EEE 486 Assignment 3 Report
Implementing Transformer Architecture for Dialogue Summarization

Emine İrem Esendemir
ID: 22202103

Bilkent University

Instructor: Aykut Koç

March 29, 2025

Abstract

This report shows the implementation of a Transformer architecture for dialogue summarization. It shows the process of building a model from scratch. The report covers data preprocessing, the implementation of scaled dot-product attention, multi-head attention, encoder and decoder layers, and the complete transformer architecture. It also includes model training results, performance evaluation using BERTScore, and a discussion of the results.

1 Introduction

Natural Language Processing (NLP) has been transformed by the introduction of the Transformer architecture, with the publication of "Attention is All You Need" [2]. Transformers have replaced recurrent neural networks (RNNs) in many places, enabling improved computational efficiency and better modeling of long-range dependencies. They form the basis of models such as BERT, GPT, and other large language models.

This report presents an implementation of a Transformer model for dialogue summarization, while preserving key information and context. The implementation, developed using TensorFlow, closely follows the original Transformer design and includes key components such as self-attention, multi-head attention, positional encoding, and encoder-decoder stacks. Experiments are conducted on the DialogSum dataset [1]. The report is structured as follows: Section 2 describes the theoretical background and implementation, Section 3 presents model training and evaluation, and Section 4 concludes with a discussion of results. Appendix includes the wanted codes for the task.

2 PART A: Theory and Implementation

2.1 Dataset Import and Preprocessing

The DialogSum dataset [1] contains dialogue-summary pairs. For preprocessing, unnecessary columns are dropped, retaining only the "dialog" and "summary" columns. Additional preprocessing steps include converting all text to lowercase, removing newlines and double spaces, and adding start-of-sentence (`[SOS]`) and end-of-sentence (`[EOS]`) tokens.

2.1.1 Tokenization

In this project, TensorFlow's Tokenizer is used to convert text into integers, where each integer represents a specific token in the vocabulary. The Tokenizer also normalizes tokens and performs lemmatization. An out-of-vocabulary `[UNK]`token is added to handle unseen words. Additionally, it prepares the data for batching by ensuring consistent sequence lengths through padding.

2.1.2 Positional Encoding

In traditional sequential models like RNNs, the order of input tokens is implicitly captured through sequential processing. However, the Transformer architecture processes all tokens simultaneously, without any sense of order. Therefore, it requires explicit positional encoding to represent the order of tokens within a sequence. This process adds position information to the embedding vectors by providing a unique encoding for each position. These positional encodings are added to the embedding vectors before they are fed into the encoder and decoder.

2.1.3 Masking

There are two types of masking used in the Transformer architecture: the padding mask and the look-ahead mask.

The padding mask is used to handle variable-length sequences. Since sequences in a batch must have the same length, shorter sequences are padded with zeros. However, these padding tokens should not influence the self-attention mechanism. This mask helps the model ignore these tokens by creating a mask where padding tokens are represented by 0 and actual tokens are represented by 1.

The look ahead mask is used to prevent the decoder from looking at future tokens during training. For this a lower triangular matrix allowing a token to attend only to itself and previous tokens is created. Without this mask, the model could just copy the target sequence instead of learning to generate it.

2.2 Scaled Dot-Product Attention

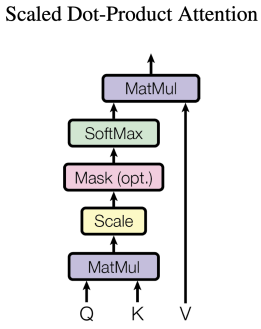

Figure 1: Scaled Dot-Product Attention [2]

The core idea of Transformers comes from the Attention Mechanism, as emphasized in the paper titled *Attention Is All You Need* [2]. Its purpose is to return rich, attention-based vector representations of the words in a sequence by allowing the model to selectively focus on important parts of the input.

The scaled dot-product attention takes three inputs: queries (Q), keys (K), and values (V):

- **Queries (Q)** represent the current token's "question" about which other tokens are relevant.
- **Keys (K)** represent each token's "identity" that can be matched against queries.
- **Values (V)** represent the actual content or information of each token that will be weighted and combined based on the attention scores.

The attention computation is formulated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where d_k is the dimensionality of the key vectors [2].

