NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Text

Description automatically generated

**CZ4042**

**Neural Networks and Deep Learning**

**Assignment 2**

Report

Group members:

WONG YUAN NENG (U1822431K)

NG YI LIANG, RANDY (U1821394D)

LOH YI XUAN RENICE (U1822247D)

15 November 2021

Table of Contents

[1. Introduction 1](#_Toc87901997)

[2. Existing techniques 1](#_Toc87901998)

[2.1. CNN 1](#_Toc87901999)

[2.1.1. Architecture 1](#_Toc87902000)

[2.1.2. Limitations 2](#_Toc87902001)

[2.2. RCNN 2](#_Toc87902002)

[2.2.1. Architecture 2](#_Toc87902003)

[2.2.2. Advantages over CNN and RNN 3](#_Toc87902004)

[3. Methodology 3](#_Toc87902005)

[3.1. CNN and RCNN models 3](#_Toc87902006)

[3.1.1. Pre-processing of input data 3](#_Toc87902007)

[3.1.2. Embedder 3](#_Toc87902008)

[3.1.3. Structure of the classifier 3](#_Toc87902009)

[3.2. BERT multiclass classifier 4](#_Toc87902010)

[3.2.1. Pre-processing of input data 4](#_Toc87902011)

[3.2.2. Embedding 4](#_Toc87902012)

[3.2.3. Structure of our BERT multiclass classifier 5](#_Toc87902013)

[4. Experiments and results 6](#_Toc87902014)

[4.1. Hyperparameter tuning for BERT 6](#_Toc87902015)

[4.1.1. Choosing batch size 6](#_Toc87902016)

[4.1.2. Choosing number of hidden layers 7](#_Toc87902017)

[4.1.3. Summary 8](#_Toc87902018)

[4.2. Comparison of model architectures 9](#_Toc87902019)

[4.2.1. BERT multiclass classifier vs. existing techniques of CNN and RCNN 9](#_Toc87902020)

[4.2.2. BERT\_false vs. existing techniques of CNN and RCNN 9](#_Toc87902021)

[4.2.3. CNN vs. RCNN 9](#_Toc87902022)

[4.2.4. Summary 10](#_Toc87902023)

[5. Conclusion 10](#_Toc87902024)

[Appendix A 1](#_Toc87902025)

[Appendix B 2](#_Toc87902026)

# Introduction

In the past decade, advances in computational power and processing speed have allowed neural networks to surge in popularity within the field of machine learning. Traditionally, the training of feed-forward neural networks involved taking in the entire batch of inputs at once and back-propagating the gradients of weights and biases between the output and target labels. However, the learning of textual data has introduced a temporal component to the learning process since the inputs are read word by word. As such, the recurrent neural network (RNN) architecture was designed to handle this [1] by having a hidden layer to store the states which captures information from the previous iterations to be used in training future iterations.

However, RNN have large drawbacks when dealing with text processing and sentiment analysis. Firstly, since words at the start of each sentence are always taken in first, they have no information on the context of subsequent words [1]. Also, since words are trained sequentially, words just preceding the current word input are given more weight in training compared to words that have appeared long before the current iteration. This may cause misinterpretations of context since a current word could be more strongly linked to words that are further away and nearer to the start of the sentence in question.

Therefore, when it comes to text emotion recognition, since RNN tend to capture immediate local information and ignore global information such as the general emotion of the entire sentence, it has reduced efficacy in predicting the overall emotion of an entire sentence. As such, our project motivation is to develop a model that is capable of capturing both local and global information, then compare its performance with the existing research models for sentence classification, such as convolutional neural networks (CNN) and recurrent convolutional neural networks (RCNN).

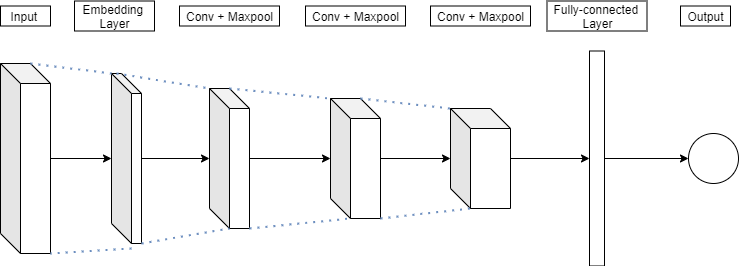
# Existing techniques

## CNN

One of the existing models used for sentence classification is the CNN model [2], which typically consists of convolution layers, pooling layers and a fully connected (FC) layer. The CNN model is best known for the field in computer vision such as object detection and image classification.

### Architecture

In this project, we build the CNN model to explore how well it performs in the field of Natural Language Processing (NLP), specifically Text Emotion Recognition (TER) relative to the BERT model. The architecture of the CNN classifier built consists of 5 layers – an embedding layer, 3 sets of convolutional with max pooling layers, followed by the FC layer.



