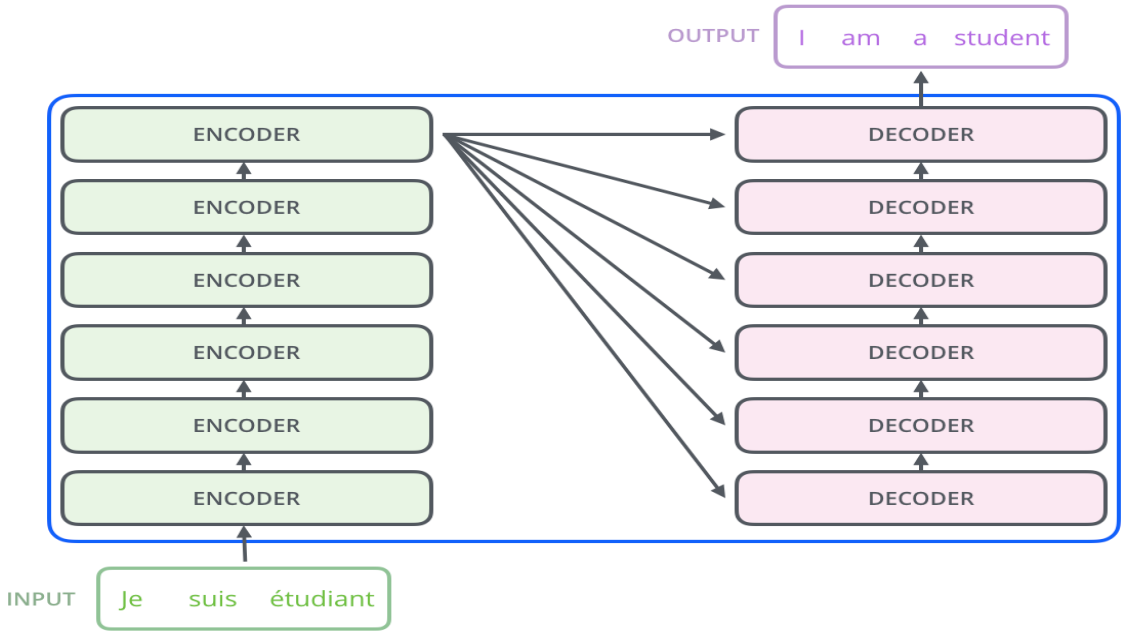
# Transformer Model Architecture

The Transformer is a model that is widely used most in NLP (Natural Language Processing). It is used in Text Recognition and conversion.

Below is the Overview of Transformer application How it works with respect to Text Conversion.

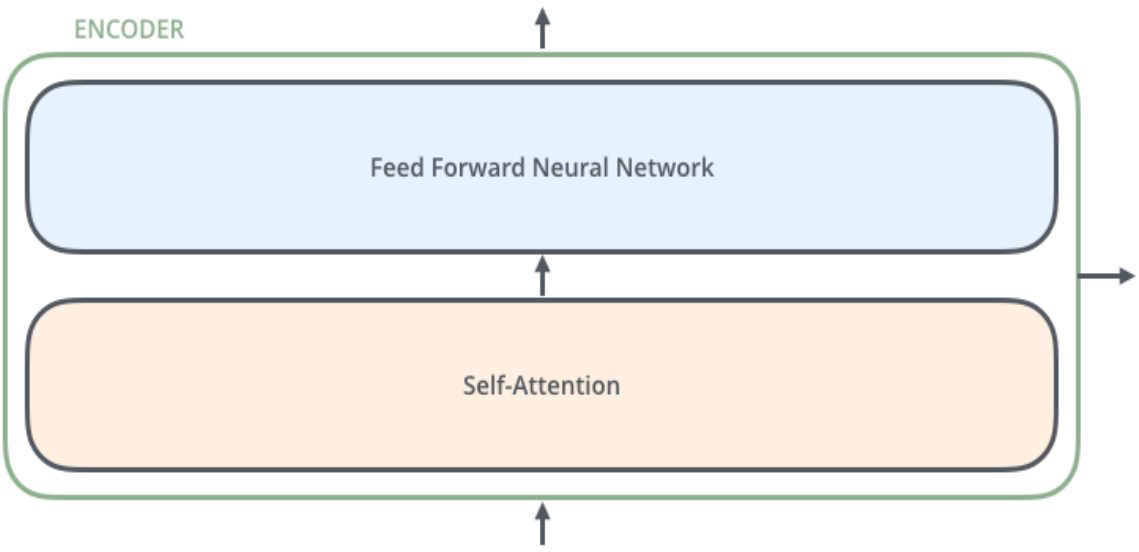

Here in the above image, input sequence is given to Transformer and Output sequence is generated.



