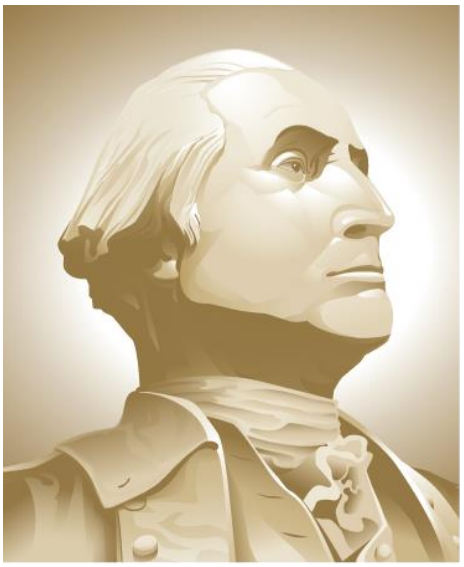


THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Final Project Individual Report: GLUE Benchmark

Course: Natural Language Processing

Professor: Dr. Amir Jafari

Author: Tristin Johnson

December 9th, 2021

Introduction

For our final project, we decided to work on the General Language Understanding Evaluation (GLUE) benchmark, which is a collection of resources for training, evaluating, and analyzing natural language understanding systems. The GLUE benchmark includes 11 different datasets, consisting of a sentence or a sentence-pair language understanding tasks. These tasks are built on established, existing datasets that cover a wide range of typical natural language processing tasks, such as sentiment analysis, textual similarity, question pairs, textual entailment, and more. We decided to work on this benchmark as all the GLUE tasks are consistent with the learning objectives of natural language processing. “The ultimate goal of GLUE is to drive research in the development of general and robust natural language understanding systems.” Throughout this report, we will highlight each of the datasets, the data preprocessing, the models we implemented for each task, along with the results of each model on each dataset.

Individual Work

In terms of how the work was divided, we split up the work based on the datasets. I was in charge of completing 4 of GLUE tasks, which include the Stanford Sentiment Treebank (SST), the Corpus of Linguistic Acceptability (CoLA), Recognizing Textual Entailment (RTE), and Winograd NLI (WNLI). The remaining 7 GLUE tasks were split up among my other two group members, Robert Hilly and Divya Parmar.

Having said that, I used a handful of different models to the 4 datasets in which I was working on. To start, I used the transformers package ‘Training Arguments’ and ‘Trainer’ to get

a base score for each of the tasks. In doing so, the transformers I used to compare for analysis was ELECTRA and ALBERT, which we will discuss more in the next section. I then took these four tasks, using BERT and ELECTRA, and implemented my own custom transformer by adding in some extra preprocessing steps. The purpose of this was to learn how to use the power and model architecture of each of the pretrained models and apply them to data that I preprocess along the way. Furthermore, this was done to help get more practice with the PyTorch framework in terms of training, testing and model parameters. Lastly, I was able to implement my own LSTM model, with my own data preprocessing and none of the pretrained models. This was to try and compete with the pretrained models and see how well I can implement a LSTM neural network.

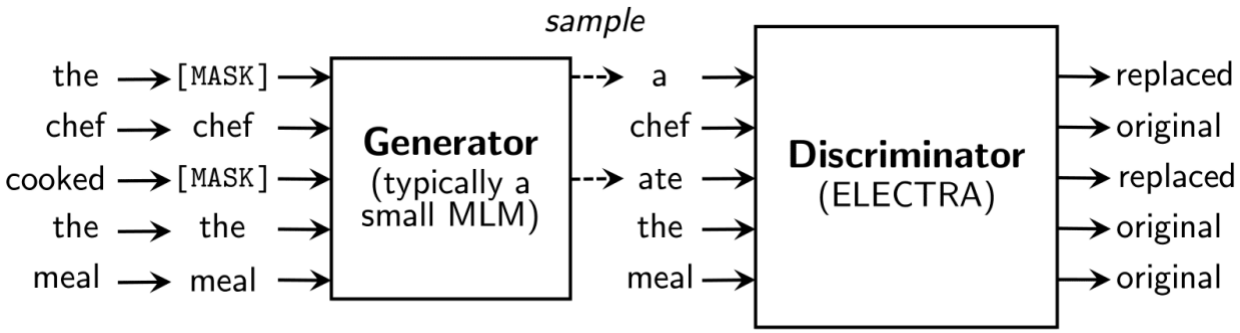
Data Preprocessing:

