*Transformer architecture*

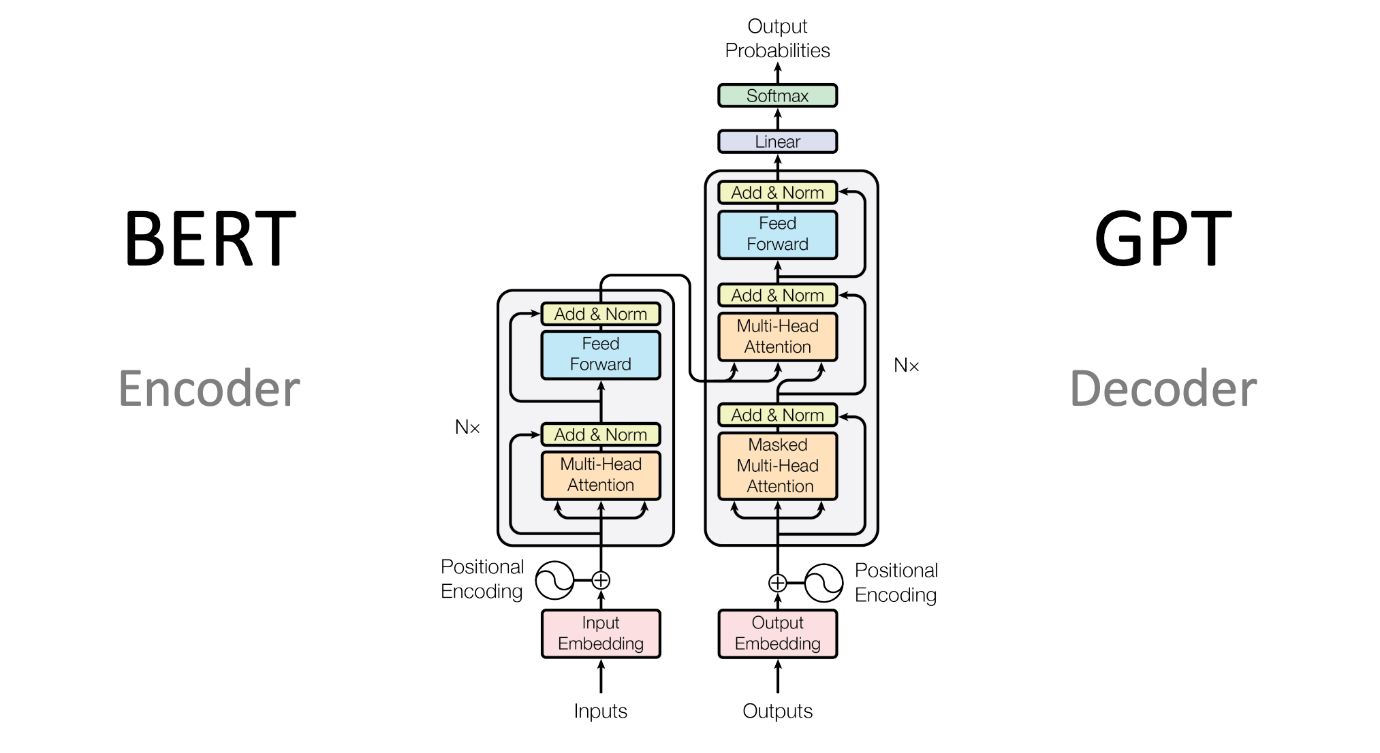


Fig. Transformer Architecture

It consists of 2 module Encoder and Decoder.

1. Encoder:

It is composed of stack of N identical layers. Each layer consists of 2 sub-layers. First, multi head self-attention and second, position wise feed forward neural network.

1. Decoder: The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence

## Step by step explanation of how Transformer works

1. **Tokenization**: Machine-learning models work with numbers, not words or text. So before passing texts into the model to process, you must first tokenize the words using Tokenizers. This converts the words into numbers, with each number representing a position in a dictionary of all the possible words that the model can work with. So, each word has been matched to a token ID
2. **Input Embedding**: Feed the input tokens to embedding layer. Embedding algorithm will convert it to the vector form which is numerical representation of that word. So, Each token is mapped into a vector. That is how each word maps to continuous representation.