Here, there are stack of Encoders and Decoders where input sequence is first given to the encoder and it is processed, and the topmost encoder output is the input for the Decoder. The Decoder will generate output as per the requirement.

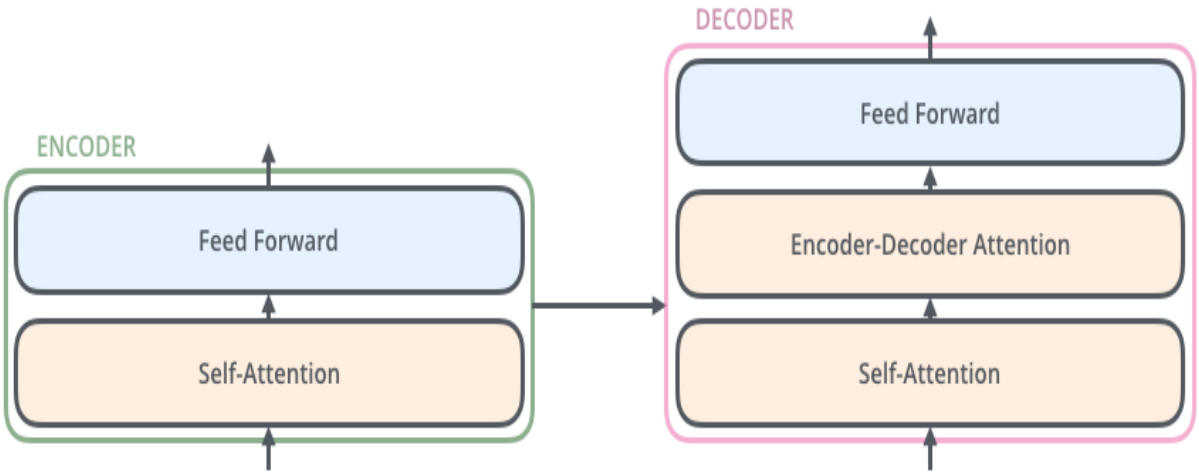
The Encoders are identical in structure and each one is broken into two layers.

1. Self - Attention
2. Feed Forward Network




The encoder input flows through the attention layer the helps to look at other words in the input sequence as it encodes specific word. The outputs of self-attention layer are fed to the FFNN.

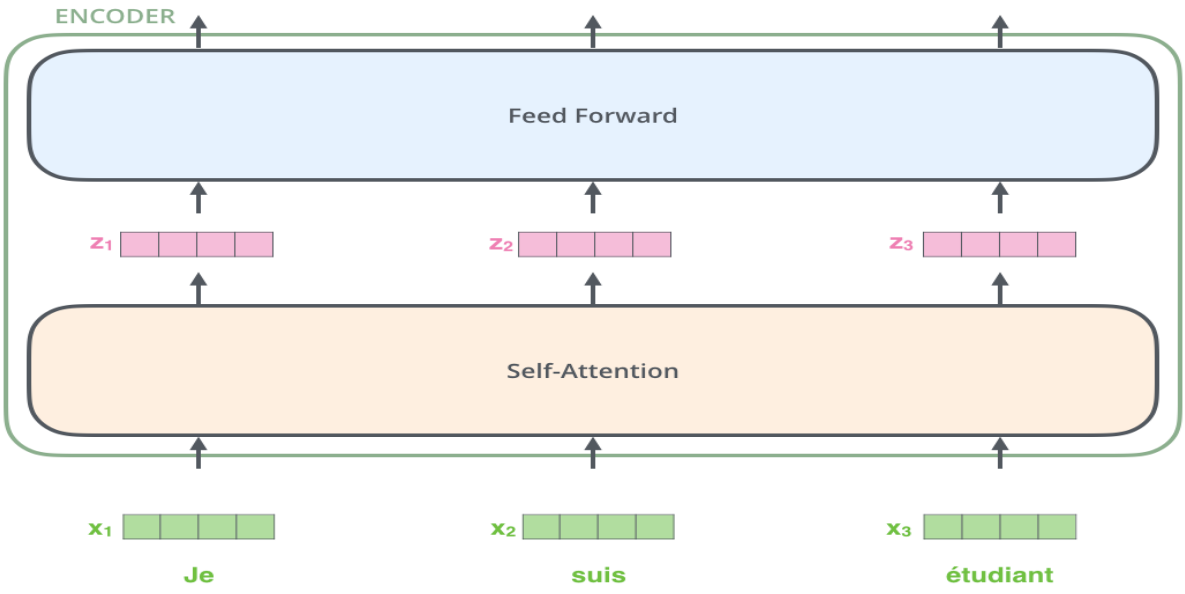
The Decoder have layers same alike encoder but between two layers there exists Encoder-Decoder attention.




