Fine Tuning a BERT Model for Text Classification

Kishan Polekar
School of Information Studies
Syracuse University
Syracuse, NY
kpolekar@syr.edu

***Abstract*—The purpose of this paper is to present an analysis of a project that introduces the concept of BERT models, provide details about sample data, the techniques used to change the data for further analysis, and select a pre-trained model upon which our model is built. We try tweaking several hyperparameters to test if the model performs well and provide results upon training the model. Finally, future work and shortcomings in this analysis are discussed.**

I. Introduction

The increasing popularity of BERT (Bidirectional Encoder Representations from Transformers) models such as ChatGPT has opened up new possibilities for natural language processing (NLP) applications. These models can generate text, predict responses, accurately guess hidden or missing words, and perform various other text and/or speech related tasks. This project aims to leverage the capabilities of BERT models to classify sentences or phrases into distinct categories.

The main goal of this project is to fine tune an existing BERT model for text classification mainly for detecting similarities between two sentences. Classifying sentences or phrases into categories would greatly benefit many industries such as public health, banking, and any other industry that utilize automated response machines.

Accurate separation of words or sentences based on certain domain would enhance customer service, minimize errors in processing, and facilitate automated responses, in turn reducing response times. The model could contribute to AI gaining a better understanding of human emotions which would lead to a reduction in human effort and optimized time for various tasks.

II. Problem and Data Description

A. Problem

Natural Language Processing (NLP) applications offer a valuable opportunity to leverage the capabilities of BERT models, especially in fields that heavily rely on automated response machines, as well as those that involve extensive human interaction, data entry, and processing—such as health and banking sectors. In these industries, effectively handling textual information is important for improving customer service, reducing errors in processing, and enhancing overall efficiency. The goal of this project is to develop a BERT model that can classify sentences or phrases into predefined categories.

To implement this classification model, the project utilizes a pre-trained BERT language model and TensorFlow which is a widely used open-source machine learning library. BERT's ability to understand contextual information and semantic relationships in text makes it well-suited for tasks involving sentence classification and similarity assessment.

B. Data Description

The dataset chosen for training and evaluating the model is the GLUE (General Language Understanding Evaluation) MRPC (Microsoft Research Paraphrase Corpus), accessed through TensorFlow Datasets (TFDS). This dataset consists of pairs of sentences collected from diverse online news sources. Each sentence pair is annotated to indicate whether the sentences are semantically equivalent or not. This annotated information serves as the base for training the model, with the two primary class labels being 'equivalent' and 'not_equivalent.'

TABLE I. Data Split GLUE MRPC

Split	***Number of Examples***
Train	3,668
Validation	1,725
Test	408

The above table lists the number of rows (sentence pairs) in each split of the original data.

The maximum sequence length of training and evaluation (validation) data is 128 characters.

Feature	Class	Shape	Dtype
	FeaturesDict		
idx	Tensor		int32
label	ClassLabel		int64
sentence1	Text		string
sentence2	Text		string

Fig. 1. Format of each observation's structure in original dataset.

III. APPROACH / ALGORITHM

Our analysis requires a few steps before we head into the model training. First, we must gather the data which is done through TFDS, and we must pre-process the data so that it is understood by our model. The pre-trained BERT model does not recognize raw text; hence two things must be done before feeding it to the model obtained from the Model Garden:

1. The text needs to be *tokenized* (split into word pieces) and converted to *indices*.
2. Then, the *indices* need to be packed into the format that the model expects.

Once we have formatted the data as expected, we can build and train the model. The following steps are involved in this process:

1. *Build the model:* Download the configuration file for the pre-trained BERT model.
2. *Restore the encoder weights:* Restore the encoder's randomly initialized weights from the checkpoint.
3. *Set up the optimizer:* Select an appropriate optimizer for training the model, set up initial hyperparameters.
4. *Fine tune hyperparameters:* Before training the model, we try out different values of hyperparameters and select the one that works best.
5. *Train the model:* Once we have the parameters tuned, we compile and fit the model to our training data and validate it against the validation data.
6. *Derive results and make predictions:* Finally, once we have our model trained, we make predictions on new unseen data and see if the model is able to perform as desired.

After all the above steps are completed, we explore the limitations with our model and analysis, and explore future steps if we continue our research.

IV. DATA & MODEL

A. Preprocessing

The pre-processing of raw text data involves the following steps:

1) Tokenize each word in the sentence using the BERT Tokenizer:

In the first step, the sentence undergoes a process called tokenization. Tokenization involves breaking down the input text into smaller units, typically words or subwords. For BERT models, a specialized tokenizer is employed to handle this task. The BERT Tokenizer segments each word into smaller units known as tokens, and these tokens form the basic building blocks for the subsequent stages of processing. The raw text is encoded into numbers or 'tokens' for the model.

