Implementing Transformer Architecture for Dialogue Summarization

Berkay Altıntaş
Department of Electrical and Electronics Engineering
Bilkent University
berkay.altintas@ug.bilkent.edu.tr

PART A: Theory and Implementation

1 Introduction

Recurrent Neural Networks (RNN) [1], Long Short Term Memory (LSTM) [2], and Gated Recurrent Unit (GRU) [3] are the models designed to process sequences by processing data step by step. Although the LSTMs and GRUs improve memory over long sequences using gating mechanisms, they still struggle with long-range dependencies and slow training owing to their sequential processing. Transformers [4] overcome these limitations by using the attention mechanism to look at all words at once, allowing faster training and better understanding of context, especially for long texts.

2 Transformer Architecture

The Transformer architecture consists of an encoder and decoder. In general, while the encoder part of the transformer takes the input sequence and generates a representation of it, the decoder generates the output sequence translated based on the encoder's representation.

- **Encoder:** The encoder consists of N = 6 identical layers, each with two parts: a multi-head self-attention mechanism and a feed-forward network. Residual connections and layer normalization are applied after each part. All layers produce outputs of the same size, helping the model learn contextual representations of the input.
- **Decoder:** The decoder also has N = 6 layers. Each layer includes the same two parts as the encoder and an attention layer that focuses on the encoder's output. It uses this information to generate the output sentence, one word at a time. Moreover, it uses masking to ensure that predictions only depend on previous words, not future ones.

2.1 Tokenization

In this work, firstly, the unnecessary symbols from the text, excluding [ ] which indicated the start of the sentence (SOS) and end of the sentence (EOS), are removed. Then, if a word in the remaining part is not the part of the vocabulary, it is replaced with [UNK]. After that the tokenizer looks at the text from the both documents and their summaries to learn which words are used. Thus, it represents each word with number in such a way that the model can understand and process it.

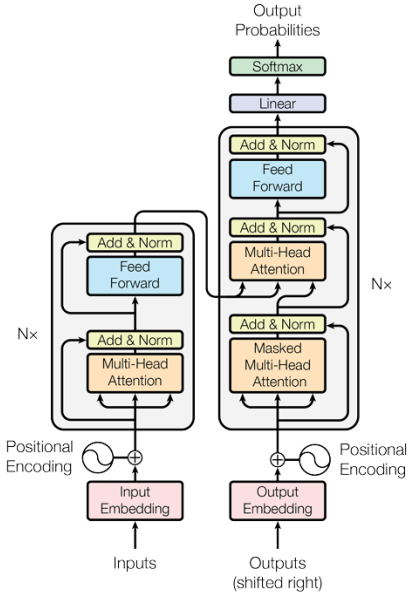

Figure 1: Transformer Architecture [4]

2.2 Embeddings and Positional Encoding

After the tokenization, in the embeddings part, each token is converted into a vector of dimension d_{model}. In the original implementation, $d_{\text{model}} = 512$. Thus, each word is represented by a dense vector of length 512.

Since the transformer architecture has no recurrence or convolution operation in it, it does not directly keep the order of the words or sequences. Thus, in order to add the relative or absolute position of the tokens, the positional encoding is used to input embeddings at the bottoms of the encoder and decoder blocks. In other words, position embeddings of size d_{model} and embeddings of size

d_{model} are addded together and become the encoder input of size d_{model}. Although there are different ways of positional encodings, the following sine and cosine functions of different frequencies are used in the original transformer implementation,

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where pos is the position and i is the dimension. These positional embeddings are computed only once and reused during training and inference.

2.3 Masking

In this work, two types of mask, which help the model to focus on the relevant parts of the sequence and ensure the proper structure of the output, are used: the padding mask and the look-ahead mask.

- **Padding Mask:** This type of mask is used to make all sequences the same length to handle them easily. Thus, it marks these positions in such a way that the model ignores them during attention calculations.
- **Look-ahead Mask:** This type of mask ensures that the model does not investigate unseen future tokens. It does this by creating a lower triangular matrix by allowing each position in the output depending on the current and previous positions.

2.4 Attention Mechanism

In the attention mechanism, the input is represented by three vectors: queries and keys of dimension d_{k} and values of dimension d_{v}. These vectors help the model to relate words to each other.

