Chatbot using Transformer

==========

Overview:

A chatbot is a computer program designed to simulate human conversation with end-users. While not all chatbots incorporate artificial intelligence (AI), modern chatbots increasingly leverage conversational AI techniques, such as natural language processing (NLP), to comprehend user queries and provide automated responses.

Chatbot technology has become ubiquitous, appearing in devices like smart speakers, consumer communication platforms such as SMS, WhatsApp, and Facebook Messenger, and workplace tools like Slack. The latest generation of AI chatbots, often referred to as "intelligent virtual assistants" or "virtual agents," not only excel at understanding natural conversations through advanced language models but also excel at automating various tasks. Prominent examples include Apple's Siri and Amazon Alexa.

In this project, we'll create a chatbot from scratch using TensorFlow, without relying on predefined APIs.

Dataset

For our chatbot, we'll utilize the Cornell Movie-Dialogs Corpus dataset. This dataset consists of over 130,000 meticulously curated dialogues from 617 films. The data has been refined, removing overly lengthy dialogues, and includes seven key features:

-ID: A serial number.

-Question: Represents user questions.

-Answer: Represents responses.

-Question_as_int: Tokenized questions.

-Answer_as_int: Tokenized answers.

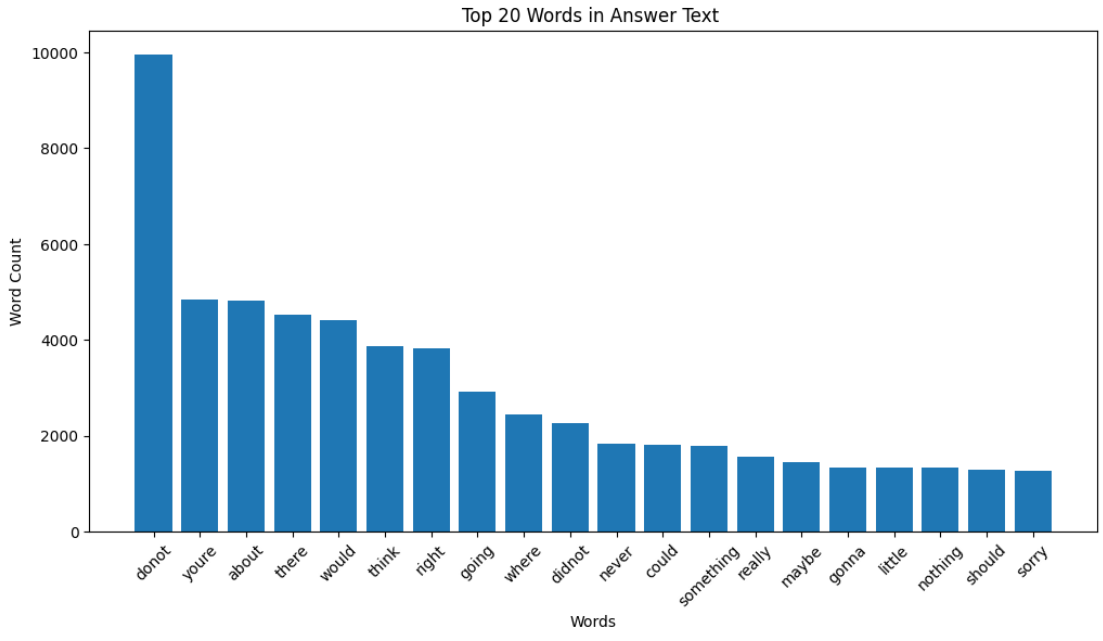
-Question_len: Word count in questions.

-Answer_len: Word count in answers.

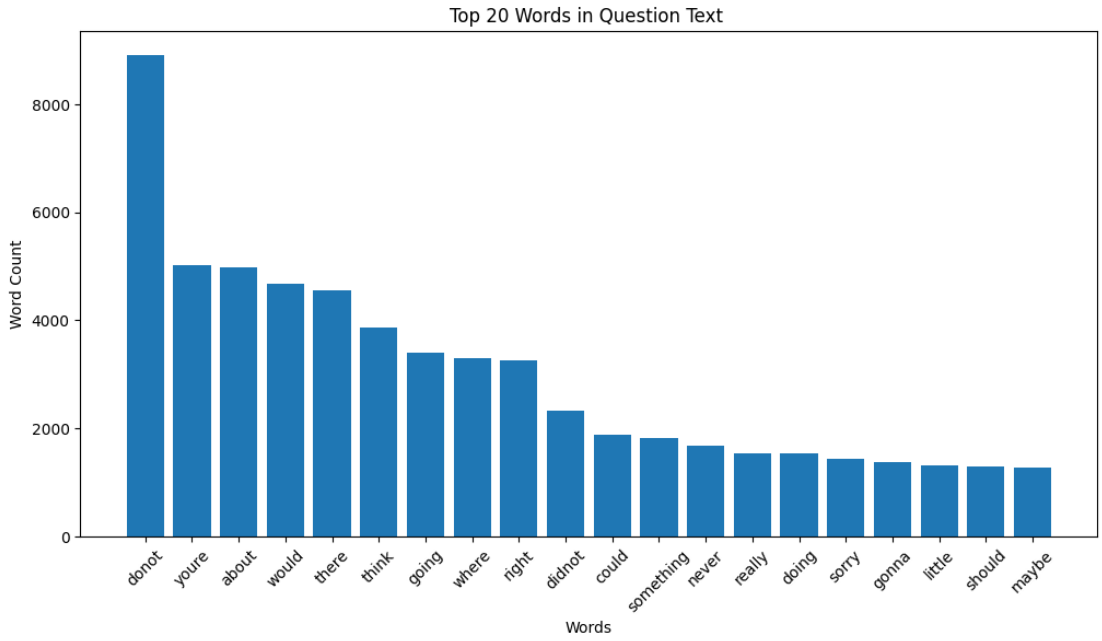
While this dataset was initially created for movie dialogues, it can be adapted to model everyday human interactions or casual conversations, we will only focus on the "question" and "answer" features during model training.