Now let's get into detail how we train the algorithm and how input and output sequences are generated:

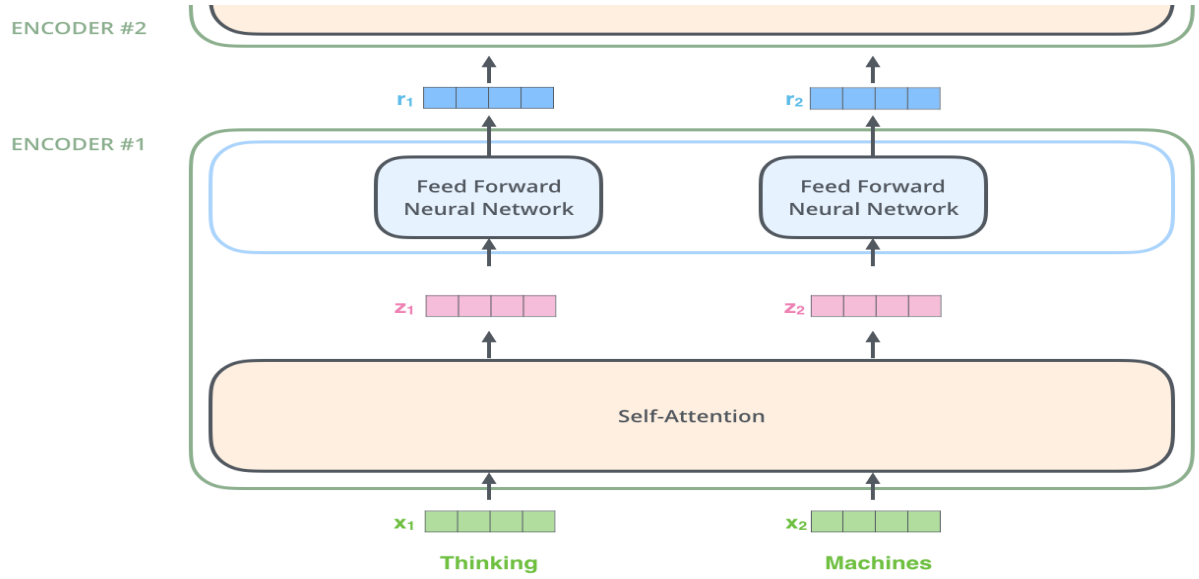
We start tuning each input word into a vector using Embedding. The embedding happens only in bottom most encoder. Each vector has size 512.

After embedding the words into input sequence, each of them flows through each of the two layers of encoder.



