Transformers

Iheb Gafsi*
INSAT Student
iheb.engineer@gmail.com

General Introduction:

The world of neural networks dedicated to NLP has gone into a lot of different stages, we've gone into simple embedding neural networks, to RNNs, to LSTMs and GRUs, but in 2017, Transformers were invented by Google and the "Attention is all you need" paper was published.

The key difference between Transformers and others is the attention mechanism that Transformers provide. Unlike the others, Transformers can "remember" everything, and parallelize the input and output processing.

Model Architecture:

The model contains essentially two big blocks which are the Encoder and the Decoder.
The Encoder's role is to basically maps the input into an abstract continuous vector representation of the inputs.
The Decoder on the other hand feeds the last output and that input representation step by step recurrently until the "end of sequence" token (typically "<end>") is generated.

Transformers look originally like the figure down below in Figure 1.

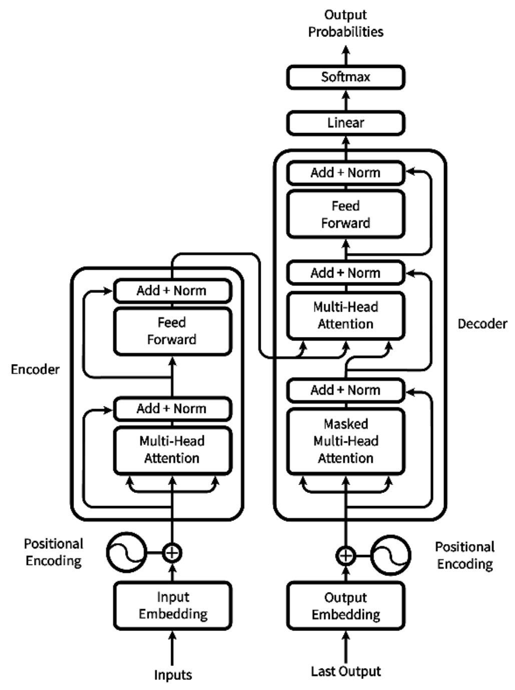

Figure 1

But I would like to illustrate them like in Figure 2

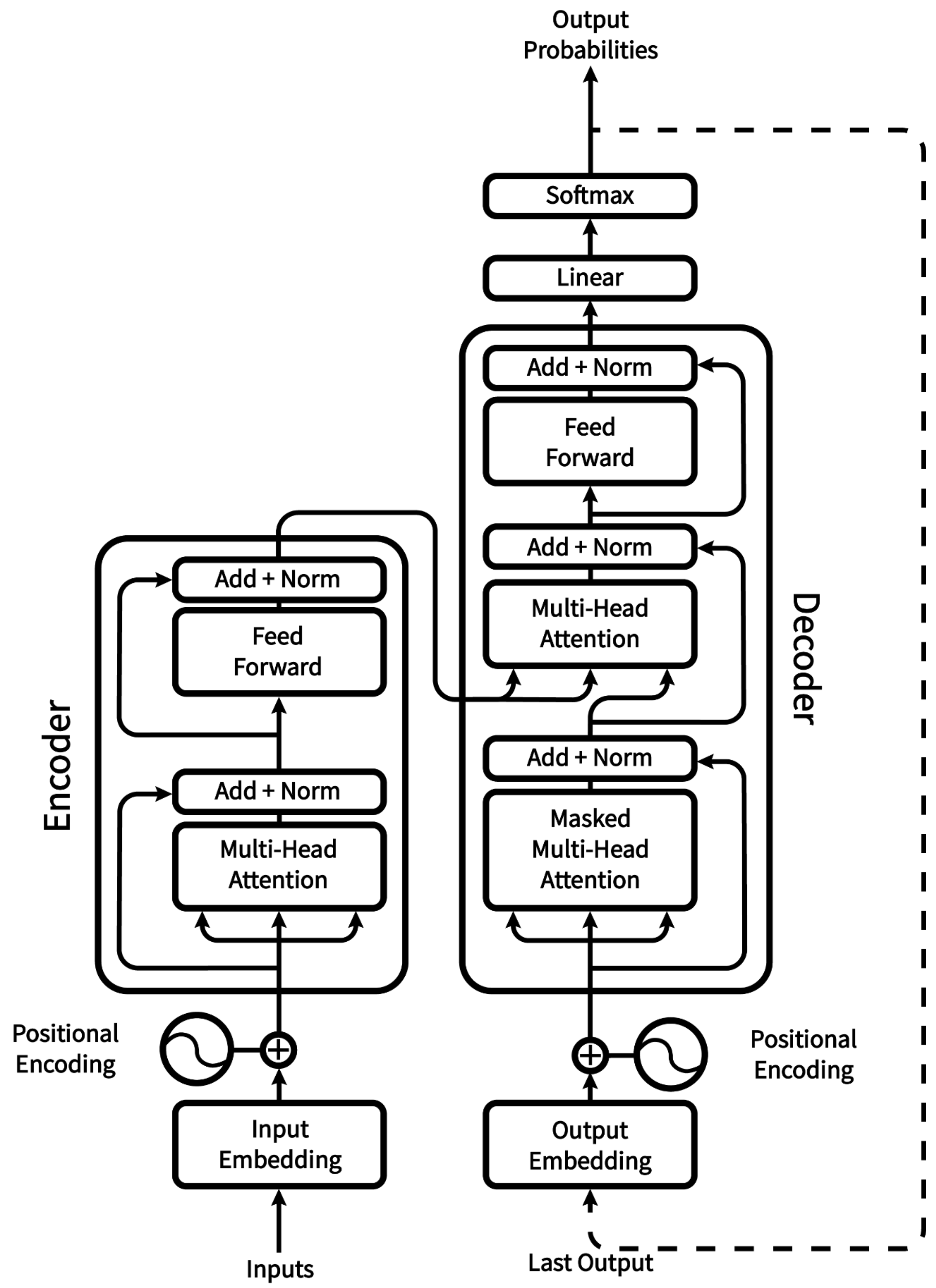

Figure 2

Input/Output Embedding:

This block associates every word to a b-dimensional vector of the dimensionality d_{model} which is typically equals to 512. All the embeddings are learned by the model during the training.

Figure 3 shows an example of Input/Output Embedding, we will consider $d_{model} = 5$.

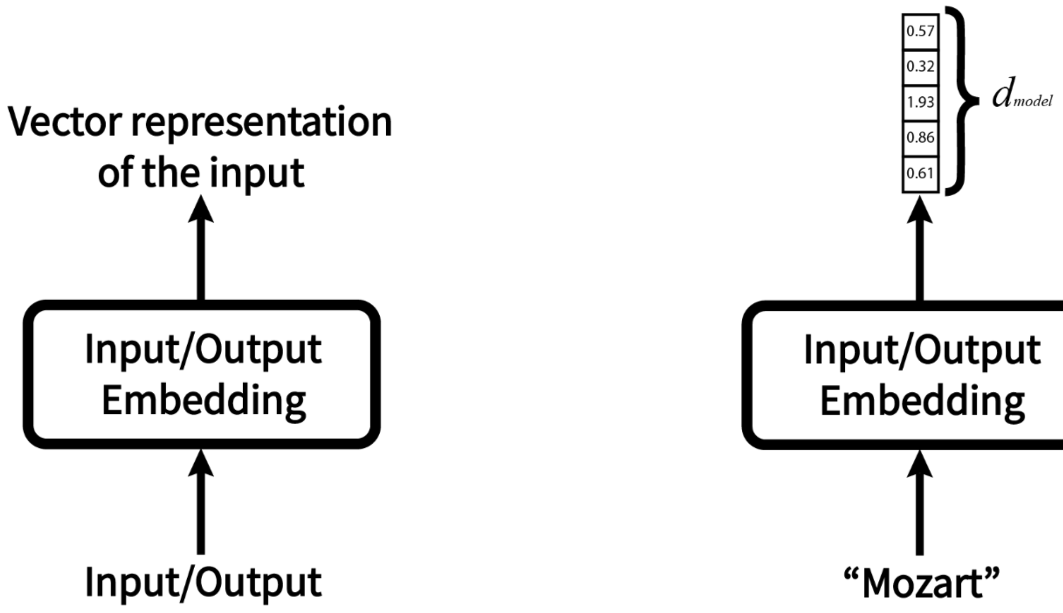

Figure 3

Positional Encoding:

Since our model isn't recurrent and to give meaning to the position of every element of the sequence, we add a position vector to every embedded element. There are several ways to implement this technique but there's a clever way to implement this. We will define a function PE that gives a unique vector to add to every embedded element.

$$\begin{cases} PE_{(pos,2i)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\ PE_{(pos,2i+1)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \end{cases}$$

Where:

- pos is the position of the element in the sequence $0 \leq pos \leq len(sequence) - 1$
- i is index of dimensionality of a position in the vector $0 \leq i \leq d_{model} - 1$

The sine and cosine functions give the orthogonality to the elements by their smoothness and unique values. It is hypothesized that this function helps the Transformer to capture the position information easily. There's an example in Figure 4.

