CS 6120 — Assignment 7

Due: March 30, 2026 (100 points)

Attention Modeling

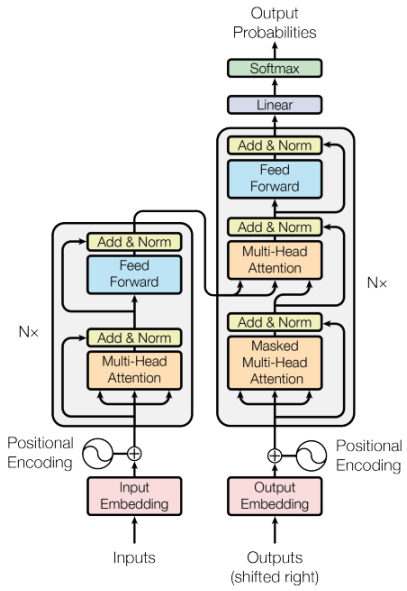

Figure 1: The Transformer Architecture

In this homework, we will be implementing and training a transformer neural network, similar to the algorithms described in class. This model is slightly different than others you have already implemented. It is heavily based on attention and does not rely on sequences, which can ultimately allow for parallel computing. Transformers are compute heavy, and it is much easier to train if you're on GPU hardware. There are a variety of ways that you can use GPU's.

In this assignment you will explore summarization using the transformer model. Summarization in natural language processing is useful for multiple consumer enterprise applications. Bots can be used to scrape articles, summarize them, and then you can use sentiment analysis to identify the sentiment about certain stocks.

You will be guided through all the steps and will find numerous hints to assist you. By the end of this homework, you will have implemented the full transformer (both encoder and decoder), but you will only be graded on the implementation of the decoder as the encoder is provided for you.

We will be working with the SAMSum dataset, which contains about 16k messenger-like conversations with summaries. Conversations were created and written down by linguists fluent in English. Linguists were asked to create conversations similar to those they write on a daily basis, reflecting the proportion of topics of their real-life messenger conversations. The style

and register are diversified - conversations could be informal, semi-formal or formal, they may contain slang words, emoticons and typos. Then, the conversations were annotated with summaries. It was assumed that summaries should be a concise brief of what people talked about in the conversation in third person. Go ahead and download it from the course website. You can unzip and extract it with:

```
$> wget -nc https://course.ccs.neu.edu/cs6120s25/data/samsum/util.py
$> wget -nc https://course.ccs.neu.edu/cs6120s25/data/samsum/corpus.tar.gz
$> tar -xvzf corpus.tar.gz
```

Transformers Utilities: Provided Functions

There are a number of necessary but provided functions interspersed throughout the templated code. Some of these, we may have implemented in class, and others we have directly written here. The functions in these sections are not graded.

Preprocessing Data

The preprocessing step is a combination of steps that includes tokenization, training / test splits, filtering, padding, and document processing. In contrast to many other datasets, the vocabulary and tokenization is derived from both the input text sequences *and* the output annotations, which happen to be text sequences themselves. Ultimately, the processing will produce a `tf.Dataset`, which is a very common dataset iterator that is quite commonly used in production systems.

Masking

There are two types of masks that are useful when building your Transformer network: the *padding mask* and the *look-ahead mask*. Both help the softmax computation give the appropriate weights to the words in your input sentence. You have already learned how to implement and use them in one of this week's labs. Here they are implemented for you.

Positional Encoding

In sequence to sequence tasks, the relative order of your data is extremely important to its meaning. When you were training sequential neural networks such as RNNs, you fed your inputs into the network in order. Information about the order of your data was automatically fed into your model. However, when you train a Transformer network using multi-head attention, you feed your data into the model all at once. While this dramatically reduces training time, there is no information about the order of your data. This is where positional encoding is useful.

You have learned how to implement the positional encoding in one of this week's labs. Here you will use the `positional_encoding` function to create positional encodings for your transformer. The function is already implemented for you.

The Transformer Encoder layer pairs self-attention and convolutional neural network style of processing to improve the speed of training and passes K and V matrices to the Decoder, which you'll build later in the assignment. In this section of the assignment, you will implement the Encoder by pairing multi-head attention and a feed forward neural network in Fig. 2.

Encoder Layer

`MultiHeadAttention` you can think of as computing the self-attention (that you will be implementing in Q1) several times to detect different features.

Feed forward neural network contains two Dense layers which we'll implement as the function `FullyConnected`.

Your input sentence first passes through a *multi-head attention layer*, where the encoder looks at other words in the input sentence as it encodes a specific word. The outputs of the multi-head attention layer are then fed to a *feed forward neural network*. The exact same feed forward network is independently applied to each position.

