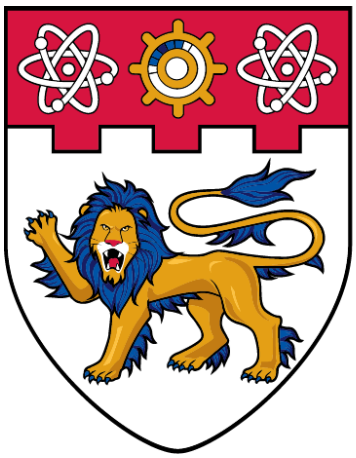

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042 Neural Network and Deep Learning

Project C: Sentiment Analysis Report

23 November 2020

Group Members:

Low Yu Benedict, U1821762E
Tan Rui Ming Raymond, U1720967K
Ng Wan Ying, U1822711E

Table of Contents

1. Introduction

This report covers our team's progress on the chosen Project Idea C, where we were tasked to develop deep learning techniques for Text Sentiment Analysis (TSA).

Natural Language Processing (NLP) has gained significant improvements since the 2010s as a result of machine learning — in particular deep neural networks — proving to be exceptionally effective in solving NLP tasks. These days, NLP has gained widespread usage; for example, email services now incorporate spam filters with the aid of NLP, while news groups as well as government agencies utilize NLP to sift through and identify fake news.

While the applications of NLP is vast, sentiment analysis stands out as a promising tool to be explored. Sentiment analysis in the form of text classification analyzes text and determines if the sentiment is positive, negative or neutral. Sentiment analysis can be used to automate systems and provide companies the utility to understand their customer's sentiments, making more informed business decisions with these new found actionable insights.

To get reliable and stable results, we decided to use the IMDB dataset. This is because the IMDB dataset is significantly large, at around 50k reviews and are used as benchmarks for several state of the art networks.

In this report, we will introduce three neural networks, Convolutional Neural Network (CNN), Recurrent Neural Network (RNN) and Bidirectional Encoder Representations from Transformers (BERT). These networks are used in our project to solve our sentiment analysis problem. At its core, this report will tackle the following challenges as specified in the project requirements:

1. To avoid using recurrent networks in order to speed up computations
2. To deal with small datasets, that is, with insufficient number of training samples
3. To deal with domain adaptation, that is, how one can adapt a network train in one domain to work in another domain

This report will explore the effectiveness of RNNs (one of the most common implementations for sentimental analysis) and CNNs (traditionally popular for image recognition tasks, CNNs are faster compared to RNN as a result of fewer computations needed). We will then continue to explore how one could apply data augmentation or transfer learning to deal with a lack of data for training. Furthermore, we will show how transfer learning can serve as an effective solution for domain adaptation. Finally, we will discuss the results of all three networks, as well as the effectiveness of data augmentation and transfer learning using empirical data.

2. Review of Existing Techniques

In recent times, Attention Models have begun gaining popularity in the field of NLP. In addition to Attention Models, both CNNs and RNNs are also considered highly viable for solving NLP problems. In this section, we will discuss what makes CNNs, RNNs and Attention Models so effective in solving our text sentiment analysis problem.

2.1. Recurrent Neural Network

RNNs were created to solve sequence prediction problems. As such, it is extremely useful in solving linguistic tasks, as language contains context in both the front and back directions of a sentence. However, for the implementation of RNN, bidirectional layers were not used and thus, only information before the data point is used in our RNNs.

2.1.1. Gated Recurrent Unit

Gated Recurrent Unit (GRU) is a variant of the RNN architecture that uses gating mechanisms to handle the flow of information between cells. GRU has the ability to hold onto long-term dependencies that stems from the computation within the GRU cell to produce the hidden state, where this hidden state is able to hold both the long-term and short-term dependencies simultaneously. A GRU cell contains the update gate and the reset gate which are responsible for regulating the information to be kept or to be selectively filtered out at each time step. The update gate is computed using previous hidden state and current input data; The reset gate is derived and calculated using both the hidden state from the previous time step and the input data at the current step.

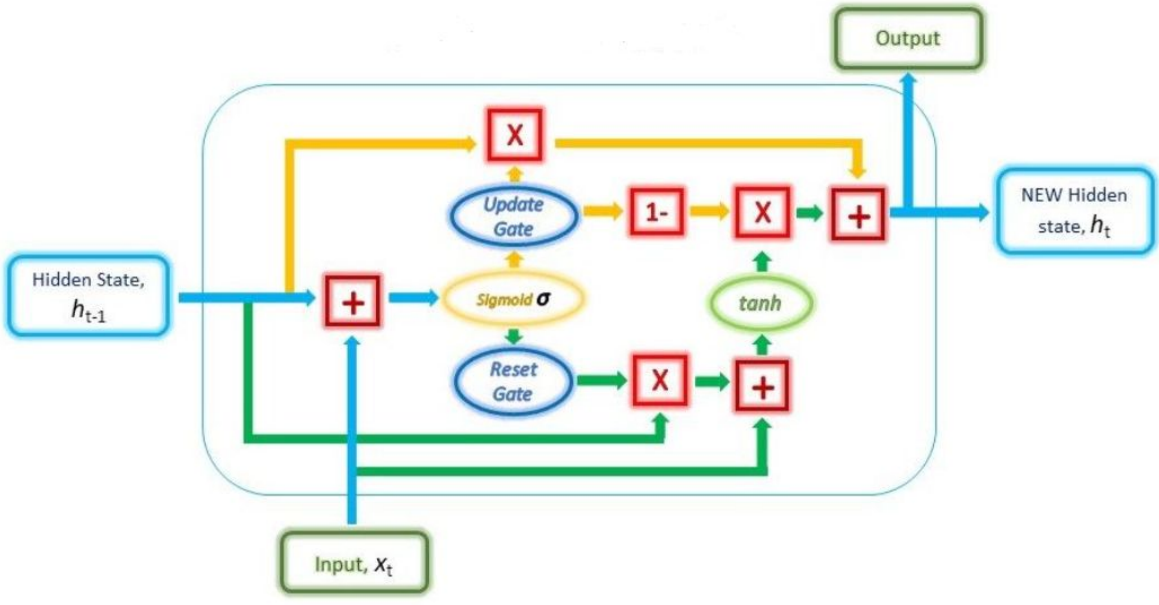

Figure 1: General architecture of GRU. Image from https://blog.floydhub.com/content/images/2019/07/image14.jpg [1]

2.1.2. Long Short-Term Memory

Long Short-Term Memory (LSTM) is yet another variant of the RNN architecture that consists of feedback connections and is designed to remember information over arbitrary time intervals. A common LSTM unit is made up of a cell, an input gate, an output gate and a forget gate. The input gate manages the magnitude to which a new value flows into the cell; the forget gate manages the magnitude to which a value remains in the cell; the output gate manages the magnitude to which the value in the cell is used to compute the output activation of the LSTM unit. The activation function of the LSTM gates used is the logistic sigmoid function.

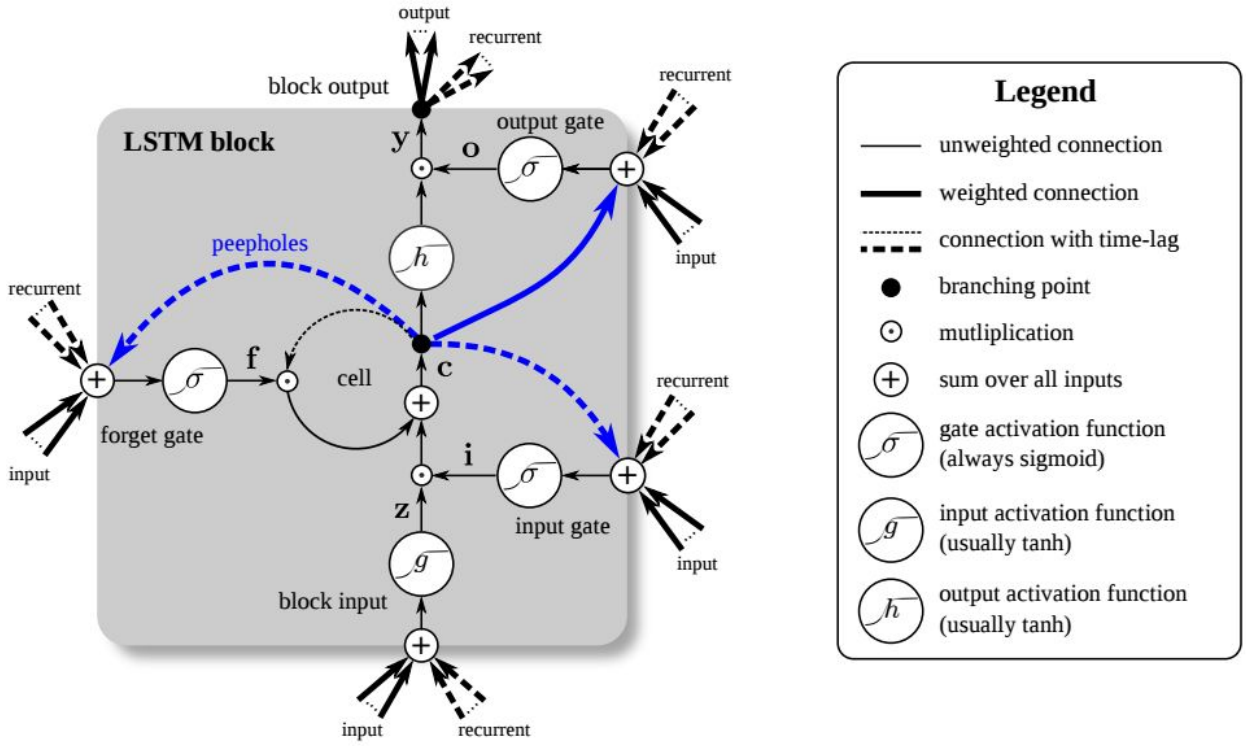

Figure 2: General architecture of LSTM. Image by Klaus Greff and colleagues as published in LSTM: A Search Space Odyssey.[2]

2.2. Convolutional Neural Network

Convolutional Neural Network (CNN) is a class of Deep Neural networks which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The applications of CNN includes image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, brain-computer interfaces, and financial time series. The preprocessing required in a CNN is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, CNN has the ability to learn these filters/characteristics.

