Final Project Report
ECEN 5060
Deep Learning

Zach Johnson
A20202995

Introduction

For my final project, I fine-tuned a pretrained BERT-Tiny language model for sentiment analysis. I then broke that fine-tuned model down to basic matrix multiplications to better understand how the architecture works. The goal of this project was to improve the chosen models sentiment understanding. The model should be able to take in a sequence and determine if the overall tone of the text is "positive" or "negative". For example, we could give the model this sequence: "That movie was terrible!". The model should then determine that the tone of this sequence is negative. To fine-tune the chosen model, the "IMDB" movie review dataset was chosen. This dataset contains movie reviews that are labeled either positive or negative. Once we have the fine-tuned model saved, it is easy to access the weights and biases for each layer. Using these weights and biases, we can build the model entirely from scratch. After the model is built from scratch, we can explore future areas of research using tiny and well-performing language models for application-specific deployment.

Project Schedule

The schedule for this project was a little unorthodox. This project topic is highly related to my thesis work, so building the model from scratch has been an ongoing project throughout this semester. However, the model itself and the desired application are different in this project. So, the main task here was understanding the architecture difference between my thesis application and this application, sentiment analysis. Because of this and the fact I worked individually, the project management features in Github were not used. But, an estimated Gantt chart is provided. Here is an estimate of task dates.

- 03/04 - 04/14 : BERT-Tiny model breakdown
- 04/15 – 04/22 : Fine-tune BERT-Tiny for Sentiment Analysis
- 04/22 – 04/26 : Adjust broken down BERT-Tiny model to fine-tuned model
- 04/27 – 04/29 : Prepare presentation materials

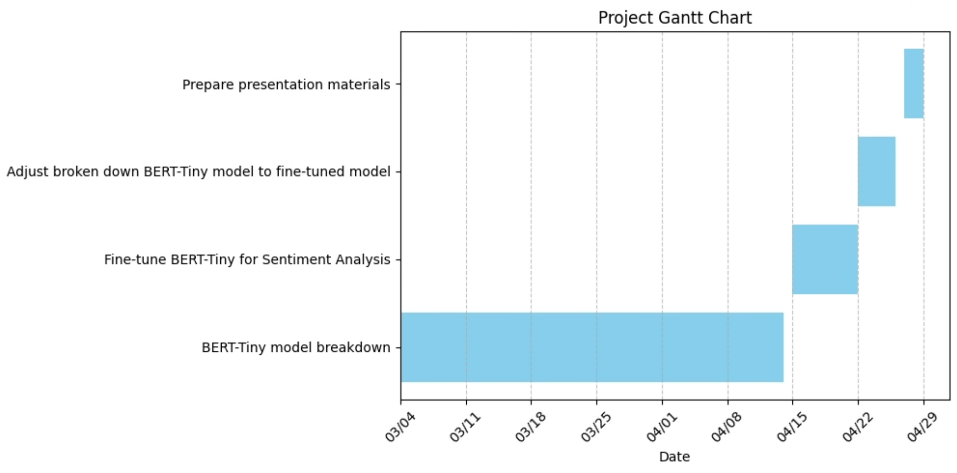


Figure 1: Final Project Gantt Chart

Dataset Description

The dataset chosen to fine-tune BERT-Tiny was the "IMDB" movie review dataset. It contains 50,000 unique and highly polar movie reviews. This means that each movie review is either clearly positive or clearly negative. This prevents any ambiguity, such as sarcasm, from changing the decision boundaries of the model. The dataset is split 50/50 between training and test, so there are 25,000 movie reviews for training and 25,000 for testing. Here is an example of a review that can be found within this dataset.

- "Probably my all-time favorite movie, a story of selflessness, sacrifice and dedication to a noble ca..."

This example is labeled as '1', or positive.

Network Details and Training

First, I will describe the details of the selected model. The model I chose for this task is BERT-Tiny, a tiny variation of the BERT language models with only 4.3 million parameters. BERT-Tiny consists of four key layers: the embedding layer, two transformer layers, and the classification head. Using the Transformers library from HuggingFace, you can print the model architecture. This can be seen in the appendix figure 1. This describes each of the linear layers that exist within the model, including the input and output shapes. Knowing this information will help when building the model from scratch.

BERT stands for Bidirectional Encoder Representations from Transformers. BERT is a discriminitive model, meaning it is deployed primarily for Natural Language Understanding (NLU) tasks. It uses bidirectional contextualization, so it can consider the context of words before and after a specific word. This concept is also known as self-attention, which we will discuss in more detail later in this report.