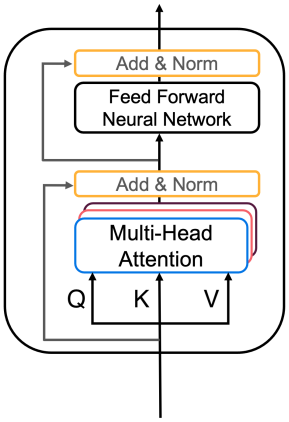

Figure 2: Encoder Layer

For the `MultiHeadAttention` layer, you will use the MultiHeadAttention implemented in Keras[1]. If you're curious about how to split the query matrix Q, key matrix K, and value matrix V into different heads, you can look through the implementation. You will also use the Sequential API with two dense layers to built the feed forward neural network layers.

Full Encoder

Now you can pair multi-head attention and feed forward neural network together in an encoder layer! You will also use residual connections and layer normalization to help speed up training.

The encoder block is already implemented for you. Take a very close look at its implementation, as you will later have to create the decoder yourself, and a lot of the code is very similar. The encoder block performs the following steps:

1. It takes the Q, V, K matrices and a boolean mask to a multi-head attention layer. Remember that to compute *self*-attention Q, V and K are the same. You will also perform Dropout in this multi-head attention layer during training.
2. There is a skip connection to add your original input `x` and the output of the multi-head attention layer.
3. After adding the skip connection, the output passes through the first normalization layer.
4. Finally, steps 1-3 are repeated but with the feed-forward neural network with a dropout layer instead of the multi-head attention layer.

Question 1: Self Attention

The use of self-attention paired with traditional convolutional networks allows for parallelization which speeds up training. You will implement **scaled dot product attention** which takes in

[1]The multi-head attention module differs between Keras v3.6 and v3.7. You will notice special cases in the test functions, which will not test certain functionality if the versioning is lower that v3.7.

a query, key, value, and a mask as inputs to return rich, attention-based vector representations of the words in your sequence. This type of self-attention can be mathematically expressed as:

$$\text{Attention } (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{1.1}$$

- Q is the matrix of queries
- K is the matrix of keys
- V is the matrix of values
- M is the optional mask you choose to apply
- d_k is the dimension of the keys, which is used to scale everything down so the softmax doesn't explode

In this question, you will fill out the function `scaled_dot_product_attention` to create attention-based representations. The boolean mask parameter M can be passed in as `None` or as either padding or look-ahead. Because M is applied additively, you will need to multiply `(1.0 - mask)` by `-1e9` before adding it to the scaled attention logits.

Question 2: Decoder Layer

In this section, we'll learn to use `tensorflow.keras`'s multi-head attention model, which you will reference at the source code website.

We have covered elements of the decoder layer in lecture, and it is very similar to the provided `Encoder` implementation. The Decoder layer takes the K and V matrices generated by the Encoder and computes the second multi-head attention layer with the Q matrix from the output (Figure 3a).

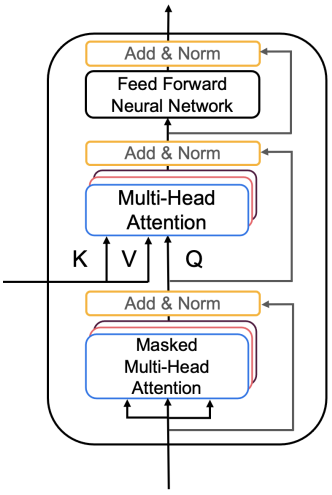

Figure 3: Decoder Layer

In Fig. 3 pair multi-head attention with a feed forward neural network, but this time you'll implement two multi-head attention layers. You will also use residual connections and layer normalization to help speed up training (Figure 3a).
Implement `DecoderLayer()` using the `call()` method

1. Block 1 is a multi-head attention layer with a residual connection, and look-ahead mask. Like in the `EncoderLayer`, Dropout is defined within the multi-head attention layer.
2. Block 2 will take into account the output of the Encoder, so the multi-head attention layer will receive K and V from the encoder, and Q from the Block 1. You will then apply a normalization layer and a residual connection, just like you did before with the `EncoderLayer`.

3. Finally, Block 3 is a feed forward neural network with dropout and normalization layers and a residual connection.

The first two blocks are fairly similar to the `EncoderLayer` except you will return `attention_scores` when computing self-attention. Implement the `call()` method in `DecoderLayer()` to utilize multiple attention head layers and output layers.

Question 3: Full Decoder

In this question you will use your `DecoderLayer` to build the full `Transformer Decoder`. You will embed your output, add positional encodings, and then feed your encoded embeddings to a stack of Decoder layers.

In this exercise, you will initialize your Decoder with an Embedding layer, positional encoding, and multiple DecoderLayers. Your `call()` method will perform the following steps:

1. Pass your generated output through the Embedding layer.
2. Scale your embedding by multiplying it by the square root of your embedding dimension. Remember to cast the embedding dimension to the data type `tf.float32` before computing the square root.
3. Add the position encoding: `self.pos_encoding[:, :seq_len, :]` to your embedding.
4. Pass the encoded embedding through a dropout layer, remembering to use the `training` parameter to set the model training mode.
5. Pass the output of the dropout layer through the stack of `DecodingLayers` using a `for` loop.