We'll begin by preprocessing the dataset, removing special characters, and analyzing word frequency.

Then we will visualize and have a look at word frequency distribution.



let's check for words length greater than 4



That’s enough for EDA, let’s get started with the modeling

There is Various approaches for the training we will use transformer

Transformer Architecture :

Transformer NLP represents a neural network architecture that has revolutionized the field of natural language processing (NLP). Transformers are proficient at learning long-range dependencies within text, enabling them to better grasp the overall context and generate more coherent text. Compared to previous NLP models, transformers are highly flexible and scalable, making them adaptable to various tasks and domains.

Transformers are built on the self-attention mechanism, allowing the model to focus on different parts of the input sequence and understand their relationships. This sets them apart from earlier NLP models that relied on recurrent neural networks (RNNs) for text sequence processing. While RNNs are adept at learning

sequential relationships, they can be slow to train and challenging to parallelize.

Data preparation

We already have a CSV file, I have converted it into a pandas data frame and took only two columns '*question*' and '*answer*', and then cleaned it i.e. removal of special characters, decentration

	question	answer
0	well i thought we would start with pronunciati...	not the hacking and gagging and spitting part ...
1	not the hacking and gagging and spitting part ...	okay then how bout we try out some french cuis...
2	you are asking me out that is so cute what is...	forget it
3	no no it is my fault we did not have a proper...	cameron
4	gosh if only we could find kat a boyfriend	let me see what i can do

Given that neural networks require numerical representations, we'll perform tokenization to convert each word into a numerical format. We'll employ TensorFlow Datasets (tfds) to create subword tokenizers for both the "answer" and "question" text data. Subword tokenization breaks text into smaller units, enhancing training speed by reducing the vocabulary size. The "tokenizer_q" and "tokenizer_a" will convert text into sequences of tokens and serve as pretrained tokenizers.

```
Tokenized string using the answer tokenizer: [27199, 10524, 13670, 667, 15302, 4410]
Original string using the answer tokenizer: Encoder decoder
Individual tokens using the answer tokenizer:
27199 --> E
10524 --> nc
13670 --> ode
667 --> r
15302 --> deco
4410 --> der
============================================================
Tokenized string using the question tokenizer: [27437, 10632, 13774, 664, 24925]
Original string using the question tokenizer: Encoder decoder
Individual tokens using the question tokenizer:
27437 --> E
10632 --> nc
13774 --> ode
664 --> r
24925 --> decoder
```

We'll use TensorFlow's dataset object to generate data during training and set up data pipelines for training and validation datasets.

This approach avoids loading the entire dataset into memory, instead generating data as needed. We'll employ the tf_encode() function to convert text into numerical representations, pad the dataset to ensure consistent sentence lengths, and use the prefetch() function to optimize data loading and model performance, particularly with large batches.

For "question," the vocabulary size is 27,513, spanning from 0 to 27,512. We'll add two additional tokens at the start and end, representing the start and end of sequences. Notably, we haven't added special characters, as they are already included in tokenized form.

With the data preprocessed and tokenized, we're ready to delve into implementing the key components of the Transformer architecture for our chatbot.

Positional encoding

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

Positional encoding is a technique used to add positional information to a sequence of tokens. This is necessary for transformers, which do not have recurrent neural networks (RNNs) to preserve the sequence information and relative positions of words.

Positional encoding is done after each token is converted into its embedding vector. The embedding vector is a dense vector that represents the meaning of a word. The positional encoding vector is added to the embedding vector to add information about the position of the word in the sequence.

There are several different methods for positional encoding. One common method is to use sine and cosine functions. The perivous equation shows how to calculate the positional encoding vector for a given token using sine and cosine functions:

positional_encoding_vector[i] = [sin(pi * i / embedding_vector_length), cos(pi * i / embedding_vector_length)]

where:

- i is the index of the token in the sequence.
- embedding_vector_length is the length of the embedding vector.

The positional encoding vector is then added to the embedding vector to create the final input to the transformer.

Positional encoding allows transformers to learn the relative positions of words in a sequence, even though they do not have RNNs.

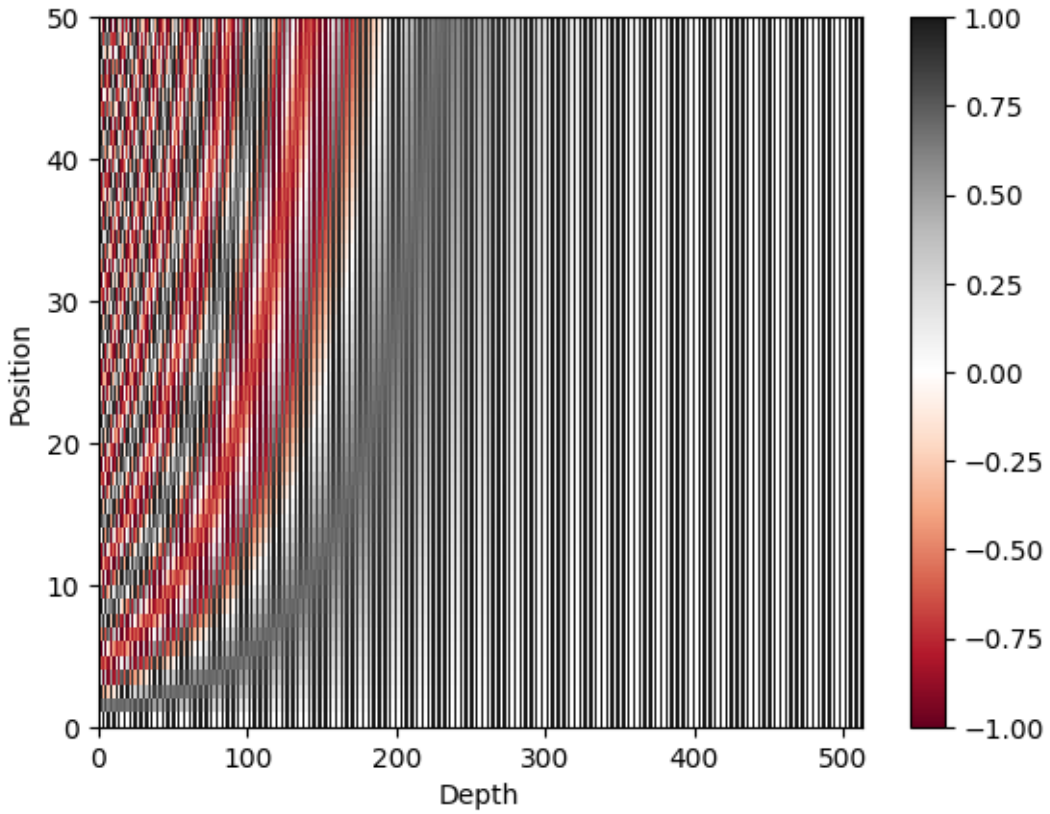
Here is an example of how positional encoding works:

```
# The embedding vector for the word "hello"
embedding_vector = [0.1, 0.2, 0.3, 0.4, ..., 0.9, 1.0]

# The positional encoding vector for the word "hello"
positional_encoding_vector = [0.0, 0.1]

# The final input to the transformer
input_vector = embedding_vector + positional_encoding_vector
```

