**Natural Language Processing with Disaster Tweets**

Matteus DiPietro, Chirag Mattel and Omar Sabbagh

Department of Computer Science, University of Windsor

COMP: 4740: Advanced Topics: Artificial Intelligence II

Dr. Robin Gras

April 26, 2022

**Natural Language Processing with Disaster Tweets (Abstract/Introduction)**

Natural language processing is a continually growing and important topic in machine learning. It is being used in every aspect of our lives, whether it be natural language processing enhanced autocomplete on our phones and computers, voice assistants to answer our questions or even grammar checkers when working on reports. As natural language processing becomes ever more present in our lives, it is becoming increasing important to understand how these models work, and how to implement them. The Kaggle competition, Natural Language Processing with Disaster Tweets is a great start on the journey of understanding natural language processing. It provides a relatively simple challenge that allows for many different model architectures to be tested and explored. The dataset consists of tweets that contain keywords that relate to disasters. These disasters range from natural ones, such as fires and tornadoes, to human caused one like crashes or shootings. The problem that comes with this task is that many keywords that are used to describe disasters are often used in a variety of other situations that a tweet might contain. Take this tweet for example, “My phone looks like it was in a car ship airplane accident”. While the tweet contains the keyword ‘accident’ in the context of the tweet it does not refer to and actual accident but is instead being used to describe the state of the tweeter’s phone. Therefore, the goal of the model is to distinguish real disaster quotes from fake ones.

**Research Questions**

1. Which model between a simple model, an LSTM model or a BERT model performs the best when trying to differentiate between natural disaster tweets.
2. How precise can we tune the hyperparameters of the three models to achieve the highest accuracy possible? How much are each model affected by the tuning of these parameters (significant, insignificant, no change)?
3. How do these models compare to others already existing models? Do they perform better? If so, is it caused by the difference of the models or preprocessing that could be applied to our model to improve performance.

**Implementation**

**Simple Model**

The first model that was created was a simple model that only utilise an Embedding layer, a GlobalAveragePooling layer and Dense layers. This model also uses the minimum amount of preprocessing required. Before the tweets can be inputted into the model, they must first be tokenized. Tokenization converts the text into vectors, each word is assigned an index and each text is represented as a collection of these tokenized words. By default, all punctuation is removed so that the text is only space separated. After the text is tokenized, all the sequences are padded so that each entry is the same size and can be passed into the model. The data can then be passed into the model where it trains on the training set and produces a final accuracy from the testing set.

**LSTM**

LSTM - Long Short-Term Memory is a type of Recurrent Neural Network model that specifically designed to solve the vanishing-gradient /long-term-dependency problem by including an additional 4 neurons at each neuron in an RNN. The 1st gate being Forget Gate -decides how much information is discarded - constructed using a sigmoid activation represents a number between 0 - keep nothing and 1 – keep everything. The 2nd gate being Input Gate – decides how much current and previous information is kept. It takes previous hidden states output and passes them to a sigmoid activation to decide what values to be updated from previous hidden input; 0 represents not important 1 represents important. Right after this gate current cell inputs & hidden state outputs are sent through a tahn activation producing values between -1 and 1. The two vectors produced from the sigmoid and tahn gates just mentioned above are then then cross multiplied as this exercise helps regulate the network. The 3rd gate being Cell State Gate - at this stage the network has sufficient information from the input gate and forget gate which then is used to produce the new cell state. The 4th and final gate is the Output Gate – This gate decides what the next \*hidden state should be by multiplying the new cell state with the product of the tahn vector produced from gate 2 to act as a filter. \*Hidden states contain information from previous inputs and acts as the neural networks' memory.

