COMP47650 Deep Learning - Sentiment Analysis - Dataset B2

Edward O'Neill - 17342691

Abstract

This paper and project overall concerns itself exploring the performance of three different models to predict the sentiment of tweets about the Coronavirus of the B2 dataset found on Kaggle.[1] There is a certain historical element on the models that were used in that they are each progressions and improvements on each other. The models which were used were : Recurrent Neural Networks (RNN), Long Short Term Memory Networks (LSTM), and then the transformer BERT purely as a reference to the power of Recurrent Neural Networks and Long Short Term Memory Networks.

1 Introduction

Sentiment Analysis is the act of analysing and extracting information and predicting that which is subjective. The most common thought that comes to mind when people think of it is predicting the sentiment of posts on social media. The purpose of tasks such as this has quite interesting consequences such as seeing the overall sentiment people have towards certain topics such as the topic of this project, the Coronavirus. To carry out the task of classifying the sentiment of tweets about the Coronavirus, I explored two different models, Recurrent Neural Networks and Long Short Term Memory Networks and then used BERT purely as a reference to judge the accuracy of the two models. These models were chosen due to their historical significance within Deep Learning and Natural Language Processing : they are all excellent for the task of Sentiment Analysis. Long Short Term Memory Networks are a natural progression on the Recurrent Neural Network architecture and the difference between the two was a fascinating observation.

Regarding model implementation, Pytorch was used as it provided quite a nice API to use and the documentation was much better than alternatives. It made the development process not as difficult as one would consider it to be; A lot of the heavy lifting such as back propagation was done by Pytorch itself which was quite a nice feature to have as it meant that the complexity was isolated to the models constructor and the forward propagation of the model.

This project will be broken into the following sections : Related Work, Experimental Setup, Results, and then Conclusion and Future Work.

2 Related Work

To familiarise myself with Pytorch and the construction of more complicated models, I looked to the following tutorial for guidance.[2] It is imperative to understand where the models that are used in this paper come from. The first paper I looked at is *Learning Internal Representation by Error Propagation*[7] by Rumelhart, Hinton, and Williams. I would say that overall problem the paper tries to address is the idea of internal representation of time and state within a network. The problem that the paper concerns itself with is that in some cases, problems can only really be solved by a sufficient

[1]https://www.kaggle.com/datatattle/covid-19-nlp-text-classification/download

[2]https://github.com/bentrevett/pytorch-sentiment-analysis

amount of input from the outside world but this has a downside. There is no internal representation. This is a problem as more difficult problems for learning a model require some sort of knowledge of previous inputs and outputs. A model without internal representations cannot make the necessary mappings required to make a correct prediction as the input and outputs would be constrained by their similarities rather than what would be wanted, their differences.

The example that they use to show this is the classic problem of learning the XOR function. They explain the problems associated with learning this function as noted by Minksy and Papert[6]. A network lacking hidden units essentially lacks context of the previous input output states. Hidden units then therefore provide quite powerful functionality to the model as predictions are now also made using the previous input output pairs giving a temporality to the models representation[4] but more importantly, it allows for a more meaningful generalisations of inputs and therefore more succinct models. This behaviour is perfect for the task on sentiment analysis but how does sentiment analysis even work?

Sentiment analysis in short is detecting the sentiment of a piece of text for example work was done in detecting hate speech on Twitter[1]. A string is broken up and tokenized. These tokens are given a score and then the overall score of the aggregated string represents the polarity (positivity, negativity) of the sentence. There are many different approaches to sentiment analysis, to name a few : rule based which analyses based on preconceived rules, automatic which relies on machine learning techniques, and then hybrid which combines the previous two. In *Automatic Sentiment Analysis in On-line Text*[2], two very interesting techniques in the overall methodology of Sentiment Analysis are given: Web Search and WordNet.

In the former, tuples are created based on words which could potentially alter the polarity of a piece of text. For example the word *brutal* would indicate a negative review if it was in a restaurant review (service was brutal) but could possibly indicate a positive review if the subject was say an action movie (the action was brutal, I loved it!). WordNet is a bit more interesting. Here, words are considered to be nodes on a graph and these nodes are connected by edges which are represented by their synonyms. This is overall meaningless if it wasn't for the creators of WordNet, Marx and Kamps considering the dimension of the appraisal. This means that sentiment of a word is calculated by comparing the distance of the word from the positive and negative ends of the dimension wherein lie prototype words using a metric called minimum path length (MPL). In different dimensions there exists different prototype words for example if the task was analysing an adverb, it would be placed on the potency dimension as that would be able to tell you whether or not it weakly or strongly (weak and strong being the prototypes) affects the sentences sentiment. For example if it was analysing the string "The movie was really bad", the word "really" would have a large affect on the overall sentiment of the string.