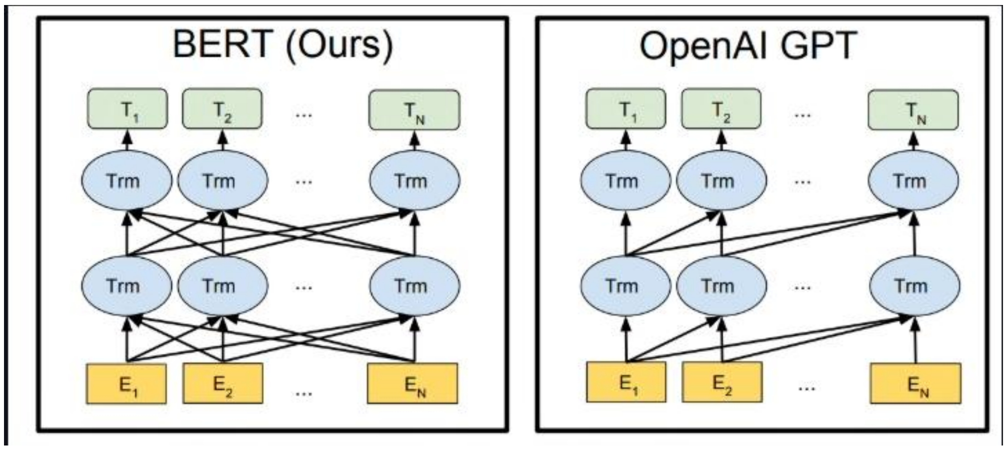


Figure 2: BERT Bidirectional Contextualization vs. OpenAI GPT

Looking at Figure 2, we can see that there is consideration of tokens before and after each token in BERT. But for OpenAI GPT, the only considered context is for tokens after that specific token. This makes BERT perfect for a task like sentiment analysis, where the context of each token must be considered. Now that we understand how the BERT language model works, we can train the model on the dataset using the Transformers library from HuggingFace.

For training the model, there were two specific functions used from HuggingFace. These functions are the TrainingArguments function and the Trainer function. Both of these functions can be found in the appendix. TrainingArguments sets different values such as number of epochs, batch size, and output directory. This saves and outputs different fine-tuned model weights throughout the training process. Once the training arguments are set, they are passed to the Trainer function. The Trainer takes in the model, the training arguments, and the data seperated between training, evaluation, and testing. The trainer also includes a data collator, which pads the input text to the max length BERT can handle (512 tokens) so that the length of each sequence is the same. This made training very intuitive and straight-forward.

Experimental Setup

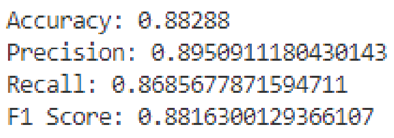
Before training the network, I split the data up strategically to allow for more training data. Originally, the dataset is split 50/50 in terms of training and testing data. To allow the model to have more training data, I split the testing data in half and concatenated it to the training data. Then, I created a 90/10 split of that training data for evaluation data. So, this was the final split of data:

- Training Samples: 33,750
- Evaluation Samples: 3,750
- Testing Samples: 12,500

This gives the model more information to train on, especially considering we are using a very small language model for this task. I found that twenty epochs was enough for the model to converge to a solid accuracy. Stopping the training before twenty epochs showed there was more for the model to learn. Any training after twenty epochs showed similar performance and no gained generalization from the model. I also found that a batch size of sixteen was perfect for fine-tuning this model in terms of performance. This was a balance of getting the best performance out of the model as well as considering memory usage. In terms of learning rate, the Trainer function from HuggingFace makes it very easy. It uses its own learning rate scheduler, and adjusts the learning rate based on the loss shown from the model during training. During the training process, the model showed no signs of overfitting, as the evaluation/testing results closely matched the results seen with the training data. Additionally, there was no data imbalance to consider, as the “IMDB” movie review dataset contains a balanced number of positive and negative movie reviews. Because of the number of samples present, I did not perform any data augmentation. Lastly, the dataset was loaded using the load_dataset function from the datasets library. This made it very easy to handle the data, as it is split and labeled by “train” and “test”. The process of loading the data in can be seen in appendix figure 3. It shows using the load_dataset function with the key word “imdb”, shuffling the dataset, then splitting the dataset up into train and test.

Training Results

The BERT-Tiny model was able to perform well on the desired sentiment analysis task. The metrics of the model after fine-tuning can be found below in Figure 3.



```
Accuracy: 0.88288
Precision: 0.8950911180430143
Recall: 0.8685677871594711
F1 Score: 0.8816300129366107
```

Figure 3: Model Metrics after Fine-Tuning

The model was able to predict the sentiment of the test movie reviews with an accuracy of 88%. While this is solid performance, especially for how small the model is, this metric could be misleading. To verify that the model is performing well in all facets, we can look at precision, recall, and the F1 score of the model on the held out test set. We see a precision of 89%, which is actually higher than the overall accuracy. This indicates that if the model is predicting that a movie review is positive, it is more than likely correct. But, we see a lower recall performance at around 86.8%. This indicates that more positive reviews are missed than negative reviews. Meaning, the model is more conservative when it comes to classifying a review as positive. If there is any level of ambiguity, such as uncertainty or sarcasm, the model will predict the review as negative. We can see this visually with the confusion matrix in Figure 4.