In text classification, the CNN is a feature-extracting architecture — meant to be integrated into a larger network, and to be trained to work in tandem with the larger network to produce an outcome. The CNNs layer responsibility is to extract meaningful sub-structures that are useful for the overall prediction task at hand.

2.2.1. Architecture

Networks with convolutional and pooling layers are efficacious for classification tasks in which we expect to find strong local clues regarding class membership even though these clues can appear in different places in the input. Convolutional and pooling layers allow the model to learn to find such local indicators, where certain sequences of words are good indicators of the topic regardless of their position in the text.

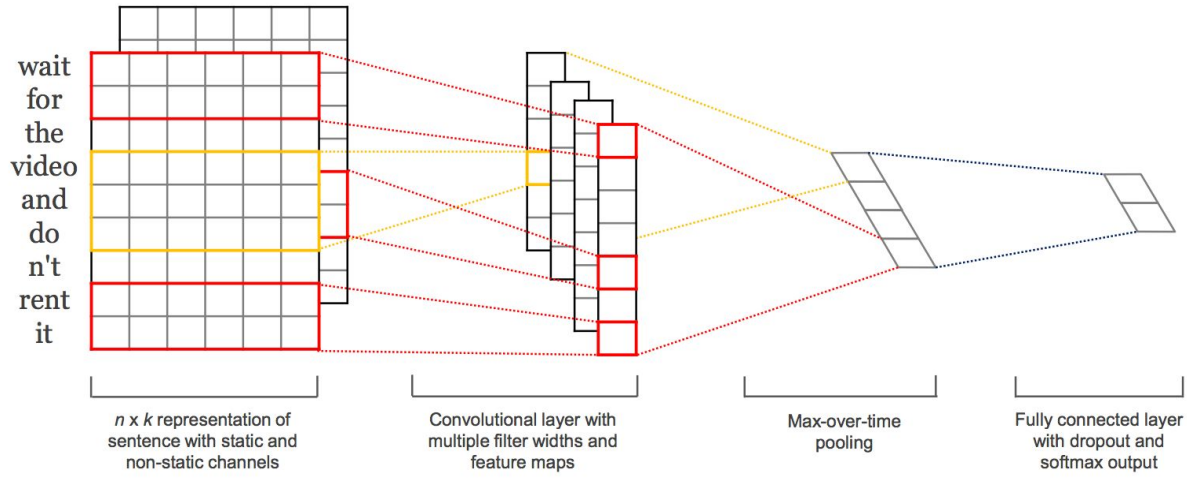

Figure 3: General architecture of CNN for NLP. Image from "Convolutional Neural Networks for Sentence Classification", 2014.[3]

2.3. BERT

BERT (Bidirectional Encoder Representations from Transformers) is a state of the art model that was developed by Google. Unlike the other networks used above, BERT is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks. In the case of BERT, the model is first pre-trained on a large amount of data using the BooksCorpus (800M words) (Zhu et al., 2015) and English Wikipedia (2,500M words). A document-level corpus rather than a shuffled sentence-level corpus is used to extract long contiguous sequences.

As such, for sentimental analysis, all we need to do is to use the pre-trained BERT model and fine-tune it with IMDB movie review dataset, reducing the time required to train while increasing the accuracy of the model.

2.3.1. Architecture

BERT makes use of Transformers, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms — an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary.

The input is a sequence of tokens, which are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors of size H, in which each vector corresponds to an input token with the same index. There are 2 main types of model sizes, being BERT-Base and BERT-Large. The BERT-Base contains 12 Transformer block layers, 768 hidden size and 12 Attention heads with a total of 110M parameters. On the other hand, the BERT-Large contains 24 Transformer block layers, 1024 hidden size and 16 Attention heads with a total of 340M parameters. The general architecture can be seen from Figure 4 below.

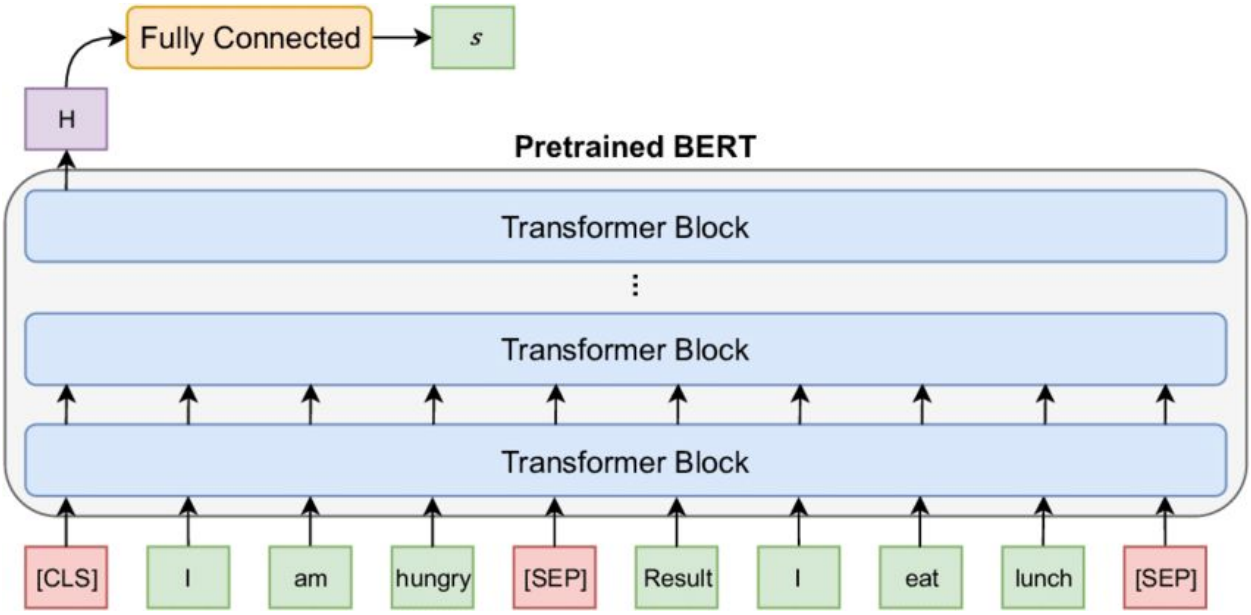

Figure 4: General architecture of BERT. Image from "BERT Explained: State of the art language model for NLP", 2018 [4]

Another huge advantage of BERT as opposed to directional models such as CNN and RNN, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word), allowing it to generalise and better deal with datasets of different domains. This is further explained in **Section 5** under Transfer Learning.

3. Methods

3.1. Pre-processing

In our exploratory data analysis (EDA) it was found that the IMDB dataset consisted of two columns, reviews and the sentiments with the sentiment being either positive or negative for any review. As the sentiments are categorical in nature but were represented as strings, we encoded the negative reviews with 0s and positive reviews with 1s.

3.1.1. Stratified Split

Through our EDA, it was also discovered that the dataset was evenly split with half of it being positive reviews and the other half being negative reviews. While stratified split is usually used as a good measure for when a dataset is unbalanced, we stuck to using stratified split nonetheless as it'd make sure that our training and testing datasets properly represented both classes of reviews. The ratio of our training to testing dataset were set to 80:20 respectively.

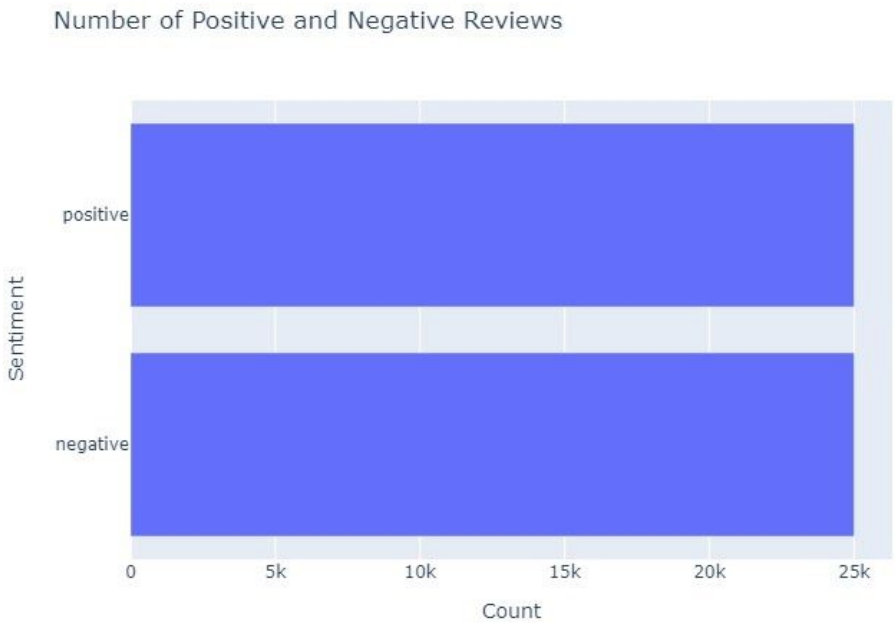

Figure 5: Count of positive and negative reviews

3.1.2. Word Tokenization

Tokenization is the process of encoding text into numerical values. This is helpful as our model interpretes numerical data, and thus can only be trained after encoding has taken place. Broadly speaking, there are two kinds of tokenization — words and character tokenization.