**Embedding Layer**: The embedding layer is the first layer, and it outputs a 2D vector of input data such that each word is integer encoded and represented by a unique vector of binary numbers. The embedding layer is used compress the input space. Using the Keras embedding layer, we use a pre-trained word embedding. The weights of the embedding layer are initialised to 0 and the optimal values are learnt during training.

**Convolutional + Max-pool Layers**: For each layer, we use the 2D convolution layer with increasing kernel size. We applied max pooling over each window. The pooling layer is responsible for reducing the output dimensionality. The computational power required is reduced while extracting only prominent features.

**Fully connected Layer**: Finally, we concatenate all the layers to form the FC layer. The input data from the previous layers are flattened and fed to the FC layer. To prevent overfitting, a 0.5 dropout rate is utilised. The FC layer then performs classification based on the activation function. Since TER is a multiclass classification problem, the SoftMax activation function is used.

### Limitations

Firstly, CNN requires the usage of kernels. These kernels are matrix that moves across the embedding layer and work as filters to extract features from the input text. The window size of these kernels must be optimized such that vital information is not lost while maintaining a good time complexity. Using a small window may result in losing critical information. In our case, using a small window for convolution means that the computation is based on a smaller number of words. In images, local structure matters which means that semantic information is contained in adjacent window. However, in sentences, words do not have to be adjacent to be related. Therefore, with small window size, the model can only capture local information but may fail to capture the emotion of the entire sentence. In order to capture the global information, the filter must consider the context of the entire sentence and not only the context of a few words. This means that a larger window size of kernels is also required. However, this would result in a large feature space, making it difficult to train. Therefore, it is difficult to determine the window size of these kernels.

Secondly, regular CNN which consists of less than 10 layers do not perform as well in the field of NLP. On the other hand, deep CNN, typically when consisting of more than 30 layers, performs well in the NLP field. However, the CNN model with just a few layers is computationally costly due to the large numbers of training parameters, hence requiring a lot of time and GPU power to train.

## RCNN

Another technique that has been used to classify texts is the RCNN model. The RCNN model is the combination of Bi-directional Long Short-Term Memory (BLSTM) and the CNN neural network [1] [3].

### Architecture

According to the reference paper, we built the RCNN model to examine how well it performs in TER compared to other models. The architecture consists of 5 layers – an embedding layer, a BLSTM with dropout layer, a 2D convolution layer, a 2D max pooling with dropout layer and lastly the output layer.

Diagram, schematic

Description automatically generated

**Embedding Layer**: The first layer is the embedding layer. It is used to store the input data in a 2D vector by encoding each word into a unique vector of binary vector. Like the CNN model, the weights of the embedding layer are initialised to 0. During training, Keras attempts to find the optimal values for the weight matrix. A dropout rate of 0.5 is added to this layer.

**BLSTM Layer**: The BLSTM layer is made up of bidirectional LSTM cells, which are a type of RNN cell that is an extension of traditional LSTMs that is useful in learning order dependence. The BLSTM layer thus allows the model to have both backward and forward information about a sequence. A dropout rate of 0.2 is added to this layer.

**2D Convolutional Layer**: Using a set of filters, the 2D Convolution Layer is used to help extracts the features from the input data for classification.

**2D Max Pooling Layer**: The 2D Max Pooling Layer helps with model sequencing. As the pooling layer summarises the dominant features detected by the previous convolution layer, it enhances the model’s invariance to irrelevant translations. A dropout rate of 0.4 is added to this layer.

**Output Layer**: The input data from the previous layers are flattened and sent to the output layer which is the last layer of the RCNN model classifier. Using the SoftMax activation function, this layer classifies the emotion of the text.

### Advantages over CNN and RNN

The positive features of RNN are used to improve the CNN model thereby making this combined framework overcome some of the limitations of the CNN and RNN model. Given the unidirectional nature of RNN model, it can only capture the context of prior words. While the CNN is unable to capture global information due to the kernel sizes as explained in Section 2.1.2. However, the RCNN model can capture both local and global information. This is due to the addition of BLSTM layer which accounts for both left and right context. This means that the model can exploit both past and future information, allowing the model to capture long-term sentence dependencies as well as more contextual information.

# Methodology

In this project, we implemented a variation of the Bidirectional Encoder Representations from Transformers (BERT) model, as well as the CNN and RCNN models mentioned in section 2. We trained the models on both CROWDFLOWER and WASSA2017 datasets to fairly compare their performances.

## CNN and RCNN models

For each of these 2 models, we implemented and trained the models in TensorFlowV2, with reference to the non-TensorFlowV2 codes published by the authors of the research papers for these 2 models [4] [5] [6]. The model structures that we have implemented can be found in Appendix A.

### Pre-processing of input data

Before the input texts can be used as input data for the training of both the CNN and RCNN models, the texts must be pre-processed in the following ways [7]:

1. Convert all alphabets into its lowercase form using in-built function lower().
2. Remove all URLs found within the texts by using RegEx functions.
3. Remove stock symbols that holds no linguistical meaning, by using RegEx.
4. Remove twitter username handles defined as any word starting with “@”, by using RegEx.
5. Remove everything that is not a letter or apostrophe with a space.
6. Remove single letter words.
7. Tokenising sentences in each text into a list of words by using the nltk package.