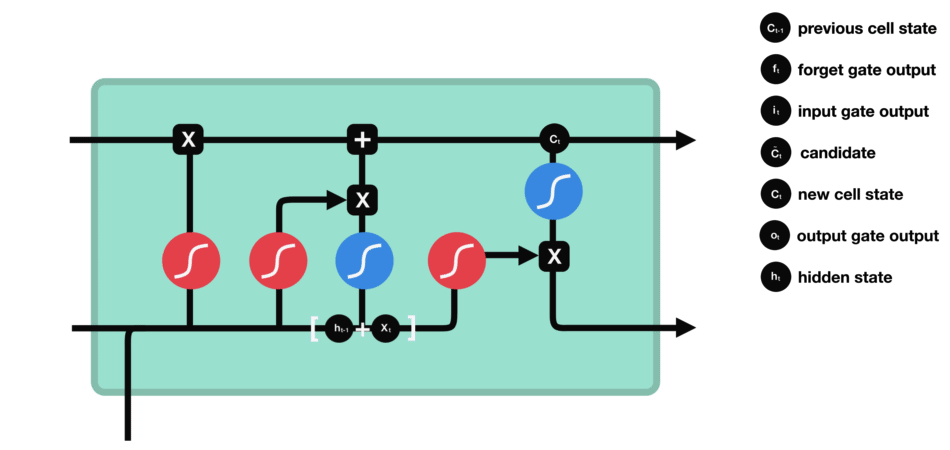


Figure 1 LSTM Model

**BERT**

The BERT model is one of the most used models and represented a leap in natural language processing. BERT stands for Bidirectional Encoder Representations from Transformers. BERT was built on the Transformer architecture, but instead of the traditional architecture, BERT stacks multiple encoders depending on the needed use. The output can then be embedded into a task specific model and fine-tuned to increase performance. To train BERT, two modeling methods were used, masked language model and next sentence prediction. In masked language model, an input is given where a certain amount of the input has been masked, or hidden. It is the job of the model to guess the masked input is and return the completed input. Doing this allows for BERT to create relationships between words by learning which words are commonly seen together. For relationships between sentences to be learned next sentence prediction is used. This training gives BERT two sentences, and it must decide the order that the sentence should be in. All this training is done using a large and diverse dataset allowing BERT to be very effective in a variety of circumstances. However, this flexibility comes with several downsides, some which were run into during the testing of this model, such as the size and training time of the model.

**Experimentation**

Once all three models were implemented using hyperparameters that were known to provide good results, it was time to see whether the performance of the models could be improved by tuning these hyperparameters. The main hyperparameters that were tested are optimizers, number of layers, activation functions, learning rates, numbers of epochs, batch size, dropout rate and batch normalization. In some cases, a certain hyperparameter was not able to be tested, this occurred frequently in the BERT model as large pretrained models are used.

**Base**

***Optimizers***

The first hyperparameter that was tested was the optimizer. The optimizers that were tested were Adam, SGD, Adamax, Adadelta, Adagrad, Nadan and RMSProp. Adam was the initial optimizer that was used in the model and achieved an accuracy of 76.34%. Of the other optimizers, SGD, Adadelta and Adagrad performed significantly worse than the other optimizers, with all achieving accuracies of 57%. Meanwhile Adamax performed only slightly less at 74.28%, but Nadam and RMSProp both performed marginally better at 77.16% and 77.93% respectively. Therefore, RMSProp will be used when determining the final accuracy of the model.

***Number of Layers***

The initial model contained two dense layers, one hidden and the other being the output layer. Additional layers were added to see how they affected the accuracy of the model, with up to six additional layers being added. While the initial model received an accuracy of 76%, adding more layers did not increase the accuracy, in fact the opposite occurred. The more layers that were added to the model the worse it performed. This could be due to overfitting caused by the additional layers along with the configuration of the model. For the final testing, the original two dense layers would be used.

***Activation Functions***

The activation functions that were tested were relu, tanh, softmax, sigmoid and selu. The initial model used relu for the hidden dense layer and sigmoid for the output layer. For testing, only the activation function of the hidden dense layer was changed. Relu achieved an accuracy of 76.18%. Tanh performed the worst out of all the activation functions, only achieving an accuracy of 73.98%. Softmax and sigmoid both outperformed relu with accuracies of 78.08% and 77.59% respectively. Selu was in the middle of the group with an accuracy of 75.05%. Out of these results, softmax performed the best with 78.08% and would be used in the final testing of the model.

***Learning Rates***

Along with testing optimizers, the learning rate of the optimizers were also tested. Learning rates were tested from 0.0005 to 0.05, where 0.001 is the default learning rate for the optimizers in the Keras library. From testing, lowering the learning rate below the default of 0.001 resulted in lower accuracies by 2-5%. This was also the case for decreasing the rate, resulting in drop of 10% in accuracy. A learning rate of 0.0005 resulted in an accuracy of 77.56% and would be used in the final testing of the model.

***Number of Epochs***

Eight different number of epochs were tested ranging from 1 to 60, with 30 epochs as the initial number of epochs. There was a slight decline in performance increasing above 30 epochs, about a 3% decrease. One epoch resulted in a very low accuracy of 57.03%, which is to be expected as the model as little time to learn. Changing to 15 epochs allowed for the model to effectively learn patterns in the dataset without overfitting and achieved an accuracy of 78.05%. Therefore, 15 epochs will be used in the final testing.