The input vector now contains information about both the meaning of the word "hello" and its position in the sequence. This information can be used by the transformer to learn the relative positions of words in a sequence and perform many different natural language processing tasks.



Pad Masking

Pad masking is a technique used to prevent the transformer model from attending to padded tokens. Padded tokens are tokens that are added to the end of a sequence to make all sequences the same length. Padded tokens do not have any meaning, so it is important to prevent the transformer model from attending to them. Pad masking is done by creating a mask that indicates which tokens in the sequence are padded. The mask is then applied to the attention weights of the transformer model. This prevents the transformer model from attending to padded tokens and ensures that the model only attends to meaningful tokens.

Looakahead mask

Lookahead masking is done by creating a mask that indicates which tokens in the sequence are future tokens. The mask is then applied to the attention weights of the transformer model. This prevents the transformer model from attending to future tokens and ensures that the model can only attend to tokens that have already been processed. this masking is only for the decoder part, as we will be generating/predicting words one by one

Self-attention

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

self-attention is a key concept that makes the Transformer model different from other natural language processing models. Self-attention allows the Transformer model to attend to different parts of the same input sequence, which helps the model to learn the relationships between words in the sequence.

The self-attention matrix is a square matrix that contains the attention weights for each word in the input sequence. The attention weights represent the importance of each word to the other words in the sequence. . The Transformer model will then use the masked self-attention matrix to attend to the different parts of the input sequence.

When we take an input sequence like {23, 38, 1, 0, 0}, the self-attention matrix, also known as the attention weight matrix, takes

the shape of 5x5. For each word within the sequence, we compute its attention concerning the other words in the same sentence—hence the term 'self-attention.' We must handle padding for those tokens that represent empty positions in the sequence. In the code, I accomplished this by assigning highly negative values to the padded tokens when passing them through the softmax function. This ensures that they have minimal influence in the attention mechanism."

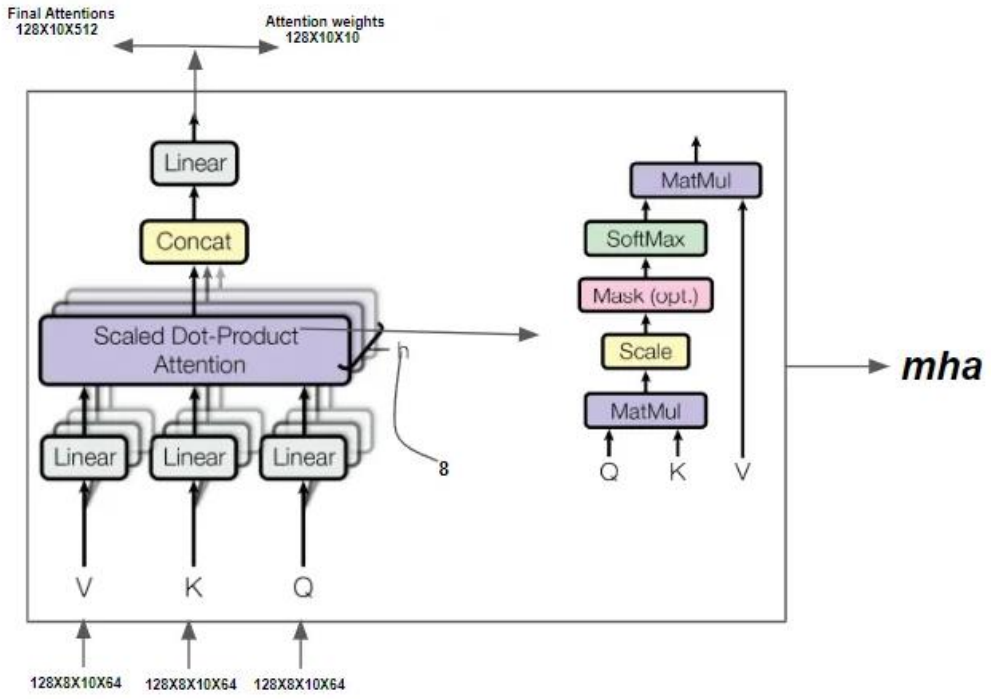
calculates scaled dot-product attention between query (q), key (k), and value (v) tensors. Scaled dot-product attention is a crucial component of the Transformer architecture.

Args:
q: Query tensor with shape (..., seq_len_q, depth)
k: Key tensor with shape (..., seq_len_k, depth)
v: Value tensor with shape (..., seq_len_v, depth_v)
mask: Float tensor with shape broadcastable to (..., seq_len_q, seq_len_k). Defaults to None.