While it is a common practice to also perform stemming and lemmatization and remove stop words, we decided not to do so in our methodology because implementation takes time and online articles argued against its effectiveness on pre-processing texts from “tweets” [7].

### Embedder

Both the CNN and RCNN models use pre-trained word embeddings through the Global Vectors for Word Representation (GloVe) vectors to provide the input texts with a pre-trained model of the dense vector representation of words [8].

We first tokenise the input texts by using Keras *Tokenizer* and convert them into sequences with the application of post-sequence padding by using Keras *pad\_sequence*s function [8]. We set the *num\_words* parameter of the *Tokenizer* to 10000 so that a maximum of 10000 new embeddings will be created for words from the dataset that cannot be found in the GloVe embedding, which we will be using to embed the sequence of tokens later.

### Structure of the classifier

The classifier is the CNN or RCNN architecture designed to take the embedded tokens of each input “tweet” as the real inputs into the neural network classifier. The neural network architectures are implemented according to the codes published as mentioned previously.

## BERT multiclass classifier

The BERT language model (BERT LM) was recently introduced by Google in 2018 and applies bidirectional training of transformers for language modelling [9]. This bidirectional training circumvents the hurdles of RNNs in linking the words from earlier in the text to the words that come later during model training. In addition, the use of a transformer, which is a popular attention model, allows the BERT LM to achieve a deeper level of language global context since extra weights can be learnt and placed on phrases far away from the current word in training, as long as they are closely related in context.

BERT LM has so far been a very popular model architecture for language modelling, and is often used as an encoder in classifier models by adding a neural network of class label predictions (ie. decoder) after the BERT LM. The resulting classifier is called the BERT classifier. Existing BERT classifiers, however, are often only binary classifiers. Thus, our project attempts to adapt the BERT binary classifiers to a multiclass situation by redesigning the decoder portion and training the newly redesigned BERT multiclass classifier on the given CROWDFLOWER and WASSA2017 multiclass datasets.

### Pre-processing of input data

Instead of performing data pre-processing in the manual way as in section 3.1.1, a BERT classifier has many accompanying pre-processors that is compatible to its BERT encoder, so we downloaded and deployed the pre-processor that we believed is the most suitable for our dataset of “tweets”. The pre-processor that we have chosen to use is bert\_en\_uncased\_preprocess/3, which is the 3rd version of the English language text pre-processor for BERT [10].

The pre-processor used is a model that was pre-trained on a vocabulary for English extracted from Wikipedia and BookCorpus, same as that of the models designed by the original BERT authors. Also, the text inputs were normalised in the “uncased” way by converting the text into lowercase before tokenising it into a list of words, and any accent markers are stripped off as well.

As such, we can instantly pre-process our input texts from our dataset by using each text as input into this pre-processor, which truncates the text to 128 tokens for pre-processing, and we will obtain the input mask, word ids, and type ids of the list of tokens originating from the respective text. Instead of reading the text input sequentially from left to right or right to left, the entire sequence of words is read together at once.

Text

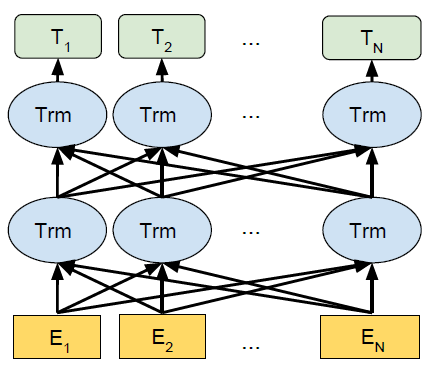
Description automatically generated

### Embedding

Instead of performing embedding on the lists of pre-processed tokens like in section 3.1.2, the BERT classifier embeds them using its BERT encoder to obtain a map of 3 important keys – pooled\_output, sequence\_output, and encoder\_outputs [11].

#### BERT encoder architecture

Different versions of the BERT encoder model architecture are readily available on the documentation site [11], each differing by the number of hidden layers (ie. transformer blocks) *L*, the hidden size *H*, and the number of attention heads *A*. The BERT encoder we have chosen is *bert\_en\_uncased\_L-12\_H-768\_A-12/4*, which is the 4th version of the English language BERT encoder made up of 12 transformer blocks (ie. *L*=12), each with hidden size 768 (ie. *H*=768), as well as 12 attention heads (ie. *A*=12). We have ensured that the chosen BERT encoder model architecture is compatible with the chosen BERT pre-processor model.



Each transformer block, labelled as *Trm* in the figure above, comprises of multiple attention heads, each using the self-attention mechanism on the different pre-processed tokens *Ei* of the same input sequence of embedded tokens *E*, which came from the output of the pre-processor, to derive the final token embeddings *Ti* in the form of a map of the 3 important keys as mentioned before. This encoder architecture is the main reason why BERT classifiers can model dependencies of words far away in the sequence of tokens of an input text, unlike the directional sequencing in CNNs and RNNs [12].