We decided to perform word-based tokenization instead of character-based tokenization as the reviews were quite long, with a median length of 176 words per review. With this in mind, we were not confident of a character-based model to perform too well.

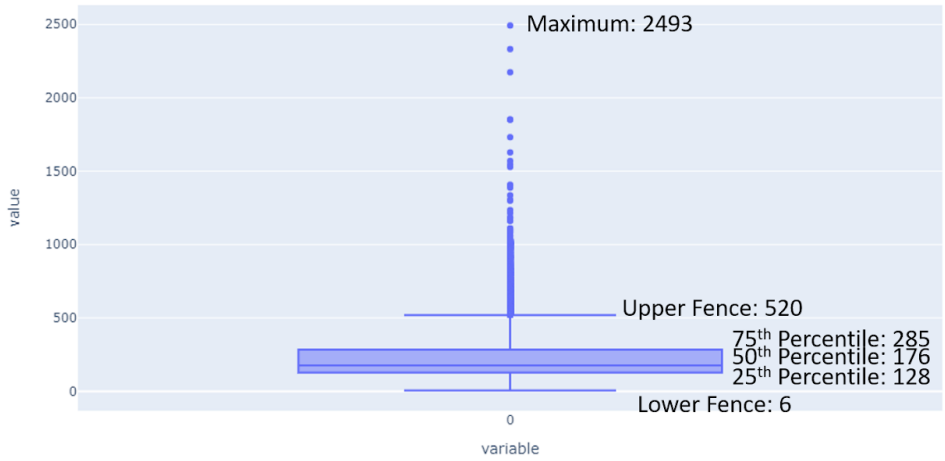

Figure 6: Distribution of word length in IMDB dataset

We used Keras's Tokenizer utility class and kept the parameters to its default. As such, the Tokenizer did several simple steps to identify words. Firstly, it removed all punctuations and converted all texts to lower-case. Subsequently, the default Tokenizer functions as a white space tokenizer — splitting words by white space and then assigning a value to it.

During our EDA, it was also discovered that our training dataset alone had 112,225 unique words. We decided to set an arbitrary value of 50,000 as our vocabulary size, which worked out well enough through the process of our experiments and thus was not changed. This means that while our Tokenizer would tokenize all 112,225 words, it would only tokenize the 50,000 most common vocabulary (in our training dataset) when actually applied on our training dataset reviews. Words that are not in the top 50,000 occurrence were tokenized as "<OOV>" and assigned an integer value of 0.

3.1.3. Padding

Padding is done to standardize the input shape for our model. For our project, we used post-padding pre-processing. For post-padding, given a padding length of *n* we perform padding by either cutting off any sentence with length longer than *n* at it's *n*th word, or adding 0s to a sentence with length shorter than *n* until it reaches length *n.*

With the information as shown in Figure 6, we decided to set our padding length at 520 — the upper fence that demarcates outliers in our training dataset.

3.2. Training and Implementation

For our project, we trained two different models, a CNN and an RNN. Furthermore, for each model we experimented with different architectures. This section covers the methods used for training our models, the architecture of these models, and the hyper-parameters involved in our grid search. The goal was to find the optimal CNN and RNN models architecture as well as the

hyper-parameters for solving our problem. Also, the “optimal” CNN and RNN models would later be used to evaluate the effectiveness of data augmentation.

3.2.1. Convolutional Neural Network (CNN)

For our CNNs, we tried a total of three different architectures:

1. 2-Layer CNN
2. 3-Layer CNN
3. 4-Layer CNN

All three architectures include dropout layers of varying probabilities determined through trial and error. The full information on the final architectures of these CNNs are included in Appendix A. The models were trained over 150 epochs, using Adam optimizer with a learning rate set to 0.01, and a batch size of 512, to study the behavior of each architecture. This was important for the subsequent grid search process where we used the EarlyStopping and ReduceLROnPlateau callbacks, and thus had to have a good understanding of how our model trains. Both callbacks work by monitoring the validation loss, and are triggered after *p* number of epochs from when the minimum validation loss was reached, where *p* is the patience value. As such, by understanding the training behaviour, we can make an informed decision on the patience value for our callbacks.

3.2.1.1. 2-Layer CNN

After plotting the loss and accuracy of the 2-layer CNN, it was clear that it had converged as early as the second epoch and from then only began overfitting. As such, we decided that the EarlyStopping patience was to be set at 3 and the ReduceLROnPlateau patience at 6. The aim of the low patience is to counteract the extremely fast overfitting behaviour, reducing the learning rate as early as possible and hopefully allow the model to learn without suffering from overfitting.

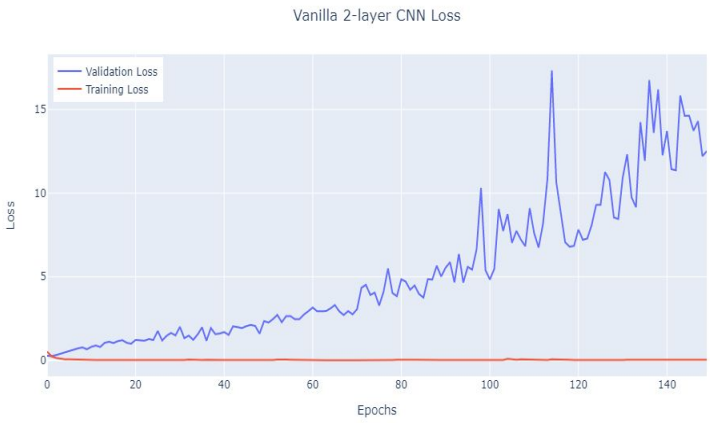

Figure 7a: Loss of Vanilla 2-Layer CNN

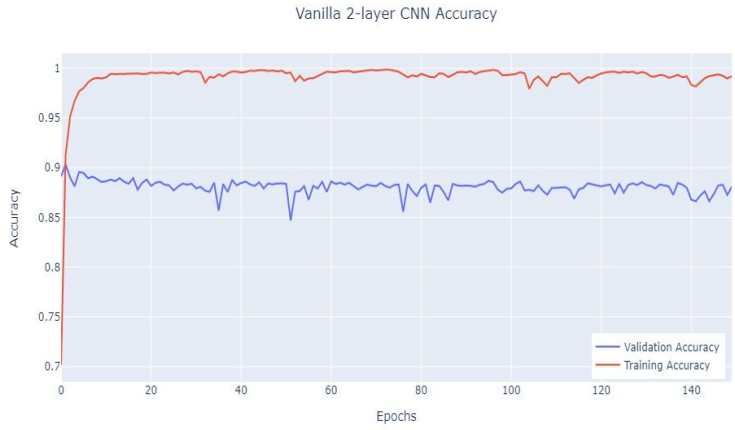

Figure 7b: Accuracy of Vanilla 2-Layer CNN

3.2.1.2. 3-Layer CNN

The 3-layer CNN showed very similar behaviour in terms of validation accuracy as the 2-layer CNN. However, the loss of the 3-layer CNN was a lot more stable before epoch 20. As such we decided to set the ReduceLROnPlateau patience at 4, and the EarlyStopping patience at 8. Ideally, this would allow for a gradual reduction in learning rate, taking advantage of the somewhat smooth loss of the architecture to allow for more epochs of training to be done before termination.

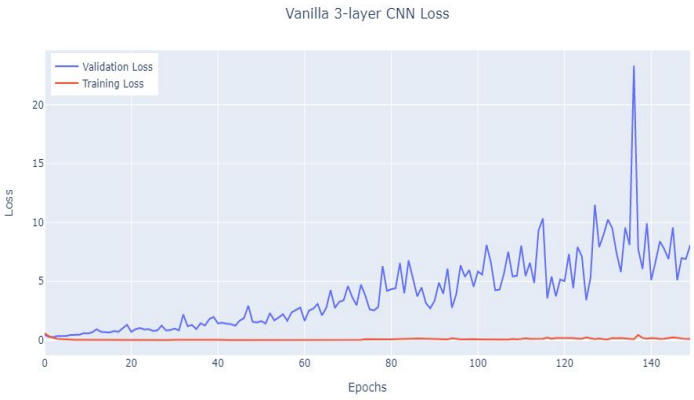

Figure 8a: Loss of Vanilla 3-Layer CNN

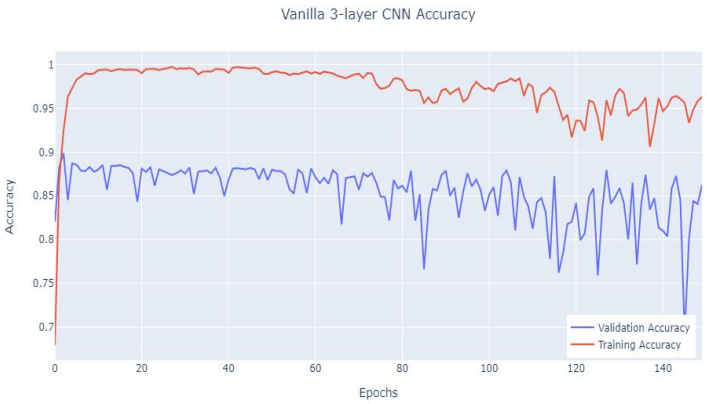

Figure 8b: Accuracy of Vanilla 3-Layer CNN

3.2.1.3. 4-Layer CNN

The 4-layer CNN displayed a similar performance in terms of accuracy as the 2-layer and 3-layer CNN. However, its loss stayed noticeably more stable, staying within range of 0.3 to a maximum of 1.84 over 150 epochs of training. We decided to set the ReduceLROnPlateau patience at 10, and EarlyStopping patience at 20.

