Architecture Breakdown of a Fine-Tuned BERT Model for Sentiment Analysis

Presented By:

Zach Johnson

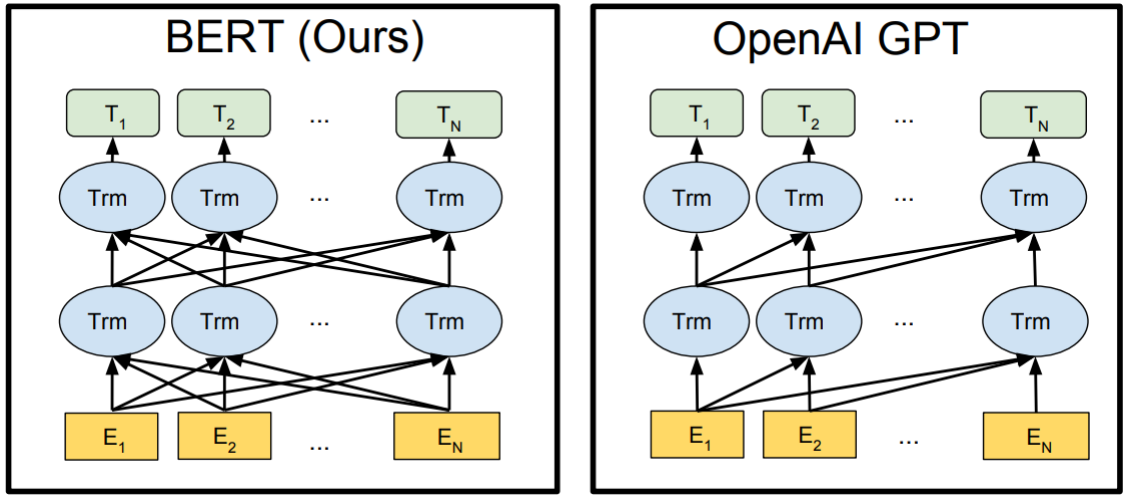
BERT-Tiny

- BERT – Bidirectional Encoder Representations from Transformers
  - Discriminative model (Natural Language Understanding)
- Bidirectional Contextualization: considers context of words before and after each word

- Built on Transformer Architecture (self-attention)

- Contextual understanding makes BERT perfect for Sentiment Analysis

- BERT-Tiny
  - 4.3 Million Parameters
  - 2 Transformer Layers



Fine-Tuning for Sentiment Analysis

- Dataset: IMDB Large Movie Review Dataset from HuggingFace
  - 50,000 highly polar movie reviews labeled positive or negative

HUGGING FACE

- Each movie review is tokenized:

```
# Tokenizing the datasets
def tokenize(batch):
    return tokenizer(batch["text"], padding="max_length", truncation=True, max_length=512)
```

- HuggingFace Transformers: TrainingArguments and Trainer

```
# 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,
)
```

```
# 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
)
```

- Train!

```
# fine tuning the tiny model
trainer.train()
```

Model Performance

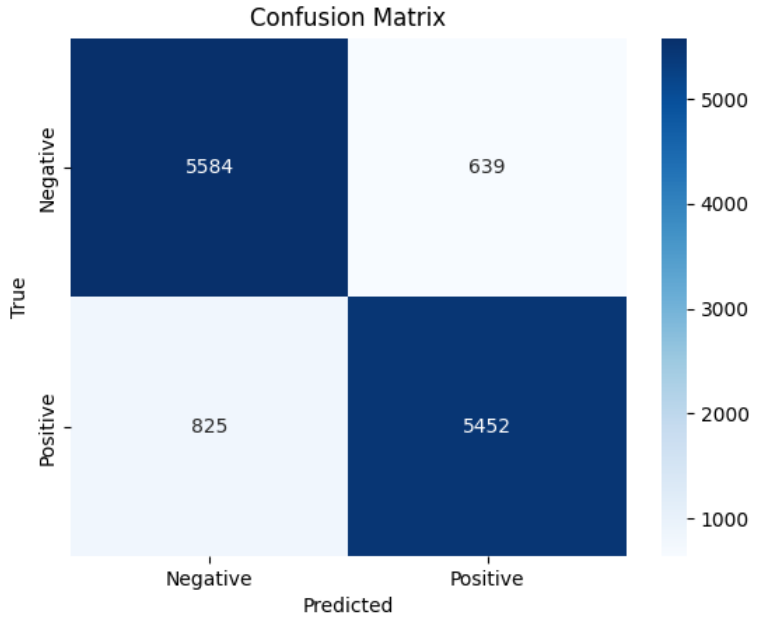
- 88% accuracy on very small BERT model (BERT-Tiny)