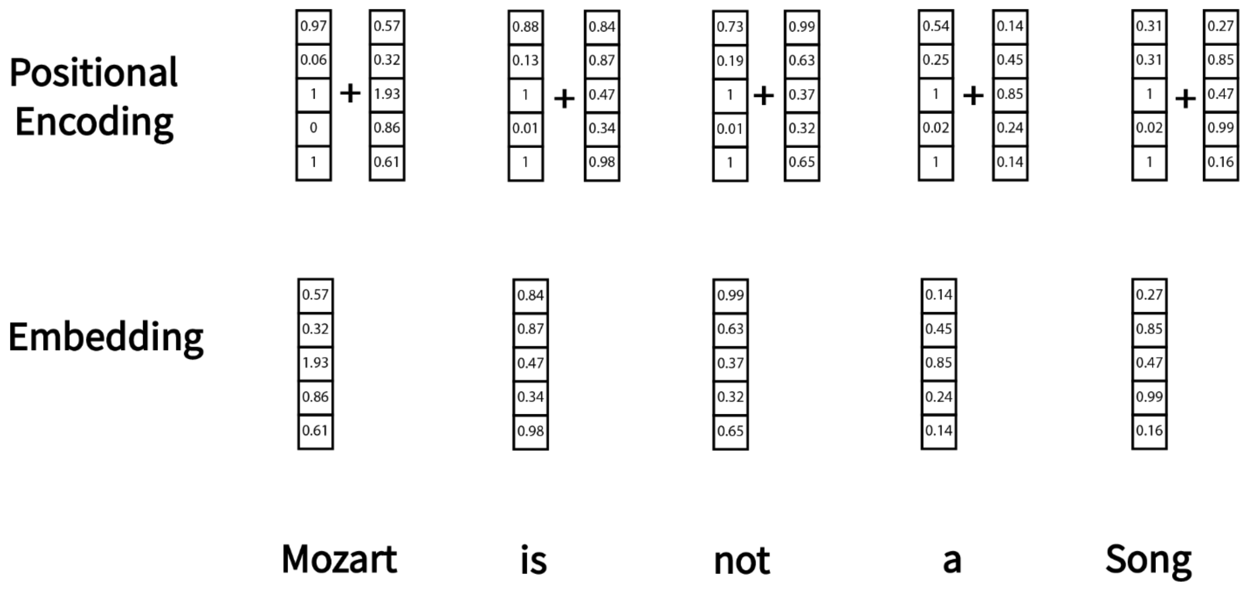

Figure 4

Here as you can see, "Mozart" is the first word and the second index of its position vector has the value of $PE_{(1,2\times2+1)} = 0.06$ (in this example we used 1 as the first index)

Encoder:

1. Multi-Head Attention:

1-1. Self-Attention:

The Self-Attention mechanism learns how to associate every word of the input to other words of the input.

Example: "Take care" should have "of" after it. Figure 5 shows what a Multi-Head Attention block looks like.

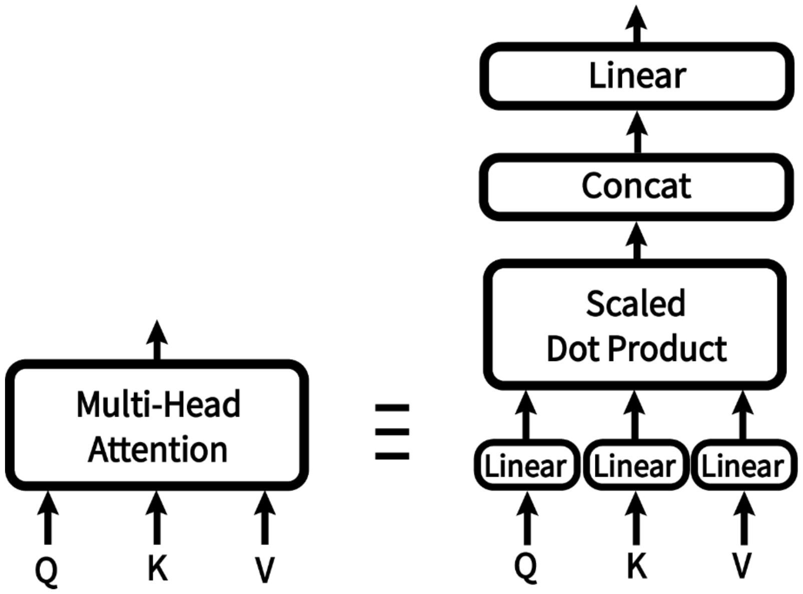

Figure 5

1-2. Queries, Keys, and Values Extraction:

The encoded vectors denoted P resulting from the positional encoding go directly to the Multi-Head Attention where we extract the queries Q, keys K, and values V matrices by multiplying them by learnable weight matrices. We get these vectors using the equations down below.
Q, K, and V have respectively the dimensions $d_q, d_k, and\ d_v$ (generally equal to 64 or 128) $h \times \left(d_q + d_k + d_v\right) = 3 \times d_{model}$
The example is also illustrated in Figure 6.

$$\begin{cases} Q = PW_Q \\ K = PW_K \\ V = PW_V \end{cases}$$

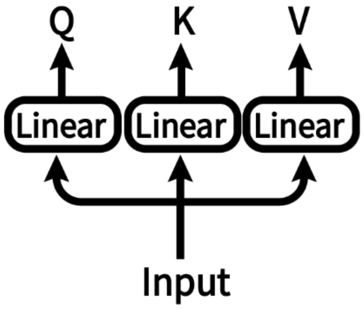

Figure 6

1-3. Scaled Dot Product Attention:

The Transposed Keys and Queries are multiplied to calculate the scores to determine how much focus should be put on words. Then we divide the score matrix by a factor of $\sqrt{d_k}$ to scale it down so that we can have more stable gradients as multiplying huge numbers can cause the vanishing problem. Then we apply a mask by multiplying our matrix with a 0s and 1s matrix to filter out what's not important, note that this step is optional. Just then, we apply the SoftMax function on the resulting matrix $Softmax(x_i) = \frac{x_i}{\sum_j x_j}$ and then we multiply our result with the Values vector to get the Attention vector. Then we get:

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

Figure 7 illustrates the steps to configure this scaled dot product attention:

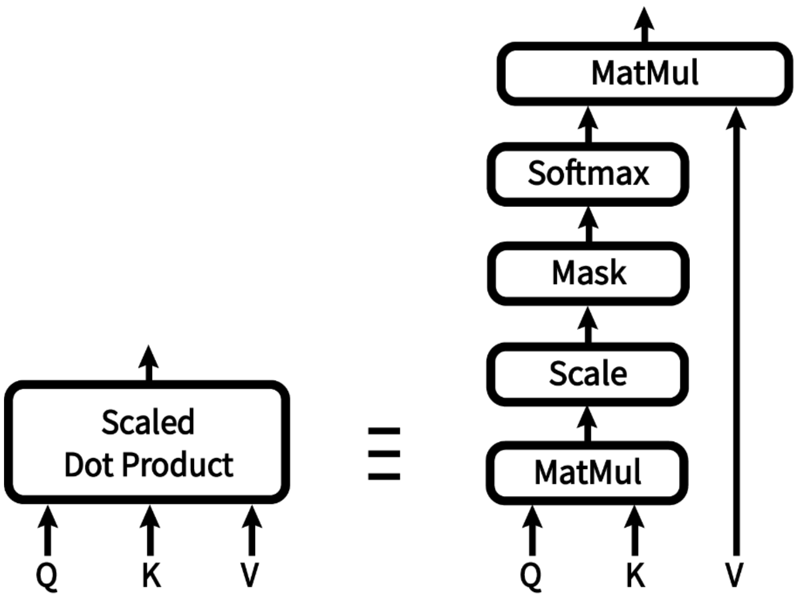

Figure 7

1-4. Concatenation:

What we were talking about was one single head attention, what we want to be doing is to do several ones with different dimensions, so we will be having h (typically 8) layers of Head Attention to get the resulting vectors as the Figure 8 shows. At the end the vectors of every Head Attention will be concatenated by length, then the final dimension of the resulting vector will be $h \times d$, d represents dimension of the resulting vector of an individual Head Attention, d is typically equal to d_k, which means the resulting vector is typically $8 \times 64 = 512 = d_{model}$