One of the biggest advantages that LSTMs have over RNNs is that they don't have to be concerned with the vanishing gradient problem. This is a problem because activation functions such as sigmoid and RELU can cause the gradients of the network to approach zero and thus "disappear" or worse, they can explode in size and grow towards infinity which ruins the training and testing results. This is done with creating a constant error which flows through the model such that the gradients neither vanish nor explode.[5] and with the use of cell states. These cell states allow the LSTM to make connections between past inputs and future in a much more meaningful way by utilising gates (forget gate, input gate, output gate). These gates allow the LSTM to keep dependencies that would otherwise disappear in normal RNN networks allowing the LSTM to be that much accurate which is helpful in the task of Sentiment Analysis wherein long term dependencies are key to the task.

Regarding the reference model BERT, it's seminal paper[3] details its goals quite well. To a degree, part of what it sets out to achieve seems to be to streamline Natural Language Processing. The BERT model itself is just a pretrained bidirectional model which can be fine tuned to the output layer such that it can train models for tasks like Sentiment Analysis. BERT belongs to a family of models called Transformers. These are improvements to the recurrent models as they aren't bound by padding and sequencing lengths. In the seminal paper introducing this architecture[8], they state that this sequential computation problem was a goal for them to solve. They use attention mechanisms which can calculate any dependency without regard of their distance from each other in the input and output sequences in conjunction with recurrent networks to construct a model which can completely bypass the problem of sequential computation.

3 Experimental Setup

3.1 Preprocessing

Regarding preprocessing, what I wanted to achieve was to be as minimal as possible and abstract a lot of the data cleaning and torchtext functionality because it made everything that much clearer and cleaner.

The first task was to just observe the data of the csv files. It was necessary to encode them such that the utf-8 within the files could be decoded. The first thing I considered before even cleaning the data was the task itself. There was a lot of data to consider such as the user name of those that wrote the tweet or even the location. One could make the argument that the location of where a tweet was sent for example would have an impact on the sentiment of the tweet. If a tweet came from New York during the first wave of Coronavirus last year, it is probably more likely to be negative than if it came from Ireland just based on how heavily each location was hit with the virus. I felt that including the location of the tweet was an incorrect path to take however due to obfuscating the act of sentiment analysis i.e we're not looking to classify which locations made predominately negative tweets for example so all of the columns bar the tweets and their sentiments were removed.

The next task was ontological : What's the difference between a positive tweet and an extremely positive tweet? I would argue that for sentiment analysis, there is no difference and to suggest that there is just creates an unneeded difficulty for the model to classify whether a string such as *"Hospital workers are doing fantastic work"* is positive or extremely positive. Although I'm sure the model could create good classifications, reducing the overall complexity of a task is a key factor to a models success therefore I grouped the sentiments together to just be left with : Negative, Neutral, Positive. These were then encoded to the values of -1, 0, 1 to make working with them easier.

To clean each dataset I opted into a more functional approach which seems to have better performance that the original iterative method. The issue with much of the data in the tweets is that many words are just not important to predicting sentiment. For example a tweet containing a link would have the same sentiment with or without the link or if there were commas or periods in it so if these were removed then the sentiment would be unchanged but the tweet would be cleaner. This can be done very easily with regex. What if after removing these kinds of words/symbols, a lot of unnecessary words still remained like 'and', 'a', 'is', and 'that'? These words don't represent anything really and can't be removed with regex and have no affect on the sentiment of a sentence. These words are called stop words and can be found in the nltk package. Functionally, their removal can be achieved with *filter*.

Exploiting Pandas' *apply* function then allows for a more intuitive and cleaner preprocessing method. Each tweet is first cleaned with regex and then the stop words are filtered out of the entire tweet.