Since I am using ALBERT and ELECTRA, I wanted to use these models but also customize the preprocessing and input of the data to see if I can beat the baseline accuracy. For both CoLA and SST, there was only one sentence with a label. Therefore, I added a CLS token to the front of the token IDs (using either ALBERT or ELECTRA tokenizer) along with a SEP token at the end. I also added attention masks to help with model performance. Regarding WNLI and RTE, these tasks included two sentences, a 'premise' and 'hypothesis', along with a label. This means that I concatenated the token IDs of both the premise and hypothesis, with a CLS token in the front, a SEP token between the premise and hypothesis token IDs, and a SEP token at the end. I then added attention masks for more information within the neural network. In doing these steps, the idea was to slightly modify the input to both the ELECTRA and ALBERT models, and I will discuss the results later in the report.

Lastly, is the LSTM model that I implemented. The first step was to clean up all the text. To do this, I removed all special characters, removed consecutive white spaces, and removed digits, all using regular expression. I also using the NLTK stop words corpus to remove stop words from the text. Now that the text is clean, I tokenized the text, created a corpus, and then create a one-hot encoded dictionary to translate all the tokens to numbers. After this was done, I had to add padding to make all the texts the same length when they are used as input into the neural network. Now that I have tokenized and padded all the texts, it is now ready to define the LSTM architecture and its model parameters, which can be seen in the next section.

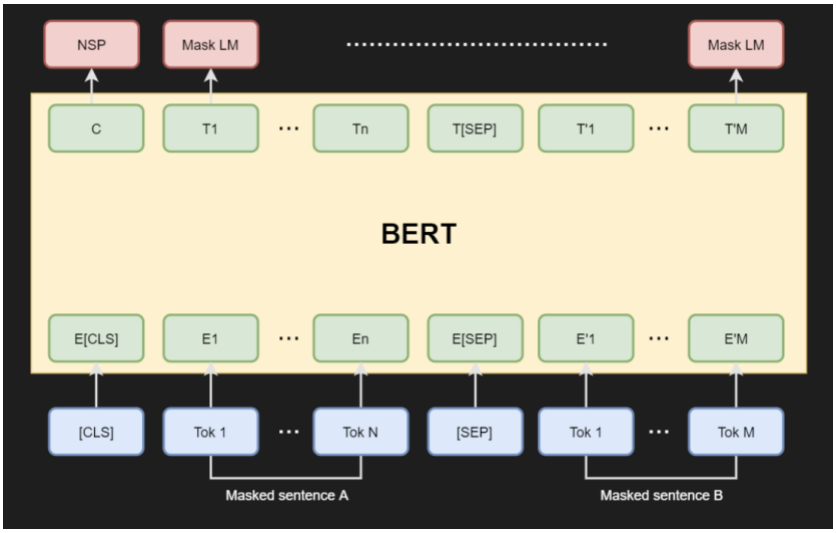
Individual Model Architectures

As mentioned above, the transformers I used for these tasks were ELECTRA and ALBERT. I have read a few interesting research papers regarding ELECTRA and its overall performance in text classification, so I wanted to work with that for this project. ELECTRA, inspired from Generative Adversarial Networks (GANs), is a new pretraining approach which trains two transformer models, being the generator and the discriminator. The generator's role is to replace tokens in a sequence, which in return, is trained as a Masked Language Model (MLM). The discriminator is what tries to identify which tokens were replaced by the generator in the sequence, which is what we're more interested in. ELECTRA workflow can be seen below:

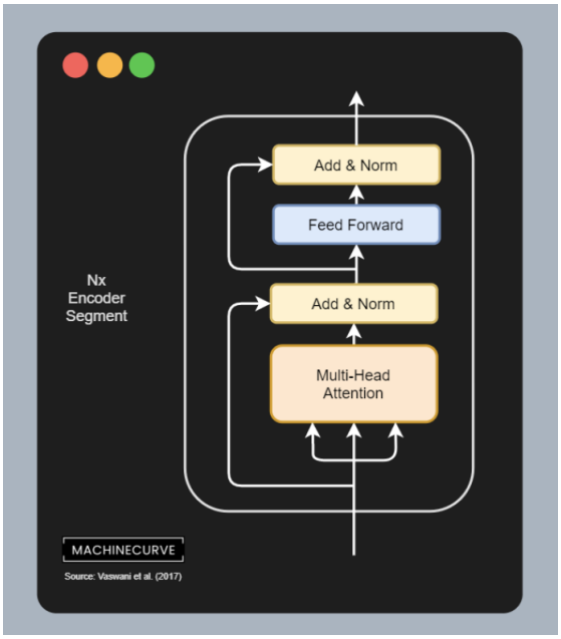


This leads to the discriminator's role, which tries to identify which tokens were replaced by the generator in the sequence, and this is the model we're interested in. The generator and the discriminator share the same input word embeddings, and after pre-training, the generator is dropped, and the discriminator is fine-tuned on downstream tasks.