- Higher precision (0.895) means positive predictions are usually correct
- Lower recall (0.868) indicates more positive reviews are missed

- The model is more conservative when predicting positive
  - Positive reviews that are ambiguous will likely be misclassified negative

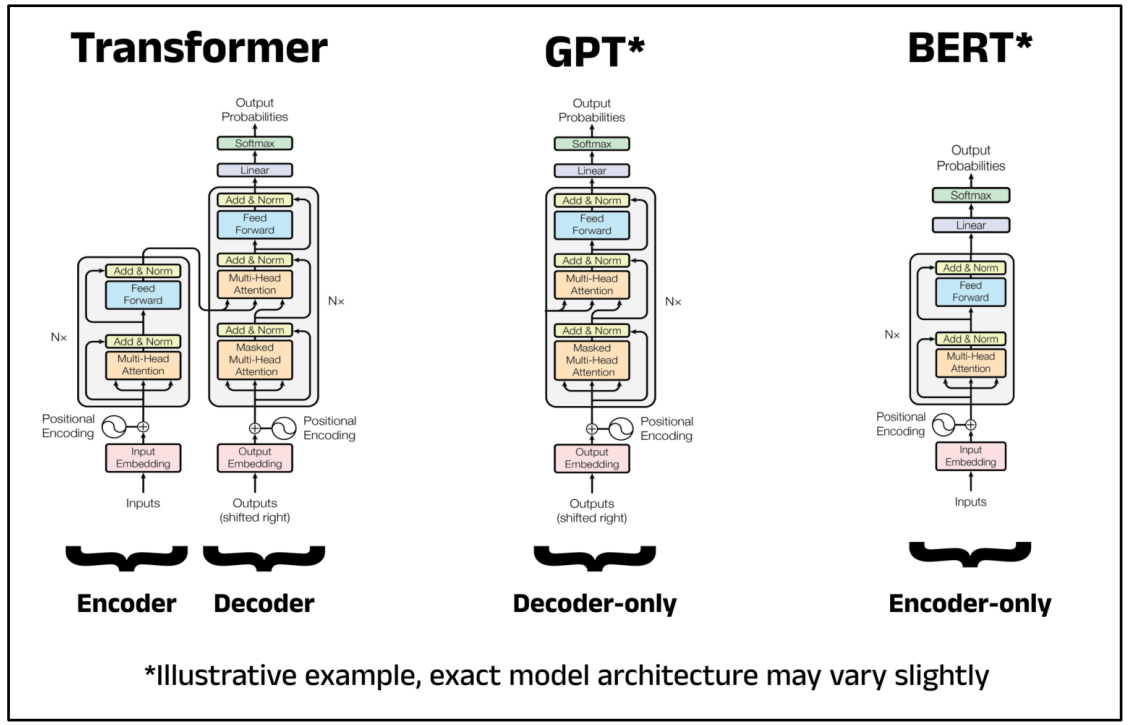
- F1 score indicates strong and balanced performance with:
  - Identifying positive sentiment
  - Minimizing missed positives

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



BERT Architecture

- BERT uses an Encoder-only Transformer Architecture
- Specifically for BERT-Tiny:
  - Input Embedding Layer
  - Two Transformer Layers
  - Classification Head
- Fine-Tuned Weights and Biases for each layer can be accessed through HuggingFace Transformer Model:
- Goal is to design model entirely from scratch using only matrix multiplication!




```
bert.embeddings.word_embeddings
bert.embeddings.position_embeddings
bert.embeddings.token_type_embeddings
bert.embeddings.LayerNorm
```

```
bert.encoder.layer.0.attention.self.query
bert.encoder.layer.0.attention.self.key
bert.encoder.layer.0.attention.self.value
bert.encoder.layer.0.attention.self.dropout
bert.encoder.layer.0.attention.output
bert.encoder.layer.0.attention.output.dense
bert.encoder.layer.0.attention.output.LayerNorm
bert.encoder.layer.0.attention.output.dropout
bert.encoder.layer.0.intermediate
bert.encoder.layer.0.intermediate.dense
bert.encoder.layer.0.intermediate.intermediate_act_fn
bert.encoder.layer.0.output
bert.encoder.layer.0.output.dense
bert.encoder.layer.0.output.LayerNorm
bert.encoder.layer.0.output.dropout
```