The BERT encoder is also a model that was pre-trained on a vocabulary for English extracted from Wikipedia and BookCorpus, just like its pre-processor model. Also, the text inputs were normalised in the “uncased” way by converting the text into lowercase before tokenising it into a list of words, and any accent markers are stripped off as well.

#### Masked Language Modelling (MLM)

The BERT encoder employs the MLM technique during its model pre-training [9] [13], where a small proportion of tokens in the sequence are masked at random before being fed into model training. The model then predicts the masked tokens using the non-masked words as context, and this increases the extent of learning the contexts of each word, therefore giving the model the ability to capture the global information of the whole sentence more accurately.

#### Next Sentence Prediction (NSP)

The BERT encoder also employs the NSP technique during its model pre-training [9] [13], where the encoder receives pairs of sentences as input and learns to predict if the second sentence is the subsequent sentence in the original document. During pre-training with NSP, the second sentence in each input pair is the real subsequent sentence for 50% of the time, and the second sentence is a random sentence from the corpus for the other 50%. This technique helps to train the BERT encoder to understand the relationships between sentences.

#### Output

The output from the BERT encoder is the map of the 3 important keys below:

1. ***pooled\_output*** represents the embedding for the input sequence of tokens for the entire “tweet”, and has the shape of [*batch\_size*, *H*]. This array is the key used for fine-tuning.
2. ***sequence\_output*** represents the contextual embedding for each input token in a “tweet”, and has the shape of [*batch\_size*, *seq\_length*, *H*].
3. ***encoder\_outputs*** are the intermediate activations of the L Transformer blocks, where each activation, which is a Tensor with the shape [*batch\_size*, *seq\_length*, ***sequence\_output***], is the output of one of the *L* Transformer blocks.

### Structure of our BERT multiclass classifier

The vanilla structure of our BERT multiclass classifier is a neural network with the sequence of layers:

1. Input layer
2. Pre-processing “layer” of the BERT pre-processor model from section 3.2.1
3. Encoder “layer” of the BERT encoder model from section 3.2.2
4. Dropout “layer” for the pooled output from the BERT encoder “layer”
5. Fully-connected linear output layer to output the class logits

We followed the BERT binary classifier tutorial by TensorFlow [14] and did not use the Softmax activation function in the output layer but instead defined the loss function later as *CategoricalCrossEntropy* with the *from\_logits* parameter set to *True*.

In addition, we introduced some hidden linear layers before the fully-connected output layer in an attempt to further optimise the model performance in the domain of text emotion recognition. We experimented with adding no hidden linear layers, adding 1 hidden linear layer, adding 1 hidden linear layer with dropout, as well as adding 2 hidden linear layers with dropout on each layer. The results of these experiments will be explained later in section 4.1.2.

During training of the BERT classifier, the main bulk of model training is at the additional hidden linear layers (if any) and the fully-connected output layer. The pre-trained parameters of the BERT encoder will only be fine-tuned if the *trainable* parameter is set to *True*.

# Experiments and results

We perform hyperparameter tuning for BERT and compare the BERT model trained with the optimum parameter values to the CNN and RCNN models via a series of experiments, each using 3-fold cross-validation.

## Hyperparameter tuning for BERT

In the first experiment, we tune the hyperparameters for BERT, using only the CROWDFLOWER dataset. The hyperparameters involved are batch size, number of hidden linear layers, and the number of hidden neurons in these linear layers.

### Choosing batch size

An optimum value for the batch size is chosen from the list [8, 16, 32, 64, 96, 128] by comparing the performances of the 6 models trained using each of these different batch sizes. The model architecture of each model is a vanilla BERT multiclass classifier, which comprises of only the BERT pre-processor, the BERT encoder, and a fully connected output layer. The output layer is a linear layer of 13 neurons, which is equivalent to the number of output classes of the CROWDFLOWER dataset. Each model is trained for 10 epochs with 3-fold cross-validation, and the model accuracy and model loss in each epoch are determined by taking the mean of the values among the 3 folds. Then, from the test loss (ie. *val\_loss*) and test accuracy (ie. *val\_accuracy*) of each of the 10 epochs, we obtain the minimum test loss (ie. *min\_val\_loss*) and the maximum test accuracy (ie. *max\_val\_accuracy*). The results for all the 6 models are as presented in the tables below.

A screenshot of a computer screen

Description automatically generated with low confidence A screenshot of a computer

Description automatically generated with low confidence

We then draw 2 simple line plots to observe how the *min\_val\_loss* values decrease and how the *max\_val\_accuracy* values increase with the increase in batch size.

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

The plots show a trend suggesting that the optimum batch size will be larger than 128, but the training of any model using batch sizes of more than 128 on the SCSE GPGPU have led to memory limit errors as the amount of memory consumed is too large for the GPGPU memory quota to handle. As such, we choose 128 as the batch size to be used for the rest of the experiments.