Listing 1: Preprocessing Dataset

```
df.Tweet = df.Tweet.apply(lambda tweet : ''.join(
                               (list(map(lambda word : clean_tweet(word),
                               filter_nonstopwords_nonpunctuation(tweet.split())))))
```

3.2 Preparing Datasets

3.2.1 Validation Dataset

Since a validation dataset wasn't provided, it needed to be constructed from the train dataset. The purpose of the validation dataset is to facilitate hyper-parameter tuning without the need of touching the test data. This is a necessary construct to have have when training a dataset. If a validation set is used, one can see how their current hyper-parameters affect the loss and accuracy of the model when it's applied to unseen data. This means one can avoid touching the test data until one can be sure that their hyper-parameters are optimal for the model. Touching the test data could have disastrous effects on the performance of the model such as over-fitting and the model learning the parameters of the test data giving the programmer an incorrect high accuracy. The validation set was created with an 80:20 split with the training dataset. The datasets were then shuffled with the sample function with frac=1 passed in to return all the rows.

3.2.2 Torchtext

Pytorchs *torchtext* library is a very interesting library. It is part of a set of libraries (torchvision, torchaudio) that perform classifications on different types of data. A key idea within torchtext is a construct called a field. A field defines the data and gives instructions for how the data should be converted to a Tensor. Since there are only two columns, Tweet and Sentiment, creating two fields of the same name therefore allows the data to be processed into torchtexts Tabular datasets.

Since the data concerns itself with text only, it still needs to be processed by torchtext even though it was processed already. *Spacy* is a library which is "for advanced Natural Language Processing in Python and Cython."[3] and so it allows for a quite efficient way to tokenize text by just passing the parameter *spacy* in the fields instantiation.

Listing 2: Field Creation

```
Tweet = data.Field(sequential = True, tokenize='spacy',
                   tokenizer_language = 'en_core_web_sm',
                   use_vocab = True, include_lengths = True)
Sentiment = data.Field(sequential = False,
                       use_vocab =  False)
```

In the case of BERT, the preparation here was slightly different. I will touch upon this difference when I come to section on BERT.

Once the fields are defined and the Tabular datasets are created, the vocabulary for the fields can be created and the maximum size in which they can be.

Listing 3: Field Vocabulary Building

```
Tweet.build_vocab(train, max_size = VOCAB_SIZE)
Sentiment.build_vocab(train)
```

3.3 Loss, Regularisation and Optimising

The loss function I opted into using was cross entropy loss. It's a great function for classification tasks whose outputs dimensions aren't binary. The Regularisation technique that I employed was L_2 Regularisation rather than the more popular Dropout. I chose to use this Regularisation technique due to wanting to explore the differences of Regularisation in models and from being more familiar with other techniques such as Dropout already. L_2 Regularisation is also much more easier to implement than other techniques by passing a value (1e-3 for example) in the weight decay parameter in the optimiser which is less obtrusive than having a dropout object instantiated in the constructor of the model. After tuning the model to different optimisers such as Ada and AdamW, I ended up using Adam as that gave the best results by far.

4 Models

4.1 Model One - Recurrent Neural Networks

The purpose of using a RNN was that this type of network is quite efficient at analysing sequences which is the perfect model to use in the task of analysing a sequence to predict sentiment. These networks are very interesting because the idea of temporal representation comes into play with them.

As stated before, the idea of Recurrent Neural Networks first appeared in Rumelharts *Learning representations by back-propagating errors*[7]. Essentially what they permit is to use the previous states input and its hidden state as context for the current prediction. For example consider the following tweet : "Quarantine is brutal"

The network takes the first word and produces an output for it, this output is then fed to what takes the second word and aggregates the first words output with the seconds words outputs and feeds this to the third word. Once the third output takes this aggregate, produces a final output, the hidden state is fed into a linear layer which produces the predicted sentiment.

[3]https://pypi.org/project/spacy/

This behaviour isn't so difficult to consider regarding implementation. Passing the text to an embedding layer embeds the input such that it reduces its dimensionality. It's more difficult to create accurate models if the dimensions of the data are too large therefore a reduction in it is helpful regardless of classification task. After the tweet is embedded, passing it to a RNN produces the output and hidden state which can be fed-forward to the linear layer.