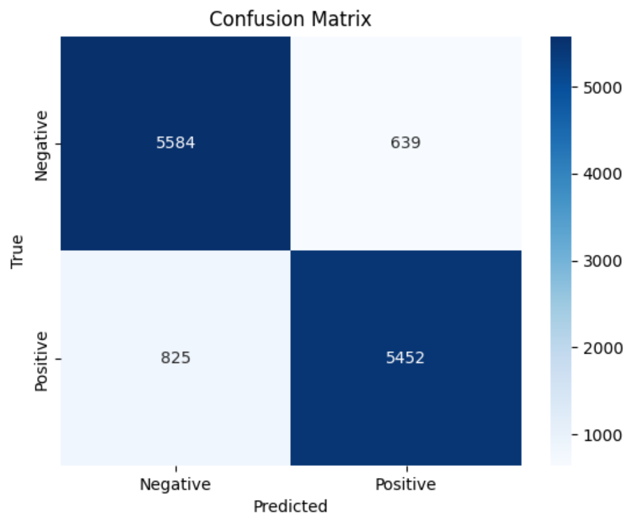


Figure 4. Confusion Matrix

We can see that when a review is a True Positive, there is a higher number of Predicted Negatives in comparison to Predicted Positives for True Negatives. This further reiterates the fact that the model is more conservative when predicting that a review is positive. Lastly, we will look at the F1 score of the BERT-Tiny model on the test set. We see a high F1 score that matches the accuracy of the model. This indicates the model is able to predict reviews as positive at a high rate successfully, while also minimizing the amount of missed positive reviews.

Model Breakdown

Now that we have BERT-Tiny fine-tuned, we can now build the model from scratch. We can access the weights and biases of the fine-tuned model easily using the Transformers library from HuggingFace. An example of accessing those weights can be found in Appendix Figure 4. There are four key layers that we need to implement within the model: the input embedding layer, two transformer layers, and the classification head. A visual of the model can be seen in Figure 5.

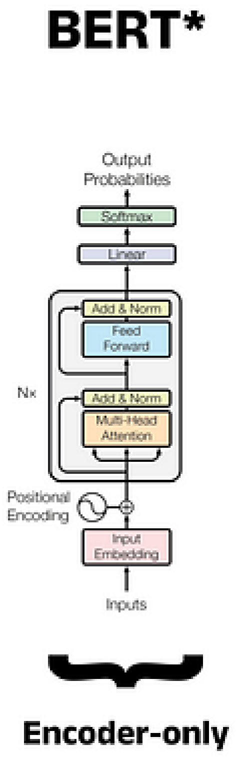


Figure 5. Visual of the BERT Architecture

The first layer we will build is the input embedding layer. The goal of this layer is to take text and convert it into vector representations that the model can understand. There are three embeddings that the model considers, the first being token embeddings. Each token in the vocabulary of the model has a unique vector representation. So these vectors are grabbed from the model for each token within the input sequence. The next embedding considered is positional embedding, which indicates to the model where in the sequence that specific token exists. The last embedding is the segment embedding, which is mainly used for tasks such as question and answering. It indicates to the model when the first sequence ends and when the second sequence starts. For this task, this is not necessary. So, each sentence in the sequence is considered as one large sentence. Once each of these embeddings are found, they are summed together and normalized. A visual of this process can be found below in Figure 6. The code for the embedding layer can be found in Appendix Figure 5.

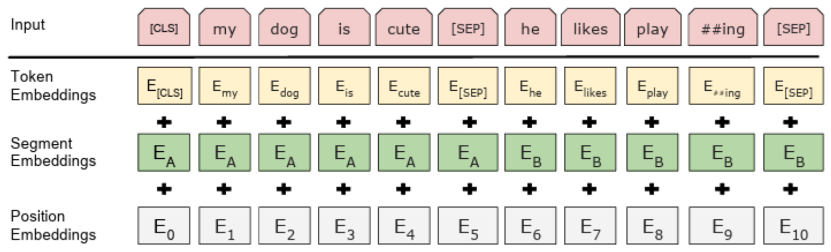


Figure 6. Input Embedding Layer Process

Once we have the output from the Embedding Layer, it is sent forward to the Transformer layer. More specifically, the embedding information is fed to the Multi-Headed Attention mechanism in the Transformer layer. There are three important matrices within the attention mechanism: Query (Q), Key (K), and Value (V). The model has fine-tuned weights and biases for each of these matrices, and the Query, Key, and Values are calculated using the input (X).