With regards to ALBERT, I wanted to use this transformer because it is a 'lighter' and 'just-as-efficient' model as BERT. ALBERT is originally derived from BERT, with a few key differences. Below is an example of BERT:



BERT takes two sets of tokens as inputs, and they are then tokenized. Next, a classification (CLS) token is added, which contains sentence-level information on the text. The tokens are then fed into BERT, with word embedding happening first. Furthermore, BERT utilizes two language tasks, a Masked Language Model for predicting output tokens, and a Next Sentence Prediction to learn sentence-level information for the next input. Furthermore, BERT base has 110 trainable parameters and BERT large has 340 million trainable parameters. The goal was to reduce the number of trainable parameters, which led to 'A-Lite-BERT', or ALBERT:



ALBERT utilizes the BERT architecture, in the sense that the ALBERT architecture itself is the encoder segment from the BERT model, with a few minor changes. The first major change is that the embeddings are factorized, meaning the parameters of embedding are decomposed into two smaller matrices in addition to embedding and hidden state size. Another is that ALBERT applies cross-layer parameter sharing, meaning the parameters of the Multi-head Self-Attention encoded segment and the Feedforward encoded segment are shared. With these changes, we now have a faster, and arguably more efficient, BERT model.

Lastly, we can look at the LSTM model that I implemented, in which the architecture can be seen below:

```
SentimentAnalysisLSTM(
  (embedding): Embedding(1501, 64)
  (lstm1): LSTM(64, 256, num_layers=2, batch_first=True)
  (lstm2): LSTM(64, 256, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (linear): Linear(in_features=256, out_features=1, bias=True)
  (act): Sigmoid()
)
```

Looking above, I first define the embedding layer. The embedding layer represents words using a dense vector representation, where the position of a word within the vector space is based on the words that surround the word when it is used. Along with the embedding layer, the vocabulary size from our corpus is provided. Next, I added two LSTM layers. These both consist of the embedding dimension, which is 64, along with the number of neurons, which is 256. I then added a dropout of 0.3 to help reduce overfitting, a linear layer to flatten the output, and sigmoid as the activation function. Now that we know the model architectures of each model we are applying to my GLUE tasks, we can get into the hyperparameters for each model.

Environment Setup

The framework I used for this project was PyTorch, as I have greatly enjoyed learning and utilizing this framework, especially with all the different variations you can apply when training a model. For the transformer models, I used the following hyperparameters: batch size of 32, a max length of 256, a learning rate of 0.001, and 5 epochs. When compiling these models, I used AdamW as the optimizer, 'ReduceLROnPlateau' for the scheduler, and Cross Entropy as the loss function. For the LSTM model, I used the following hyperparameters: batch size of 32, 20 epochs, 2 layers, embedding dimension of 64, 256 neurons, and a max length of 256. When compiling this model, I used a learning rate of 0.001, the Adam optimizer, the 'ReduceLROnPlateau' scheduler, and a Binary Cross Entropy loss function. Now that we know how the data was preprocessed, all about the models and architectures, along with the model parameters and hyperparameters, we can look at the results of each model.

Results

Below is a table of the results that I was able to obtain while training and testing on the SST task:

	Train Accuracy	Test Accuracy	Training Time per Epoch	Testing Time per Epoch
ELECTRA	N/A	90.137%	N/A	3 min 52 sec
ALBERT	N/A	86.811%	N/A	11 min 17 sec
Custom Model (ELECTRA)	55.783%	51.927%	2 min 19 sec	3 sec
Custom Model (ALBERT)	52.433%	50.917%	22 min 27 sec	16 sec
LSTM	88.419%	77.315%	52 sec	2 sec

Looking above, we can see that ELECTRA was able to achieve the highest accuracy score out of all the models, with the ALBERT coming second. I was happy that the LSTM accuracy score on the test set was able to beat the custom model using the transformers, and the speed that the LSTM model has was very quick. Next, we can look at the results for CoLA:

	Matthew's Correlation Coefficient (Test)	Training Time per Epoch	Testing Time per Epoch
ELECTRA	0.5508	N/A	33 sec
ALBERT	0.4187	N/A	1 min 29 sec
Custom Model (ELECTRA)	0.212	16 sec	11 sec
Custom Model (ALBERT)	0.1981	1 min 20 sec	14 sec
LSTM	0.11951	7 sec	1 sec