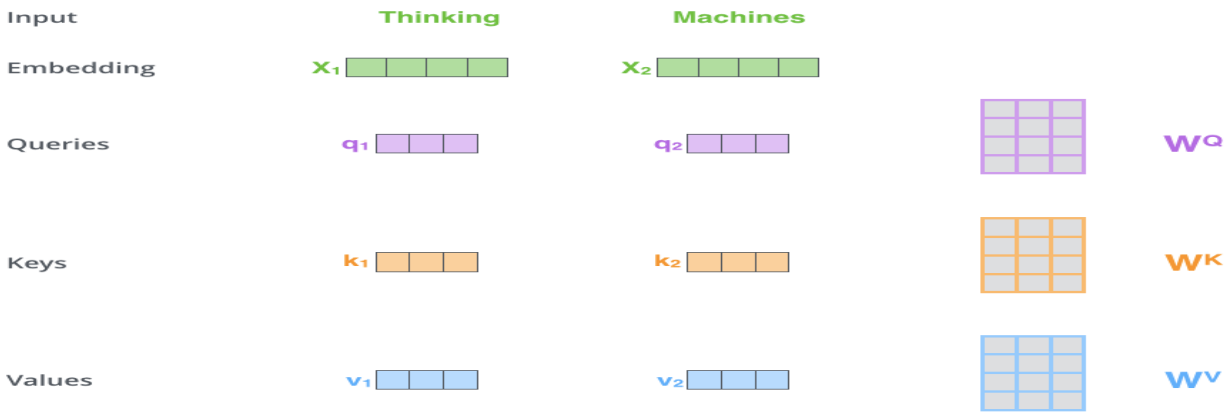
Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder.

There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies; however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.



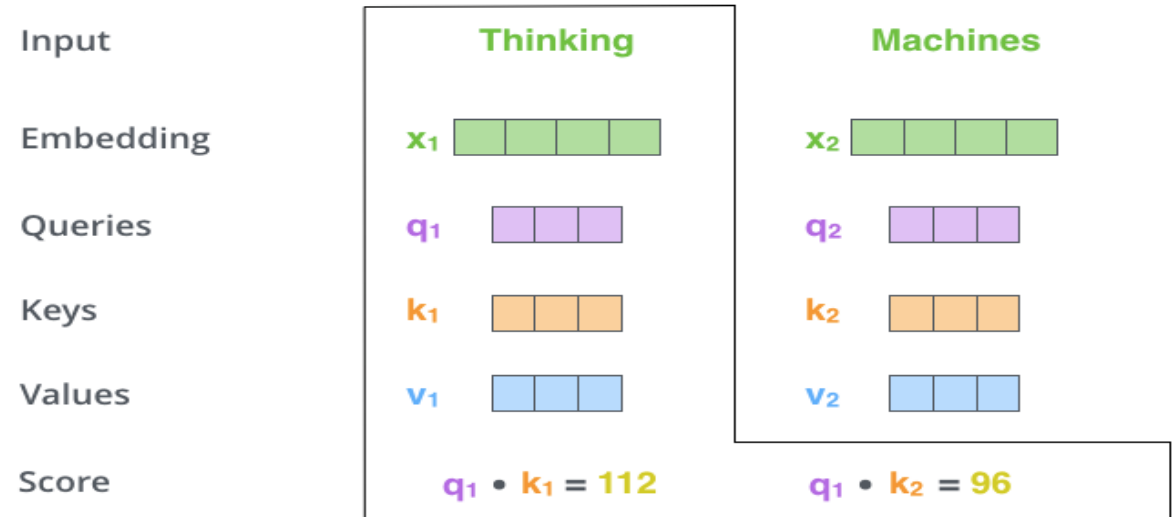

**Now we discuss Self-Attention in Details as follows:**

The first step to calculate self-attention is to create three vectors (Query Vector, Key Vector, Value Vector) from each word of encoding input vectors. These vectors are created by multiplying embedding with three matrices. These vectors are smaller in dimension of 64 when compared to embedded vectors.