$$Q = X \cdot W_Q + b_Q$$

$$K = X \cdot W_K \ + b_K$$

$$V = X \cdot W_V \ + b_V$$

Once this is calculated, each token in the input sequence has a designated Query, Key, and Value vector. To calculate the attention score between two tokens, we take the Query of one token and the Key of the other and compare them (multiply). This indicates how relevant those two tokens are on each other in the input sequence. To calculate the attention score of the entire sequence:

$$Attention\ Score_{ij} = Q_i \cdot K_j^T \text{ (relevance of token j on token i)}$$

Once we have calculated the attention scores for the entire sequence, we scale these scores. This prevents the scores from growing too fast.

$$Scaled\ Score_{ij} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}} \text{, where dk is the dimension of K}$$

After scaling the attention scores, we can apply the softmax function on these values. This will output a list of probabilities for each token, indicating how likely each token is relevant to the desired token.

Attention Weights = softmax(Scaled Score)

Lastly, we multiply the Attention Weights with the Value (V) matrix calculated earlier. The Value matrix indicates the information about that token that should be passed on to the next layer if it is found to be relevant. So, if we see a high probability from the softmax function, that information will be passed on. If the probability that token is relevant is low, that information will not be passed on. The context calculation is below.

$$Context_i = Attention\ Weights \cdot V_j$$

This is the entire self-attention mechanism. The passed on context indicates to the model which tokens are relevant, and which are not. The last caveat of the attention mechanism is the aspect of “Mult-Headed” attention. It is important to note that it is considered within the fine-tuned weights and biases that there are two attention heads. The calculated Query, Key, and Value matrices are reshaped and transposed for this consideration. The implementation of the attention mechanism can be found in Appendix Figure 6.

After going through the attention mechanism, there is a Feed-Forward Network which consists of two dense (linear) layers and a GELU activation function. GELU is approximated using the function found in Figure 7.

$$\mathrm{GELU}(x) \approx 0.5x\left(1+\tanh\left(\sqrt{\frac{2}{\pi}}\left(x+0.044715x^3\right)\right)\right)$$

Figure 7. GELU Activation Function Approximated

This takes the inherently linear output from the attention mechanism and makes more complicated and sophisticated decision boundaries for the model. The implementation of the Feed-Forward Network can be found in Appendix Figure 7. This the entirety of the transformer layer. There are two of these layers in BERT-Tiny, so it is instantiated twice.

The last layer is the classification head. This layer consists of a pooler, a tanh activation function, and a classifier dense layer. The entire sequence is summarized into the “[CLS]”, a BERT specific token, by going through the pooler. This information then goes through a tanh activation function, which squashes the values seen between -1 and 1. Then, it goes through the classifier linear layer, which outputs unnormalized values indicating the model prediction. Lastly, these values are softmaxed, indicating the model prediction in terms of probability. The implementation of the classification head can be found in Appendix Figure 8. This is the entirety of the model, built entirely from scratch.

Conclusion

In this final project, I took a BERT language model variant, BERT-Tiny, and fine-tuned it for sentiment analysis. Then, I took the fine-tuned weights and biases from that model and built it entirely from scratch. We see good performance results from the model, even when the architecture of the model is very small. There are several things I learned, including adjusting a pretrained model for a specific task, handling data, and how the transformer architecture actually works. In terms of improvements, it is possible that introducing more data not related to the “IMDB” dataset could improve the performance of the model.

Now that we have the model built from scratch, we can look at some future areas of research. It is possible to implement the entire model and the multiplications required in digital hardware. This would allow us to deploy a small, but well-performing, language model for a specific application. With quantization, it would only require around 4 MB of memory to store the model parameters. This is an exciting area of research, especially considering the current state of the art language models continue to grow in parameters and power consumption.

References

[1] A. Vaswani *et al.*, “Attention Is All You Need,” 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, K. Google, and A. Language, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” 2019.