Since CoLA used Matthew's Correlation Coefficient as the metric, we can see that ELECTRA was able to achieve the highest score, with the ALBERT model getting second. Regarding accuracy, I was able to achieve the highest score with ELECTRA in the custom transformer model, with a test accuracy of 70.12%. Next, we can take a look at the RTE task results:

	Train Accuracy	**Test Accuracy**	**Training Time per Epoch**	**Testing Time per Epoch**
ELECTRA	N/A	63.176%	N/A	58 sec
ALBERT	N/A	54.151%	N/A	2 min 40 sec
Custom Model (ELECTRA)	50.281%	52.708%	23 sec	3 sec
Custom Model (ALBERT)	49.116%	51.818%	2 min 26 sec	5 sec

Once again, ELECTRA came in first, with ALBERT coming second. What surprised me here is the speed at which ELECTRA is able to accurately predict values. Finally, we can look at the WNLI task:

	Train Accuracy	**Test Accuracy**	**Training Time per Epoch**	**Testing Time per Epoch**
ELECTRA	N/A	56.828%	N/A	8 sec
ALBERT	N/A	57.38%	N/A	18 sec
Custom Model (ELECTRA)	51.496%	56.338%	3 sec	1 sec
Custom Model (ALBERT)	49.606%	56.338%	12 sec	1 sec

Looking above, ALBERT was finally able to outperform ELECTRA and achieve the highest accuracy score. One thing to note, for this task in particular, the other three scores came oddly

close together, with the custom models using ELECTRA and ALBERT having the exact same accuracy score, which is quite interesting.

Summary of Results & My Work

Looking at these accuracy scores, overall ELECTRA was the best performer out of all four of these GLUE tasks. What impressed me is ELECTRA's speed in terms of epoch time in both training and testing, along with the fact that it was able to achieve high accuracy scores. Regarding ALBERT, I was a little disappointed with some of the scores that ALBERT achieved. After reading some research papers on this model, the idea is that decreasing the trainable parameters from BERT has the potential to make the model better, but in these tasks, that was not the case. Furthermore, I was content with the scores that my LSTM model was able to achieve and was able to outperform some of the other models' scores.

In completing these tasks, a lot of code was written in order to have a file for each task. In total, there were 2,023 lines of code. Of that code, I used approximately 790 lines of code from resources online. Therefore, in total, I wrote about 60.949% of the code myself. A lot of this was quite repetitive, especially with just loading the original models, and you will notice that in the code.

Conclusion

Overall, I was content with the outcome of doing this project. I had the ability to learn about two pretrained transformers (ELECTRA and ALBERT), customize these transformers, implement an LSTM model, and compare the scores of all these models against each other for

each task. Furthermore, I learned a lot about applying different neural networks to natural language understanding tasks, in which these tasks are applicable in the real-world. Also, I learned a lot about the PyTorch framework, and have been becoming more and more of a fan of this framework for implementing neural networks. All in all, this was a great project to test my understanding of handling textual data, and I am looking forward to applying this knowledge in the future.

References

1. "Transformers." *Hugging Face*. https://huggingface.co/docs/transformers/index
2. "Sentiment Analysis using LSTM – PyTorch". *Kaggle,* 8 June 2021, https://www.kaggle.com/arunmohan003/sentiment-analysis-using-lstm-pytorch
3. "Text Classification on GLUE". *Google Colab*. https://colab.research.google.com/github/huggingface/notebooks/blob/master/examples/text_classification.ipynb - scrollTo=HFASsisvIrIb
4. Verma, Dhruv. "Fine-tuning Pre-Trained Transformer Models for Sentence Entailment." *Towards Data Science,* 14 Jan. 2021. https://towardsdatascience.com/fine-tuning-pre-trained-transformer-models-for-sentence-entailment-d87caf9ec9db
5. Shekhar, Shraddha. "LSTM for Text Classification in Python." *Analytics Vidhya,* 14 June 2021. https://www.analyticsvidhya.com/blog/2021/06/lstm-for-text-classification/
6. Clark, Kevin. "ELECTRA." *GitHub*. https://github.com/google-research/electra
7. Versloot, Christian. "ALBERT explained: A Lite Bert." *Machine Curve,* 6 Jan. 2021. https://www.machinecurve.com/index.php/2021/01/06/albert-explained-a-lite-bert/