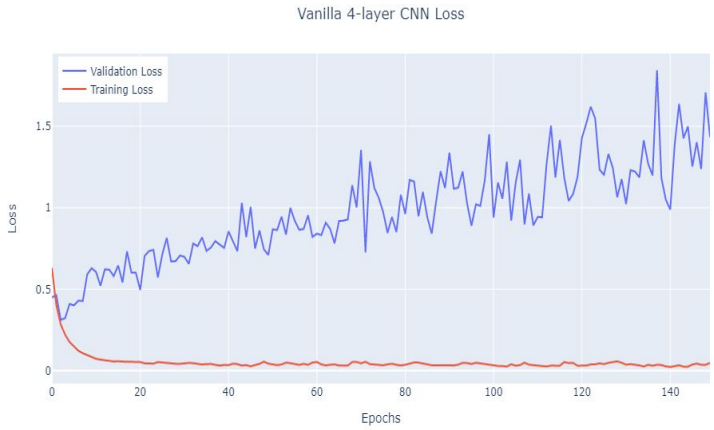

Figure 9a: Loss of Vanilla 4-Layer CNN

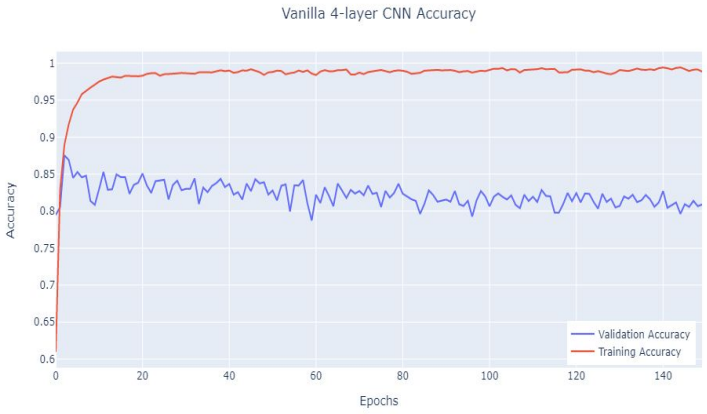

Figure 9b: Accuracy of Vanilla 4-Layer CNN

Table 1. Patience value for each architecture in CNN

Architecture	ReduceLROnPlateau Patience	EarlyStopping Patience
2-Layer CNN	3	6
3-Layer CNN	4	8
4-Layer CNN	10	20

3.2.2. Recurrent Neural Network (RNN)

For our RNNs, we tried a total of four different architectures:

1. 1-Layer GRU
2. 2-Layer GRU
3. 1-Layer LSTM
4. 2-Layer LSTM

All four RNN architectures include dropout layers of varying probabilities determined through trial and error. The full information on these architectures are included in Appendix B. We trained all four models over 250 epochs, using Adam optimizer with a learning rate set to 0.01, and a batch size of 512, to understand the behaviour of each model so as to decide on an optimal patience value for our callbacks during the grid search. 250 epochs were used instead of 150 as LSTMs required a longer time to converge as compared to CNNs, in our experiment.

3.2.2.1. 1-Layer GRU

The 1-Layer GRU converged at about epoch 9, where the validation accuracy stayed stable until about epoch 31. However, the model deteriorated very rapidly from epoch 32 onwards and did not recover. With such a trend in mind, we had a window of about 20 epochs of patience for our callbacks, and it takes about 10 epochs to converge. We decided to set the ReduceLROnPlateau patience at 5 as it should probably kick in shortly after convergence (minimum loss occurs at epoch 8). We set our EarlyStopping callback at 10 to allow for the GRU model to train for longer epochs with a lower learning rate.

Figure 10a: Loss of Vanilla 1-Layer RNN GRU

Figure 10b: Accuracy of Vanilla 1-Layer RNN GRU

3.2.2.2. 2-Layer GRU

The 2-Layer GRU was more stable than its 1-layer counterpart, with the accuracy dipping after epoch 140. However, its loss behaves similarly to its 1-layer counterpart. As such, we set the ReduceLROnPlateau patience at 4, but set the EarlyStopping patience to be longer, at 12.

Figure 11a: Loss of Vanilla 2-Layer RNN GRU

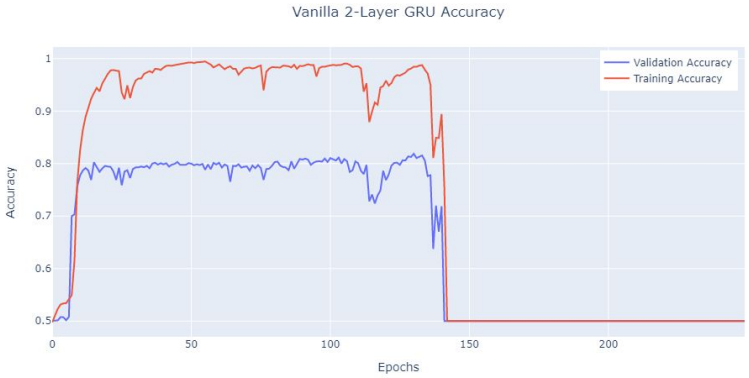

Figure 11b: Accuracy of Vanilla 2-Layer RNN GRU

3.2.2.3. 1-Layer LSTM

The 1-layer LSTM model converged at about epoch 13 and maintained a steady validation accuracy up till epoch 86. In addition, the loss stayed relatively low despite being quite erratic. Ultimately the 1-layer LSTM was quite stable and as such the ReduceOnLRPlateau patience was set to 8, and the EarlyStopping patience at 16.

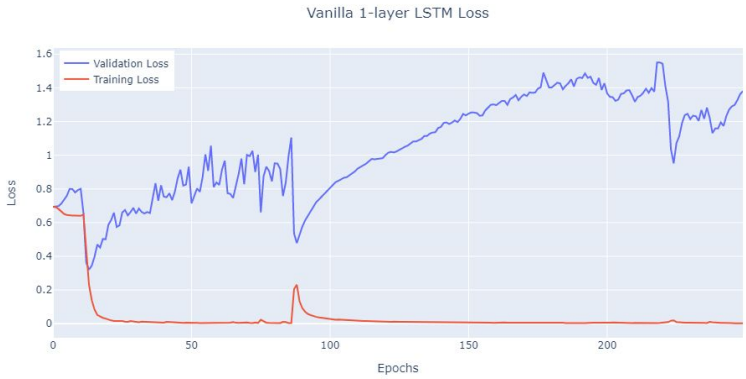

Figure 12a: Loss of Vanilla 1-Layer RNN LSTM

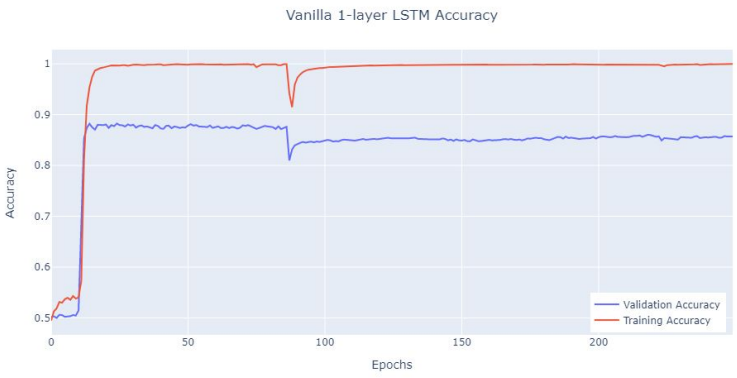

Figure 12b: Accuracy of Vanilla 1-Layer RNN LSTM

3.2.2.4. 2-Layer LSTM

For the 2-layer LSTM, it was discovered that the learning rate was too high and resulted in very poor performance. The training accuracy stayed almost consistently at 0.5, and the validation accuracy fluctuated around 0.5 as well. It was clear the model was not learning any patterns. As such, a learning rate of 0.005 was used instead.

With a new learning rate of 0.005, the model's accuracy improved significantly. By about epoch 18 the validation accuracy had already converged and subsequently, the model began overfitting. Despite this, the loss only gradually increased and did not fluctuate very greatly. As such, we set the ReduceLROnPlateau patience to 8 and the EarlyStopping patience to 16.

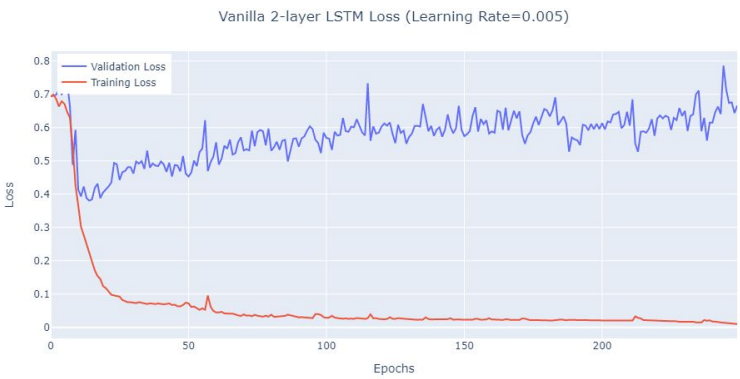

Figure 13a: Loss of Vanilla 2-Layer RNN LSTM with 0.005 learning rate

Figure 13b: Accuracy of Vanilla 2-Layer RNN LSTM with 0.005 learning rate

Table 2. Patience value for each architecture in RNN

Architecture	ReduceLROnPlateau Patience	EarlyStopping Patience
1-Layer GRU	5	10
2-Layer GRU	4	12
1-Layer LSTM	8	16
2-Layer LSTM (0.005 learning rate)	8	16

3.2.1. Grid Search (CNN)

Grid search was performed to find the optimal hyper-parameters for each CNN architecture. The objective was to find the architecture and optimal hyper-parameters that yields the highest validation accuracy at the end of training.

The hyper-parameters tuned for each CNN architecture is shown in **Table 3**.