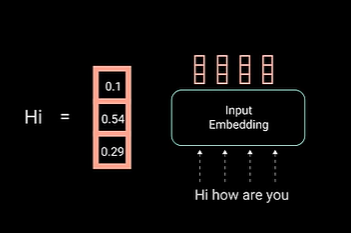


Fig. Embedding Vector

1. **Positional Encoding:** Add positional encoding into the input embedding because transformer does not have recurrence state like RNN so need to add positional information about the input words. This is done using positional encoding algorithm. Other method is to use sin and cosine function.

For every Odd timestamp word, create a vector using COS function. And For every Even timestamp word, create a vector using SIN function. And add those vectors to the corresponding input embeddings vectors. This gives the network information on positions of each vectors.

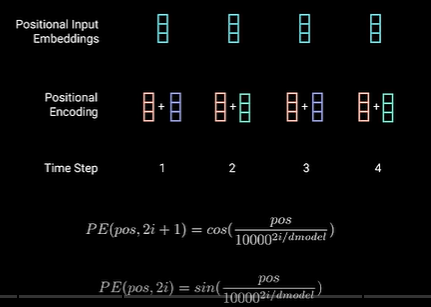
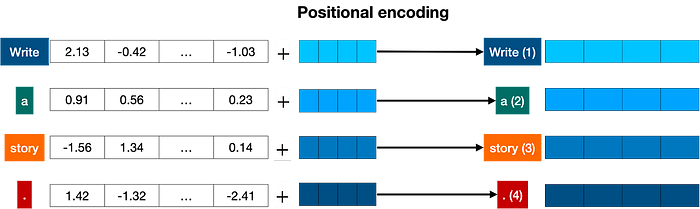


Fig – Positional Encoding Algorithm

Note:-> Once we have the vectors corresponding to each of the tokens in the sentence, the next step is to turn all these into one vector to process. The most common way to turn a bunch of vectors into one vector is to add them, component-wise. That means, we add each coordinate separately. For example, if the vectors (of length 2) are [1,2], and [3,4], their corresponding sum is [1+3, 2+4], which equals [4, 6]. This can work, but there’s a small caveat. Addition is commutative, meaning that if you add the same numbers in a different order, you get the same result. In that case, the sentence “I’m not sad, I’m happy” and the sentence “I’m not happy, I’m sad”, will result in the same vector, given that they have the same words, except in different order. This is not good. Therefore, we must come up with some method that will give us a different vector for the two sentences. Several methods work, and we’ll go with one of them: positional encoding. Positional encoding consists of adding a sequence of predefined vectors to the embedding vectors of the words. This ensures we get a unique vector for every sentence, and sentences with the same words in different order will be assigned different vectors. In the example below, the vectors corresponding to the words “Write”, “a”, “story”, and “.” become the modified vectors that carry information about their position, labeled “Write (1)”, “a (2)”, “story (3)”, and “. (4)”.



*Positional encoding adds a positional vector to each word, in order to keep track of the positions of the words.*

1. **Encoder**:

It is composed of stack of N identical layers. Each layer consists of 2 sub-layers. First, multi head self-attention and second, position wise feed forward neural network. There are also residual connections surrounding each of the two-sub module followed by layer normalization.

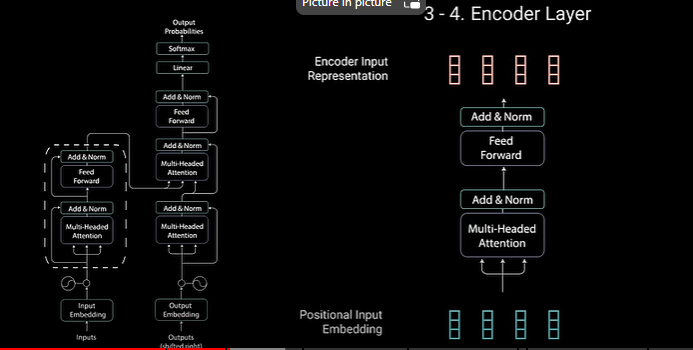


Fig. Encoder

1. Multi headed Self-Attention :
   1. Self attention:- It allows the model to associate each individual word to other word in the input sentence. For ex. “Hi How are you”. In this sentence model will learn that the word “you” is associated to “are” & “How”. And probably this is the question form.