The overall implementation process can be visualized in Figure 1. The steps are as follows:

1. **MatMul**: First, the transpose of the Key is taken and multiplied by the Query . This computes a similarity score between each query and all keys, measuring how much attention each word should give to others.
2. **Scale**: The result of the dot product is divided by the square root of the key dimension $\sqrt{d_k}$. Scaling prevents the dot product values from growing too large.
3. **Mask**: If a mask is provided, it is applied to the scaled scores to prevent the model from attending to certain positions, such as padding tokens or future tokens (during decoding). This is achieved by adding a very large negative value ($-1e9$) to the masked positions, which ensures that their softmax output becomes near zero.
4. **SoftMax**: The softmax function is applied along the axis corresponding to the keys to normalize the scores into a probability distribution across all keys for each query.
5. **MatMul**: Finally, the attention weights are multiplied by the Value matrix. This produces a weighted sum of values, allowing each query to gather relevant information from the sequence based on the computed attention.

These calculations produce output vectors for each input token, enriched with information from relevant tokens. Scaled dot product attention mechanism implementa- tion is as follows:

```
def scaled_dot_product_attention(q, k, v, mask):
    matmul_qk = tf.matmul(q, k, transpose_b=True)
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

    if mask is not None:
        scaled_attention_logits += (1 - mask) * -1e9
    attention_weights = tf.keras.activations.softmax(
       scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

Testing

The above method is tested with the provided sample inputs and the resulting output and attention weights are shown below:

```
Output:
 [[[0.5  0.75]
  [0.31 0.84]
  [0.27 1.  ]]]