### Choosing number of hidden layers

An optimum value for the number of hidden layers is chosen from the list [16, 32, 64, 96, 128, 256] and an optimum number of hidden layers is chosen, both by comparing the performances of the 18 models trained using each permutation of the pair of hyperparameters.

Again, each model is trained for 10 epochs with 3-fold cross-validation, and the minimum test loss (ie. *min\_val\_loss*) and the maximum test accuracy (ie. *max\_val\_accuracy*) are obtained in the same way as in section 4.1.1.

#### BERT multiclass classifier with no hidden layers

This model structure, named *no\_Linear*, is equivalent to that of the vanilla BERT multiclass classifier used in choosing the batch size. An illustration of this can be found in Appendix B. The results for the minimum test loss and maximum test accuracy for this model are as shown below.

Graphical user interface

Description automatically generated

Because there are no hidden linear layers added, the number of hidden neurons is set to 0 and is not tuned, thus only 1 model is trained so only 1 row of result is observed. As the vanilla model, the maximum test accuracy of 0.398 is not an optimistic value, which is also why we decided to investigate the effects of adding hidden linear layers to the model on the model performance.

#### BERT multiclass classifier with 1 hidden layer without dropout

This model structure, named *1\_Linear*, involves the addition of 1 hidden linear layer with ReLUactivation function. An illustration of this can also be found in Appendix B. The results for the minimum test loss and maximum test accuracy for this model are as shown below.

A screenshot of a computer

Description automatically generated with low confidence

Due to the addition of the hidden linear layer, 6 different models can be trained by adjusting the number of hidden neurons in the added layer. As such, we obtain 6 different sets of minimum test loss and maximum test accuracy values. According to the results shown, the model trained with 64 neurons in the added hidden linear layer is the best out of the 6 models in terms of both minimum test loss and maximum test accuracy.

#### BERT multiclass classifier with 1 hidden layer with dropout

This model structure, named *1\_Linear\_dropout*, involves the further application of the dropout function with probability 0.1 onto the added hidden linear layer of the model in section 4.1.2.2. An illustration of this model can also be found in Appendix B. The results for the minimum test loss and maximum test accuracy for this model are as shown below.

A screenshot of a computer screen

Description automatically generated with low confidence

In theory, the application of the dropout function should have improved the model performances as compared to the models in section 4.1.2.2 as it helps with reducing model overfitting. However, the results proved otherwise as the minimum test loss and maximum test accuracy worsened for almost all the 6 models except for when 96 and 128 hidden neurons were used. The observation for the failure of the dropout function can be attributed to its probability being set at a value too low for the function to be effective, as well as the probable lack of training data [15]. It is also crucial to note that although the minimum test loss improved for the model trained using 96 hidden neurons, the maximum test accuracy worsened. The opposite is true for the model trained with 32 hidden neurons.

According to the results shown, the model trained with 96 neurons in the added hidden linear layer is the best in terms of minimum test loss, and the model trained with 128 neurons in the added hidden linear layer is the best in terms of maximum test accuracy. We choose the model trained with 96 hidden neurons as the best of these 6 models as it has a lower test loss although the test accuracy is almost the same as that of 128 neurons.

#### BERT multiclass classifier with 2 hidden layers with dropout

This model structure, named *2\_Linear\_dropout*, involves the further addition of another hidden linear layer, also with the dropout function of probability 0.1, onto the model in section 4.1.2.3. An illustration of this model can also be found in Appendix B. The results for the minimum test loss and maximum test accuracy for this model are as shown below.

A screenshot of a computer

Description automatically generated with low confidence

According to the results shown, the addition of another hidden linear layer with dropout worsened the model performances for all the 6 models in terms of minimum test loss as well as maximum test accuracy. The model trained with 256 hidden neurons is an exception, which saw a slight improvement in its maximum test accuracy, but its minimum test loss did not improve. It can be observed that the model trained with 256 hidden neurons is the best of these 6 models as it has both the lowest minimum test loss and the highest maximum test accuracy.

### Summary

In summary, for the design of the BERT multiclass classifier, the optimum batch size to be chosen is 128, and it can then be observed from the table of results below that the best performing model is *no\_Linear*, which uses 0 additional hidden layers and thus the number of hidden neurons is also 0.

Text

Description automatically generated

We also illustrated the test loss and test accuracy results from sections 4.1.2.1 to 4.1.2.4 in the plots below. The plot for the *no\_Linear* model is not observable because it is only a single data point as there is no need to tune the vanilla BERT model in terms of the number of hidden neurons.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

As such, although the *no\_Linear* model is also the vanilla model which has a subpar performance with only a minimum test loss of 1.775218 and a maximum test accuracy of 39.8197%, it is already the best model as compared to those trained from tuning the important model hyperparameters.

## Comparison of model architectures

In this second experiment, we train, compare, and discuss the performance of our optimum BERT model from section 4.1 in comparison to the CNN and RCNN models from the given research papers, which we have implemented in TensorFlow2 with reference to the codes published by the authors. We also included the *BERT\_false* model in our comparison, which is a BERT model with the trainable parameter set to false so that model training will not retune the pre-trained BERT encoder layers but only the hidden linear layers (if any) and the fully connected output layer.