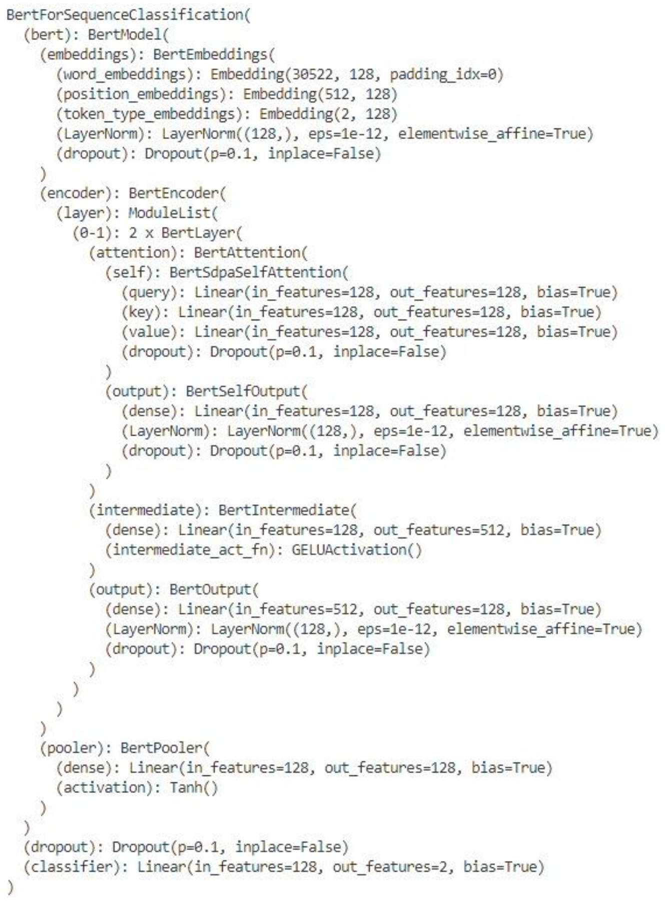
[3] arunmohan003, “Transformer from scratch using pytorch,” *Kaggle.com*, Jan. 09, 2023. https://www.kaggle.com/code/arunmohan003/transformer-from-scratch-using-pytorch (accessed May 06, 2025).

[4] Lakshmipathi N, “IMDB Dataset of 50K Movie Reviews,” *www.kaggle.com*. https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

[5] Bharath K, “BERT Transformers for Natural Language Processing,” *Paperspace Blog*, May 18, 2022. https://blog.paperspace.com/bert-natural-language-processing/

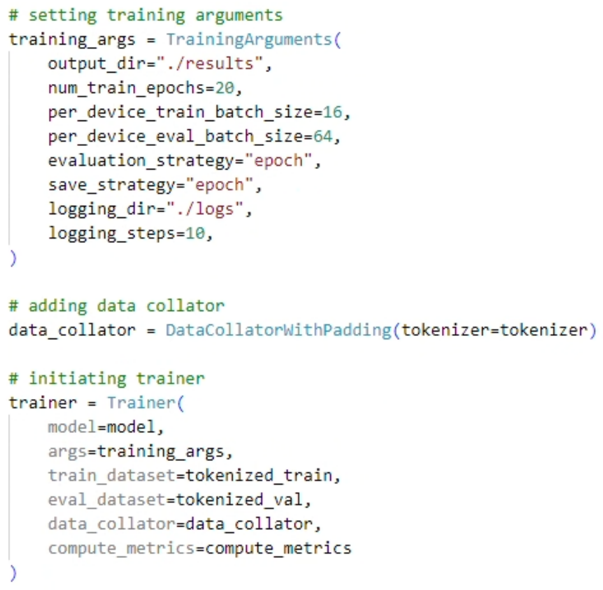
[6] “Trainer,” *huggingface.co*. https://huggingface.co/docs/transformers/en/main_classes/trainer

Appendix



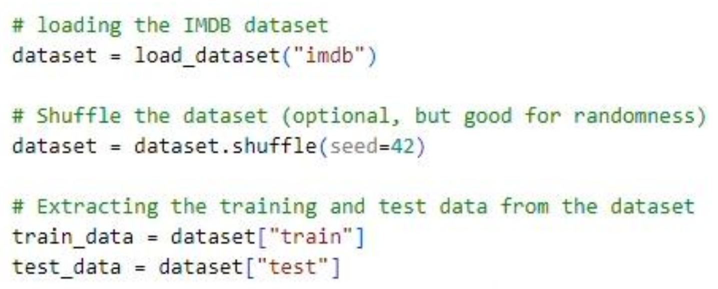
```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 128, padding_idx=0)
      (position_embeddings): Embedding(512, 128)
      (token_type_embeddings): Embedding(2, 128)
      (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-1): 2 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=128, out_features=128, bias=True)
              (key): Linear(in_features=128, out_features=128, bias=True)
              (value): Linear(in_features=128, out_features=128, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=128, out_features=128, bias=True)
              (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=128, out_features=512, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=512, out_features=128, bias=True)
            (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=128, out_features=128, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=128, out_features=2, bias=True)
)
```