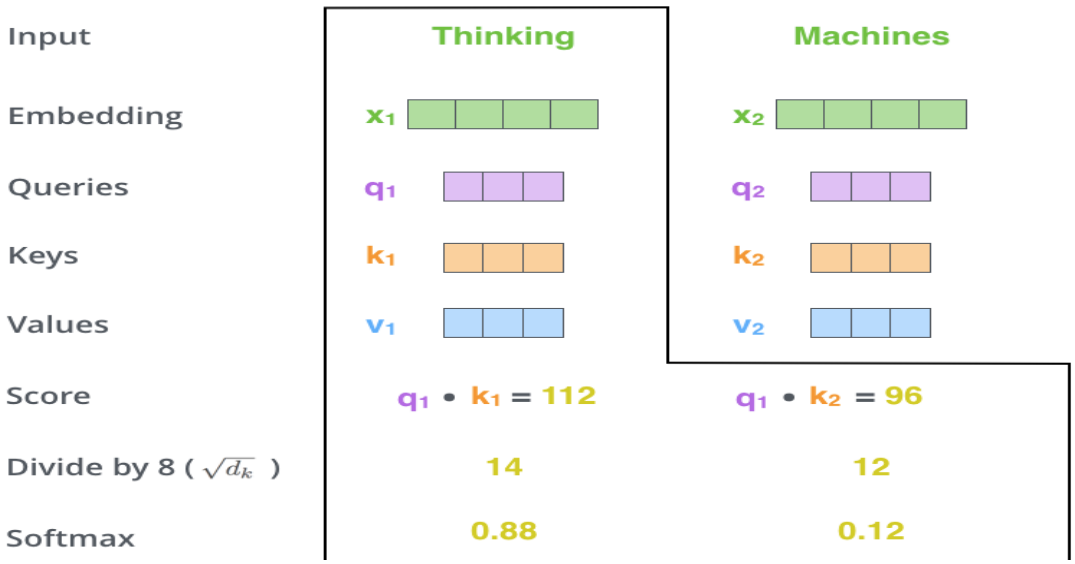


The second step is calculating self-attention is to calculate score. The score determines how much focus to place on the other parts of input sequence as we encode a word at certain position.

Here, the score is calculated by taking the dot product of Query Q and Key K.



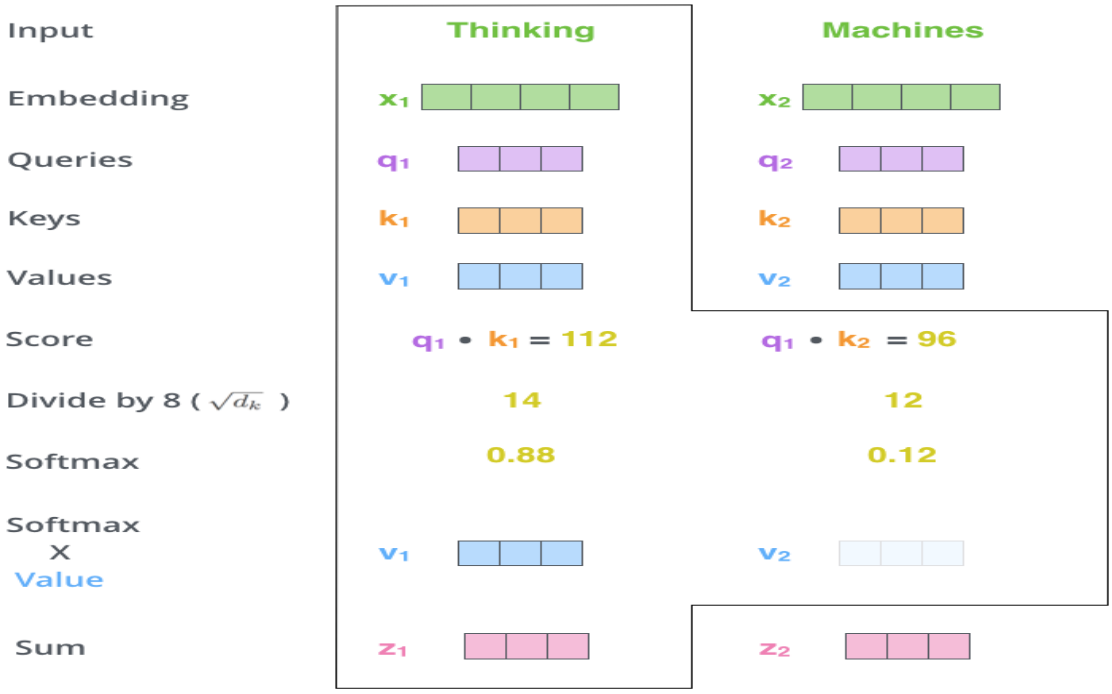

The third and fourth steps are to divide the score by 8(Square root of 64). This leads to having more stable gradients. The result then passes to SoftMax where it adds up value to 1.




The SoftMax score determines how much each word is expressed at this position.

The fifth step is to multiply each value vector by the softmax score.