We also used both of the 2 datasets provided to us – CROWDFLOWER and WASSA2017 – to separately train the 4 models, which then gave us more basis for comparison. The performance results from the training all 4 models are as shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | CROWDFLOWER | | WASSA2017 | |
| Max. Test Accuracy | Min. Test Loss | Max. Test Accuracy | Min. Test Loss |
| BERT | 39.8 | 1.78 | 87.0 | 0.569 |
| BERT\_false | 24.3 | 2.13 | 33.0 | 1.35 |
| CNN | 34.8 | 1.91 | 82.4 | 0.520 |
| RCNN | 33.8 | 1.92 | 71.5 | 0.855 |

### BERT multiclass classifier vs. existing techniques of CNN and RCNN

We observe that the performance of our BERT multiclass classifier is superior to both the CNN and RCNN across both datasets in terms of test accuracy, but CNN fares better than BERT in terms of test loss for the WASSA2017 dataset. This implies that although more test inputs of WASSA2017 may have been accurately classified by BERT, they are predicted with less certainty as compared to test inputs of WASSA2017 that are accurately classified by CNN [16]. As such, we still consider our BERT multiclass classifier to be the best model.

Our BERT multiclass classifier’s achievement can be attributed to the state-of-the-art training strategies that the BERT encoder in our classifier employs, such as the use of transformers. Transformers allow the entire sequence of words in an input to be read at once, hence removing the directional and temporal limitations faced by the sequential nature of RNNs and CNNs.

Also, as mentioned in section 3.2.2.2, the BERT encoder uses the MLM training method, which is a complex strategy that causes the computational resources required to train the model to increase significantly. For the CROWDFLOWER dataset, the average time taken to train each epoch of the BERT model using batch size of 128 was about 360 seconds on average, 10 times as long as that for the training of one epoch of the RCNN model which took about 35 seconds on average, and about 20 times as long as that for the training of one epoch of the CNN model which took about 16 seconds on average. Therefore, the trade-off to better model performance by BERT is significant, especially on very large datasets.

### BERT\_false vs. existing techniques of CNN and RCNN

On the other hand, the BERT\_false model performed more poorly than even the CNN and RCNN models, likely because the input dataset are “tweets” obtained from Twitter, which often comprise of sentences tainted with informal languages and intentional misspellings. As a result, the input dataset has a type of language that is very different from that of the dataset that the BERT model was pre-trained on, although both datasets are technically in English. As such, since the BERT\_false model does not retune the pre-trained BERT encoder layers, the BERT encoder was unable to accurately encode the input “tweets” and thus, the classifier layer(s) was unable to accurately predict the emotions of each of the “tweets”.

### CNN vs. RCNN

We also observe that the CNN model performed better than the RCNN model in terms of both test accuracy and test loss. While such results are unexpected due to the bidirectional nature of the RCNN model compared to the unidirectional CNN model, this could be explained by the difference in datasets used for this experiment as compared to the ones used in the research papers. The CROWDFLOWER and WASSA2017 datasets used here are comparably much smaller than the Stanford Sentiment Treebank used by the research papers, thus the RCNN model in this experiment could have overfitted to a greater extent than the CNN model, resulting in lower test accuracies and greater test losses.

### Summary

In summary, our BERT multiclass classifier gives the best prediction accuracies than the existing techniques explored in section 2. Between the CROWDFLOWER and WASSA2017 datasets, we also see that the performances of all the models are better for the WASSA2017 than the CROWDFLOWER dataset. This is as expected since WASSA2017 has fewer target labels, 4, than CROWDFLOWER, 13. Having 13 class labels also results in labels that are semantically very similar, such as *‘hate’* and *‘anger’*, or *‘fun’* and *‘enthusiasm’*, and this increases the degree of uncertainty and inaccuracy when predicting labels.

# Conclusion

Our BERT multiclass classifier has shown to be extremely competitive in performance after training. By being able to capture emotions expressed within both words and sentences, it surpasses slightly more outdated models such as the CNN and RCNN for text emotion recognition, which is as proven in our experiments in section 4.

However, the current model performance is still unsatisfactory in classification tasks involving a larger number of classes as the test accuracy in our experiments for the CROWDFLOWER dataset is still too low at a value that is not even more than 40%. This shows that our classifier has many rooms for improvement, and the ability to handle larger number of classes is one of them.

Our experiments on the CROWDFLOWER dataset in section 4.1.2 have also proved that the addition of hidden linear layers did not further improve the performance of our classifier. This implies that future works may need to look at other ways of modifying the model architecture to improve the model performance.

As mentioned in the previous sections, the datasets studied in this project are extracted from the social media platform Twitter, hence the type of English language involved are likely to be semantically different from the written form of English language from Wikipedia articles and BookCorpus that the BERT encoder of our classifier was pre-trained on. Thus, future works can look at pre-training the BERT encoder using texts extracted from social media platforms before using it in BERT classifiers, in order to investigate the truth to this assertion.