The attention function on a set of queries are computed simultaneously which groups them into a matrix Q. Moreover, the keys and values are grouped as matrices K and V respectively. Therefore, the attention score is calculated like as follows,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This formula compares the query with each key using a dot product to calculate the similarity score. These similarity scores are scaled by the squared root of the d_{k} to prevent extremely large values. After that, a softmax function is applied to convert the similarity scores into weights which are used to compute a weighted sum of the value vectors. This final result represents the attention output for that word by summarizing what it should focus on from the entire sentence.

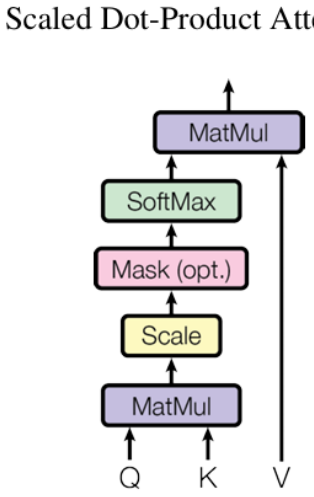

Figure 2: Scaled Dot-Product Attention [4]

3 Encoder Block

Before entering the encoder block of the Tranformer, tokenized inputs are passed through the embedding layer where each token converted into a dense vector representations of size d_{model}. Then as explained previously, to keep the information about token positions, the positional encodings are added to input embeddings. Thus, the input of the encoder becomes the following,

$$\text{Input} = \text{Embedding}(x) + \text{PE}$$

Structure

The encoder is composed of a stack of $N = 6$ identical layers in the original implementation. Each layer consists of two main sub-layers:

1. **Multi-Head Self-Attention Mechanism**
2. **Position-Wise Feed-Forward Network**

Each sub-layer is followed by:

- A **residual connection**
- **Layer normalization**

The output of each sub-layer is defined as:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

where x is the input to the sub-layer, and $\text{Sublayer}(x)$ is the output of either the self-attention mechanism or the feed-forward network. This residual structure helps with gradient flow and stabilizes deep networks.

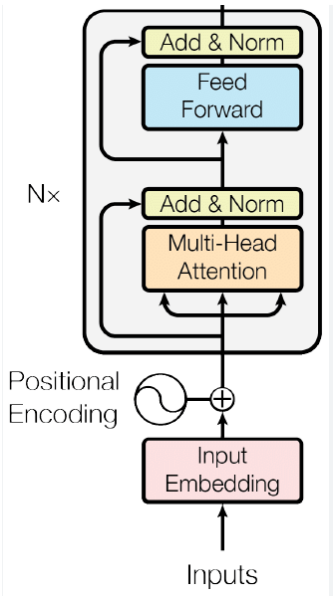

Figure 3: Encoder Block of the Transformer Architecture[5]

Multi-Head Self-Attention

The core of each encoder layer is the multi-head self-attention mechanism, which allows the model to compute relationships between all pairs of tokens in the input.

For a single head, attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Where:

- $Q = XW^Q$,
- $K = XW^K$,
- $V = XW^V$, and $X \in \mathbb{R}^{n \times d_{\text{model}}}$ is the input to the layer.

However, instead of calculating a single attention function, it is more effective to apply multiple linear projections to the queries, keys and value in parallel. For example, they can be projected h times into subspaces of d_{k}, d_{k} and d_{v} respectively to capture the information represented in the subspaces. Thus, the attention calculation becomes a multi-head attention calculation with h heads like as follows,

$$\begin{aligned}
\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\
\text{where} \quad \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \\
\text{and} \quad W_i^Q &\in \mathbb{R}^{d_{\text{model}} \times d_k} \\
W_i^K &\in \mathbb{R}^{d_{\text{model}} \times d_k} \\
W_i^V &\in \mathbb{R}^{d_{\text{model}} \times d_v} \\
W^O &\in \mathbb{R}^{hd_v \times d_{\text{model}}}
\end{aligned}$$

Multi-head attention mechanism serves several advantages. Since it has multiple heads, different heads may have the opportunity to attend the different aspects of the input. Moreover, the use of parallel processing improves training efficiency and helps the model learn dependencies more effectively while reducing the risk of overfitting.