Appendix Figure 1: BERT-Tiny Model Configuration



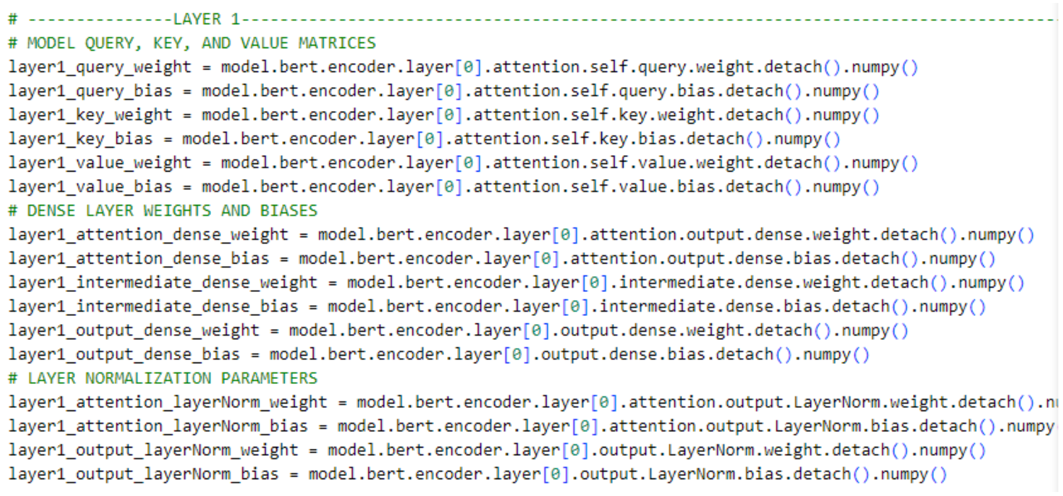
```
# setting training arguments
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=64,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    logging_steps=10,
)

# adding data collator
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# initiating trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_val,
    data_collator=data_collator,
    compute_metrics=compute_metrics
)
```

Appendix Figure 2: TrainingArguments and Trainer from HuggingFace



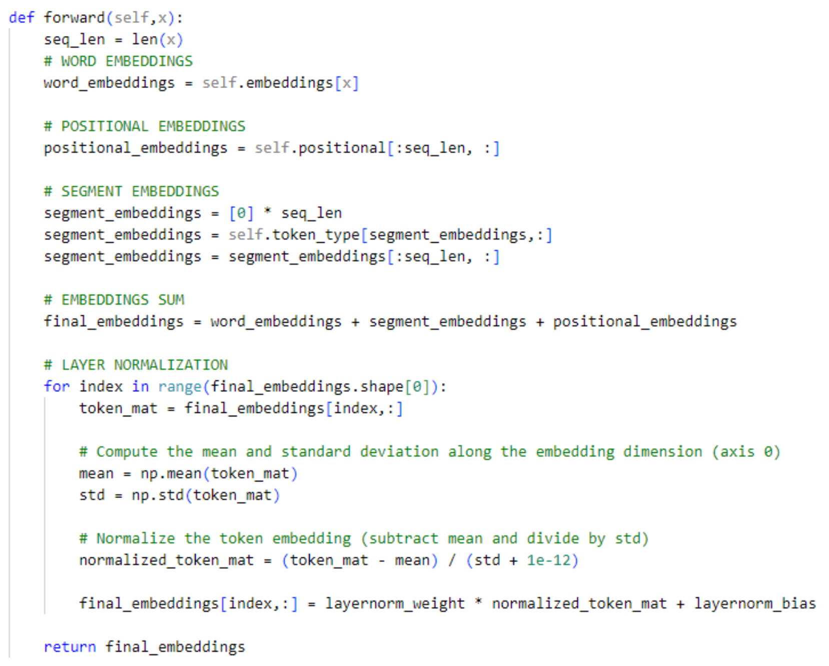
```python
# loading the IMDB dataset
dataset = load_dataset("imdb")

# Shuffle the dataset (optional, but good for randomness)
dataset = dataset.shuffle(seed=42)

# Extracting the training and test data from the dataset
train_data = dataset["train"]
test_data = dataset["test"]
```