Also, one of the limitations of BERT encoders is that it lacks the ability to handle long text sequence as it only supports up to 512 tokens [17]. Hence, future works can look at designing new variants of the BERT encoder to accommodate a larger limit on the number of input tokens.

To address its limitation of being computationally heavy to train, it is also possible to explore transfer learning methods where the bulk of the model is pre-trained on more general datasets and weights of neurons are frozen, before fine-tuning the last few layers of the model on a more specialised domain. This allows for significantly reduced time required for training since most of the model has already been trained.

Given how large the BERT encoder is with the normal BERT encoder having over 100 million parameters, it also prevents its application on other modern machine learning techniques like federated learning and edge-computing, where model sizes must be kept as small as possible [18]. Therefore, further work can be done to look at how the BERT encoder can be compressed into a smaller size and yet retain its performance in accuracy especially on domain specific texts.

All in all, while our BERT multiclass classifier performs better than the existing techniques of CNNs and RCNNs in the text emotion recognition task, it still has many limitations and weaknesses that can be improved on to perform even better in terms of the test loss and test accuracy as well as the model training time.

References

|  |  |
| --- | --- |
| [1] | P. Zhou, Z. Qi, S. Zheng, J. Xu, H. Bao and B. Xu, “Text Classification Improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling,” Institute of Automation, Chinese Academy of Sciences; Center for Excellence in Brain Science and Intelligence Technology, China, 2016. |
| [2] | Y. Kim, “Convolutional Neural Networks for Sentence Classification,” New York University, New York, 2014. |
| [3] | S. Lai, L. Xu, K. Liu and J. Zhao, “Recurrent Convolutional Neural Networks for Text Classification,” in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, China, 2015. |
| [4] | Manuel, “NNForTextClassification/pzhou\_tf.py at master · ManuelVs/NNForTextClassification,” GitHub, 26 June 2019. [Online]. Available: https://github.com/ManuelVs/NNForTextClassification/blob/master/fen/pzhou\_tf.py. [Accessed 14 November 2021]. |
| [5] | Y. Kim, “CNNs for sentence classification,” GitHub, 2 February 2016. [Online]. Available: https://github.com/yoonkim/CNN\_sentence. [Accessed 14 November 2021]. |
| [6] | D. Britz, “Convolutional Neural Network for Text Classification in Tensorflow,” GitHub, 21 July 2018. [Online]. Available: https://github.com/dennybritz/cnn-text-classification-tf. [Accessed 14 November 2021]. |
| [7] | Y. Takahashi, “LSTM vs BERT — a step-by-step guide for tweet sentiment analysis,” towards data science, 5 November 2020. [Online]. Available: https://towardsdatascience.com/lstm-vs-bert-a-step-by-step-guide-for-tweet-sentiment-analysis-ced697948c47. [Accessed 14 November 2021]. |
| [8] | D. Mwiti, “Guide to Using Pre-trained Word Embeddings in Natural Language Processing,” PaperspaceBlog, June 2021. [Online]. Available: https://blog.paperspace.com/pre-trained-word-embeddings-natural-language-processing/. [Accessed 14 November 2021]. |
| [9] | J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Google AI Language, 2019. |
| [10] | TensorFlow, “bert\_en\_uncased\_preprocess,” TensorFlow Hub, [Online]. Available: https://tfhub.dev/tensorflow/bert\_en\_uncased\_preprocess/3. [Accessed 14 November 2021]. |
| [11] | TensorFlow, “bert\_en\_uncased\_L-12\_H-768\_A-12,” TensorFlow Hub, [Online]. Available: https://tfhub.dev/tensorflow/bert\_en\_uncased\_L-12\_H-768\_A-12/4. [Accessed 14 November 2021]. |
| [12] | M. S. Z. Rizvi, “Demystifying BERT: A Comprehensive Guide to the Groundbreaking NLP Framework,” Analytics Vidhya, 25 September 2019. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/. [Accessed 14 November 2021]. |
| [13] | R. Horev, “BERT Explained: State of the art language model for NLP,” towards data science, 11 November 2018. [Online]. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270. [Accessed 14 November 2021]. |
| [14] | TensorFlow, “Classify text with BERT | Text | TensorFlow,” 12 November 2021. [Online]. Available: https://www.tensorflow.org/text/tutorials/classify\_text\_with\_bert. [Accessed 14 November 2021]. |
| [15] | StackExchange, “Convolutional neural network overfitting. Dropout not helping,” 22 August 2017. [Online]. Available: https://datascience.stackexchange.com/questions/22494/convolutional-neural-network-overfitting-dropout-not-helping. [Accessed 14 November 2021]. |
| [16] | StackExchange, “How is it possible that validation loss is increasing while validation accuracy is increasing as well,” 28 May 2017. [Online]. Available: https://stats.stackexchange.com/questions/282160/how-is-it-possible-that-validation-loss-is-increasing-while-validation-accuracy. [Accessed 14 November 2021]. |
| [17] | E. Ma, “Why does XLNet outperform BERT?,” Medium, 24 September 2019. [Online]. Available: https://medium.com/dataseries/why-does-xlnet-outperform-bert-da98a8503d5b. [Accessed 14 November 2021]. |
| [18] | M. Ravikumar, “Efficient BERT: Finding Your Optimal Model with Multimetric Bayesian Optimization, Part 1,” nVIDIA DEVEL0PER, 18 August 2020. [Online]. Available: https://developer.nvidia.com/blog/efficient-bert-finding-your-optimal-model-with-multimetric-bayesian-optimization-part-1/. [Accessed 14 November 2021]. |