Implement the `call()` method in `Decoder()` to embed your output, add positional encoding, and implement multiple decoder layers.

Question 4: Transformer

The flow of data through the Transformer Architecture can be seen in Fig. 4 and can be reviewed as follows:

1. First your input passes through an `Encoder`, which is just repeated `EncoderLayers` that you implemented:
   a) embedding and positional encoding of your input
   b) multi-head attention on your input
   c) feed forward neural network to help detect features
2. Then the predicted output passes through a `Decoder`, consisting of the decoder layers that you implemented:
   a) embedding and positional encoding of the output

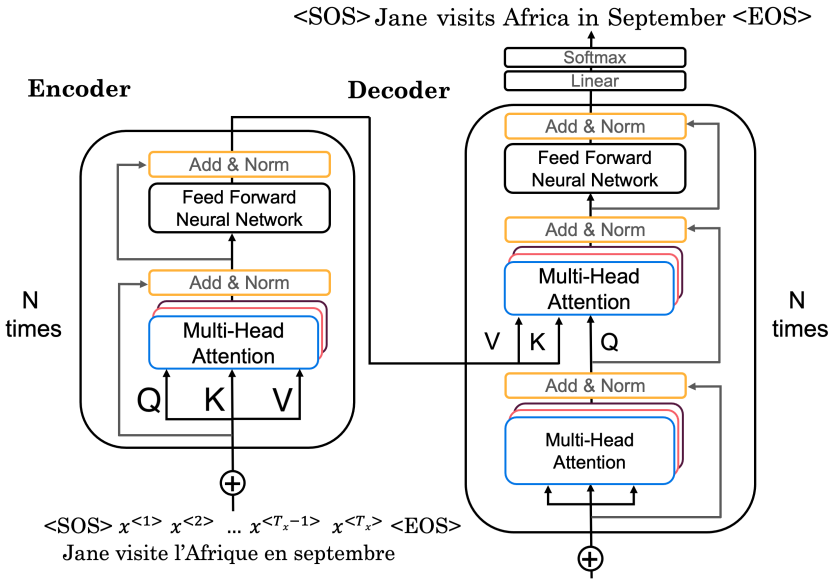

Figure 4: Transformer Architecture

b) multi-head attention on your generated output

c) multi-head attention with the Q from the first multi-head attention layer and the K and V from the `Encoder`

d) a feed forward neural network to help detect features

3. Finally, after the N^{th} `DecoderLayer`, one dense layer and a softmax are applied to generate prediction for the next output in your sequence.

In this question, you will implement `Transformer()` using the `call()` method. This will:

1. Pass the input through the `Encoder` with the appropriate mask.
2. Pass the encoder output and the target through the Decoder with the appropriate mask.
3. Apply a linear transformation and a softmax to get a prediction.

Question 5: Text Summarization

The last thing you will implement is inference. With this, you will be able to produce actual summaries of the documents. You will use a simple method called greedy decoding, which means you will predict one word at a time and append it to the output. You will start with an `[SOS]` token and repeat the word by word inference until the model returns you the `[EOS]` token or until you reach the maximum length of the sentence (you need to add this limit, otherwise a poorly trained model could give you infinite sentences without ever producing the `[EOS]` token.

Write a helper function that predicts the next word, called `next_word` so you can use it to write the whole sentences. Hint: this is very similar to what happens in the provided function

`train_step` (in the section of the template labeled `Provided Functions: Part III`), but you have to set the training of the model to `False`.

Train and Test Your Model

One of the provided functions is `train_step`. It is a loop that will train your model for 20 epochs. On a GPU, it should take about 20 seconds per epoch (with the exception of the first epoch, which may take twice as long). Note that after each epoch the code performs the summarization on one of the sentences in the test set and prints it out, so you can see how your model is improving.

If you critically examine the output of the model, you can notice a few things:

- In the training set the model output is (almost) identical to the real output (already after 20 epochs and even more so with more epochs). This might be because the training set is relatively small and the model is relatively big and has thus learned the sentences in the training set by heart (overfitting).
- While the performance on the training set looks amazing, it is not so good on the test set. The model overfits, but fails to generalize. While it is a small training set and a comparatively large model, but there might be a variety of other factors.
- Look at the test set example 3 and its summarization. Would you summarize it the same way as it is written here? Sometimes the data may be ambiguous. And the training of **your model can only be as good as your data**.

You are only using a small dataset, to show that something can be learned in a reasonable amount of time in a relatively small environment. Generally, large transformers are trained on more than one task and on very large quantities of data to achieve superb performance.

Go ahead and train your Transformer for 20 epochs with the provided dataset by running `train_model()`.

Submission Instructions

You will notice there are several unit test functions in your homework. Feel free to use them to verify your implementation. (Note that the random initialization of neural network parameters may depend on Tensorflow Version, where we have used TFv2.18.0).

When you have finished, submit your Python file, named `assignment7.py` to Gradescope.