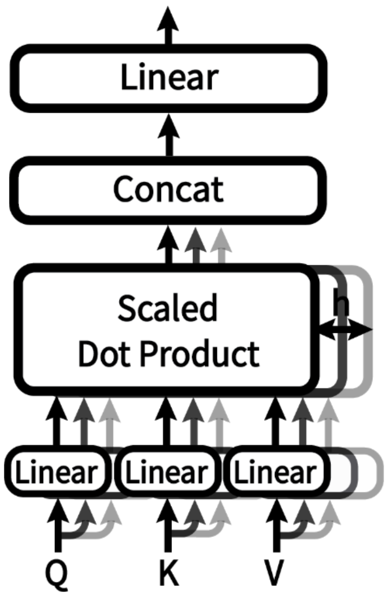

Figure 8

2. **Add + Norm:**
   - **Residual Addition:**
     Now we add the input and the output of the last cell which is the Multi-Head Attention. This allows our model to retain the original information and prevent the loss of the information during the transformation.
   - **Normalization:**
     To stabilize the training process, we normalize the values distribution and reduce the impact of scale differences. Therefore, we will eventually use
     $output = \gamma \frac{input - mean}{\sigma + \epsilon}$
     Where γ is a learnable coefficient, σ is the square root of the variance, and ϵ is a constant to prevent dividing by 0.
3. **Feed Forward:**
   Now what we just have to do is a simple Feed Forward Neural Network which is simply going to be two linear layers and a ReLU activation function in between as the Figure 9 shows.

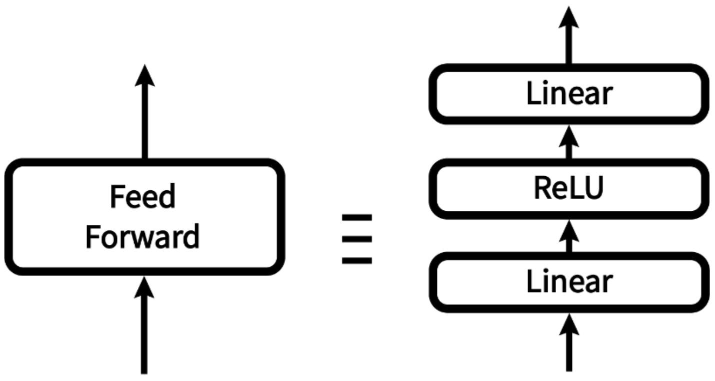

Figure 9

   As a result, we will have an output vector with dimensionality $d = d_{model}$ (typically) and the inner layer dimensionality d_{ff} typically equals to 2048
   $FFN(x) = ReLU(\mathrm{x}\mathrm{W}_1 + \mathrm{b}_1)W_2 + b_2$
   Note that $ReLU(x) = \max(0, x)$
4. **Note:**
   We can have N stacks (typically 6) of identical encoders to process information differently and combine them using averaging, concatenation, summation, weighted summation, but most commonly feeding the outputs into the inputs of the next layer.

Decoder:

1. **Output Embedding and Positional Encoding:**
   Just like the Input Embedding, the Output Embedding learns how to map every element of the sequence to a d_{model}-dimensional vector and the Positional Encoding gives the value for every position.

2. **Masked Multi-Head Attention:**

   This has the same functionality as other Multi-Head Attention cells with some differences. The Triangular Mask plays a crucial role to prevent leftward information flow to preserve the auto-regression property of the Transformer. This mask works on eliminating the focus on the future tokens so that the predicted token can only depend on the last tokens. So, we add $-\infty$ to the upper right triangle of the matrix to become 0 with the SoftMax function.

$$Softmax\left(\begin{bmatrix} 0.2 & 0.1 & 0.1 \\ 0.4 & 0.3 & 0.7 \\ 0.9 & 0.2 & 0.3 \end{bmatrix} + \begin{bmatrix} 0 & -inf & -inf \\ 0 & 0 & -inf \\ 0 & 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0.7 & 0.3 & 0 \\ 0.7 & 0.1 & 0.2 \end{bmatrix}$$

3. **Multi-Head Attention:**

   This Multi-Head Attention combines both the decoder's input and the encoder's output to determine which of the encoder's output to put focus on. The outputs of the encoder will be considered as queries and the others of the decoder as keys and values.
   Next, the resulting vector will be fed to a Feed Forward Neural Network to process the information.

4. **Linear and SoftMax:**

   The Linear Layer acts as a classifier that results a vector of n elements, n represents the vocabulary data size (like 10k), and then gets passed to a SoftMax layer that outputs the vector of probabilities for every word. Just like the encoder, we have N layers of the decoder that we can feed their outputs as inputs to the next layers.

Optimizers:

It is the best in the case of Transformers to use the Adam Optimizer with $\beta_1 = 0.9, \beta_2 = 0.98, \varepsilon = 10^{-9}$ with a learning rate that follows the formula:

$$l_{rate} = d_{model}^{-0.5}.\min(steps^{-0.5}, steps.warmupsteps^{-1.5})$$