***Batch Size***

Batch sizes ranging from 2 to 64 testing with all powers of two being tested to see how they compare. A batch size of 32 was used in the initial model. Reducing the batch size from 32 resulted in a decrease of 2% among the batch sizes 2,4,8 and 16. Increasing the batch sizes gave better results with a batch size of 64 giving the best results with an accuracy of 77.1% and would be used in the final testing of the model. Interestingly, batch sizes of 40 and 50 gave no real improvements over a batch size of 32, with the only real improvement coming from a batch size of 64.

***Dropout Rate***

Adding dropout layers tends to increase accuracy, so a dropout layer is added between the two dense layers and the dropout rate is adjusted between 0.1 and 0.9 at 0.1 increments. Dropout rates of 0.1, 0.5 and 0.9 performed the best out of the tested rates, with the rates in between often performing worse than the model without a dropout layer. A rate of 0.9 achieved an accuracy of 77.47% and will be used in the final testing.

***Batch Normalization***

As a final parameter, batch normalization was added between the two dense layers to see how it would affect the accuracy of the model. Using batch normalisation, the accuracy of the model dropped to 70.56%. So, no batch normalisation would be used in the final model.

**LSTM**

***Hyper parameter automated testing***

The hyper parameters we chose to play with for our model are epoch, units, dropout, as well batch size. We will also construct a way to automatically test these hyper parameters by testing every possible iteration of parameters given the following lists that hold the values for each parameter.  
We will test each parameter setting on how it affects results, if increasing yields positive/negative feedback or if some settings have diminishing returns while others can sway the model significantly. In order to aid the study of hyper parameter optimization we will print the loss and accuracy for each model to help us understand visually the relationship of each hyperparameter. We constructed a final Excel document named RESULTS which houses the statistics of every varied model as well in column K contains a hyperlink that can be used to open the respected accuracy and loss curve. In total we tested 584 different hyper parameters,

***GPU Acceleration***  
 In order to train multiple models to verify the optimal hyper parameters we need low training times. To achieve this, we will utilize the GPU hardware we have on hand. For our project and experiment will thoroughly setup TensorFlow to utilize the CUDA architecture available to us. Using GPU accelerated DNN that runs locally will be a milestone for the develop.   
***Epoch***

We tested from a range of 1 – 100 epochs and found that at 10 epochs the model seems to plateau in returns, we recommend going forward to always utilized 10 epochs for all models.

***Dropout***

We had tested our drop out from a range of 0.1 to 0.55 in intervals of 0.5. We found dropouts to be optimal from ranges 0.1-0.4, anything above 0.4 dropout tended to be low performing models.

***Batch Size***

We had tested our batch size from ranges of [2,4,8,16,32,64,128,256], we found that batch sizes between 64 and 96 preformed the best, while excessively high batch sizes performed the worst due to over training. Infact, we noticed that at unreasonable batch sizes such as 4096 the model performs at the absolute worst, that being less than 60% accurate.

Table

Description automatically generated

**BERT**

***Activation Functions***

Four different activation functions were tested, elu, selu, gelu and relu. Selu and gelu underperformed with accuracies of 54.1% and 54.4%. Elu and relu performed very similarly with accuracies of 55.71% and 55.66%.

***Learning Rates***

Learning rates of 0.0001, 0.001, 0.01 and 0.1 were tested. Learning rates of 0.1 and 0.0001 were the worst performing with accuracies of 51.19% and 52.16% respectively. While a learning rate of 0.001 performed the best with 55.66%

**Final Model**

**Base**

After changing the parameters to the ones found through testing, the new model was tested, but performed significantly worse than the original model with an accuracy of 57.03%. Using the other results of the testing as well as new knowledge gained throughout the length of testing, the accuracy of the model was able to achieve an accuracy of 78.21% using RMSprop with a learning rate of 0.001, two dense layers, a dropout rate of 0.5, 30 epochs and a batch size of 32.

**LSTM**

After testing a multitude of hyperparameters the model seemed to still perform very well regardless of the parameters. The lower performing models were as low as 74% accurate which is still incredibly high. Of course, we had a few outlier results such as 56% accuracy, but we were intentionally trying to set unreasonable hyper parameters such as 4096 batch size. We found that our top performing models capped at 80% accuracy which isn’t farm from the lowest performing models only 6% more accurately after hyper parameter optimization. This may incur us to look further into other parameters that we may have missed in our model to see if they lead to significant changes in our results. Overall, we can conclude that hyper parameters weren’t as sensitive to changes as though it would. This means that this model is very accurate under multitude of circumstances. Our loss ranged from 0.46 to 0.53, and accuracy ranged from 74% to 80%. Meaning the model will be consistent with other setups. Our most accurate model had 80% accuracy and 0.46 loss, while our worst (NON out Lier) had 74% accuracy and .53 loss. Our training times were incredibly low averaging less than 10 seconds per run.