4.2 Model Two - Long Short Term Memory & Bidirectionality

LSTMs are a lot more interesting than Recurrent Neural Networks. They are a natural progression on the RNN architecture and are a lot more powerful than them regarding sentiment analysis. The LSTM that was used was a bidirectional LSTM however it is possible to use a bidirectional RNN. The LSTM architecture itself is more interesting as it avoids the problem of vanishing and exploding gradients that RNNs have with the introduction of cells and gates mentioned before. Bidirectionality however is extremely powerful. In the previous case the model looks to the previous output for context as to what to produce next but if the model is bidirectional, then we look to the previous case but there's also another output hidden state product being produced from the future back towards the current input. Within the Pytorch implementation, the output and hidden states that are produced are stacked upon each other purely for the sake of making everyone's life easier because the final output hidden state pair can be concatenated together and sent to the linear layer for prediction.

4.3 Model Three - BERT

BERT, short for Bidirectional Encoder Representations from Transformers is a pre-trained Transformer from Google that revolutionised problems such as Natural Language Processing and the ilk. The main difference between a Transformer and a Recurrent Neural Network is that data doesn't have to be processed in order. A problem which existed within the creation of batches and their iterators in the previous models was that the tensors had to be processed in order of length. This was a serious problem in the earlier stages of the project because the model expected to process the tensors in this way and since errors are at run-time, it proved to be difficult to mediate the bug.

Since transformers aren't limited like previous models, it can then be trained in a much more meaningful way. BERT in particular is already a pre-trained model that can be altered to ones specific needs to give great performance.

As I mentioned in the beginning, the fields had to be changed when BERT was used as the model due to how it works. It was trained originally such that each sequence of words started with an initialise token (init token) and ended with an end of sequence token (eos token). Each sentence as well had a pad token that would be ignored by the model but would pad each sequence to the same length. Each of these tokens had to have been passed to the Tweet field function along with a preprocessing directive to convert each word to an integer representation.

Much of the model is quite like the previous implementations except for the inclusion of the BERT parameter. Before the model is constructed, the BERT pre-trained model (bert) is instantiated. Then in the model itself, the only thing that is changed is how the data is embedded. In this case, the data is just fed to bert. The gradients of transformers are set to zero however before the embedding purely due to the possibility of gradients being calculated during the training phase which would alter results drastically.

5 Results

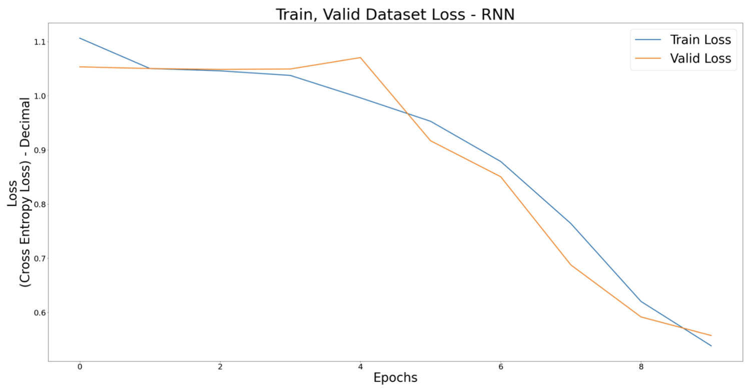
A few experiments were carried out to see the effects that the hyper-parameters have on the efficiency of the networks. I found that the hyper parameters which gave the most meaningful change in loss and accuracy was the vocabulary size and the hidden dimension. The other parameters such as the number of layers, batch size, learning rate, weight decay (L_2), and the embedding dimension were kept at a constant of 2, 64, 1e-3, 1e-3, and 128 respectively. In the case of BERT the batch size was reduced to thirty two and the hidden dimension was 64 because I found the training was quite slow. The graphs in the RNN experiments and the LSTM experiments are those that have the best hyper-parameters in their respective experiments.

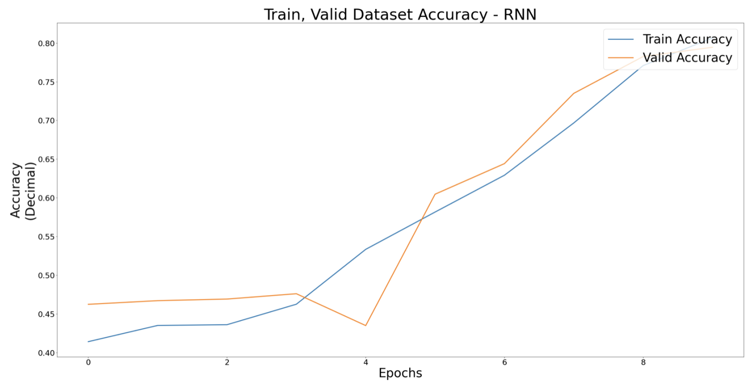
5.1 Recurrent Neural Networks

The results of the Recurrent Neural Networks were surprising.