Table 3. Hyperparameters of various CNN architecture

	Embedding Dimensions	Number of Filters in Convolutional Layer 1	Number of Filters in Convolutional Layer 2	Number of Filters in Convolution al Layer 3	Number of Filters in Convolutional Layer 4
2-Layer CNN	{10, 20, 30}	{20, 30, 40}	{5, 10, 20}		
3-Layer CNN	{20, 80, 100}	{100, 150, 200}	{70, 90, 110}	{40, 50, 60}	
4-Layer CNN	{20,30}	{100, 200}	{60, 90}	{30, 50}	{10, 20}

3.2.2. Grid Search (RNN)

Grid search was also performed to find the optimal hyper-parameters for each RNN architecture. Similarly, the objective was to find the architecture and optimal hyper-parameters that yields the highest validation accuracy at the end of training.

The hyper-parameters tuned for each RNN architecture is shown in **Table 4**.

Table 4. Hyperparameters of various RNN architecture

	Embedding Dimensions	RNN Layer 1 (GRU/LSTM) Units	RNN Layer 2 (GRU/LSTM) Units
1-Layer GRU	{128, 256}	{100, 200}	
2-Layer GRU	{128, 256}	{100, 200}	{100, 200}
1-Layer LSTM	{128, 256}	{100, 200}	
2-Layer LSTM	{128, 256}	{100, 200}	{100, 200}

3.2. BERT

For the implementation of BERT, we used Pytorch to train and test the model. This is because there are alot more resources for BERT on pytorch than tensorflow. As for the BERT model, we used the pytorch-pretrained-bert package from python. As the BERT models in the package have been initialized and pretrained, we would just need to feed the relevant input into the initialized model.

For the tokenizing process of the input, additional steps have to be done compared to that of RNN and CNN. Other than splitting the inputs into individual words and converting them into word vectors, the [CLS] and [SEP] token must also be added to mark the start and the end of each input. Furthermore, all of the words have to be converted to lowercase. To simplify the tokenizing process, we used the bert tokenizer provided in the pytorch-pretrained-bert package on the IMDB dataset.

Due to the constraints of the GPU in Google colab and the GPU Cluster, we used the BERT-base-uncased model instead of the more accurate BERT-large-uncased model. As BERT contains a huge number of parameters, we needed to reduce the batch size to 8 to fit the entire batch of data into the GPU. For the implementation of BERT, we also limit the input length to 512 while we pad input sizes that are less than 512. Despite reducing the model and the batch size, BERT still gives a higher accuracy than CNN and RNN.

3.3. Data Augmentation

Data is quite possibly one of the most crucial elements in machine learning. As such, sometimes the largest constraint for a researcher might be the lack of data. Data augmentation comes into play by creating data out of the existing dataset. As an added bonus, data augmentation can be helpful in creating more robust models as it adds noise to the dataset.
For our project, we closely followed the key ideas of this research paper [6] and implemented four different NLP data augmentation techniques.

1. **Synonym Replacement (SR)**
   In SR, we randomly choose *n* number of words in a sentence that are not stop words and replace these with a random synonym.
2. **Random Insertion (RI)**
   In RI, we randomly choose a word that is not a stop word and insert a new random synonym into the sentence. This is performed *n* number of times
3. **Random Swap (RS)**
   In RS, we randomly choose a pair of words in a sentence and swap the positions of these words. This is performed *n* number of times.
4. **Random Deletion (RD)**
   In RD, we randomly delete a word from a sentence with a probability of *α.*

The number of iterations *n* is determined by the length of a sentence. As such, we use the formula $n = \alpha l$ where α is a number between 0 and 1, and l is the length of a sentence.
Stop words were determined using the nltk library. The researchers suggested the values of α and the number of times to run the augmentations based on the size of the dataset. [6]
Comparing our training dataset size of 40,000 to the 5 datasets used by the researchers in the aforementioned paper we decided that it was acceptable to follow their proposed metrics, as the datasets they used were similarly sized like ours, ranging from a few thousand data points to over 30,000.

Table 5. Proposed metrics on recommended usage parameters based on dataset size

Training Dataset Size	α	Number of Data Augmentation Performed (iterated)
500	0.05	16
2,000	0.05	8
>= 5,000	0.1	4

5. Transfer Learning

Transfer learning which is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. In the case of BERT, it has already been pre trained to understand language structure and so it can be used as a starting point for sentimental analysis. This means that only the fine-tuning with the IMDB dataset is needed.

The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. For example, CNN and RNN, uses a left-to-right architecture, where every token can only attend to previous tokens. It is crucial to incorporate context from both directions for the model to generalise better. This is where BERT excels as it uses a bidirectional architecture. However, this means that BERT cannot be pre-trained using traditional left-to-right or right-to-left language models. Instead BERT will be pre-trained using two unsupervised tasks as described below.

5.1 Masked LM (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

1. Adding a classification layer on top of the encoder output.
2. Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
3. Calculating the probability of each word in the vocabulary with softmax.

The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. The process of the unsupervised task is illustrated below in Figure 14.

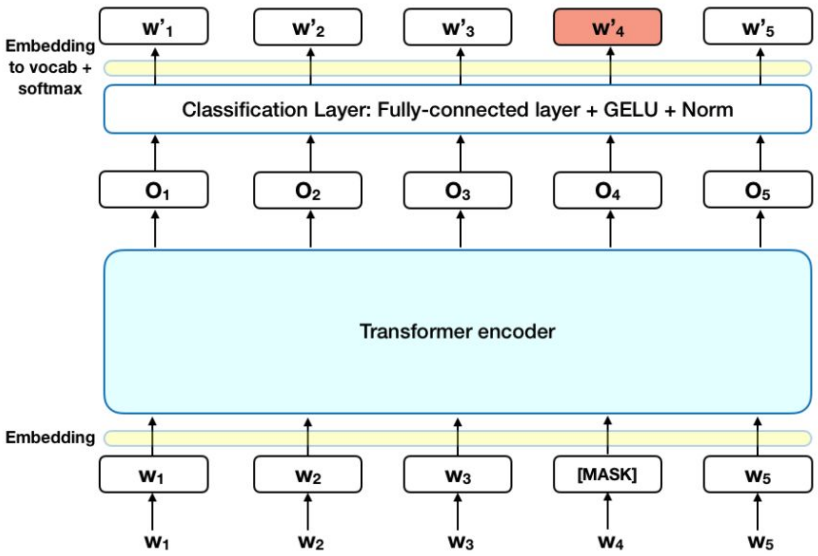

FIgure 14: illustrates the the process of the unsupervised task, MLM. Image from "BERT Explained: State of the art language model for NLP", 2018 [4]

5.2 Next Sentence Prediction (NSP)

Another unsupervised method that was used is NSP, which trains the model to understand the relationship between two sentences,which is not directly captured by language modeling. To achieve this, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the next sentence in the original document. 50% second sentence in the pair is the next sentence in the original document, while the other 50% is a random sentence from the corpus and is disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.
3. A positional embedding is added to each token to indicate its position in the sequence.

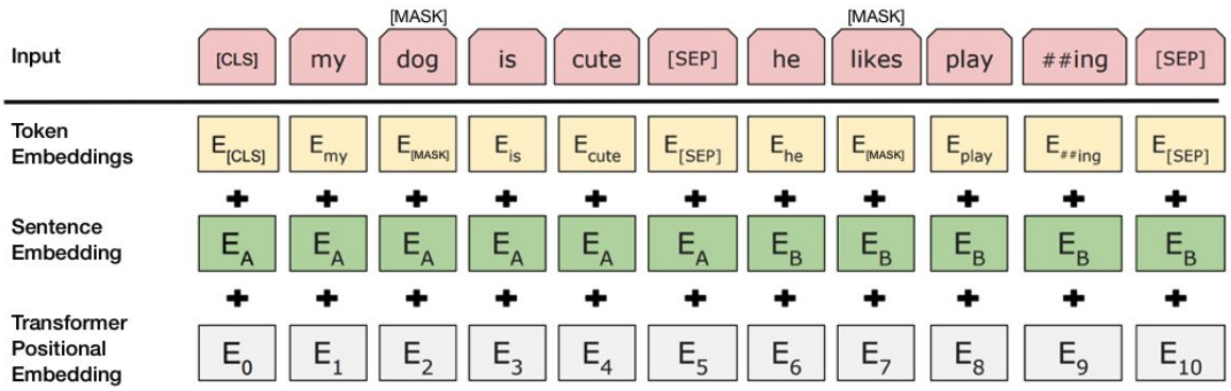

Figure 15: Shows BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings. Image from "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2019 [5]

To predict if the second sentence is indeed connected to the first, the following steps are performed:

1. The entire input sequence goes through the Transformer model.
2. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
3. Calculating the probability of IsNextSequence with softmax.

By using the above 2 methods, Masked LM and Next Sentence Prediction, the pre-trained model is able to generalise across different domains and perform sentiment analysis with a high accuracy even with little data.

5.3 Benefits of Transfer Learning

Training the model from scratch takes a very long time and requires a huge amount of data. However, using a pre-trained model removes the problem of small datasets as only fine tuning of the model is needed. As such, even a small dataset is able to properly fine-tune a model and achieve significantly better results. Furthermore, this allows the BERT to generalise and perform sentiment analysis with different domains. As BERT is pre-trained with a general and diverse dataset, the BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, without substantial task specific architecture modifications. This is demonstrated below in the results section.

A distinctive feature of BERT is its unified architecture across different tasks. There is minimal difference between the pre-trained architecture and the final downstream architecture. As such, it is able to domain adaptation easily as the same pre-trained model can be easily fine-tuned by datasets in another domain and still produce a high accuracy.

6. Experiments and Results

6.1. Grid Search

In this section, we will discuss the results of the grid search and select the best CNN and RNN architecture with its optimal hyper-parameters based on the validation accuracy recorded on the last epoch. To minimize clutter in our graphs, models that have a validation accuracy lower than 0.7 at the end of training are not considered here.