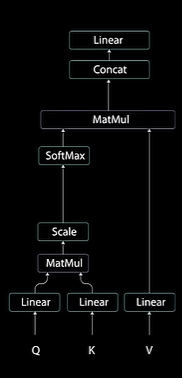
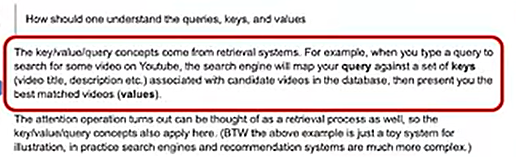


Fig. Self Attention Module

* To achieve self attention, we feed the positional embedding input to 3 distinct fully connected layers to create a Query, Key and Value vectors.



**Below steps are performed in self-attention block**

* + 1. Query and keys goes in dot product matrix multiplication to generate score matrix.

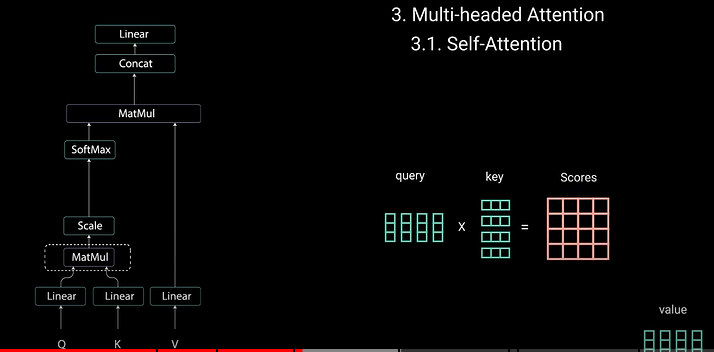
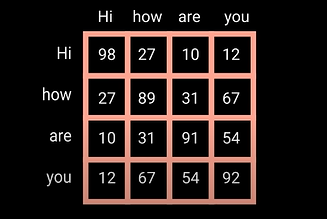
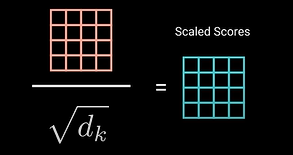


Fig. Score Matrix

Score matrix determines how much focus should each word have on other word. So each word will have a score for other word at a timestep. High the score more the focus, this is how Queries are mapped to Keys.



* + 1. Then score get divide by the dimension of key vector to scale it down . this is to allow more stable gradients as multiplying values in backpropagation can have exploding affects.



* + 1. Then we take the softmax of the scaled score to get the attention weights. This gives probability scores between 0 & 1. Higher score gets high and lower score gets depressed. This allows model to be confident on which word to attend to.

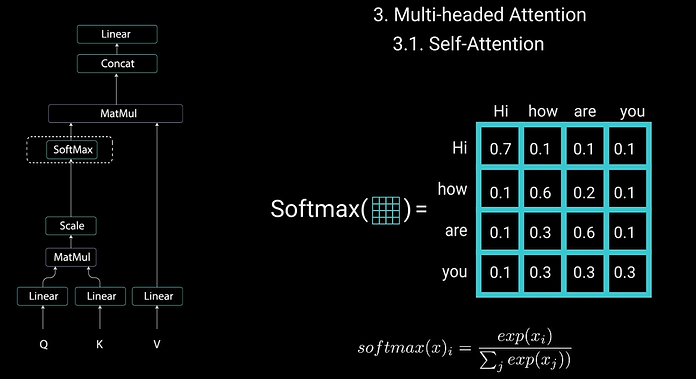


Fig. Attention Weights.