Returns:
output: Attention-
weighted output tensor with shape (..., seq_len_q, depth_v)
attention_weights: Attention weights tensor with shape (..., seq_len_q, seq_len_k)

Note This Q, K, and V are nothing but encoded values of input into the lower dimension(depth), input(eg. 10X100) will be passed to 3 dense layers each for Q, K, and V.

Multi head Attention layer



Multi-head attention is a powerful technique that allows the Transformer model to learn to attend to different aspects of the input sequence. This is important because different words in a sentence can have different meanings and relationships to each other. For example, the word "I" can refer to the speaker or the writer, and the word "a" can be an article or an indefinite pronoun. By attending to different parts of the sentence and learning the relationships between words, the Transformer model is able to better understand the meaning of the sentence.

Multi-head attention is a technique that calculates self-attention multiple times to extract multiple types of attention for each word in a sentence. In the original Transformer paper, 8 heads were used,This means that self-attention was calculated 8 times for each word in the input sequence. The output of each self-

attention calculation was an attention weight matrix of shape 5x5. The attention weight matrices from the 8 different heads were then concatenated, which resulted in an attention weight tensor of shape 8x5x5

In the code, you will get to see how amazingly this's been implemented (multiple times self-attention calculation)

---

if we have version of the text that focuses solely on the reshaping example:

Input: (64, 10, 512) -> (BATCH, words, embedding)

Dense Layer: (64, 10, 512) -> (64, 10, 512) (three dense layers for Q, K, V encodings)

Reshaping: (64, 10, 512) -> (64, 8, 10, 64) -> (BATCH, attention head, #words, encode)

Self-Attention: (64, 8, 10, 64) -> (64, 8, 10, 10) (attention weights)

Concatenating: (64, 8, 10, 10) and (64, 8, 10, 64) -> (64, 10, 512)

Note embedding dimension must be divisible by no. of heads and always embedding_dimn/heads => encodin_dimn(here 64)

So Multihead attention will be one of the transformer layers that's why I have inherited it from tf.keras.layers, and the multi-head attention weight matrix will be created during the forward pass via call(). Custom layer attributes include 3 dense layers each for Q, V, and K encodings, 1 final dense layer for converting output shape to match it with input shape, num_heads, and d_model.

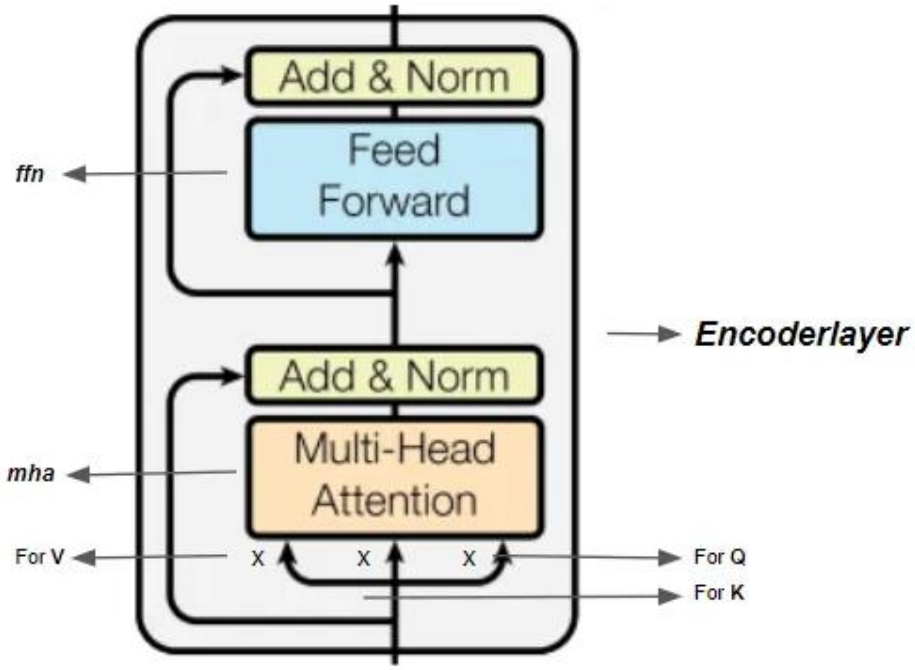
One thing to keep in mind d_model must be divisible by num_head to achieve multi-head attention with this code implementation. In forward pass (calling call()) we can see v, k, q, and mask as parameters these three values are nothing but input itself, let's say its (input embedded batch) dimensions are 128 X 10 X 512 (BATCH, input_seq_length, embedding dimn) for each. After passing these three through their corresponding dense layers we will get (128 X 10 X 512) shaped matrix (as no. of dense units : d_model(512)) for each, Now split_head() will do the job, it will reshape the (128 X 10 X 512) — > (128 X 8 X 10 X 64)

---

what we mean by (128 X 8 X 10 X 64) represents. '128': batch_size, '8': number of attention heads, '10': input sentence length, '64': encoding of embedding vector of each word into lower dimension (as depth), like I said before Q, K, V are the encodings in a lower dimension. So now we have Q, K, V of shapes (128 X 8 X 10 X 64), passing Q, K, V values for self_attention calculation (nothing changed there, same transformation equation) we will get (128 X 8 X 10 X 10) as self-attention weights and Attention matrix as (128 X 8 X 10 X 64). Now we need to convert its shape equal to the input matrix, so these '8' attention heads will be concatenated over the last dimension (64), and after the concatenation, the shape will be (128 X 10 X 512) and then pass it into the Dense layer(512 units). Final output matrix is of size (128 X 10 X 512) and attention weight matrix is of size (128 X 8 X 10 X 10). See in multi head self-attention calculation we haven't used any fancy algorithm that's the beauty of the transformer.