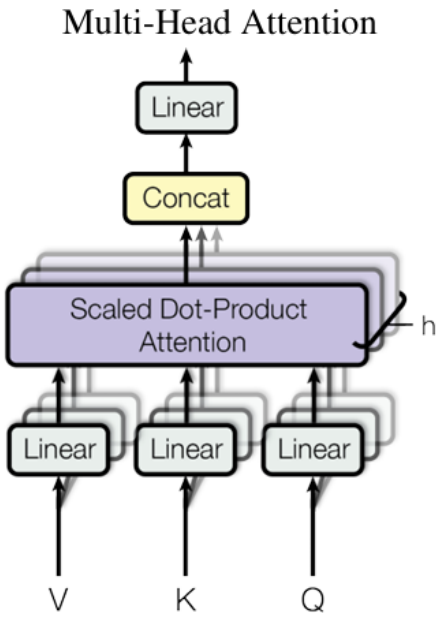

Figure 4: Multi-Head Attention [4]

Feed-Forward Network (FFN)

After the attention sub-layer, a fully connected feed-forward network is applied to each position independently. Moreover, this consists of transformations with a ReLU activativation function in between

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Typically:

- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$
- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ with d_{ff} usually larger than d_{model}

Output of the Encoder

After passing through all N encoder layers, the output is a set of contextualized representations for each input token. These are passed to the decoder during the encoder-decoder attention phase.

Implementation of the Encoder Block

The implementation of the scaled dot product attention can be seen in the Listing 1. The obtained results are as follows,

$$\text{Output:} \quad \begin{bmatrix} 0.5 & 0.75 \\ 0.31 & 0.84 \\ 0.27 & 1.0 \end{bmatrix}$$

$$\text{Attention Weights:} \quad \begin{bmatrix} 0.25 & 0.25 & 0.0 & 0.25 & 0.25 \\ 0.43 & 0.0 & 0.16 & 0.16 & 0.26 \\ 0.45 & 0.0 & 0.0 & 0.27 & 0.27 \end{bmatrix}$$

The implementation of the encoder layer and encoder classes can be seen in Listing 2 and Listing 3 respectively. In this work, unlike the original Transformer implementation, the following parameters are used.

Component	Implementation
Embedding dimension d_{model}	16
Number of heads h	4
Feed-forward dimension d_{ff}	32
Number of layers N	2
Vocabulary size	500
Max position length	20

Table 1: Transformer encoder configuration used in the implementation.

Moreover, the tensor shapes are as follows,

- **Input shape:** $(1, 10)$
- **Mask shape:** $(1, 1, 1, 10)$
- **Output of encoder has shape:** $(1, 10, 16)$

4 Decoder Block

Before entering the decoder block of the Transformer, like the encoder block, there are token embeddings and positional encoding stages. Thus, the input of the decoder block becomes.

$$\text{Input} = \text{Embedding}(y) + \text{PE}$$

Structure

The decoder consists of a stack of $N = 6$ identical layers in the original implementation. Each layer consists of three main sub-layers:

1. **Multi-Head Self-Attention Mechanism**
2. **Multi-Head Encoder-Decoder Attention**
3. **Position-Wise Feed-Forward Network**

Each sub-layer is followed by:

- A **residual connection**
- **Layer normalization**

The output of each sub-layer is defined as:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Multi-Head Attention

The first attention block in each decoder layer is a masked multi-head self-attention mechanism. It allows each position in the output sequence to attend only to previous positions . This is essential during training to maintain the autoregressive property of the decoder.

To enforce this, a look-ahead mask is applied that blocks attention to future tokens. This prevents the decoder from accessing tokens it should not yet have generated.

For a single attention head, masked self-attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + \text{mask}\right)V$$

Where:

- $Q = YW^Q$,
- $K = YW^K$,
- $V = YW^V$, and $Y \in \mathbb{R}^{n \times d_{\text{model}}}$ is the decoder input embedding matrix.

Rather than using a single head, the decoder employs multiple parallel attention heads. Each head projects the queries, keys, and values into separate subspaces of dimensions d_k, d_k, and d_v, respectively:

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{where} \quad \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \\ W_i^Q &\in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \\ W_i^V &\in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}} \end{aligned}$$

This allows each head to learn different patterns or relationships in the sequence, enhancing the model's representational power.

Multi-Head Encoder-Decoder Attention

The second attention block in the decoder is the encoder-decoder attention mechanism. This layer allows the decoder to attend to the encoder's output and focus on relevant parts of the input sequence.

- Queries Q come from the output of the previous decoder sub-layer,
- Keys K and values V come from the encoder output.

There is *no masking* applied here, since the encoder output is fully available.