6.1.1. Grid Search (CNN)

It is apparent that the 2-Layer CNN architecture outperforms the 3-Layer CNN architecture and 4-Layer CNN architecture, clinching the top three spots with the highest validation accuracy. Furthermore, the best model is ahead of the runner up by about 0.019 accuracy. As such, we chose the 2-Layer CNN with 30 embedding dimensions, 40 filters in the first convolutional layer and 5 filters in the last convolutional layer as our best CNN model.

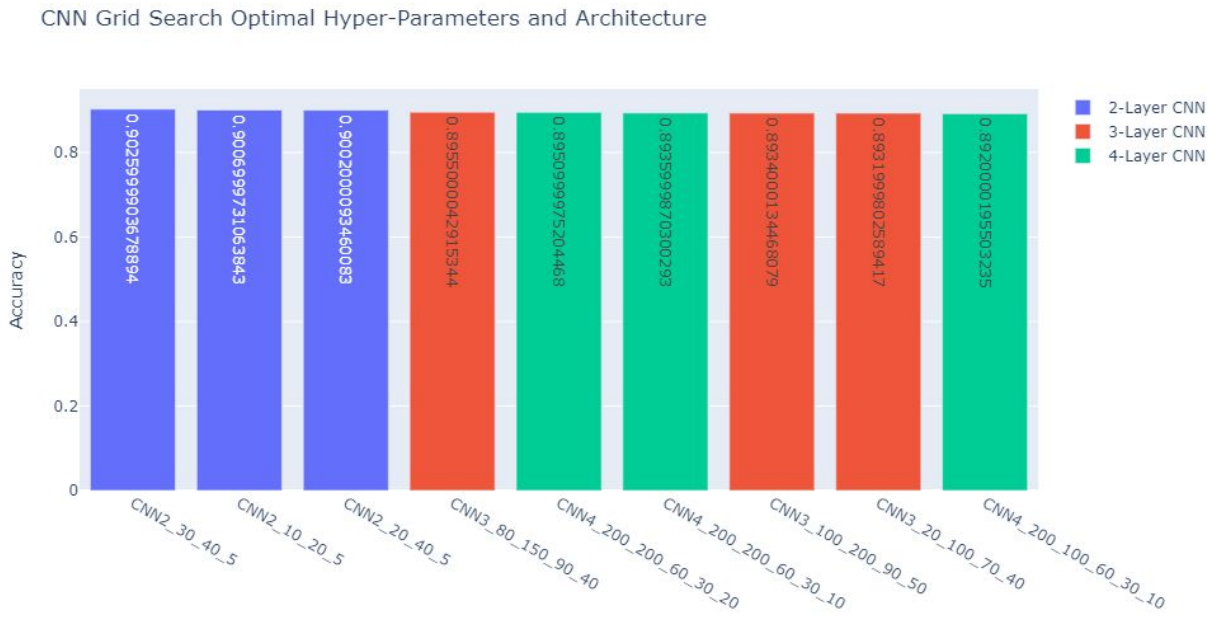

Figure 16: Illustrate the accuracy of the CNN models in descending order

6.1.2. Grid Search (RNN)

Comparing the validation accuracies of the various RNNs we trained, the highest validation accuracy is tied at 0.8816 for three different architectures with different hyperparameters.

1. **2-Layer LSTM**: 128 Embedding Dimensions, 100 filters at the first Convolutional Layer and 100 filters at the second Convolutional Layer.
2. **1-Layer LSTM**: 128 Embedding Dimensions with 100 filters at the first convolutional layer.

3. **2-Layer GRU:** 128 Embedding Dimensions, 100 units at the first GRU layer and 100 units at the second GRU layer.

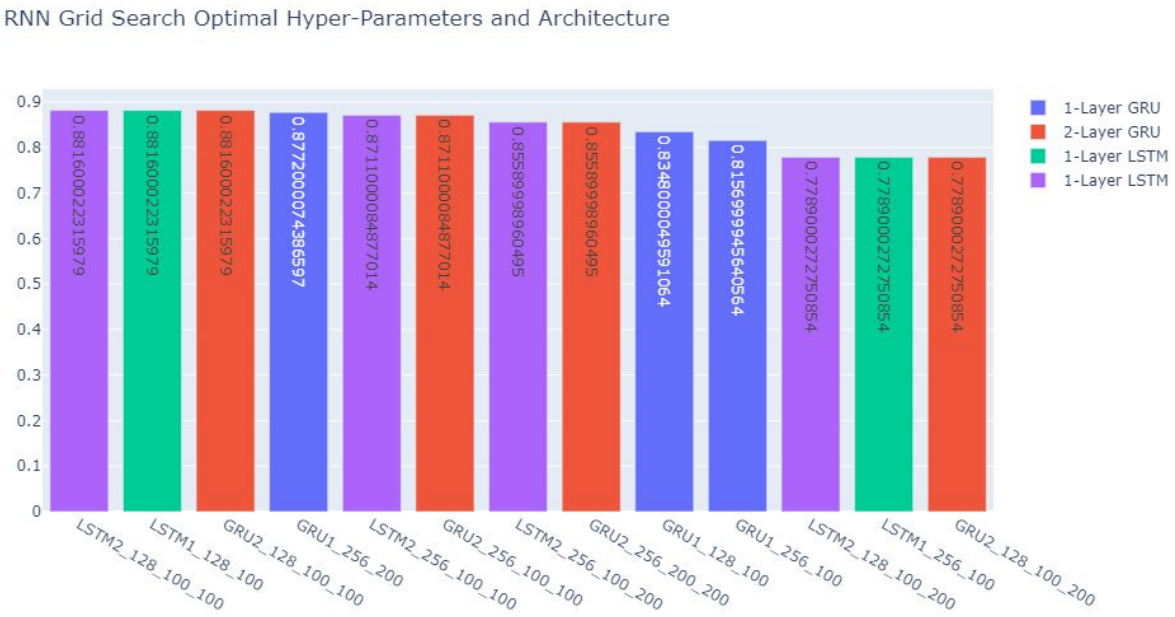

Figure 17: Illustrates the accuracy of GRU and LSTM models in descending order

We next compared the losses of these three models. The 1-layer LSTM model had the highest loss of all three. We considered the 2-layer LSTM and 2-layer GRU to have comparatively similar losses, as it is possible that the patience value for callbacks could have been better optimized thus resulting in a lower loss on the 2-layer LSTM model when trained.

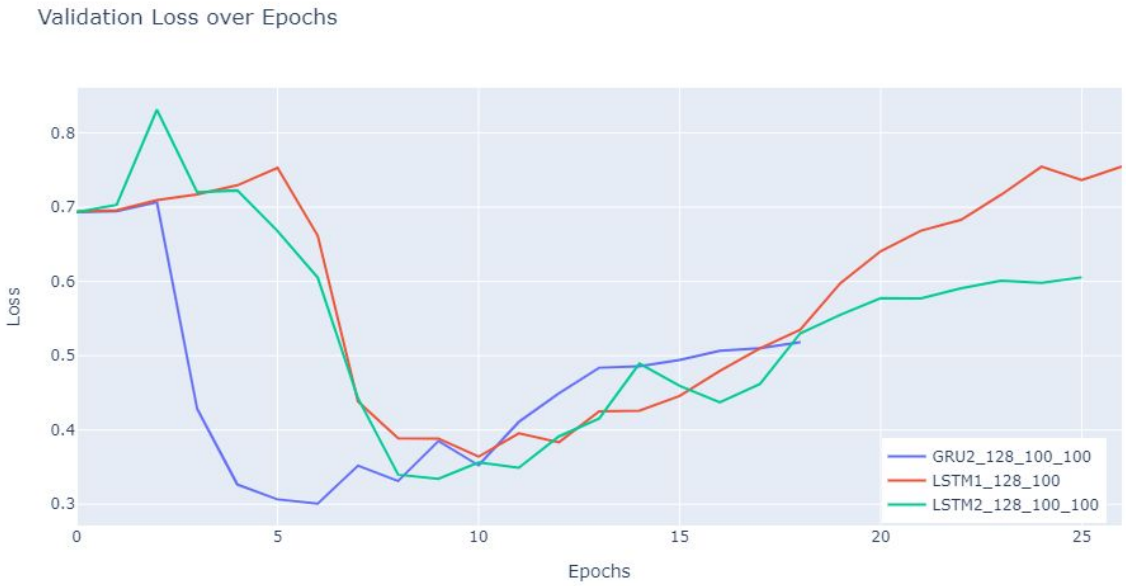

Figure 18: Compares the validation loss over epoch for GRU2, LSTM1 and LSTM2

To determine the better model (between the 2-Layer LSTM and 2-Layer GRU), we decided to study the time it took to train per epoch. This was necessary as the project had a rather tight timeline, and the limited resources as a result of having a 5-hour cap when using the GPU cluster meant that we had to be sensitive when it came to the time it took to train any model.

Figure 19: Compares the time taken between a 2 layer GRU and a 2 layer LSTM

From the results, it was clear that the 2-Layer LSTM trained much faster than the 2-Layer GRU model, shaving off 8 seconds per epoch of training. As we are choosing a model that will have to be trained using all 12 subsets of data (for data augmentation later) over three iterations, we decided that the 2-Layer LSTM model would be optimal for our project as we would not have to worry about exceeding the limited amount of time given on the GPU Cluster.

6.2. Data Augmentation

Using the best CNN and RNN model determined in Section 6.1.1 and Section 6.1.2 respectively, we trained the two models with varying subsets of our training dataset ranging from {1%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%}. The training was performed three times, and the average accuracy at the end of training is recorded.

After which, we applied data augmentation on the same subsets of data using the parameters proposed in **Table 5**. We then trained the same CNN and RNN model on these augmented datasets three times, and recorded the average accuracy at the end of training.

It is worth noting that our training dataset is of size 40,000 and as such a 1% subset of that contains 400 data points, while a 30% subset would already have 12,000 data points.