2) Pack the tokenized inputs into a BERT-recognized nest of tensors:

After tokenization, the individual tokens need to be organized in a structure that BERT can comprehend. BERT recognizes a specific format for input data, typically organized as a nest of tensors. This nesting includes essential elements like the input_word_ids, input_mask, and input_type_ids. The packed input is formatted according to the requirements of the BERT model, ensuring that the information is presented in a way that allows the model to understand and process the input effectively.

3) Tokenize special characters using a special tokens dictionary containing 30k+ samples:

Special characters and tokens, such as [CLS] (classification token), [SEP] (separator token), and [MASK] (mask token), play crucial roles in BERT's architecture. In this step, special characters within the text are tokenized. Additionally, a special tokens dictionary containing a substantial number of samples (30k+) is used. This dictionary includes pre-defined embeddings for special characters or entities, contributing to the model's ability to handle a broad range of language patterns and expressions.

4) BertPackInputs expects the following format:

a) Type of problem: [CLS] in our case

b) Sentence 1 (tokenized): with SEP token (denoting Sentence Separation)

c) Sentence 2 (tokenized): with SEP token (denoting End of Sentence)

5) Returns a dictionary containing:

a) ***input_word_ids****:* The tokenized sentences packed together.

b) ***input_mask****:* The mask indicating which locations are valid in the other outputs.

c) ***input_type_ids****:* Indicating which sentence each token belongs to.

For example, the data includes sentences "hello tensorflow" and "goodbye tensorflow," with corresponding tokenization results:

“hello tensorflow”: [[[7592], [23435, 12314]]]
“goodbye tensorflow”: [[[9119], [23435, 12314]]]

Packing the two sentences together, the format becomes:

[[101, 7592, 23435, 12314, 102, 9119, 23435, 12314, 102, 0, 0, 0]]
[[[CLS], sentence_1, [SEP], sentence_2, [SEP], special_tokens_location]]

After we pack the inputs to pass in the model, we get the following input shapes:

```
input_word_ids  shape: (32, 128)
input_mask      shape: (32, 128)
input_type_ids  shape: (32, 128)
labels          shape: (32,)
```

Fig. 2. Input shapes to feed in the pre-trained BERT model.

B. Build the model

When building our model, we first have to download the config file for the pre-trained BERT model that we will base our model on. Once we have that, we test it out on a sample of 10 pairs of sentences taken from the training data. We get an output of the ‘similarities’ between the pairs, and this output is in the form of logits of the two label classes (targets). This shows that the pre-trained BERT model has successfully been downloaded and is ready for further use. The structure of the BERT model is as follows:

- Activation function (hidden layers): GELU
- num_classes: 2
- num_hidden_layers: 12
- hidden_dropout_prob: 0.1
- input_shape: (32,128)
- Size of the network: 768 nodes