The attention is computed the same way as before, but using separate sources for Q, K, and V:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

And the multi-head version:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where} \quad \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

This cross-attention mechanism allows the decoder to generate output that is conditioned on the input sequence, enabling meaningful sequence-to-sequence modeling.

Feed-Forward Network (FFN)

Same procedure holds as in the encoder block.

Output of the Decoder

After all N decoder layers, the output is passed to a final linear layer followed by a softmax:

$$\hat{y}_t = \text{softmax}(z_t W^T + b)$$

Where:

- z_t is the decoder output at position t,
- W is the weight matrix shared with the embedding layer (optional, but often used).

Implementation of the Decoder Block

The implementation of the decoder layer and encoder classes can be seen in Listing 4 and Listing 5 respectively. In this work, unlike the original Transformer implementation, the following parameters are used.

Component	Implementation
Embedding dimension d_{model}	15
Number of heads h	19
Feed-forward dimension d_{ff}	16
Number of layers N	7
Vocabulary size	300
Max position length	6

Table 2: Decoder configuration used in the implementation.

Moreover, the tensor shapes are as follows,

- **Input x shape:** $(3, 4)$
- **Output of encoder:** $(3, 7, 9)$
- **Output of decoder:** $(3, 4, 15)$

Attention Weights

- `decoder_layer1_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer1_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer2_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer2_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer3_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer3_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer4_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer4_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer5_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer5_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer6_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer6_block2_decenc_att`: $(3, 19, 4, 7)$
- `decoder_layer7_block1_self_att`: $(3, 19, 4, 4)$
- `decoder_layer7_block2_decenc_att`: $(3, 19, 4, 7)$

Implementation of Transformer Class

The Transformer class in the code defines a model that takes an input sentence and generates a corresponding output sentence. It is made up of two main parts: an encoder and a decoder. The encoder processes the input sentence and creates a meaningful internal representation of it. This representation is then passed to the decoder which uses it along with the already generated part of the output sentence to predict the next word. During this process, the model uses special masks to ignore unnecessary padding and to make sure it does not look ahead at future words while generating the output. After the decoder completes its work, the result is passed through a final layer that helps choose the most likely words for the output. This setup allows the model to learn how to generate accurate and meaninful sentences from training examples. The total transformer implementation can be seen in Listing 6.

Component	Implementation
Number of layers N	7
Embedding dimension d_{model}	13
Number of heads h	19
Feed-forward dimension d_{ff}	8
Input vocabulary size	300
Target vocabulary size	350
Max positional encoding (input)	12
Max positional encoding (target)	12

Table 3: Transformer configuration used in the implementation.

Input and Output Shapes

- **Input sentence_a shape:** $(1, 6)$
- **Input sentence_b shape:** $(1, 6)$
- **Output of Transformer (summary):** $(1, 6, 350)$

Attention Weights

- `decoder_layer1_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer1_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer2_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer2_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer3_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer3_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer4_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer4_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer5_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer5_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer6_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer6_block2_decenc_att`: $(1, 19, 6, 6)$
- `decoder_layer7_block1_self_att`: $(1, 19, 6, 6)$
- `decoder_layer7_block2_decenc_att`: $(1, 19, 6, 6)$

PART B: Model Training, Results and Discussion

1 Model Training

The Transformer-based summarization model takes a dialogue as input, tokenizes it, and converts it into embeddings with positional information provided from the positional embedding to keep the word order. The encoder processes this sequence to capture contextual meaning, while the decoder generates the summary starting from a special start token ([SOS]). At each step, it predicts the next word by attending to the encoder's output and its past predictions. During training, the model uses the actual previous words and learns by minimizing prediction errors. In inference, it generates the summary word by word until it outputs the end token ([EOS]). This model was trained for 300 epochs and the training loss curve can be seen below.

As it can be seen in the figure above, the training loss starts to decrease from 7.5 to 0.19 which is a indicator of learning.