**BERT**

Due to the size of BERT, it was difficult to perform tests on it with the available hardware. These hardware limitations resulted in limitations on the model that prevented it from performing as it should have and thus had a very lack luster performance, only achieving an accuracy of 55.71%, being outperformed by the base model by over 20%.

**Comparisons**

**Simple BERT Model**

As we were unable to effectively implement the BERT model, we looked for someone who was to see how it compared to our models. For this we found a model made by Kaggle user Mitra Mirshafiee who implemented a simple BERT model that achieved an accuracy of 77%. Mitra achieved this using a single BERT layer and a Dense layer for the final output of the model.

Along with the BERT layer, Mitra also helped improve the performance of the model by preprocessing the text input. To this effect, many text cleaning techniques were used. These usually meant the removal of unnecessary information that is not helpful to the model for understanding the tweet. These include removing web links, stop words, punctuation and extra spaces. Lemmatization is also applied to the input text so that the meaning of the text can be better understood in fewer words.

Overall, our best model achieved an accuracy of 80% with the LSTM model, which is better than this BERT model achieved even with its text preprocessing.

**ELECTRA Model**

Another model that was found that used a different base model with great success was a model implemented by Kaggle user Ozcangundes who created an ELECTRA model. ELECTRA stands for Efficiently Learning an Encoder that Classifies Token Replacements. It is trained in a similar manner to masked language models, but instead of masking a certain amount of the input, a certain amount is replaced. This replacing instead of masking still means that ELECTRA can learn from all inputs, but it allows it to train more efficiently and therefore requires less resources to train.

Along with the ELECTRA model, Ozcangundes also applied preprocessing to the input data, similar to what Mitra Mirshafiee used in their model. As a result, the model achieved a perfect score of 1.00 in the Kaggle competition, which far exceeds our best accuracy of 80%.

**Conclusion**

**Research Questions**

Out of the three models that we tested, the LSTM model performed the best, slightly beating out the base model and outperformed the BERT model. While it was expected that the LSTM model would outperform the base model, the BERT model was not able to be fully represented due to hardware limitations, so the BERT model could beat out the LSTM model if adequate resources were available.

For each model, tuning the hyperparameters had a small, but noticeable difference. Each model also responded differently to the hyperparameter changes. The base model only increased by 2% after testing, while the LSTM varied by 6%. These changes could also be due to better starting settings in the models.

Overall, our best model performed respectively when compared to other models even if it did not employ the same preprocessing as the other models. In the BERT comparison, the LSTM model again outperformed a BERT model, if only by a few percentages. Other models drastically outperformed the LSTM model, such as the ELECTRA model that achieved a perfect score.

**Future Work**

Based off the comparisons of other models, it would seem that using the ELECTRA model would be much better and easier to implement than BERT, because of the resources that are required in order to run the BERT model. From looking for other models to compare to, there are many other models that could be implemented to see how they compare to each other. Comparing to other models also showed the importance of preprocessing the input to improve performance and training time. Along with testing other models, more experimentation could be done to the see if there are any other combination of hyperparameters that could achieve a higher accuracy than what was found during testing.

**References**

Clark, K., & Luong, T. (2020, March 10). More Efficient NLP Model Pre-training with ELECTRA. Retrieved from Google AI Blog: https://ai.googleblog.com/2020/03/more-efficient-nlp-model-pre-training.html

Kaggle. (2019, December 20). Natural Language Processing with Disaster Tweets. Retrieved from Kaggle: https://www.kaggle.com/c/nlp-getting-started/overview

Mirshafiee, M. (2020, October 6). Simple Bert with Video. Retrieved from Kaggle: https://www.kaggle.com/mitramir5/simple-bert-with-video

Ozcangundes. (2020, August 27). NLP\_Disaster\_Tweets with ELECTRA Base. Retrieved from Kaggle: <https://www.kaggle.com/ozcan15/nlp-disaster-tweets-with-electra-base>

Project Pro. (2022, April 22). *BERT NLP Model Explained for Complete Beginners*. Retrieved from Project Pro: <https://www.projectpro.io/article/bert-nlp-model-explained/558>