We have just understood the First layer of the transformer, the Mulithead attention layer

ENCODER layer



The encoder layer in a transformer is a set of sublayers that are used to encode an input sequence into a sequence of hidden states. Each encoder layer consists of two sublayers:

Self-attention: This sublayer takes the input sequence as input and produces a sequence of attention weights and context vectors as output. The attention weights represent the importance of each word in the input sequence, and the context vectors represent the weighted sum of the input encodings. Feed-forward: This sublayer takes the output of the self-attention sublayer as input and produces a sequence of hidden states as output. The feed-forward sublayer is typically a two-layer neural network. The encoder layer is repeated multiple times in a transformer model. This allows the model to learn long-range dependencies in the input sequence.

self Multihead attention -> Residual+Norm -> Feed forward neural network -> Residual+Norm we will create this layer by inheriting TensorFlow tf.keras.Layer class as usual. The encoder layer has two components Multihead attention layer and feed forward neural network, and each component's output will be layer normalized with residual connection.

In my code
In the Encoder Layer, the forward pass, implemented in the call() method, involves the following steps:

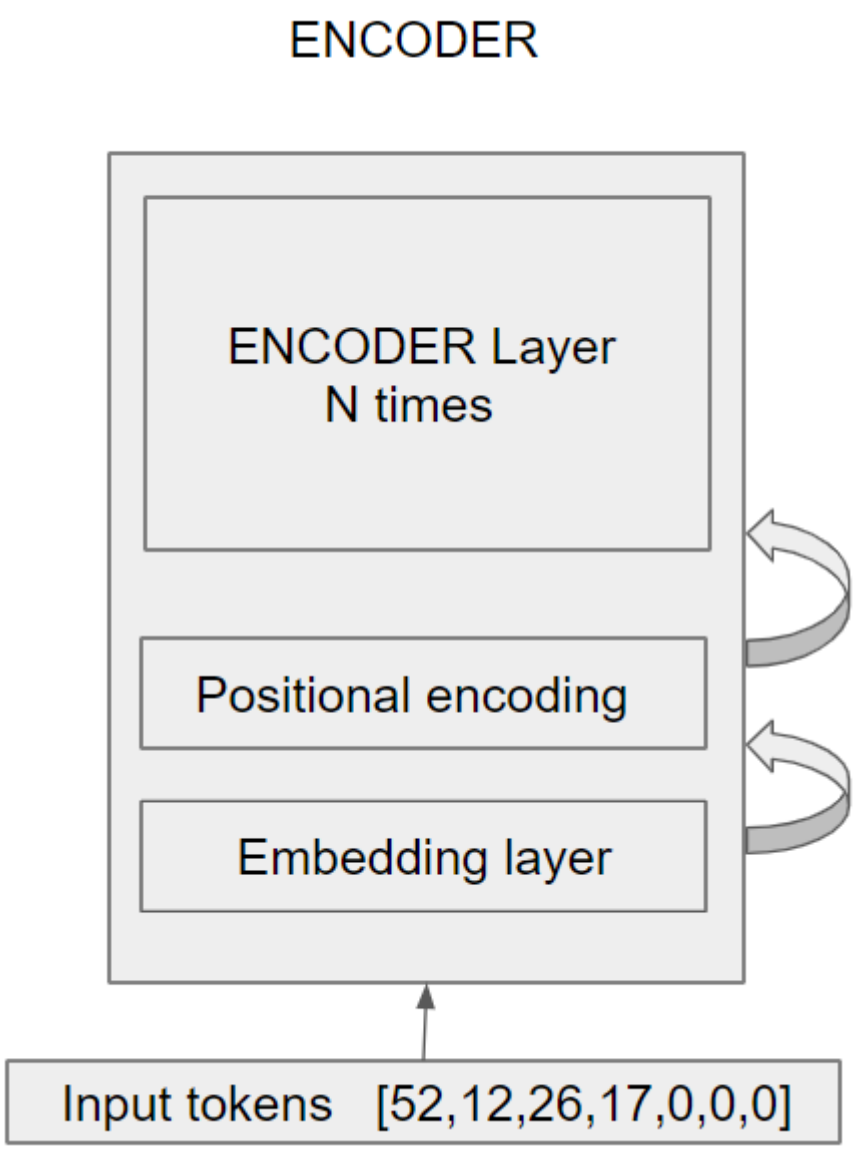
Multi-Head Self-Attention (mha): The input token embeddings are passed through the Multi-Head Self-Attention layer, resulting in an attention weight matrix and an attention matrix. The input is added to the output of the Multi-Head Self-Attention layer, which is a residual connection. Layer normalization is applied to this combination.

Point-Wise Feed-Forward Network (ffn): The output from the previous step is passed through a point-wise feed-forward neural network. This network applies non-linear transformations to the data.

Layer Normalization and Residual Connection: After the feed-forward network, the output is combined with the original input through a residual connection. This means the information from the original input is preserved. Layer normalization is applied again to the combination.

These steps make up the forward pass of an encoder layer in the Transformer model. The use of residual connections and layer normalization helps stabilize training and allows the model to learn complex relationships within the input sequences.

ENCODER



as you can see that Encoder in a transformer is nothing but a repetition of the encoder layer, the number of repetitions is