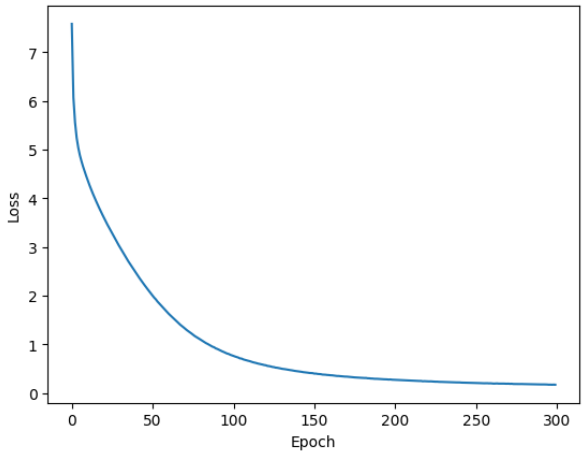

Figure 5: Training Loss Curve

2 Sample Summarization Results

Training Set Example

[SOS] #person1#: i'm tired of watching television. let's go to cinema tonight. #person2#: all right. do you want to go downtown? or is there a good movie in the neighborhood? #person1#: i'd rather not spend a lot of money. what does the paper say about neighborhood theaters? #person2#: here's the list on page... column 6. here it is. where's the rialto? there's a perfect movie there. #person1#: that's too far away. and it's hard to find a place to park there. #person2#: well, the grand theater has gone with the wind. #person1#: i saw that years ago. i couldn't wait to see it again. moreover, it's too long. we wouldn't get home until midnight. #person2#: the center has a horror film. you wouldn't want to see that? #person1#: no, indeed. i wouldn't be able to sleep tonight. #person2#: that's about all there is. unless we change our decision and go downtown. #person1#: no, we just can't pay for it. there must be something else we haven't seen. #person2#: here, look for yourself, i can't find anything else. #person1#: look at this! #person2#: what? #person1#: in the television timetable, there's a baseball game on television tonight. #person2#: i wasn't looking for a tv program. i was looking at the movie ads. #person1#: i know, but i just happened to notice it. new york is playing boston. #person2#: that must be good. i wouldn't mind watching that. #person1#: ok. let's stay home. we can go to the cinema friday. [EOS]

Human-Written Summary

[SOS] #person1#'s tired of watching television, so #person1# and #person2# search on the paper to choose a movie to watch. but they don't decide a suitable one. #person1# finds there will be a baseball game tonight, so they decide to stay at home. [EOS]

Model-Written Summary

[SOS] person1 's tired of watching television so person1 and person2 search on the paper to choose a movie to watch but they don't decide a suitable one person1 finds there will be a baseball game tonight so they decide to stay at home [EOS]

Since the transformer-based summarization model was trained for 300 epochs, it reaches relatively small loss value which helps model to predict better. This can be proved by comparing the human-written summary and model summary regarding the training set example. By comparing them, it can be observe that the model prediction uses the almost the same words as human-written summary although it does not capture the punctuations as did in the human-written summary.

Howwever, the actual performance of the model is evaluated on the test data. Thus, the following blocks can be investigated to understand how well the trained transformer-based summarization model performs in unseen data. As it can be seen in the following blocks, the model summary misses most of the details that human-being recognize and capture. Getting such a result unlike the training case may be a indicator of overfitting.

Test Set Example

[SOS] #person1#: there's a car waiting for you just outside the door. right this way, please. #person2#: ok! #person1#: let me put your cases into the trunk and please get in the back. #person2#: thanks! #person1#: how was your flight? #person2#: it's comfortable, but now i'm a little tired. #person1#: we'll reach the beijing hotel in another ten minutes. when we arrived there, you can go up and have a rest. the hotel has very good service, and it's considered as one of the best hotels here. #person2#: thank you! i lived there when i came to beijing last time. it's comfortable and beautiful. #person1#: if it's convenient for you, mr. wu would like to invite you to the banquet in honor of you in the evening. #person2#: thank you! i will. when and where will the dinner be? #person1#: at six o'clock in the international hotel. we'll pick you up this afternoon. besides, if you care for visiting, we'll arrange some sightseeing for you. #person2#: oh, that's nice. thank you for arranging all of this. [EOS]

Human-Written Summary

[SOS] #person1# is driving #person2# to the beijing hotel. #person2# will attend a banquet at six o'clock in the international hotel. #person1# warmly welcomes #person2# and drives #person2# to the beijing hotel. mr. wu has arranged a banquet for #person2# in the evening. #person1# has arranged everything for #person2# after #person2# arrives in beijing and invites #person2# to a banquet. [EOS]

Model-Written Summary

[SOS] mr recommends person2 to go into the park because of the 15 moves person2 tells him the additional suitcase will be a great hotel for dinner and gives person2 some time [EOS]

3 BERTScore on Test and Training Set

BERTScore is a metric for evaluating text generations task like summarization or translation based on semantic similarity. It uses pre-trained language model to embed the predicted and ground truth sentences and calculates the precision, recall and F1 metrics. In this work, the pre-trained model is chosen as 'roberta-large'.