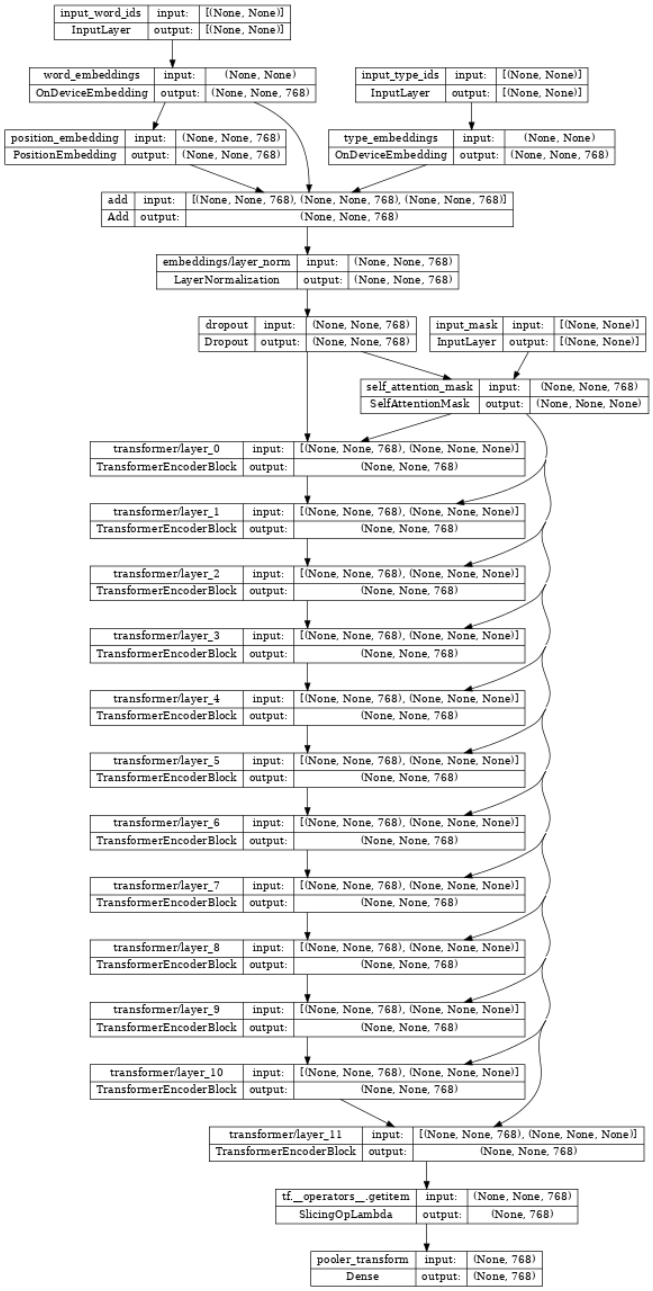

Fig. 3. BERT encoder model.

As seen in the above figure, the BERT model has an input layer that accepts data in the form of nest of tensors which we have built in the previous steps. This is then passed through a couple of embedding layers, and then fed to a pooling layer that flattens our sentence pairs in the shape of (input_size1, input_size2, 768). After this, there is layer normalization and some dropout that is applied. Finally, after masking the input, it is passed through a series of 12 hidden transformer layers which in turn gives us an output through the dense layer. This output is in terms of logits which have to be converted to actual probabilities later on.

When the model is set up, the encoding weights are randomly initialized. We restore these weights from a checkpoint using the tensorflow Checkpoint function and pass the bert_model's checkpoint parameter. Once this is done, we are ready to move on to the optimizer setup.

C. Set up the optimizer

BERT usually uses the Adam optimizer with weight decay (AdamW) which is what we will stick to for this analysis. It employs a concept of learning rate starting from 0, then 'warming up' to a certain value, and then gradually degrading/decaying back to 0. For the model, we will use a Polynomial Decay function, and a Linear Warmup scheduler. The optimizer also initializes the batch size for our training and evaluation dataset to 32 and the number of epochs to 5. For our model, we use accuracy as an evaluation metric and the SparseCategoricalCrossEntropy loss function since we have a classification problem and almost all the sentences do not meet the maximum size requirement and need padding, so we have a lot of empty data when passing to the model, We try out different values of batches and learning rates, but they do not seem to have a significant effect on the output when tested on the validation dataset (metrics compared: loss & accuracy). Below we have a graph showing the weight decay timeline over the number of training steps for some initial learning rate (1e-5).

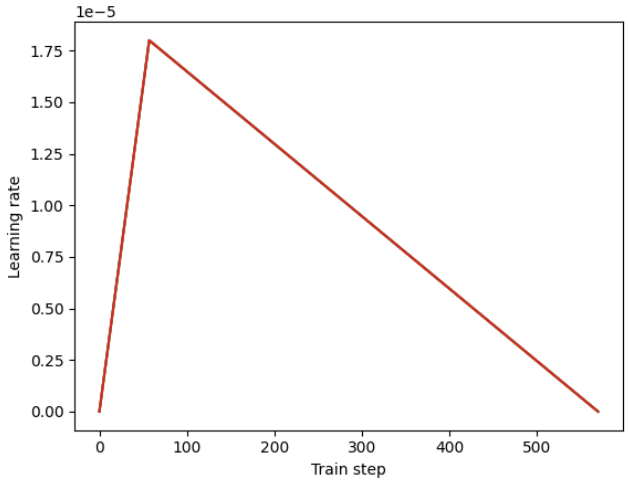

Fig. 4. Weight decay over time with learning rate 0.00001 (1e-5).

D. Fine tune hyperparameters

The different learning rates tried out and compared against validation data for loss and accuracy through base model optimization resulted in the below outcomes:

TABLE II. BASE MODEL PERFORMANCE ON VALIDATION DATA WITH DIFFERENT LEARNING RATES

Learning Rate	***Learning Rate (Actual)***	***Loss***	***Accuracy***
1e-5	0.00001	0.6671	68.38%
2e-5	0.00002	0.6490	68.63%
1e-3	0.0001	0.6251	68.38%
2e-2	0.002	0.6217	68.35%

E. Train the model

After we have the hyperparameters setup and fine-tuned, we compile the model and fit it to our training data. We test out the results on both training and validation data. Over 5 training epochs, we achieve very good accuracies, and the losses also go down by a lot. As compared to the base BERT model that we downloaded earlier, our model seems to perform much better.