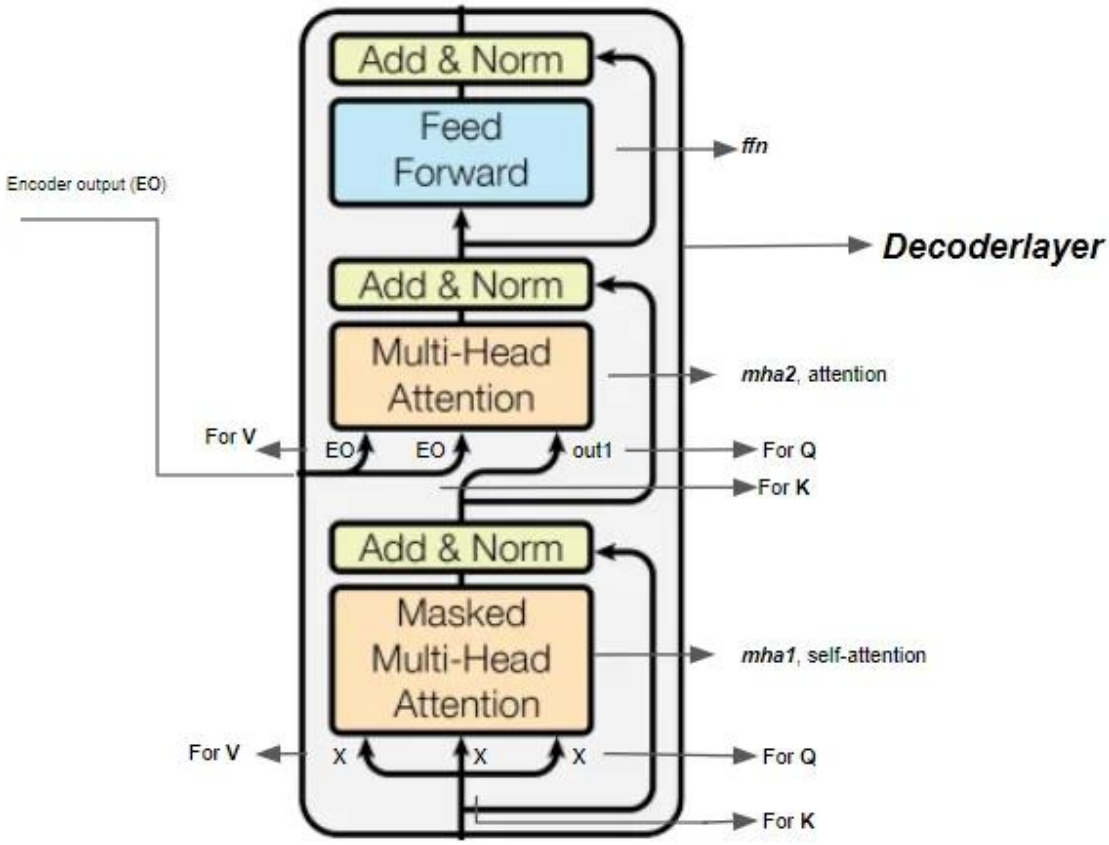
actually a hyperparameter and depends on the complexity of the problem you are solving, we using 2 repetitions.

we are creating embeddings and then adding position encodings and then just blindly calling encoder_layer (pay attention to the loop while setting self.enc_layers attribute). we inheriting tf.keras.layer class as usual, as the Encoder is also a layer of the transformer.

DECODER LAYER

The decoder layer in a transformer is a set of sublayers that are used to generate an output sequence from an input sequence. Each decoder layer consists of three sublayers:

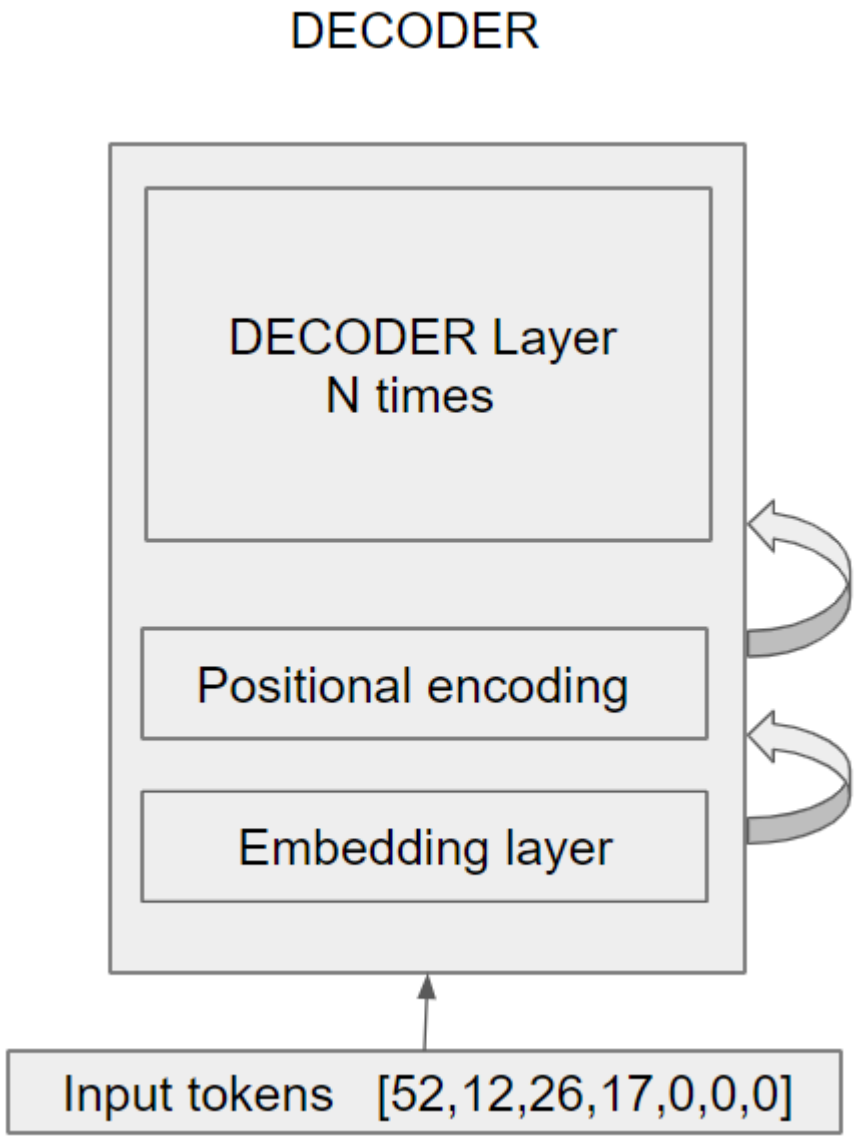
Masked self-attention: This sublayer takes the output of the previous decoder layer as input and produces a sequence of attention weights and context vectors as output. The attention weights represent the importance of each word in the output sequence so far, and the context vectors represent the weighted sum of the output encodings so far. Encoder-decoder attention: This sublayer takes the output of the previous decoder layer and the output of the encoder layer as input and produces a sequence of attention weights and context vectors as output. The attention weights represent the importance of each word in the input sequence, and the context vectors represent the weighted sum of the input encodings. Feed-forward: This sublayer takes the output of the encoder-decoder attention sublayer as input and produces a sequence of hidden states as output. The feed-forward sublayer is typically a two-layer neural network.