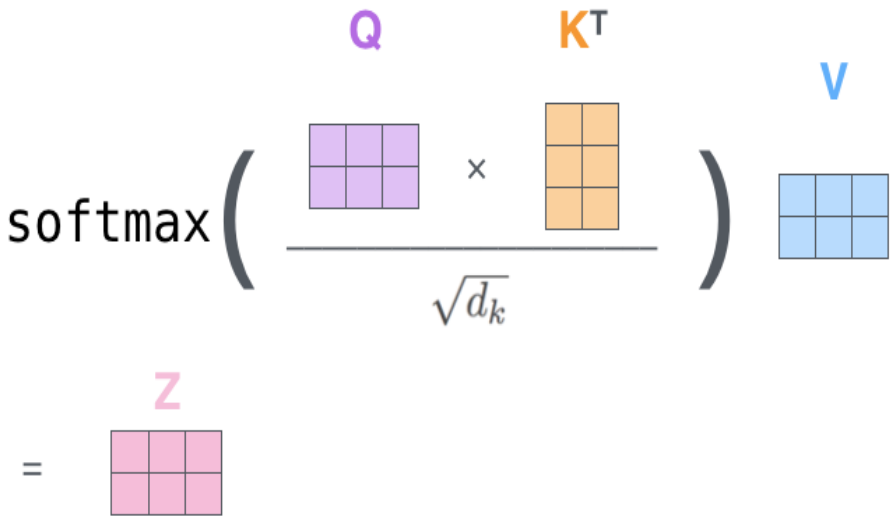
The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).




The self-attention calculation is also calculated using matrix form. It is explained as follows:

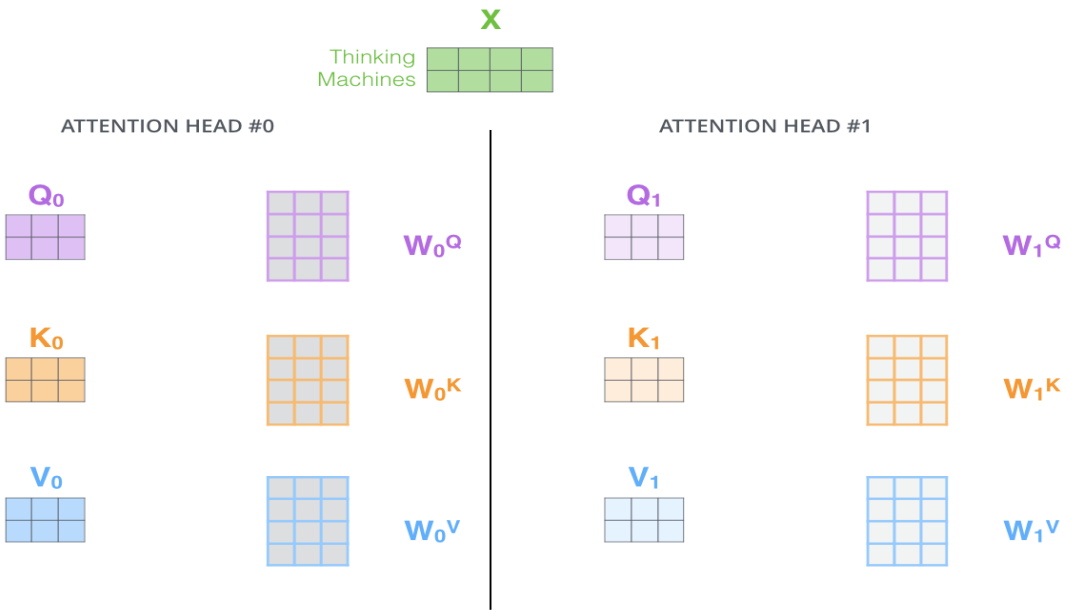
The first step is calculating Query, Key and Value matrices. It is calculated by packing our embeddings into a matrix X and multiplying it by the weight matrices we trained (WQ, WK, WV).

Self-attention matrix calculation formula is as follows:




Multiheaded attention:

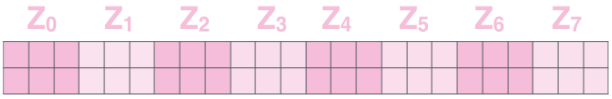
Here in multiheaded attention, it contains multiple sets of Query, Key and Value weight matrices.




If we apply above softmax calculation formula here where we end up with multiple Z matrices.

