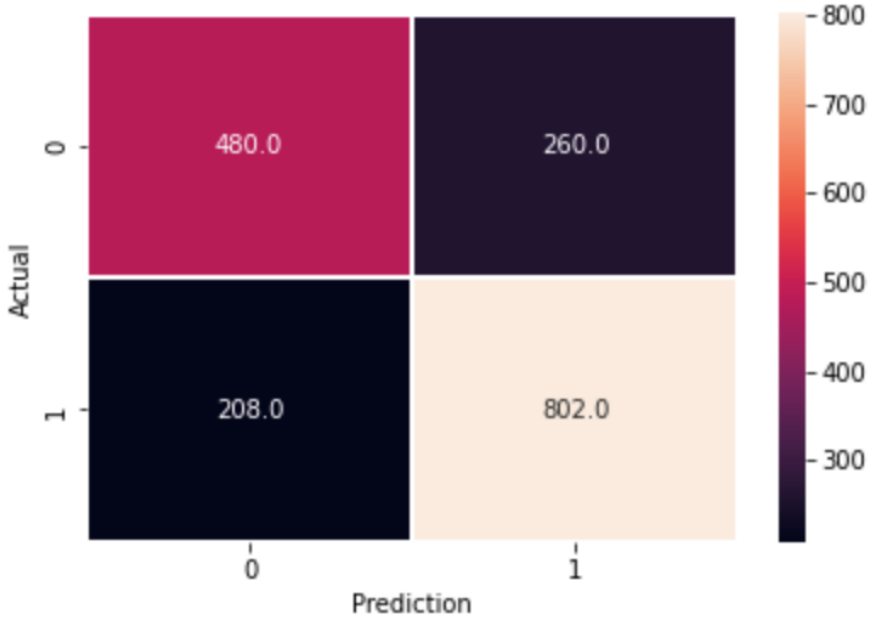

Figure 21: Confusion matrix of fine-tuned model

With a final accuracy score of 73.26%.

5. Test Results and Conclusion

5.1. Test Results

In this section, we describe the test results from the 3 network implementations described in this report. The models have been relabeled for clarity:

- *Baseline* refers to the model described in section 2.
- *Benchmark* refers to the pre-existing base model we described in section 3.1.
- *Tuned Benchmark* (Final Model) refers to the model we described in sections 3 and 4.

The tests we performed used a test set containing 10% of the data. The test set was randomly chosen across all 3 models.

Table 10. Performance of the described models

Metric	*Network Implementations*		
	Baseline	**Benchmark**	**Tuned Benchmark**
Accuracy (%)	57.60	69.77	73.26
False Positive Rate (%)	42.30	25.68	24.48
True Positive Rate (%)	40.00	63.86	69.77

In Table 10, we can observe that the Tuned Benchmark created in this report outperformed the benchmark by approximately 3.5%.

Whilst this managed to outperform the baseline significantly in most of the metrics above, it was relatively similar to the benchmark in Miss-Rate as indicated by the False Positive Rate, with values of 25.68% for the benchmark and 24.48% for the tuned benchmark. This indicates that roughly 25% of the predicted Positive (1) reviews are incorrectly classified as Positive/Recommended reviews, in the tuned model. This suggests that the probability that a true negative will be missed by the test is 25% in the tuned model: an approximate 1% improvement from the benchmark.

However, the Tuned Benchmark beats out the Benchmark significantly by approximately 6% in Sensitivity as indicated by the True Positive Rate. This indicates that 69.77% of the Tuned Benchmark predictions were actually correctly classified. This suggests that the probability of an actual positive will test positive is 69% for the Tuned Benchmark and 63% for the Benchmark.

5.2. Limitations

The limitations have been described in the individual sections up above but for a complete view of the test model, we will briefly reiterate them here. The usage of a small dataset relative to the average machine learning problem provided a limit on how well the data could be learnt without overfitting. The need to change optimization strategies due to technical difficulties with Google Colab and local hardware led to a less rigorous validation scheme. More rigorous pre-processing for the removal of stop words could not be completed owing to the GPU usage limit of Colab, as this takes at the

minimum of $O(n^2)$ time. These main issues provide the most significant impact on the outcome of the final model testing.

5.3. Conclusion

In this report, we set out to create a model that classifies the sentiment of reviews in the form of a binary recommendation. We started by creating a baseline model that classified more than half of the reviews correctly. We then used an existing codebase as a benchmark and a starting point for improvement over our baseline model. The benchmark model performed well, as it was a specialized implementation. Despite the limitations mentioned above, the final model managed to outperform both the baseline and the benchmark model in all metrics. While the degree to which the final model outperformed the benchmark is not huge it does show how careful tuning can make a difference in a model with smaller datasets.

5.4. Future Investigations and Research

Possible improvements to the optimisation method performed could involve running the algorithm multiple times and taking the mode of each of the optimised parameters. This would aid in reducing the statistical variation seen from run to run in predicting these hyperparameters and would ensure the most frequently accurate hyperparameters are being used instead of using hyperparameters generated by one optimisation. Other activation functions could also be tested against each other in the optimisation process in future iterations. Lastly, other parameters such as batch size and weight decay should be accounted for and optimised. Due to time constraints surrounding Google Colab, not too many hyperparameters could be optimised, but in future iterations of this optimisation algorithm, these should be taken into account.

6. References

[1] Steam Usage and Catalog Stats for 2021. (2021). https://backlinko.com/steam-users

[2] Bailey, D. (2021). Steam just reached 50,000 total games listed. https://www.pcgamesn.com/steam/total-games

[3] Introducing Steam Reviews. (2021). From https://store.steampowered.com/reviews/

[4] Feldman, R. (2013). Techniques and applications for sentiment analysis. Communications of the ACM, 56(4), 82-89.

[5] Protection of Personal Information Act of 2013, No. 4 of 2013 (2013). https://www.gov.za/sites/default/files/gcis_document/201409/3706726-11act4of2013protectionofpersonalinforcorrect.pdf

[6] How should I choose the optimum number for the neurons in the input/hidden layer for a recurrent neural network?. (2021). https://www.researchgate.net/post/How_should_I_choose_the_optimum_number_for_the_neurons_in_the_input_hidden_layer_for_a_recurrent_neural_network

[7] Brownlee, J. (2016). How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras. https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/

[8] Brownlee, J. (2020). Hyperparameter Optimization With Random Search and Grid Search. https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/

[9] Tutorials | TensorFlow Core. (2021). https://www.tensorflow.org/tutorials