as you can see It is almost the same as the encoder layer, now it has mha1 and mha2. If you notice mha1 has input 'answers' with lookahead mask as we want to ignore self-attention calculation for future words and then same residual+layer normalization, mha2 has encoder output for V and K dense layers and mha1 output for Q dense layer, that's why mha2 doesn't calculate 'self-attention'. So in the equation (Q.transpose(K)).V, Q is from the decoder, and K and V are the encoder's output i.e. if the encoder input length is 7 and the decoder input length is 15 then the final attention weight matrix will be of shape 15 X 7 while the self-attention weight will be of shape 15 X 15, After getting output from mha2 doing residual + layer normalization and then finally flowing resultant through the feed-forward network, So the only difference between encoder and decoder layer is of 'mha2' and 'lookahead mask'

Decoder

Nothing but Repetition of decoder layers + positional encoder + embedding layer

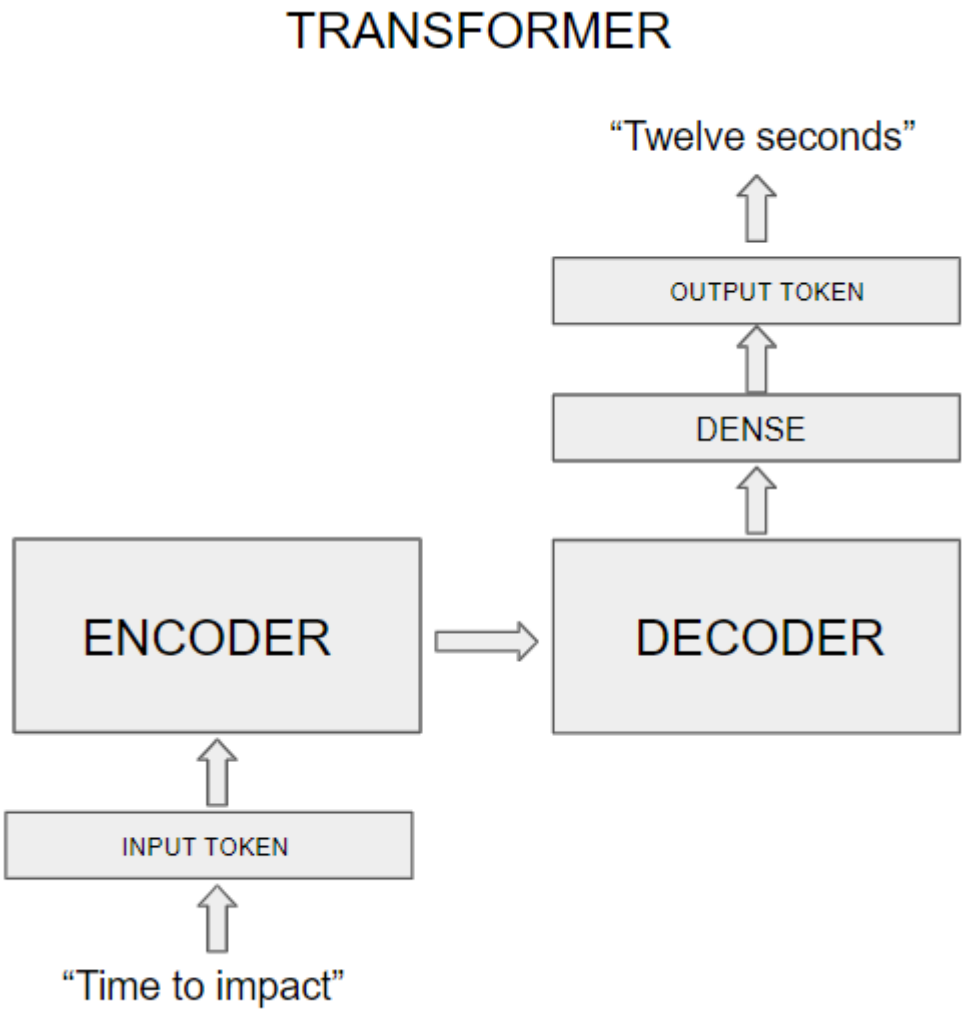


The Decoder class defines the decoder part of the Transformer model, which includes multiple decoder layers.

creating embeddings of ‘answers’ and then adding position encodings and then just blindly calling decoder_layer. we are

inheriting tf.keras.layer class as usual, as the decoder is also a layer of the transformer. One thing to keep in mind each repetition of the decoder layer will be given encoder output(to mha2), and self-attention and standard_attention weights returned by each decoder layer(from mha1 and mha2) will be saved into attention_weights dict, we will be using this for plotting attentions! finally, the output matrix has returned for word prediction.

TRANSFORMER



Encoder call --> decoder call --> ***dense layer*** --> word tokens, the final dense layer will of '*answer*' vocab size, 128 X 10 X 512 shaped

tensor as the decoder output passed through a final dense layer having '*answer*' vocab_size number of dense units, so final dense layer output tensor will be of shape 128 X 10 X 27360 (tokenizer_a.vocab_size +2), where '**128**':batch_size, '**10**': *answer input* seq length. I have inherited the Transformer class from tf.keras.Model as this will be our model.