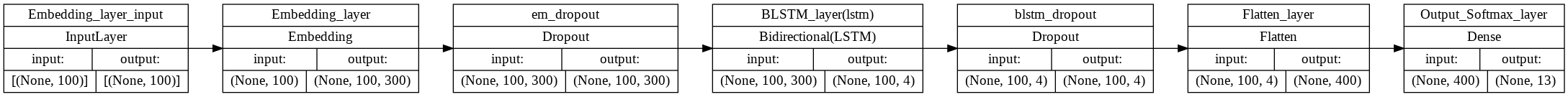
# Appendix A

1. Structure of the CNN model that we have implemented:

Chart, scatter chart

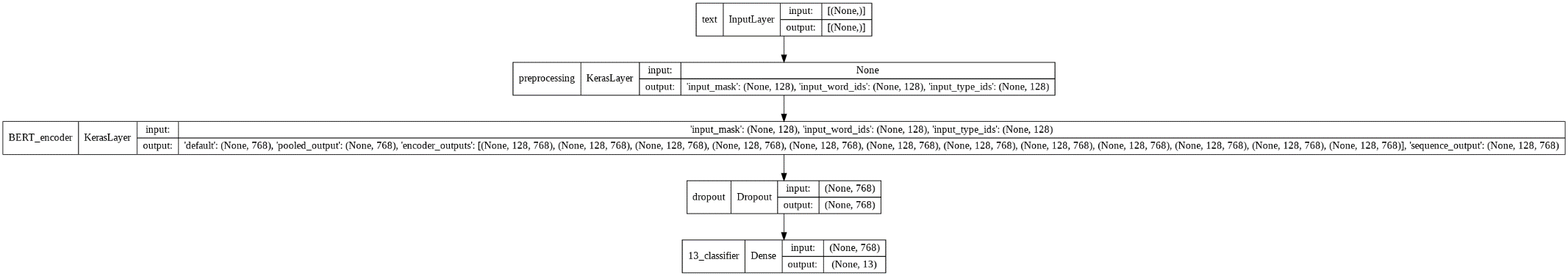
Description automatically generated

1. Structure of the RCNN model that we have implemented:

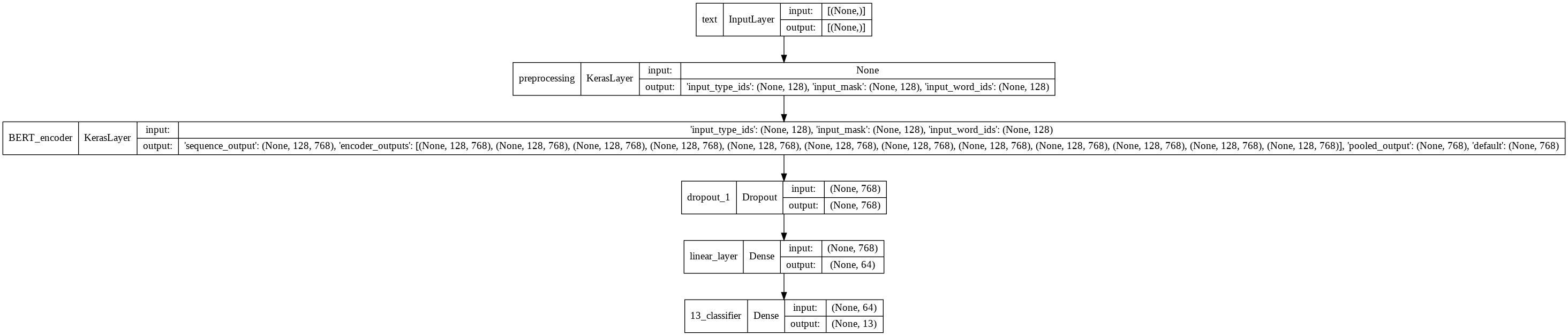


# Appendix B

1. Structure of the *no\_Linear* model that we have implemented:



1. Structure of the *1\_Linear* model that we have implemented, using 64 hidden neurons as example:



1. Structure of the *1\_Linear\_dropout* model that we have implemented, using 96 hidden neurons as example:

Table

Description automatically generated

1. Structure of the *2\_Linear\_dropout* model that we have implemented, using 256 hidden neurons as example:

Graphical user interface, table

Description automatically generated