Attention weigths:
 [[[0.25 0.25 0.   0.25 0.25]
  [0.43 0.   0.16 0.16 0.26]
  [0.45 0.   0.   0.27 0.27]]]
```

2.3 Encoder Architecture

The Encoder is one of the main component of the Transformer architecture. It takes the sequences as input and generates context-aware vectors for each token, which can later be used by the decoder. The Encoder consists of a stack of identical layers, each including key sub-layers such as multi-head self-attention, layer normalization, feed-forward networks, and layer normalization again, all combined with residual connections.

The general structure of the Encoder is illustrated in the left part of the Figure 3.

The input to the Encoder is a batch of tokenized sentences. At first, the tokens are just embedded into vectors with positional information. Each sub-layer helps the model improve token representations while keeping training stable and efficient.

2.3.1 Multi-Head Attention Mechanism

Multi-Head Attention is an important part of the Encoder Layers. Instead of performing a single attention operation, the model projects the queries (Q), keys (K), and values (V) multiple times using different learned linear projections. It means multiplying the input vectors by a learnable weight matrix and applies attention independently in each head. After computing attention in each head, the outputs of all heads are concatenated along the feature dimension. Then, a final linear transformation with another learned matrix $W^O \in R^{hd_v \times d_{model}}$ is applied to merge the information back into the original d_{model} size.

Mathematically, multi-head attention is defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{2}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

where $W_i^Q \in R^{d_{model} \times d_k}$, $W_i^K \in R^{d_{model} \times d_k}$, $W_i^V \in R^{d_{model} \times d_v}$ are learned projection matrices for each head, and $W^O \in R^{hd_v \times d_{model}}$ is the output projection matrix [2].

This mechanism enables the model to attend to information from different representation jointly.

The Multi-Head Attention output retains the input shape $(\text{batch\_size}, \text{sequence\_length}, d_{model})$ but with more expressive, content-aware features.

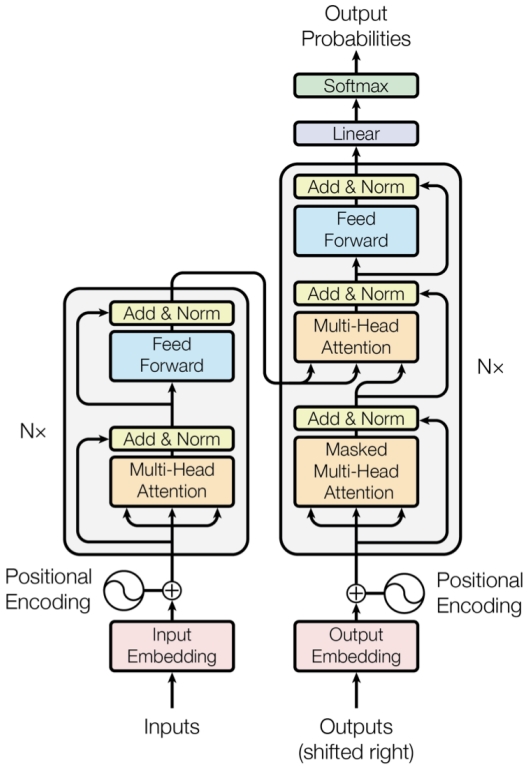

Figure 2: Transformer Architecture [2]

2.3.2 Layer Normalization

Layer Normalization is applied after each residual connection to stabilize training. It normalizes the values within the hidden dimension for each token vector independently, ensuring that each vector has mean 0 and variance 1. Its main purposes are keeping activations in a stable range and allows deeper networks with residual connections to train without problems like exploding or vanishing gradients.

2.3.3 Position-wise Feed-Forward Network (FFN)

Besides the attention sub-layers, each layer of the encoder and decoder includes a fully connected feed-forward network. It consists of two linear transformations with a ReLU activation function in between [2].

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{4}$$

In the feed-forward network, the input and output dimensions are both d_{model}, while the inner layer has a dimensionality of d_{ff}, which is greater than d_{model}.

2.3.4 Full Encoder

To summarize, the encoder first embeds the input token IDs into vectors through a learned embedding layer. These vectors are then scaled by the $\sqrt{d_{model}}$. After scaling, positional encoding is added to each token vector to keep information about the position of each word in the sequence. The dropout layer is applied to the result to prevent overfitting. The output of this process has represent each token as context aware vector of dimension d_{model}.

This output is then passed through several identical Encoder layers. Each Encoder layer enhances the token representations by applying multi-head self-attention, feed-forward transformations, and residual connections combined with layer normalization. Finally, the output of the full encoder is passed to the decoder to help with the generation of the output sequence.

The following code shows the implementation of the Encoder Layer, and the Encoder:

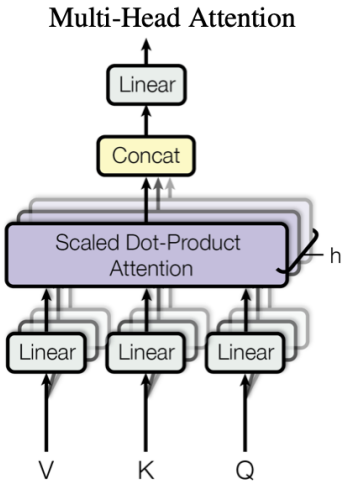

Figure 3: Multi-Head Attention [2]

```
class EncoderLayer(tf.keras.layers.Layer):
    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):
        self_mha_output = self.mha(x, x, x, mask)
        skip_x_attention = self.layernorm1(x + self_mha_output)
        ffn_output = self.ffn(skip_x_attention)
        ffn_output = self.dropout_ffn(ffn_output, training=training)
        encoder_layer_out = self.layernorm2(skip_x_attention +
            ffn_output)
        return encoder_layer_out
```

```
class Encoder(tf.keras.layers.Layer):
    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):
        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
```

Test

This encoder is tested with the provided test code and the result is shown below:

```
Using embedding_dim=16, num_heads=4, num_layers=2:

Input shape: (1, 10)
Mask shape: (1, 1, 1, 10)
Output of encoder has shape: (1, 10, 16)
```

2.4 Decoder Architecture

The Decoder is the second key component of the Transformer architecture. Its main function is to generate the output sequence based on the encoded input, such as producing a summary in this dataset. The Decoder operates autoregressively, predicting one token at a time. It consists of several Decoder layes. Each prediction is informed by both the previously generated tokens and the complete output from the Encoder.

2.4.1 Masked Multi-Head Self-Attention

The first sub-layer in each Decoder layer is the masked multi-head self-attention. It allows each token in the output sequence to attend to earlier tokens in the sequence, helping the model to learn dependencies between previously generated words. However, to have the autoregressive property of the Decoder, the model must not have access to future tokens during training. To achieve this, a look-ahead mask is applied to the attention weights. This mask ensures that, at each position, the model can only attend to allowed positions. As a result, each token is generated based only on the known history, similar to the sequential generation process used at inference time.

The input to this sublayer is the embedded decoder input, the summary tokens generated so far, and the output is a context-aware representation of each token that encapsulates information from its leftward context. The output shape remains the same as the input: $(\text{batch\_size}, \text{target\_seq\_len}, d_{\text{model}})$.

2.4.2 Encoder–Decoder Cross-Attention

The second sub-layer in each Decoder layer is the encoder–decoder cross-attention. Its role is to allow the decoder to attend to relevant parts of the input sequence, which was processed by the encoder. This is essential for generating output tokens that are contextually aligned with the source input.

In this attention block, the queries come from the decoder. To be more specific, the output of the masked self-attention sub-layer, while the keys and values come from the encoder output. This structure enables the decoder to determine which parts of the input sequence are most relevant, based on the tokens generated so far.

The input to this sub-layer is the output of the masked self-attention, and the encoder output. The result is a context-aware representation of each target token, now containing both its previous decoder context and relevant input features coming from the Encoder. The output shape remains consistent: $(\text{batch\_size}, \text{target\_seq\_len}, d_{\text{model}})$.

2.4.3 Feed-Forward Network (FFN)

The third sub-layer in each Decoder layer is the position-wise feed-forward network (FFN). Its structure is identical to the FFN used in the Encoder. It consists of two fully connected dense layers applied independently to each tokenin the sequence.

The first dense layer expands the feature dimension from d_{model} to a larger intermediate dimension, often denoted as d_{ff}, and applies a ReLU activation. The second dense layer projects the features back down to d_{model}. Mathematically, the feed-forward network can be expressed as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{5}$$

where $W_1 \in R^{d_{\text{model}} \times d_{\text{ff}}}$, $W_2 \in R^{d_{\text{ff}} \times d_{\text{model}}}$, and b_1, b_2 are the bias terms [2].

The purpose of the FFN is to introduce additional non-linearity and depth to the model, developing its capacity to capture complex patterns. Since it is applied independently to each token, it does not mix information across different positions.

The input to this sublayer is the output of the cross-attention mechanism, and the output shape remains the same as the input: $(\text{batch\_size}, \text{target\_seq\_len}, d_{\text{model}})$.

2.4.4 Decoder Layer

Each Decoder layer is composed of the three previously described sub-layers connected sequentially: masked multi-head self-attention, encoder–decoder cross-attention, and a position-wise feed-forward network.

Each sub-layer is followed by a residual connection and a layer normalization step to stabilize training and improve gradient flow.

The output of one Decoder layer serves as the input to the next Decoder layer. By stacking multiple Decoder layers, the model progressively refines the representations of each target token, combining information from both the previously generated tokens and the input sequence.

The final output of the last Decoder layer provides richer, context-aware embeddings for each position in the target sequence. These embeddings are then passed to a linear projection and softmax layer to produce the predicted output tokens.

The implementation for the Decoder is shown below:

```
class DecoderLayer(tf.keras.layers.Layer):
    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):
        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)
        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)
        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
```

2.4.5 Full Decoder

The full Transformer Decoder is composed of three main components: an input embedding layer, a series of stacked Decoder layers, and a final linear projection with softmax to generate output tokens as it can be seen from the right side of the Figure 3. .

At the beginning of the decoder, the target token IDs, the tokens generated so far in the output sequence, are mapped to vectors using a learned embedding layer. These embeddings are then scaled by the square root of the model dimension d_{model} to keep stable the gradient magnitudes. Positional encoding is then added to add information about the position of each token in the sequence. A dropout layer is applied for regularization.

The resulting tensor is passed through a sequence of N identical Decoder layers. Each layer enhances the token representations by attending to the previous output tokens through masked self-attention and to the encoder's output with help of cross-attention. The output of one Decoder layer becomes the input to the next, allowing the model to build increasingly rich contextual embeddings for each target token.

During training, the decoder is provided with the entire ground truth target sequence, shifted right by one position, and uses a look-ahead mask to prevent attending to future tokens. This strategy, known as teacher forcing, keep process efficient by parallel computation while preserving the autoregressive property.

During inference, however, the decoder generates one token at a time, starting with a special start-of-sequence token. After each predicted token, it is appended to the input and the decoder is called again to generate the next token. This process continues until a special end-of-sequence token is generated or a predefined maximum length is reached.

The implementation for the decoder is shown below:

```
class Decoder(tf.keras.layers.Layer):
    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):
        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['decoder_layer{}_block1_self_att'.format
                (i+1)] = block1
            attention_weights['decoder_layer{}_block2_decenc_att'.
                format(i+1)] = block2

        return x, attention_weights
```

Test

The Decoder is tested with the provided code and the results are shown below:

```
Using num_layers=7, embedding_dim=15 and num_heads=19:

x has shape:(3, 4)
Output of encoder has shape:(3, 7, 9)

Output of decoder has shape:(3, 4, 15)

Attention weights:
decoder_layer1_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer1_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer2_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer2_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer3_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer3_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer4_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer4_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer5_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer5_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer6_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer6_block2_decenc_att has shape:(3, 19, 4, 7)
decoder_layer7_block1_self_att has shape:(3, 19, 4, 4)
decoder_layer7_block2_decenc_att has shape:(3, 19, 4, 7)
```

2.5 Full Transformer Architecture

Having described the components of the Transformer in detail, it is needed to summarize how the complete model operates as a system. The Transformer consists of an encoder and a decoder, each built by stacking multiple layers with self-attention, feed-forward networks, and normalization mechanisms. These components work together to process input sequences and generate structured, contextually informed output sequences.

The input tokens, such as dialog turns in the DialogSum dataset, are first tokenized and converted into embedding vectors. Positional encodings are added to inject order information. The resulting sequence is passed through a series of encoder layers, each enhances the representation of each token by attending to the entire sequence using multi-head self-attention. The final encoder output contains rich, context-aware representations of the input sequence.

The decoder takes the output tokens generated so far (or the ground truth summary during training) and processes them similarly through embedding, positional encoding, and multiple decoder layers. Each decoder layer uses masked self-attention to model dependencies in the output sequence and cross-attention to align with the encoder output. This allows the decoder to generate output tokens conditioned on both the target context and the original input.

At the top of the decoder, the final output is passed through a linear projection and softmax layer to produce probabilities over the vocabulary. During training, the full target sequence is used with masking to ensure proper autoregressive learning. During inference, the model generates tokens one-by-one, feeding the previously generated token back into the decoder at each step.

The implementation for the transformer is shown below:

```
class Transformer(tf.keras.Model):
    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):
        enc_output = self.encoder(x=input_sentence, training=training,
            mask=enc_padding_mask)
        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_output = self.final_layer(dec_output)
        return final_output, attention_weights
```

Test

This Transformer is tested with the provided code and the result is shown below:

```
Using num_layers=7, target_vocab_size=350 and num_heads=19:
sentence_a has shape:(1, 6)
sentence_b has shape:(1, 6)
Output of transformer (summary) has shape:(1, 6, 350)
Attention weights:
decoder_layer1_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer1_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer2_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer2_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer3_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer3_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer4_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer4_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer5_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer5_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer6_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer6_block2_decenc_att has shape:(1, 19, 6, 6)
decoder_layer7_block1_self_att has shape:(1, 19, 6, 6)
decoder_layer7_block2_decenc_att has shape:(1, 19, 6, 6)
```

3 PART B: Model Training and Results

3.1 Model Training

After implementing the encoder, decoder, and full Transformer, the model was trained to perform summarization by learning contextualized representations of dialogue. Prior to training, the necessary hyperparameters were defined based on the model.

The following hyperparameter values were used in the training process:

```
num_layers = 2
embedding_dim = 128
fully_connected_dim = 128
num_heads = 2
positional_encoding_length = 256
```

The model was trained for 30 epochs using the Adam optimizer with a custom learning rate schedule. Dropout was applied to both the encoder and decoder sub-layers to reduce overfitting. The loss function used was categorical cross-entropy, computed over the predicted token probabilities.

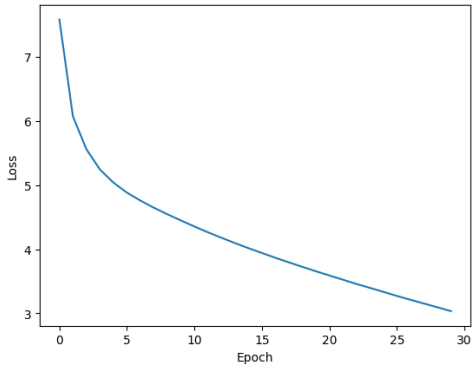

Figure 4: Training loss over 30 epochs

As shown in Figure 4, the training loss decreased over time, suggesting that the model was able to effectively learn patterns in the training data and improve its predictions.

3.2 Sample Summarization Results and Discussion

To evaluate the model's performance, one example from the training set and one from the test set are shown below. Each example includes the input dialogue, the human-written reference summary, and the model-generated summary, followed by a discussion.

3.2.1 Training Set Example

Input Dialogue:

> [SOS] person1: i'm tired of watching television. let's go to cinema to- night. 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 pa- per 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 ell 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 Generated Summary:

[SOS] person1 and person2 are planning to go to the movie but they are going to see a movie but they are going to go to the cinema but person2 refuses [EOS]

Explanation:

In this training example, lacks coherence and factual accuracy. While it identifies the general topic of going to the cinema, it repeats ideas and fails to mention the actual decision to stay home. Additionally, the phrase "person2 refuses" is not supported by the dialogue and seems hallucinated. This result suggests that while the model can capture surface-level information it struggles to track conversation dynamics and generate a concise, accurate summary.

3.2.2 Test Set Example

Input Dialogue:

[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 Generated Summary:

[SOS] mr black tells person1 the way to the airport and the hotel is going to the hotel in beijing person1 asks mr black to come in the hotel for the hotel [EOS]

Explanation:
In the test example, the model-generated summary diverges more from the actual content of the dialogue. It introduces a non existing entity ("mr black"), misrepresents speaker roles, and contains repetitive or incoherent phrases. The main points of the dialoge is not mentioned in the summary.

Overall Discussion:
The summaries often contain repetitions, hallucinations, or loss of keys of the dialog. These limitations may be attributed to the use of a small number of layers and attention heads. Additionally, full generalization might be hindered by the relatively short training time and limited dataset size.

3.3 BERTScore Evaluation

To quantitatively evaluate the quality of the generated summaries, BERTScore was used. BERTScore computes similarity between reference and candidate texts using contextual embeddings from a pre-trained BERT model. It provides Precision, Recall, and F1 Score metrics that reflect how semantically close the model-generated summary is to the ground truth.

Due to the large size of the training set, BERTScore was computed on a randomly selected 1/20 subset of the training to avoid computational overhead. The average BERTScore results for both the test set and this sampled training subset are reported in Table 1.

Table 1: BERTScore Results on Training and Test Sets

Dataset	**Precision**	**Recall**	**F1 Score**
Test Set	0.8870	0.8474	0.8667
Sampled Training Set (1/20)	0.9015	0.8848	0.8930

The results indicate that the model performs slightly better on the sampled training data, as expected, due to having seen similar patterns during training. The performance on the test set remains reasonably strong, with an average F1 score of 0.8667, suggesting that the model is able to generate summaries that are semantically close to the reference summaries even on unseen dialogues.

4 Discussion and Conclusion

In this assignment, a Transformer-based model was implemented from scratch for the task of dialogue summarization using the DialogSum dataset [1]. The architecture included key components such as positional encoding, scaled dot-product attention, multi-head attention, encoder and decoder stacks, and a final linear projection layer. The model was trained for 30 epochs, and its performance was evaluated using both qualitative examples and quantitative metrics.

Qualitative results showed that while the model captured some domain-specific cues, it often struggled with coherence, factual grounding, and decision tracking. The generated summaries sometimes included hallucinated content or omitted key events.

Quantitative evaluation using BERTScore indicated that the model produced semantically meaningful summaries, achieving an average F1 score of 0.8930 on a sampled subset of the training set and 0.8667 on the test set. These results show that the model generalizes reasonably well.

References

[1] Yulong Chen, Yang Liu, Liang Chen, and Yue Zhang. DialogSum: A real-life scenario dialogue summarization dataset. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 5062–5074, Online, August 2021. Association for Computational Linguistics.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.