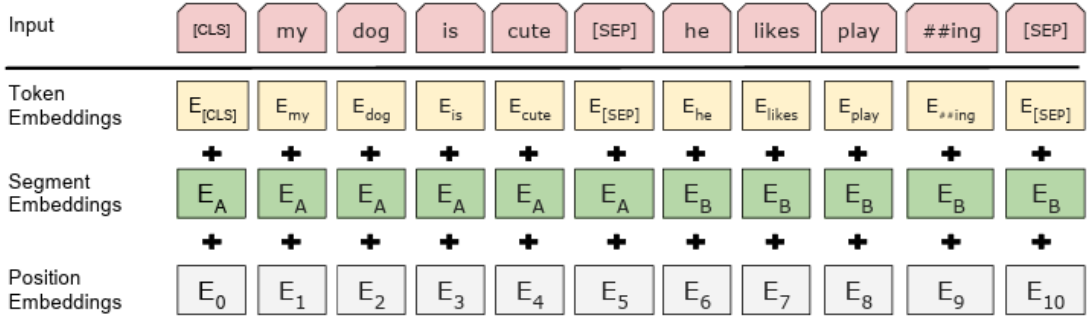
```
bert.pooler
bert.pooler.dense
bert.pooler.activation
dropout
classifier
```

Embedding Layer

- Converts input text to vector representations that the model can understand

- Considers three embeddings:
  - **Token Embeddings** (unique vector for every token in vocabulary)
  - **Segment Embeddings** (separates multiple sentences for applications like question/answering)
  - **Position Embeddings** (indicates to the model the position of each token in the sequence)

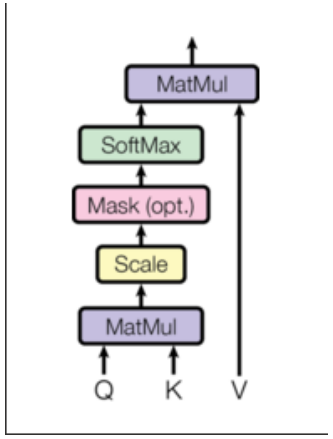
- Each embedding is summed together and normalized




```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
```

Transformer Layer: Self-Attention

- Query (Q) – Represents what a specific token is trying to find
- Key (K) – Carries information for a specific token that is compared against the Query
- Value (V) – If token is considered relevant, Value information is passed



1. Compute Query (Q), Key (K), and Values (V) with the Input (X)
   - $Q = X \cdot W_Q + b_Q$
   - $K = X \cdot W_K + b_K$
   - $V = X \cdot W_V + b_V$
2. Calculate Attention Scores between tokens i and j
   - $Attention\ Score_{ij} = Q_i \cdot K_j^T$ (relevance of token j on token i)
3. Scale the Attention Scores
   - $Scaled\ Score_{ij} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}}$, where dk is the dimension of K
4. Apply SoftMax
   - Attention Weights = softmax(Scaled Score)
5. Weight the Values (Context)
   - $Context_i = Attention\ Weights \cdot V_j$

```
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)
```

Transformer Layer: Feed-Forward Network

- Feed-Forward Network processes calculated attention and models more complicated patterns for each token

- FFN consists of:
  - Intermediate Dense Layer

  - GELU activation function
    - Gaussian Error Linear Unit – smoothly pushes toward 0 for small/negative inputs and makes output close to linear for positive inputs. (not hard-cutting like poslin/ReLU)
    - Nonlinearizes attention input that is inherently linear, allowing for more sophisticated decision boundaries

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

  - Output Dense Layer

```
# 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
```

Classification Head

- Consists of:
  - Pooler
    - Summarizes entire sequence into [CLS] token
  - Tanh activation function
    - Puts output from pooler in range [-1, 1]
  - Classifier Dense Layer
    - Outputs unnormalized scores for each class (2)
  - Softmax
    - Puts output in terms of probability each class is correct

```
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
```

Why?

- We now have BERT-Tiny designed entirely using only Matrix Multiplication/Addition using fine-tuned weights and biases.

- BERT-Tiny is very small – only around 4.3 million parameters
  - Assuming each parameter is stored as FP32 (32-bit floating point)
  - This means around 16.4 MB of memory is required to store all parameters
  - Can be reduced even further with quantization

- Enables a lightweight, application-specific language model for efficient digital IC deployment!