* + 1. Then multiply the attention weights and value vectors, to get the output vector. The higher softmax score will keep the value of the word the model learned. Lower score are drawn from irrelevant word.



Fig. Output Vector

* + 1. Then feed the output vector to linear layer to process.

To make it multi headed self-attention, split the Q, K and V into N-vectors before applying self-attention. Split vectors then again goes into the self-attention layers individually. Each self-attention is called a head.

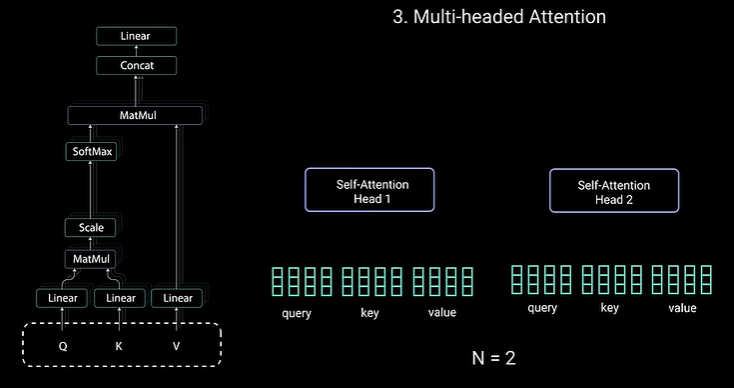


Fig – Multi-Head Self Attention

Each head gives a output vector that gets concatenated and send it to the linear layer for further processing. In theory, each head would learn something different so given the encoder module more representation power.

**Summary** of multi headed attention module : it’s a module that computes attention weights for the input and produces output vector with encoded information and how each word should attend to all other word in sequence.

**Next step, residual connection :** the original input is added to the multi headed attention output vector.

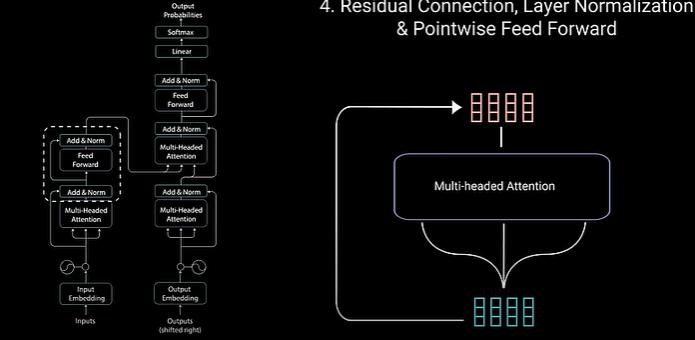


Fig . Residual Connection

These connections goes through Normalization layer. Normalized residual output is further passed through feed forward neural network. It consist of couple of linear layers having ReLU activation function in between that. Again, original input of layerNorm is added to the output of feed forward layer.

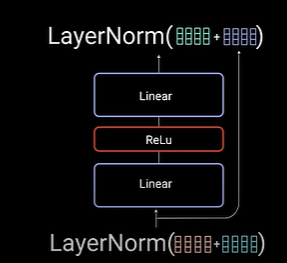


Fig. Normalization Layer

Note – residual connection helps the network train by allowing gradient to flow through them directly. Layered Norm layer stabilizes the network which results in producing the training time necessary. And a feed forward layer used to further process the output and give richer representation.

**Summary of Encoder layer**

All these operations are done to encode the input to the continuous representation with attention information. It helps the decoder to focus on information going in the decoding process. Likewise, we can stack the multiple encoder layers to further encode the information where each layer has the way to learn different attention weights. So, boosting the predictive power of Transformer architecture.



## Decoder

The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its “encoder-decoder attention” layer which helps the decoder focus on appropriate places in the input sequence:

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

## The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That’s the job of the final Linear layer which is followed by a Softmax Layer.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let’s assume that our model knows 10,000 unique English words (our model’s “output vocabulary”) that it’s learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.