3.1 Precision

Precision measures how much of the information in the generated summary (the candidate) is relevant.

In BERTScore, this is done by comparing each token in the generated summary to the most similar token in the reference summary using contextual embeddings.

Let $C = \{c_1, c_2, \ldots, c_m\}$ be the tokens of the generated summary, and let $R = \{r_1, r_2, \ldots, r_n\}$ be the tokens of the human summary. Let $\text{sim}(c_i, r_j)$ be the cosine similarity between the contextual embeddings of tokens c_i and r_j.

$$\text{cosine\_similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}$$

$$\text{Precision} = \frac{1}{|C|} \sum_{c_i \in C} \max_{r_j \in R} \text{sim}(c_i, r_j)$$

This computes the average similarity between each generated token and the most similar token in the reference.

3.2 Recall

Recall measures how much of the information in the reference summary is captured by the generated summary.

This is done by comparing each token in the reference to the most similar token in the candidate.

$$\text{Recall} = \frac{1}{|R|} \sum_{r_j \in R} \max_{c_i \in C} \text{sim}(r_j, c_i)$$

This is the average similarity of each reference token to its best-matching token in the generated summary.

3.3 F1 Score

The F1 Score combines Precision and Recall into a single formula.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

This is the harmonic mean, which favors models that perform well on both Precision and Recall rather than just one.

4 BERTScore Results

The obtained BERTScore results for test and training can be seen in the tables below.

Metric	**Score**
Average Precision	0.8747
Average Recall	0.8427
Average F1 Score	0.8583

Table 4: BERTScore Results on the Test Set

Metric	**Score**
Average Precision	0.9681
Average Recall	0.9489
Average F1 Score	0.9584

Table 5: BERTScore Results on the Training Set

By looking at the tables above, it can be seen that the expected results are obtained. Remember in the model summary for the test data, the model misses some of the details compared to human being. However, for the training data, the model summary is almost same with the human-written summary which indicates that the model learns well in the training stage. As a result, getting lower values for the metrics in the test data is expected regarding the their performances.

5 Discussion and Conclusion

In this work, we implemented a transformer-based summarization model step by step starting from the tokenization. After the whole implementation, the model is trained with documents which are the trainin data of the model. The model was trained for 300 epochs and evaluated at the training data. In this evaluation, the average precision was 0.9681, the average recall was 0.9489 and average F1 score was 0.9584 which indicate the good learning performances of the model. This results can be validated by comparing the human summary and model summary. However, it was not the case in the test data which was previously unseen. In this evaluation, it can be observed that there was a decrease in scores. To be more precise, in the test data, the average precision was 0.8747, average recall was 0.8427 and the average F1 Score was 0.8583 which might be a signal of overfitting. Regarding these results, it can be said that the model still can be improved with different ways. One of the ways might be to use more harsh regularization methods to avoid overfitting. As a second way, the early stopping method can be used to avoid overfitting. Thirdly, the pre-trained model for the BERTScore can be changed. In other words, instead of using 'roberta-large', any other pre-trained model could be used. Besides these, the size and diversity of the training dataset can be increased and developed to help the model to learn different patterns and contexts. As a result, it can be said that the trained transformer-based summarization model performs well but there is a still room for the improvement and this model can be improved regarding the methods mentioned above.

References

[1] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. EMNLP*, pp. 1724–1734, 2014.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, pp. 5998–6008, 2017.

[5] P. Nguyen, "The Transformer encoder structure," *ResearchGate*, [Online]. Available: `https://www.researchgate.net/figure/The-Transformer-encoder-structure_fig1_334288604`

Appendix

The implementations of the missing block can be seen in this part.

```python
def scaled_dot_product_attention(q, k, v, mask):
    """
    Calculate the attention weights.
      q, k, v must have matching leading dimensions.
      k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
      The mask has different shapes depending on its type (padding or look ahead)
      but it must be broadcastable for addition.

    Arguments:
        q (tf.Tensor): query of shape (..., seq_len_q, depth)
        k (tf.Tensor): key of shape (..., seq_len_k, depth)
        v (tf.Tensor): value of shape (..., seq_len_v, depth_v)
        mask (tf.Tensor): mask with shape broadcastable
              to (..., seq_len_q, seq_len_k)

    Returns:
        output -- attention_weights
    """
    # Multiply q and k transposed
    matmul_qk = tf.matmul(q, k, transpose_b=True)

    # Scale matmul_qk with the square root of dk
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

    # Add the mask to the scaled tensor
    if mask is not None:
        scaled_attention_logits += (1 - mask) * -1e9

    # Apply softmax to get attention weights
    attention_weights = tf.keras.activations.softmax(scaled_attention_logits, axis=-1)

    # Multiply the attention weights by v
    output = tf.matmul(attention_weights, v)

    return output, attention_weights
```