The results of the CNN with and without data augmentation can be seen in Figure 20. For our dataset and CNN model, data augmentation shows a decent improvement in results when the dataset is small (at about < 40% of original training dataset size of 40,000).

Subsequently, it would seem that adding data augmentation when the dataset is already large enough can result in slightly poorer performance; although it is possible that if more iterations were run before averaging out the accuracies, we might get a clearer picture on the effects of data augmentation on large datasets.

One problem with data augmentation is that it can run into the risk of mislabelling an augmented sentence; although in theory, deep learning should be considered robust enough to handle label noise [7]. Ultimately, data augmentation on our CNN model has shown to be helpful in smaller datasets, which is generally the reason for using data augmentation in the first place.

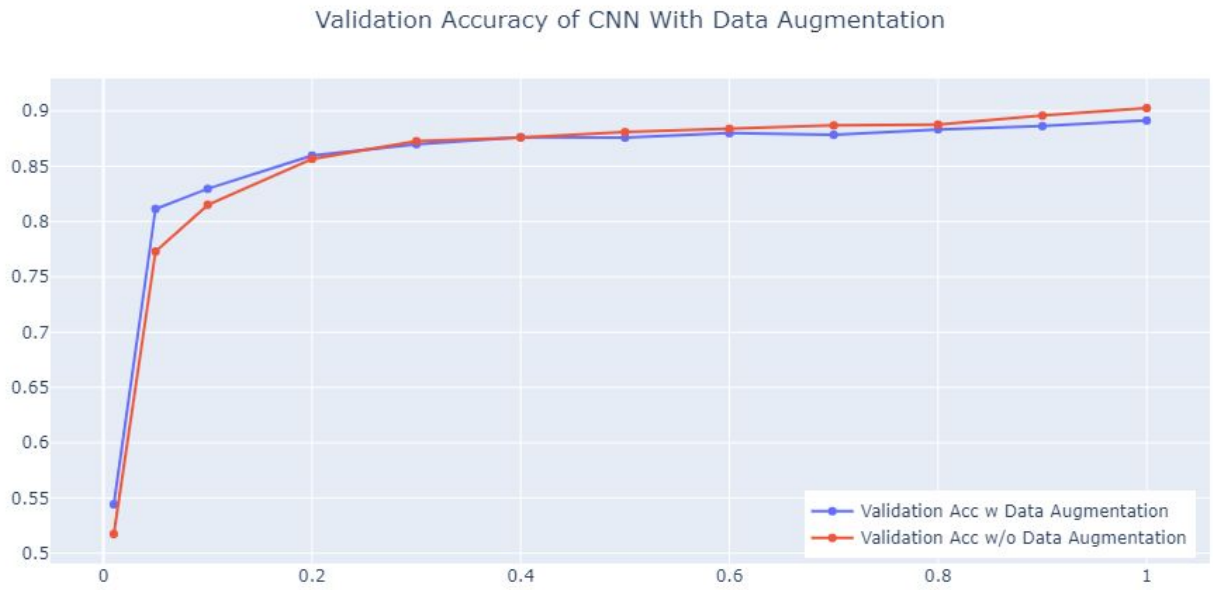

Figure 20: Compares the Validation Accuracy of the CNN model with and without Data Augmentation over the Size of the Dataset Split

In our RNN, data augmentation showed significant improvements across the board for every subset of our dataset. The dip in validation accuracy at 0.4 can be attributed to a training anomaly as the three iterations using augmented data with a subset size of 0.4 yielded validation accuracies of [0.8657, 0.5165, 0.7487] respectively. The effect of data augmentation is exceptionally noticeable when the dataset is small - from 1% to 30% of our dataset.

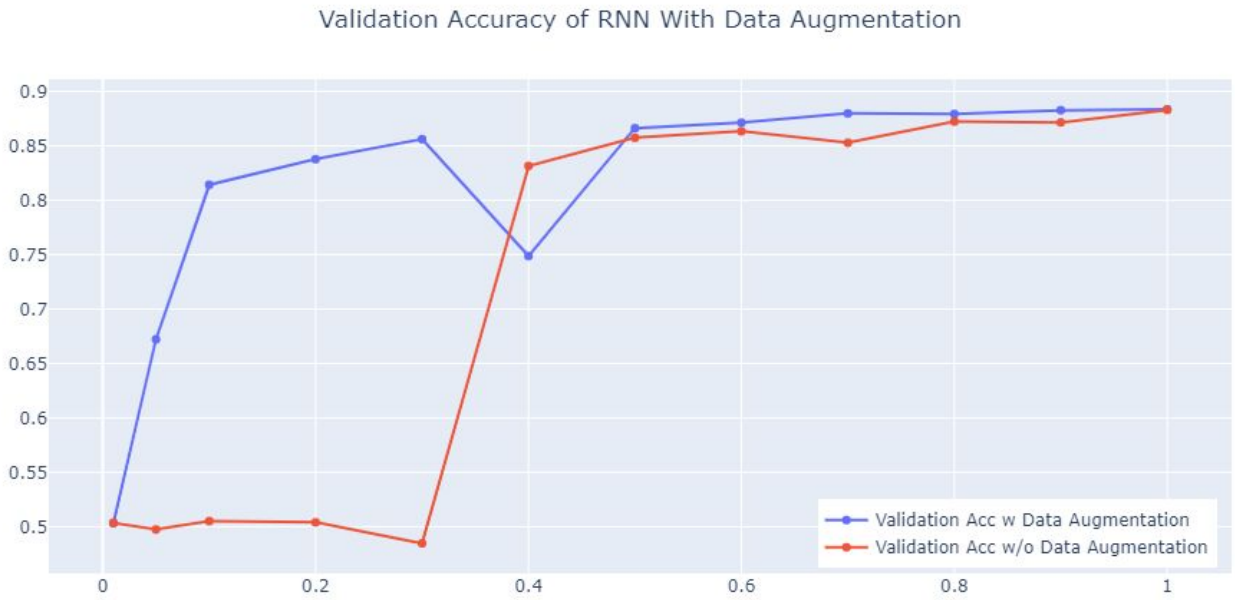

Figure 21: Compares the Validation Accuracy of the RNN model with and without Data Augmentation over the Size of the Dataset Split

With the results shown in this section, we can conclude that data augmentation is a viable solution for training models with a relatively small dataset (above 400 data points, as using 1% of the training dataset for augmentation yielded poor results).

6.3. Comparing Data Augmentation and Transfer Learning (BERT) Results

In this section we will compare the effectiveness of data augmentation and BERT. From the graph, it is clear that BERT provides significantly better performance over data augmentation. However, the simple implementation of data augmentation and relatively low cost of implementation (i.e. computational power, time taken) can keep data augmentation as a viable strategy in solving limited NLP dataset problems.

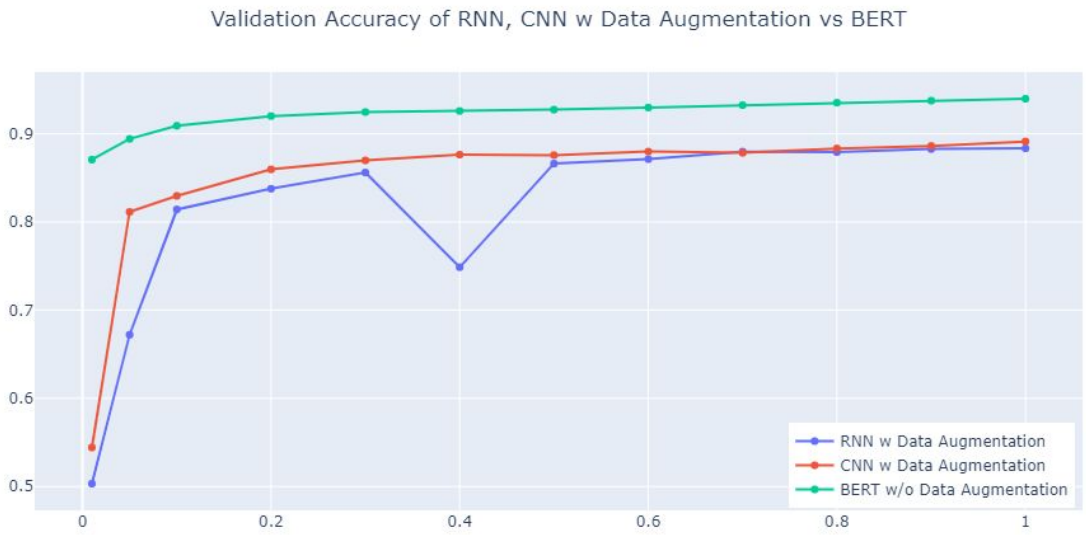

Figure 22: Compares the Validation Accuracy of BERT, RNN and CNN over the Size of the Dataset Split

6.4. Comparing RNN, CNN and BERT Results

In this section, we will compare the performance of BERT with our selected “best” CNN and RNN models. Comparing these models, BERT unsurprisingly comes on top by a large margin. It is worth noting that both models did not utilize and pre-trained embeddings and incorporated no use of transfer learning.

