**Tensor Flow Explanation**

**Transformer:**

**Masking:**

We used **create\_padding\_mask**and **create\_look\_ahead**as helper functions to creating masks to mask out padded tokens, we are going to use these helper functions as **tf.keras.layers.Lambda** layers.

We should mask all the pad tokens (value 0) in the batch to ensure the model does not treat padding as input.

**Positional encoding:**

Since this model doesn't contain any recurrence or convolution, positional encoding is added to give the model some information about the relative position of the words in the sentence.

The positional encoding vector is added to the embedding vector. Embeddings represent a token in a d-dimensional space where tokens with similar meaning will be closer to each other. But the embeddings do not encode the relative position of words in a sentence. So after adding the positional encoding, words will be closer to each other based on the similarity of their meaning and their position in the sentence, in the d-dimensional space.

The formula for calculating the positional encoding is as follows:

**PE(pos,2i)=sin(pos/100002i/dmodel)**

**PE(pos,2i+1)=cos(pos/100002i/dmodel)**

**Encoder Layer:**

Each encoder layer consists of sublayers:

**1- Multi-head attention (with padding mask)**

**2- 2 dense layers followed by dropout**

Each of these sublayers has a residual connection around it followed by a layer normalization. Residual connections help in avoiding the vanishing gradient problem in deep networks.

The output of each sublayer is LayerNorm(x + Sublayer(x)). The normalization is done on the d\_model (last) axis.

**Encoder:**

The Encoder consists of:

1- Input Embedding

2- Positional Encoding

3- num\_layers encoder layers

The input is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the encoder layers. The output of the encoder is the input to the decoder.

**Decoder Layer:**

Each decoder layer consists of sublayers:

1- Masked multi-head attention (with look ahead mask and padding mask)

2- Multi-head attention (with padding mask). value and key receive the *encoder output* as inputs. query receives the output from the masked multi-head attention sublayer*.*

3- 2 dense layers followed by dropout

Each of these sublayers has a residual connection around it followed by a layer normalization. The output of each sublayer is LayerNorm(x + Sublayer(x)). The normalization is done on the d\_model (last) axis.

As query receives the output from decoder's first attention block, and key receives the encoder output, the attention weights represent the importance given to the decoder's input based on the encoder's output. In other words, the decoder predicts the next word by looking at the encoder output and self-attending to its own output. See the demonstration above in the scaled dot product attention section.

**Decoder:**

The Decoder consists of:

1- Output Embedding

2- Positional Encoding

3- N decoder layers

The target is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the decoder layers. The output of the decoder is the input to the final linear layer.

**Transformer:**

Transformer consists of the encoder, decoder and a final linear layer. The output of the decoder is the input to the linear layer and its output is returned.

We discussed about what we did in the **Transformer** section from masking to decoder and transformer function it self , now we will discuss **Train Model** section.

**Train Model:**

**Loss function:**

So since the target sequences are padded, it is important to apply a padding mask when calculating the loss.

**Learning rate:**

We use the Adam optimizer with a custom learning rate scheduler , here is the formula :

**lrate=d−0.5model∗min(step\_num−0.5,step\_num∗warmup\_steps−1.5)**

**Initialize and compile model:**

Initialize and compile model with our predefined custom learning rate and Adam optimizer under the strategy scope.

**Fit model:**

Simple function to train our transformer by simply calling model.fit() .

**Save Model:**

Don`t forget to save your model using **tf.keras.models.save\_model** function.

**Now we are going to evaluate and predict some examples:**

**Evaluate and predict:**

**The following steps are used for evaluation:**

* Apply the same preprocessing method we used to create our dataset for the input sentence.
* Tokenize the input sentence and add **START\_TOKEN** and **END\_TOKEN.**
* Calculate the padding masks and the look ahead masks.
* The decoder then outputs the predictions by looking at the encoder output and its own output.
* Select the last word and calculate the argmax of that.
* Concatentate the predicted word to the decoder input as pass it to the decoder.
* In this approach, the decoder predicts the next word based on the previous words it predicted.

**Note:** The model used here has less capacity and trained on a subset of the full dataset, hence its performance can be further improved.

**Here are some testing for our model:**

**Input:** Where have you been?

**Output:** i have been here. i have been thinking , have not i ?

**Input:** It's a trap

**Output:** okay , you loved that all four .

**Here we test the model with another experiment by feeding the model with its previous output:**

**Input:** I am not crazy, my mother had me tested.

**Output:** i thought you said that was a very big questioned er who would be right for a little while . you would be better if you were a clue .

**Input:** i thought you said that was a very big questioned er who would be right for a little while . you would be better if you were a clue .

**Output:** the other one .

**Input:** the other one .

**Output:** who s the boss ?

**Input:** who s the boss ?

**Output:** i do not know . i do not know . i just feel a little sorrier i am on the unicorn .

Input: i do not know . i do not know . i just feel a little sorrier i am on the unicorn .

Output: oh . well , you know .

**Summary:**

Here we are, we have implemented a Transformer in TensorFlow 2.0 using the conversations in movies and TV shows provided by Cornell Movie-Dialogs Corpus.

We focused on the two different approaches to implement complex models with Functional API and Model subclassing, and how to incorporate them.

**Thanks for reading!**