Listing 1: Scaled Dot-Product Attention Function

```python
class EncoderLayer(tf.keras.layers.Layer):
    """
    The encoder layer is composed by a multi-head self-attention mechanism,
    followed by a simple, positionwise fully connected feed-forward network.
    This architecture includes a residual connection around each of the two
    sub-layers, followed by layer normalization.
    """
    def __init__(self, embedding_dim, num_heads, fully_connected_dim,
                 dropout_rate=0.1, layernorm_eps=1e-6):

        super(EncoderLayer, self).__init__()

        self.mha = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embedding_dim,
            dropout=dropout_rate
        )

        self.ffn = FullyConnected(
            embedding_dim=embedding_dim,
```

```
            fully_connected_dim=fully_connected_dim
        )

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)

        self.dropout_ffn = tf.keras.layers.Dropout(dropout_rate)

    def call(self, x, training, mask):
        """
        Forward pass for the Encoder Layer
        """

        self_mha_output = self.mha(x, x, x, mask)

        skip_x_attention = self.layernorm1(self_mha_output + x)

        ffn_output = self.ffn(skip_x_attention)

        ffn_output = self.dropout_ffn(ffn_output, training=training)

        encoder_layer_out = self.layernorm2(ffn_output + skip_x_attention)

        return encoder_layer_out
```

Listing 2: Encoder Layer Implementation

```
class Encoder(tf.keras.layers.Layer):
    """
    The entire Encoder starts by passing the input to an embedding layer
    and using positional encoding to then pass the output through a stack of
    encoder layers.
    """
    def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim,
                 input_vocab_size, maximum_position_encoding,
                 dropout_rate=0.1, layernorm_eps=1e-6):
        super(Encoder, self).__init__()

        self.embedding_dim = embedding_dim
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(input_vocab_size, self.embedding_dim)
        self.pos_encoding = positional_encoding(maximum_position_encoding, self.
    embedding_dim)

        self.enc_layers = [
            EncoderLayer(embedding_dim=self.embedding_dim,
                         num_heads=num_heads,
                         fully_connected_dim=fully_connected_dim,
                         dropout_rate=dropout_rate,
                         layernorm_eps=layernorm_eps)
            for _ in range(self.num_layers)
        ]

        self.dropout = tf.keras.layers.Dropout(dropout_rate)

    def call(self, x, training, mask):
        """
        Forward pass for the Encoder
        """
        seq_len = tf.shape(x)[1]

```

```
        x = self.embedding(x)
        x *= tf.math.sqrt(tf.cast(self.embedding_dim, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
        x = self.dropout(x, training=training)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x, training=training, mask=mask)

        return x
```

Listing 3: Encoder Class Implementation

```

class DecoderLayer(tf.keras.layers.Layer):
    """
    The decoder layer is composed by two multi-head attention blocks,
    one that takes the new input and uses self-attention, and the other
    one that combines it with the output of the encoder, followed by a
    fully connected block.
    """
    def __init__(self, embedding_dim, num_heads, fully_connected_dim, dropout_rate=0.1,
    layernorm_eps=1e-6):
        super(DecoderLayer, self).__init__()

        self.mha1 = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embedding_dim,
            dropout=dropout_rate
        )

        self.mha2 = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embedding_dim,
            dropout=dropout_rate
        )

        self.ffn = FullyConnected(
            embedding_dim=embedding_dim,
            fully_connected_dim=fully_connected_dim
        )

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
        self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)

        self.dropout_ffn = tf.keras.layers.Dropout(dropout_rate)

    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        """
        Forward pass for the Decoder Layer
        """
        # BLOCK 1: Masked self-attention
        mult_attn_out1, attn_weights_block1 = self.mha1(
            query=x, key=x, value=x, attention_mask=look_ahead_mask,
    return_attention_scores=True
        )
        Q1 = self.layernorm1(mult_attn_out1 + x)

        # BLOCK 2: Encoder-decoder attention
        mult_attn_out2, attn_weights_block2 = self.mha2(
            query=Q1, key=enc_output, value=enc_output, attention_mask=padding_mask,
    return_attention_scores=True
```

```
        )
        mult_attn_out2 = self.layernorm2(mult_attn_out2 + Q1)

        # BLOCK 3: Feed-forward network
        ffn_output = self.ffn(mult_attn_out2)
        ffn_output = self.dropout_ffn(ffn_output, training=training)
        out3 = self.layernorm2(ffn_output + mult_attn_out2)

        return out3, attn_weights_block1, attn_weights_block2
```