Appendix Figure 3: Loading in the “IMDB” Dataset



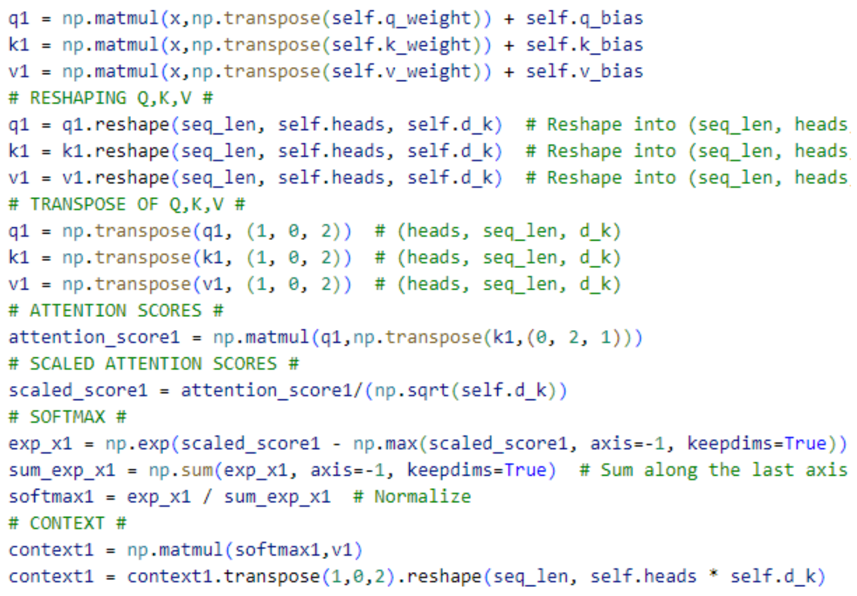
```python
# ---------------LAYER 1------------------------------------------------------------------------------------
# MODEL QUERY, KEY, AND VALUE MATRICES
layer1_query_weight = model.bert.encoder.layer[0].attention.self.query.weight.detach().numpy()
layer1_query_bias = model.bert.encoder.layer[0].attention.self.query.bias.detach().numpy()
layer1_key_weight = model.bert.encoder.layer[0].attention.self.key.weight.detach().numpy()
layer1_key_bias = model.bert.encoder.layer[0].attention.self.key.bias.detach().numpy()
layer1_value_weight = model.bert.encoder.layer[0].attention.self.value.weight.detach().numpy()
layer1_value_bias = model.bert.encoder.layer[0].attention.self.value.bias.detach().numpy()
# DENSE LAYER WEIGHTS AND BIASES
layer1_attention_dense_weight = model.bert.encoder.layer[0].attention.output.dense.weight.detach().numpy()
layer1_attention_dense_bias = model.bert.encoder.layer[0].attention.output.dense.bias.detach().numpy()
layer1_intermediate_dense_weight = model.bert.encoder.layer[0].intermediate.dense.weight.detach().numpy()
layer1_intermediate_dense_bias = model.bert.encoder.layer[0].intermediate.dense.bias.detach().numpy()
layer1_output_dense_weight = model.bert.encoder.layer[0].output.dense.weight.detach().numpy()
layer1_output_dense_bias = model.bert.encoder.layer[0].output.dense.bias.detach().numpy()
# LAYER NORMALIZATION PARAMETERS
layer1_attention_layerNorm_weight = model.bert.encoder.layer[0].attention.output.LayerNorm.weight.detach().n
layer1_attention_layerNorm_bias = model.bert.encoder.layer[0].attention.output.LayerNorm.bias.detach().numpy
layer1_output_layerNorm_weight = model.bert.encoder.layer[0].output.LayerNorm.weight.detach().numpy()
layer1_output_layerNorm_bias = model.bert.encoder.layer[0].output.LayerNorm.bias.detach().numpy()
```

Appendix Figure 4: Grabbing Weights and Biases from the Fine-Tuned Model



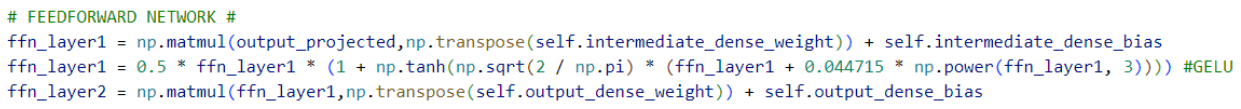
```python
def forward(self,x):
    seq_len = len(x)
    # WORD EMBEDDINGS
    word_embeddings = self.embeddings[x]

    # POSITIONAL EMBEDDINGS
    positional_embeddings = self.positional[:seq_len, :]

    # SEGMENT EMBEDDINGS
    segment_embeddings = [0] * seq_len
    segment_embeddings = self.token_type[segment_embeddings,:]
    segment_embeddings = segment_embeddings[:seq_len, :]

    # EMBEDDINGS SUM
    final_embeddings = word_embeddings + segment_embeddings + positional_embeddings

    # LAYER NORMALIZATION
    for index in range(final_embeddings.shape[0]):
        token_mat = final_embeddings[index,:]

        # Compute the mean and standard deviation along the embedding dimension (axis 0)
        mean = np.mean(token_mat)
        std = np.std(token_mat)

        # Normalize the token embedding (subtract mean and divide by std)
        normalized_token_mat = (token_mat - mean) / (std + 1e-12)

        final_embeddings[index,:] = layernorm_weight * normalized_token_mat + layernorm_bias

    return final_embeddings
```