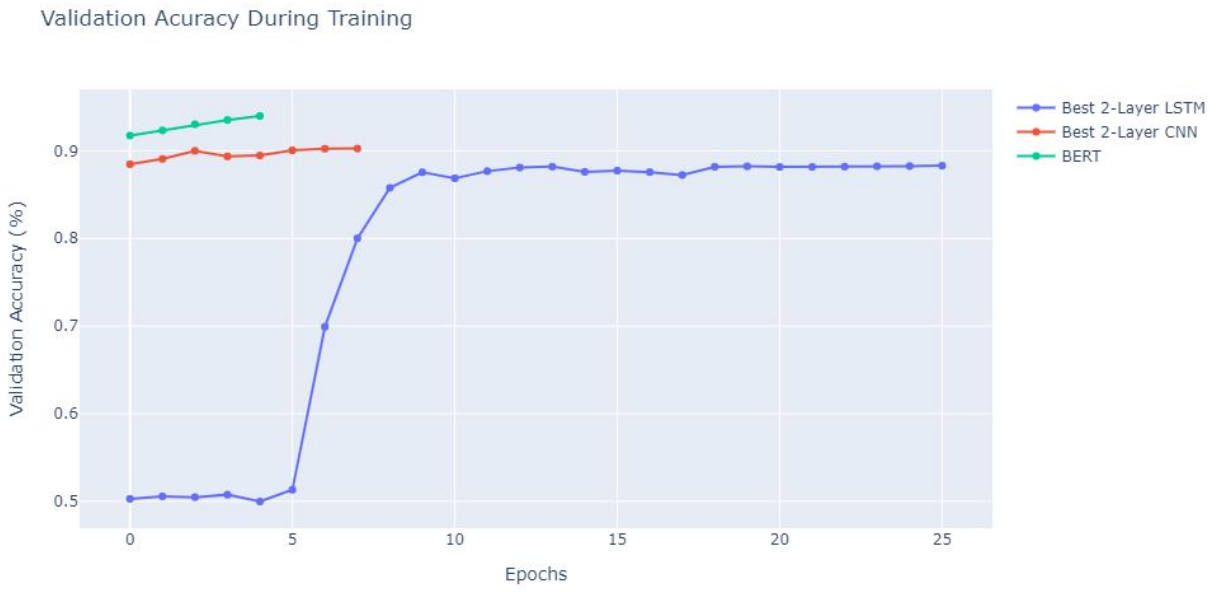

Figure 23: Compares the Validation Accuracy of BERT, the best 2-layer CNN and LSTM over epoch

While BERT is significantly more powerful than both RNNs and CNNs implemented in this project, it's downside is its extremely expensive computation - taking an average of 60seconds per epoch. Comparatively, our CNN model only took an average of 4.6 seconds per epoch.

As such, where BERT terminated its training at epoch 12, it would have taken an estimated 12 minutes to train. On the other hand, the CNN model terminated training at epoch 7 and would have only taken 32.2 seconds to train.

Figure 24: Compares the average time taken per epoch for BERT, best 2-layer LSTM and CNN

7. Discussion

In this section, we will suggest possible experiments that might improve the overall performance of our models. We will then discuss the rationale behind these suggestions and the predicted outcome such measures will produce.

7.1. RNNs and CNNs

The following suggestions will possibly improve the overall reliability of our research simply by training our models for more iterations and averaging out the results to reduce any uncertainty. These were not done due to limited computational resources and time constraint.

- **Perform more iterations during Grid Search**
  Currently, grid search is performed only once. It is possible that performing grid search 5-10 times and recording the average values may result in a different optimal CNN and RNN architecture as well as hyper-parameters. Ultimately, regardless of the outcome, running grid search multiple times will provide better insurance that the chosen model is indeed the best out of all that was trained.
- **Perform more iterations when testing the effect of Data Augmentation**
  In the same vein as the previous suggestion, performing more iterations during the testing of Data Augmentation can provide more reliable results. This project had only run the training of each subset (of dataset) over three iterations and averaged it out. As such we get anomalous training behaviour much like the one seen in Figure 21, where the subset size of 40% had one training which ended in 0.51 accuracy thus pulling the entire averaged out accuracy down severely.

The next few suggestions can possibly improve the overall performance of our models.

- **Incorporating a More Complex Tokenization**
  As a result of the simple Tokenization implemented in this project, we could have possibly lost some valuable information. It is not unthinkable that reviews may have typos, or typos in the form of blank spaces between words that would have otherwise made sense.

  Furthermore, our Tokenization removed apostrophes and hyphens, two punctuations that may provide greater insight into the intended meaning of a word. Lastly, our Tokenization changed words into lower-case. It is not uncommon for angry reviews to be written ALL IN CAPS or for names to start with a capital letter (i.e. “Grab” could refer to the ride-hailing service, but by changing it to “grab” instead we match it with the actual verb instead).

  We would thus recommend experimenting with different kinds of Tokenization that may account for more possibilities and might reduce the loss of contextual information.

- **Using Bi-Directional RNNs**
  In this project all RNN layers were unidirectional, and as such only have information references from the past. English as a language might have valuable information in a sentence that is found in the later part of a sentence. For example, in the sentence "That's a hot dog", the first three words would not provide the intended sentiment until the word "dog" appears, then we'd know that hot was not meant to mean the heat, but was instead referring to a food.

  Therefore it is possible that through the use of Bi-Directional RNN layers we might achieve a better accuracy on our RNN models.

- **Using Pre-Trained Embedding Layers**
  This suggestion is similar to the idea of transfer learning, but only uses the pre-trained embedding layer (such as GloVe word embeddings). This can provide a more accurate dimension for each word and can improve the overall accuracy of our model.

- **Adding Dropout Layers and Adjusting Dropout Probabilities**
  Some models like our 2-Layer CNN were extremely vulnerable to overfitting. As such we had to set the patience for EarlyStopping to be low, and this might have resulted in less time for the model to train. By adjusting the dropout probabilities and adding more dropout layers (i.e. recurrent dropout within the RNN layer itself) we might be able to tackle the overfitting problem and allow for our model to learn more over time.

7.2. BERT

The model that was used in this experiment is BERT-Base. As we only had a 12GB GPU available for the experiment, we needed to use a smaller BERT model, BERT-Base while decreasing the batch size to 8. With the limitation of a 5 hrs runtime for each job in the GPU cluster, we had to reduce the number of epochs from 10 to 5. Although this led to a slight reduction in the accuracy, it still performed way better than CNN and RNN. Given 2 or more GPU with a runtime of 24 hours, we could have used BERT-Large while running 10 epochs with a larger batch size. This would further improve the performance of the BERT implementation.

8. Conclusion

We performed various experiments involving the IMDB dataset. Firstly, performed grid search to obtain the optimal depth and hyperparameters for the CNN and RNN. We then collected and compared the time and accuracy of the three models. Furthermore, we illustrated how RNN and CNN performs badly with small datasets and how data augmentation improves the accuracy. The findings also show how BERT solves the issue of having little training samples and performs better than CNN and RNN without the need of data augmentation.

However, through our findings, we found that different types of models may be more suitable for sentiment analysis depending on the following limitations and constraints.

1. CNN is most suitable when time and resources are the main constraints and there is a sufficiently large amount of data available. This is because the CNN model is able to train within a shorter amount of time to give a considerable accuracy.

2. BERT is most suitable when very data is sparse and there are insufficient training samples. BERT is also versatile and able to deal with domain adaptation just by fine-tuning the pretrained BERT models. However, BERT should not be used if time or resources is a constraint. This is because BERT takes a considerable longer time and requires a lot more resources than both RNN and CNN.

In our experiments, we also realised the RNN performed slightly worse than CNN while giving a more unstable validation accuracy and loss. However, with the improvements mentioned in Section 7.1 such as the implementation of Bi-Directional RNNs and using a pre-trained embedding layer, we are confident that RNN will perform better than the current CNN used in the experiment.

9. References

[1] G. Loye, “Gated Recurrent Unit (GRU) With PyTorch”, Jul. 2019, FloydHub. Available: https://blog.floydhub.com/gru-with-pytorch/
[2] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, ‘‘LSTM: A search space odyssey,’’ IEEE Trans. Neural Netw. Learn. Syst., vol. 28, no. 10, pp. 2222–2232, Oct. 2017
[3] Y. Kim, ‘‘Convolutional neural networks for sentence classification,’’ presented at the Conf. Empirical Methods Natural Lang. Process. (EMNLP), Doha, Qatar, Oct. 2014, pp. 1746–1751.
[4] Rani Horev, “BERT Explained: State of the art language model for NLP”, Nov. 2018, TowardsDataScience. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270
[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” arXiv: 1810.04805, 2018. Available: https://arxiv.org/pdf/1810.04805.pdf
[6] J. W. Wei and K. Zou, ‘‘EDA: Easy data augmentation techniques for boosting performance on text classification tasks,’’ 2019, arXiv:1901.11196. [Online]. Available: https://arxiv.org/abs/1901.11196
[7] D. Rolnick, A. Veit, S. Belongie, and N. Shavit, “Deep learning is robust to massive label noise,” 2017, arXiv:1705.10694.

Appendix A

2-Layer CNN (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.Conv1D(40, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.3), keras.layers.Conv1D(20, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.3), keras.layers.Flatten(), keras.layers.Dense(20, activation='relu'), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')
3-Layer CNN (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.Conv1D(100, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.2), keras.layers.Conv1D(70, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.3), keras.layers.Conv1D(40, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.5), keras.layers.Flatten(), keras.layers.Dense(100, activation='relu'), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')
4-Layer CNN (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.Conv1D(100, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.2), keras.layers.Conv1D(70, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.3), keras.layers.Conv1D(40, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.4), keras.layers.Conv1D(20, 3, activation='relu'), keras.layers.MaxPooling1D(padding='same'), keras.layers.Dropout(0.5),

	keras.layers.Flatten(), keras.layers.Dense(100, activation='relu'), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')

Appendix B

1-Layer GRU (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.GRU(100), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')
2-Layer GRU (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.GRU(100, return_sequences=True), keras.layers.Dropout(0.5), keras.layers.GRU(100), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')
1-Layer LSTM (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.LSTM(100), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')
2-Layer LSTM (Random Weights Initialized)	keras.layers.Embedding(112226, 20, input_length=520), keras.layers.LSTM(100, return_sequences=True), keras.layers.Dropout(0.5), keras.layers.LSTM(100), keras.layers.Dropout(0.5), keras.layers.Dense(1, activation='sigmoid')