Experiment	Vocabulary Size	Hidden Dimension	Test Loss	Test Accuracy
1	5000	64	92.433%	51.340%
2	5000	128	92.704%	48.059%
3	10000	64	70.203%	71.892%
4	10000	128	102.376%	40.559%
5	25000	64	102.280%	41.446%
6	25000	128	129.349%	24.893%
7	50000	64	62.875%	75.874%
8	50000	128	138.629%	16.219%

For example a vocabulary size like five thousand and a hidden dimension of sixty four gave unfavourable results of a test loss of 92.433% and a test accuracy of 51.340% . Raising the vocabulary size to ten thousand and the hidden dimension to one hundred and twenty eight only serves to increase the test loss. The best parameters for the Recurrent Neural Network ended up being fifty thousand for the vocabulary size with a hidden dimension of sixty four. This gave a test loss of 62.875% and an accuracy of 75.874% .



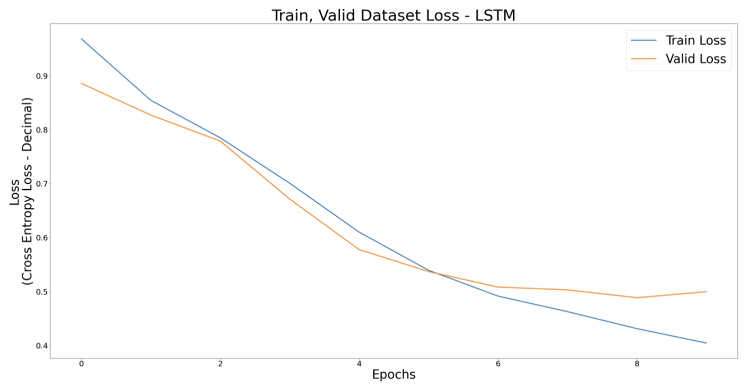


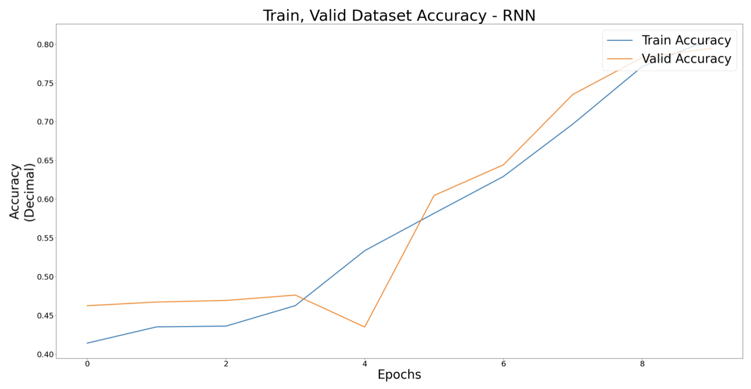
I think in part that the strangeness of results could be down to the RNN architecture. The RNN (unlike the LSTM in the next section) wasn't bidirectional so it could only make predictions based on previous input and contexts/predictions. If the LSTM can use the previous context of a word and the future context of a word due to the bi-directionality, then it would make sense that the RNN wouldn't be able to predict that well because it is limited. Furthermore the RNN is susceptible to vanishing or exploding gradients which would have played a part in the training.

5.2 Long Short Term Memory Networks

Similar tests to Recurrent Neural Networks were carried out again for this kind of network. Unlike in the previous case, the results here were a lot more consistent. As expected however the Long Short Term Memory network gave much better results.