Than we will **custom optimizer, loss function, training, and inference**

Custom learning rate

rate learning rate will be high and then after some epochs it will be decreasing ONLY

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

---

the learning rate is typically high at the beginning of training and then decreases over time. This is because the goal of training is to find the optimal values for the model's parameters, and a high learning rate allows the model to quickly explore the parameter space. However, as the model gets closer to the optimal values, a smaller learning rate is needed to avoid overshooting the optimal values.

The custom learning rate schedule allows for dynamic adjustment of the learning rate during training, which can be beneficial for convergence.

Custom loss function

A custom loss function that is the same as sparse categorical cross entropy but considers only non-padded values.

And Creating pad mask(encoder), pad mask(decoder), lookahead mask(decoder)

And Creating check point and Using gradient tape for getting derivatives of loss functions pplyiing to optimizer => BACKPROPAGATION

training the model for a specified number of epochs, displaying training progress with metrics, and saving checkpoints periodically. Checkpoints can be used to resume training or make predictions with a trained model

Results

Model1

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
input_vocab_size = tokenizer_q.vocab_size + 2
target_vocab_size = tokenizer_a.vocab_size + 2
dropout_rate = 0.1
```

```
Model: "transformer_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_1 (Encoder)         multiple                  4329216

 decoder_1 (Decoder)         multiple                  4563968

 dense_135 (Dense)           multiple                  3533052

=================================================================
Total params: 12426236 (47.40 MB)
Trainable params: 12426236 (47.40 MB)
Non-trainable params: 0 (0.00 Byte)
```

With this setting got **12M** trainable params, Epochs:**25** but wasn't learning very good , it's obvious as transformers are massive models, they are designed to be trained with large trainable params for perfect predictions

```
[ ] inp_sentence = "i am doing great"
    a, b = evaluate(inp_sentence)
```

```
[ ] for i in a[1:]:
        print(tokenizer_a.decode([i]))
```

```
i
am
not
going
to
be
a
little
late
for
a
while
```

model-2

```
num_layers = 2
d_model = 256
dff = 512
num_heads = 8
input_vocab_size = tokenizer_q.vocab_size + 2
target_vocab_size = tokenizer_a.vocab_size + 2
dropout_rate = 0.1
```

```
Model: "transformer_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_2 (Encoder)         multiple                  8100096

 decoder_2 (Decoder)         multiple                  8602112

 dense_168 (Dense)           multiple                  7047968

=================================================================
Total params: 23750176 (90.60 MB)
Trainable params: 23750176 (90.60 MB)
Non-trainable params: 0 (0.00 Byte)
```

Despite having only 24 million trainable parameters, the model's performance improved significantly after 350 epochs of training. Although the loss and accuracy metrics weren't particularly impressive until the 350th epoch, the model surprisingly generated excellent responses throughout the entire training process. This suggests that the model was able to learn meaningful representations from the data even with a relatively small number of parameters.

function for evaluating the model's output for a given input sentence. It takes an input sentence, a model, and tokenizers for questions and answers.

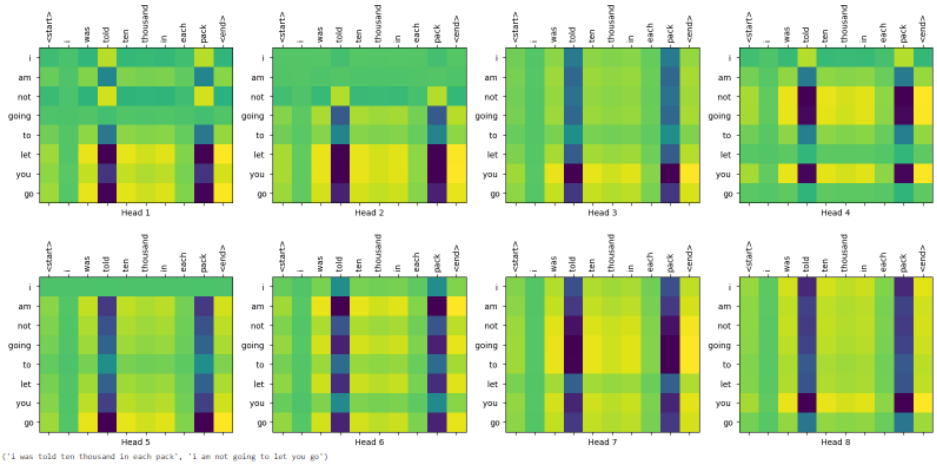
then

evaluation function performs sequence generation by iteratively predicting each word in the sequence. It keeps adding predicted words to the decoder input until either the maximum sequence length is reached, or the end token is predicted. It returns the generated sequence and the attention weights, which can be useful for analyzing how the model generates the output.

This function is designed to take an input sentence, use the model to generate a response

```
inp_sentence = "i was told ten thousand in each pack"
reply(inp_sentence, transformer,  tokenizer_q, tokenizer_a, "decoder_layer2_block2")
```

```
=============
Got end token
=============
Input: i was told ten thousand in each pack
Predicted translation: i am not going to let you go
```



('i was told ten thousand in each pack', 'i am not going to let you go')

about

used to calculate the BLEU (Bilingual Evaluation Understudy) score for machine translation evaluation. It compares the generated response from the model with the actual reference answer to assess the quality of translation.

it selects a subset of test questions and answers, then uses the reply function to generate responses for the input questions. It calculates the BLEU score by comparing the generated translation with the actual reference answer for each example.

```
=============
Input: jeanne let me introduce the king is half brother the dogged lord dunois
Predicted translation: then lord have given my lord is name in rome
Actual: then lord dunois show me the way to the other side of the river
=============
Got end token
=============
Input: father martineau but i do not see him as a candidate
Predicted translation: could there have been anyone else
Actual: could there have been anyone else
=============
Got end token
=============
Input: i have not read you your rights
Predicted translation: would you mind saying that into your bag
Actual: would you mind saying that into your bag
=============
Got end token
=============
Input: is this a good spot
Predicted translation: i am not sure look at it  time is it  time to look at it
Actual: i am not burying him here
=============
Got end token
=============
```