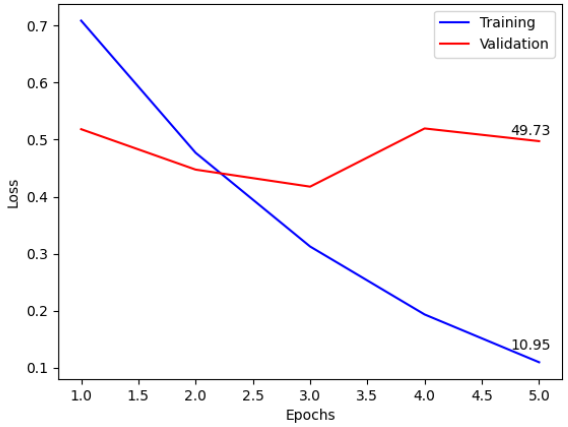

Fig. 5. Training and validation loss over 5 epochs on optimized BERT model.

In the above plot, we see how the training loss goes down rapidly with each epoch, but the validation loss seems to stay more or less the same, but this validation loss is still lower as compared to the base model's validation loss (0.649).

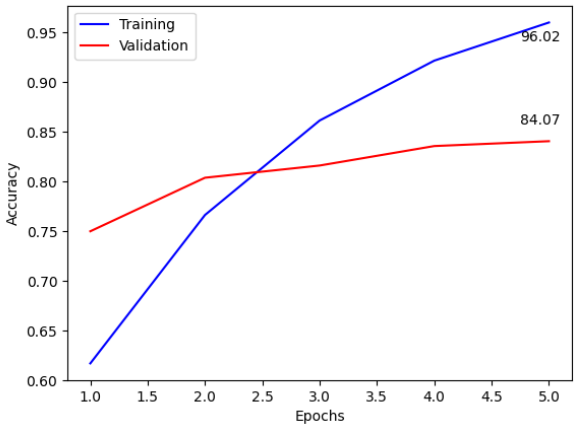

Fig. 6. Training and validation accuracies over 5 epochs on optimized BERT model.

From the graph above, we can see that our model works very well on both our training and validation datasets. The training accuracy seems to go up from about 60% in the beginning stages to about 96% when we are done fitting the model to our training data.

V. Results

The fine-tuned BERT model was trained using a specific dataset to adapt its parameters for predicting similarities between pairs of sentences. The fine-tuned model successfully predicted similarities between pairs of sentences using the TensorFlow Datasets library and GLUE BERT configurations.

The pre-trained BERT model was further enhanced through hyperparameter tuning which involves adjusting the configuration settings of the model to optimize its performance. In this context, hyperparameters such as weight decay, learning rate, and others were fine-tuned. The fine-tuning process aims to strike a balance that maximizes the model's predictive accuracy on unseen data.

The validation loss is a measure of how well the model generalizes to new, unseen data. A decrease in validation loss from 0.649 to 0.4973 indicates an improvement in the model's ability to make accurate predictions. Additionally, the accuracy on the training set reached a peak of 96%.

An example of how the model performs on new data in terms of predicting whether a pair of sentences are equivalent to one another or not is shown in the table below:

TABLE III. Sample Sentence Pairs and Outputs

Sentence Pair 1	***Sentence Pair 2***
'The rain in Spain falls mainly on the plain.'	'Look I fine tuned BERT.'
'It mostly rains on the flat lands of Spain.'	'Is it working? This does not match.'
'equivalent'	'not_equivalent'

VI. Next Steps

The model currently classifies sentence pairs into two groups: 'Equivalent' and 'Non-equivalent'. Future improvements and extensions could include:

1. Domain-Specific Classification: Expand the model to classify sentences into more specific categories within a particular domain, such as the 'banking77' model with 77 class labels for phrases in the banking sector.

2. GPU Utilization: Consider using a GPU instead of a CPU for model training to reduce training time and enhance efficiency.

These enhancements can contribute to the model's adaptability to different domains and improve its performance in real-world applications.

References

[1] "Fine-tuning a BERT model," TensorFlow, 17 10 2023. [Online]. Available: https://www.tensorflow.org/tfmodels/nlp/fine_tune_bert. [Accessed 12 2023].

[2] "glue," TensorFlow, 06 12 2022. [Online]. Available: https://www.tensorflow.org/datasets/catalog/glue#gluemrpc. [Accessed 12 2023].

[3] PolyAI, "banking77," HuggingFace, [Online]. Available: https://huggingface.co/datasets/banking77.