Experiment	Vocabulary Size	Hidden Dimension	Test Loss	Test Accuracy
1	5000	64	54.605%	79.517%
2	5000	128	96.952%	49.290%
3	10000	64	70.801%	72.77%
4	10000	128	53.566%	80.036%
5	25000	64	55.427%	79.801%
6	25000	128	53.145%	79.931%
7	50000	64	56.181%	78.707%
8	50000	128	55.917%	79.020%

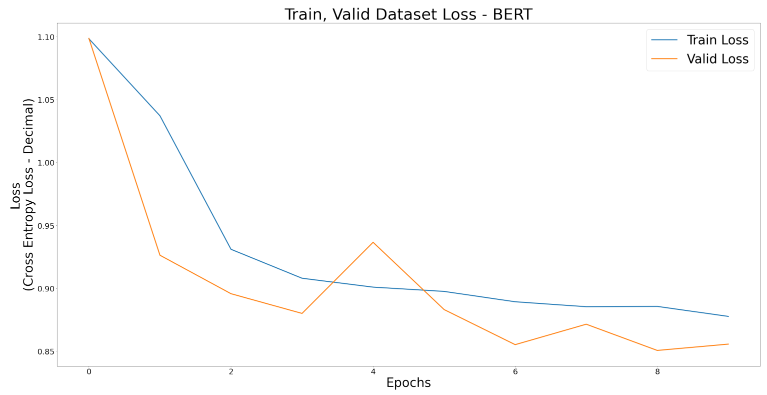


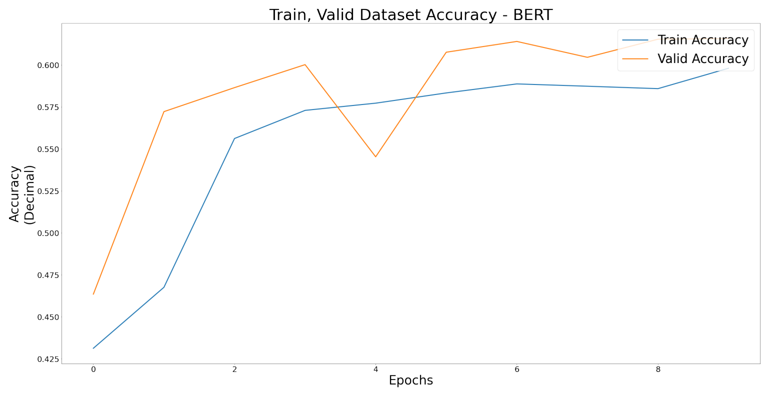


Tuning the parameters to a max vocabulary size of twenty-five thousand and the hidden dimension to one hundred and twenty eight gave the best results for the LSTM at a test loss and accuracy of 53.145% and 79.931% respectively. The LSTM far outperformed the RNN. I suspect that the reason for this is due to amount of data available. The data itself is quite small and from my experiments, it seems the RNN isn't too adept at performing Sentiment Analysis. The LSTM on the other hand far outperforms it which has to be due to its architecture. If sentiment is classified based on previous context and context from the future, then it makes more sense that the LSTM would be able to perform better on a smaller dataset because it has the extra power of bidirectionality. If the dataset were larger however, I suspect that the discrepancy between the two architectures would be even more apparent.