Listing 4: Decoder Layer Implementation

```
class Decoder(tf.keras.layers.Layer):
    """
    The entire Decoder starts by passing the target input to an embedding layer
    and using positional encoding to then pass the output through a stack of
    decoder layers.
    """
    def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim,
                 target_vocab_size, maximum_position_encoding,
                 dropout_rate=0.1, layernorm_eps=1e-6):
        super(Decoder, self).__init__()

        self.embedding_dim = embedding_dim
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(target_vocab_size, self.embedding_dim)
        self.pos_encoding = positional_encoding(maximum_position_encoding, self.
    embedding_dim)

        self.dec_layers = [
            DecoderLayer(embedding_dim=self.embedding_dim,
                         num_heads=num_heads,
                         fully_connected_dim=fully_connected_dim,
                         dropout_rate=dropout_rate,
                         layernorm_eps=layernorm_eps)
            for _ in range(self.num_layers)
        ]
        self.dropout = tf.keras.layers.Dropout(dropout_rate)

    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        """
        Forward pass for the Decoder
        """
        seq_len = tf.shape(x)[1]
        attention_weights = {}

        x = self.embedding(x)
        x *= tf.math.sqrt(tf.cast(self.embedding_dim, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
        x = self.dropout(x, training=training)

        for i in range(self.num_layers):
            x, block1, block2 = self.dec_layers[i](
                x, enc_output, training=training,
                look_ahead_mask=look_ahead_mask,
                padding_mask=padding_mask
            )
            attention_weights[f'decoder_layer{i+1}_block1_self_att'] = block1
            attention_weights[f'decoder_layer{i+1}_block2_decenc_att'] = block2

        return x, attention_weights
```

Listing 5: Decoder Class Implementation

```python
class Transformer(tf.keras.Model):
    """
    Complete transformer with an Encoder and a Decoder
    """
    def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim,
                 input_vocab_size, target_vocab_size,
                 max_positional_encoding_input, max_positional_encoding_target,
                 dropout_rate=0.1, layernorm_eps=1e-6):
        super(Transformer, self).__init__()

        self.encoder = Encoder(
            num_layers=num_layers,
            embedding_dim=embedding_dim,
            num_heads=num_heads,
            fully_connected_dim=fully_connected_dim,
            input_vocab_size=input_vocab_size,
            maximum_position_encoding=max_positional_encoding_input,
            dropout_rate=dropout_rate,
            layernorm_eps=layernorm_eps
        )

        self.decoder = Decoder(
            num_layers=num_layers,
            embedding_dim=embedding_dim,
            num_heads=num_heads,
            fully_connected_dim=fully_connected_dim,
            target_vocab_size=target_vocab_size,
            maximum_position_encoding=max_positional_encoding_target,
            dropout_rate=dropout_rate,
            layernorm_eps=layernorm_eps
        )

        self.final_layer = tf.keras.layers.Dense(target_vocab_size, activation='softmax')

    def call(self, input_sentence, output_sentence, training,
             enc_padding_mask, look_ahead_mask, dec_padding_mask):
        """
        Forward pass for the entire Transformer
        """
        # Encode the input sentence
        enc_output = self.encoder(
            x=input_sentence,
            training=training,
            mask=enc_padding_mask
        )

        # Decode using encoder output and partial output sentence
        dec_output, attention_weights = self.decoder(
            x=output_sentence,
            enc_output=enc_output,
            training=training,
            look_ahead_mask=look_ahead_mask,
            padding_mask=dec_padding_mask
        )

        # Final linear + softmax layer to get word probabilities
        final_output = self.final_layer(dec_output)

        return final_output, attention_weights
```

Listing 6: Transformer Class Implementation