Appendix Figure 5. Embedding Layer Implementation



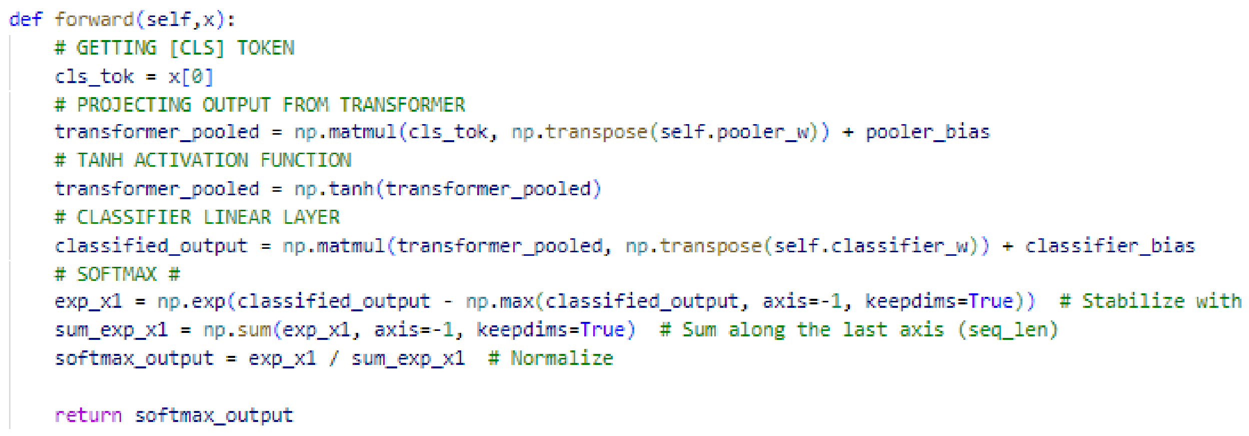
```
q1 = np.matmul(x,np.transpose(self.q_weight)) + self.q_bias
k1 = np.matmul(x,np.transpose(self.k_weight)) + self.k_bias
v1 = np.matmul(x,np.transpose(self.v_weight)) + self.v_bias
# RESHAPING Q,K,V #
q1 = q1.reshape(seq_len, self.heads, self.d_k)  # Reshape into (seq_len, heads
k1 = k1.reshape(seq_len, self.heads, self.d_k)  # Reshape into (seq_len, heads
v1 = v1.reshape(seq_len, self.heads, self.d_k)  # Reshape into (seq_len, heads
# TRANSPOSE OF Q,K,V #
q1 = np.transpose(q1, (1, 0, 2))  # (heads, seq_len, d_k)
k1 = np.transpose(k1, (1, 0, 2))  # (heads, seq_len, d_k)
v1 = np.transpose(v1, (1, 0, 2))  # (heads, seq_len, d_k)
# ATTENTION SCORES #
attention_score1 = np.matmul(q1,np.transpose(k1,(0, 2, 1)))
# SCALED ATTENTION SCORES #
scaled_score1 = attention_score1/(np.sqrt(self.d_k))
# SOFTMAX #
exp_x1 = np.exp(scaled_score1 - np.max(scaled_score1, axis=-1, keepdims=True))
sum_exp_x1 = np.sum(exp_x1, axis=-1, keepdims=True)  # Sum along the last axis
softmax1 = exp_x1 / sum_exp_x1  # Normalize
# CONTEXT #
context1 = np.matmul(softmax1,v1)
context1 = context1.transpose(1,0,2).reshape(seq_len, self.heads * self.d_k)
```

Appendix Figure 6. Attention Mechanism Implementation



```
# FEEDFORWARD NETWORK #
ffn_layer1 = np.matmul(output_projected,np.transpose(self.intermediate_dense_weight)) + self.intermediate_dense_bias
ffn_layer1 = 0.5 * ffn_layer1 * (1 + np.tanh(np.sqrt(2 / np.pi) * (ffn_layer1 + 0.044715 * np.power(ffn_layer1, 3)))) #GELU
ffn_layer2 = np.matmul(ffn_layer1,np.transpose(self.output_dense_weight)) + self.output_dense_bias
```

Appendix Figure 7. Feed-Forward Network Implementation



```
def forward(self,x):
    # GETTING [CLS] TOKEN
    cls_tok = x[0]
    # PROJECTING OUTPUT FROM TRANSFORMER
    transformer_pooled = np.matmul(cls_tok, np.transpose(self.pooler_w)) + pooler_bias
    # TANH ACTIVATION FUNCTION
    transformer_pooled = np.tanh(transformer_pooled)
    # CLASSIFIER LINEAR LAYER
    classified_output = np.matmul(transformer_pooled, np.transpose(self.classifier_w)) + classifier_bias
    # SOFTMAX #
    exp_x1 = np.exp(classified_output - np.max(classified_output, axis=-1, keepdims=True))  # Stabilize with
    sum_exp_x1 = np.sum(exp_x1, axis=-1, keepdims=True)  # Sum along the last axis (seq_len)
    softmax_output = exp_x1 / sum_exp_x1  # Normalize

    return softmax_output
```

Appendix Figure 8. Classification Head Implementation