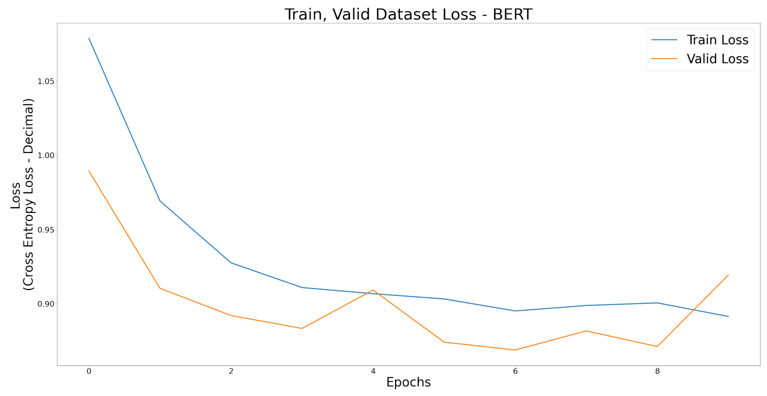
5.3 Reference Model : BERT

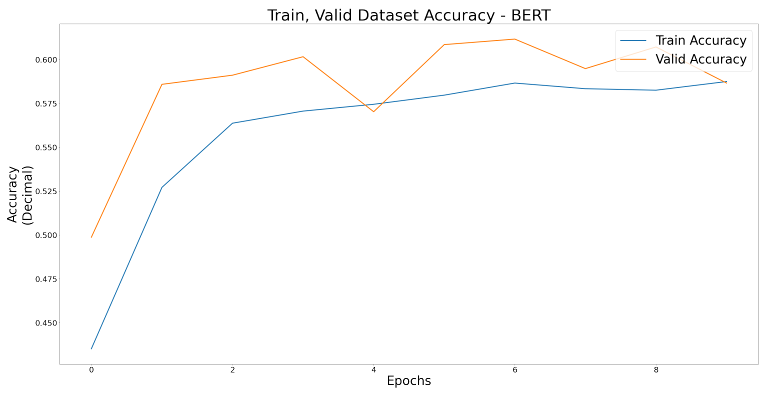
Using the state-of-the-art model BERT as a reference was surprising. It was extremely slow running on my local machine so I opted into using Google Colab where performance was much better and to lowering the batch size to reduce the models complexity. It was surprising to see that BERT performed worse comparatively than the other two models though this is probably down to the size of BERT despite the only parameter being needed to change was the hidden dimension. Take the case in which the hidden dimension was sixty-four. This gave the best results. The test loss was 85.63% and the accuracy was 61.08% .





In the case of where the hidden dimension was one hundred and twenty eight, the test was 89.99% and the test accuracy was 58.32% .





The learning is only just marginally better. It is a very complex model therefore it stands to reason, in my opinion that its complexity proved a disservice to this dataset. I suspect that if the dataset was much larger again then the performance regarding speed and accuracy would be better as it is probably too large to use for such a small dataset. However I can recognise its power. There's a lot less to consider when constructing BERT because nearly everything is taken care of for you whereas in the other models there exists a bit more due diligence that has to be carried out. Overall however it was interesting to see how a state-of-the-art model performs.

6 Conclusion and Future Work

In conclusion, the results of this project have been eye opening. I was surprised to see such a drastic difference between the Recurrent Neural Network and the Long Short Term Memory Network. I can't say for definite if the cause for the difference was down to the challenges of the dataset (sequential data, small dataset) but as stated before, the difference probably stems from the architectural differences. If the dataset was larger I suspect that the RNN would become a bit more accurate as it seems that the size of the dataset did play a significant part in the accuracy of the RNN. A larger dataset would probably help the LSTM too but overall I'm happy with the results of both models.

Regarding future work for this project including constructs that tied to the Functional Programming paradigm could an interesting avenue to explore than just the main ideas like map, filter, and lambda. There are many ideas in the paradigm that could lend itself nicely to both the performance of models and their efficiency. Immutability for example is such a small feature that lends itself nicely to the idea of parallelism. If data is immutable then we can paralleise the program because the concern of state is no longer present. A prevalent idea within Haskell is lazy evaluation/call-by-need. In a resource intensive process like training, it would be an interesting feature to implement because what it does is essentially only calculate something when it's needed. In Haskell a great example is an infinite list which can be represented in the following way : [1..]. In Python, iterators are lazy but I'm not sure if this extended to Pytorches iterators which would benefit heavily from the idea if they weren't.

I think given the chance however I would like to include monads in future work to these models. What we can exploit from them is how they allow for a more elegant way to compose functions. This would be a great feature to have within Deep Learning because all we're doing really is calling functions. A banal way to describe the field (and I suppose Computer Science overall) but what I mean is that if all we're doing is calling functions to update gradients and other parameters repeatedly then being able to call these functions in such a way that errors could be anticipated in a more elegant way would aid to the speed of the model. One such monad that can be used is the Maybe monad. This monad allows for error exceptions to be done in a really fascinating way. Instead of causing a runtime error if there was a division by zero or a difference in tensor shapes for example, it would instead return 'Nothing'. This is a representation of no value existing and if we had this, we could just remove it or employ backtracking and go back to before 'Nothing' was calculated and skip thus effectively abstracting a lot of problems away.[4][5][6]

7 References

[1] P. Badjatiya, S. Gupta, M. Gupta, and V. Varma. Deep learning for hate speech detection in tweets. In *Proceedings of the 26th international conference on World Wide Web companion*, pages 759–760, 2017.

[2] E. Boiy, P. Hens, K. Deschacht, and M.-F. Moens. Automatic sentiment analysis in on-line text. In *ELPUB*, pages 349–360, 2007.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://www.aclweb.org/anthology/N19-1423`.

[4] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] M. Minsky and S. Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 1969.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

[4]https://www.infoq.com/articles/fn.py-functional-programming-python

[5]https://senkorasic.com/articles/maybe-monad-in-python/

[6]https://wiki.haskell.org/Maybe