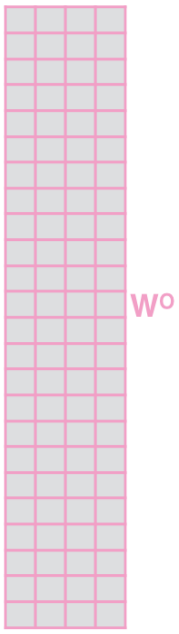
The FFNN is not expecting multiple matrices so we sum up all the Z matrices into single matrix.

1) Concatenate all the attention heads



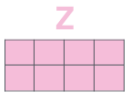

2) Multiply with a weight matrix $W^O$ that was trained jointly with the model

X




3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

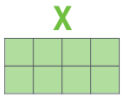
=



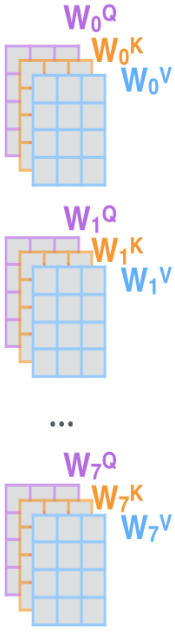

1) This is our input sentence*

Thinking Machines

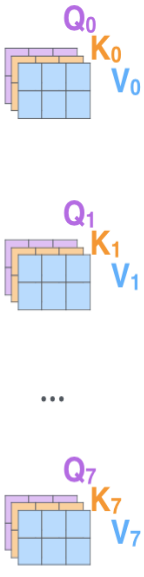
2) We embed each word*



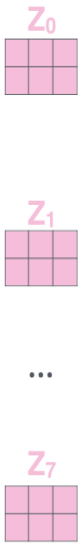

3) Split into 8 heads. We multiply X or R with weight matrices

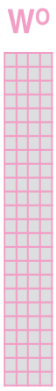


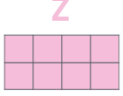

4) Calculate attention using the resulting Q/K/V matrices




5) Concatenate the resulting Z matrices, then multiply with weight matrix $W^O$ to produce the output of the layer



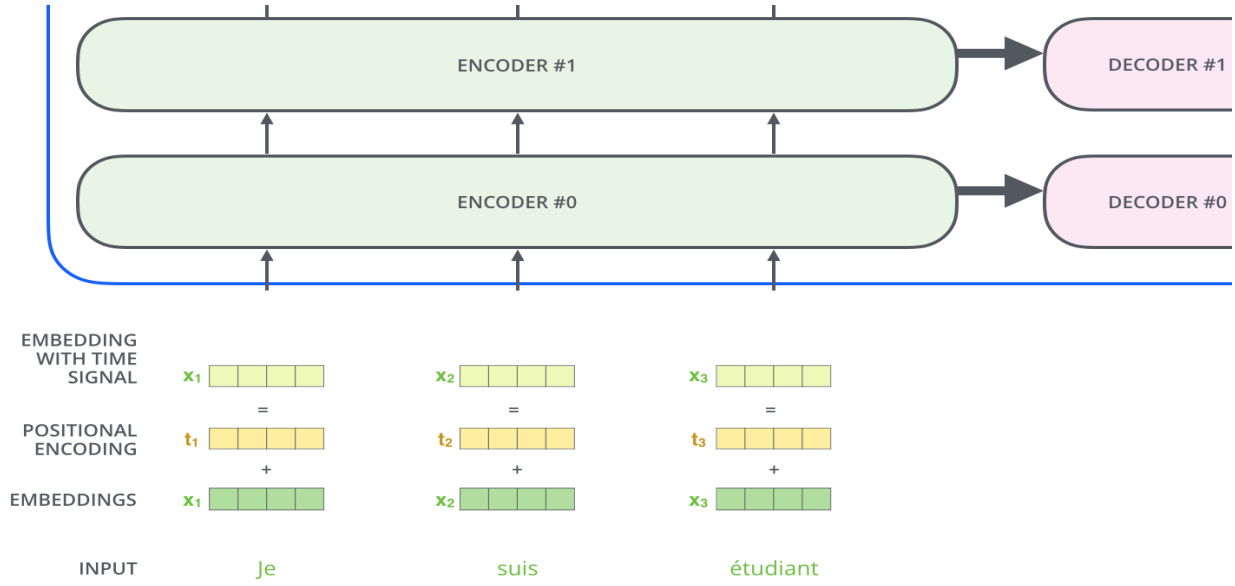



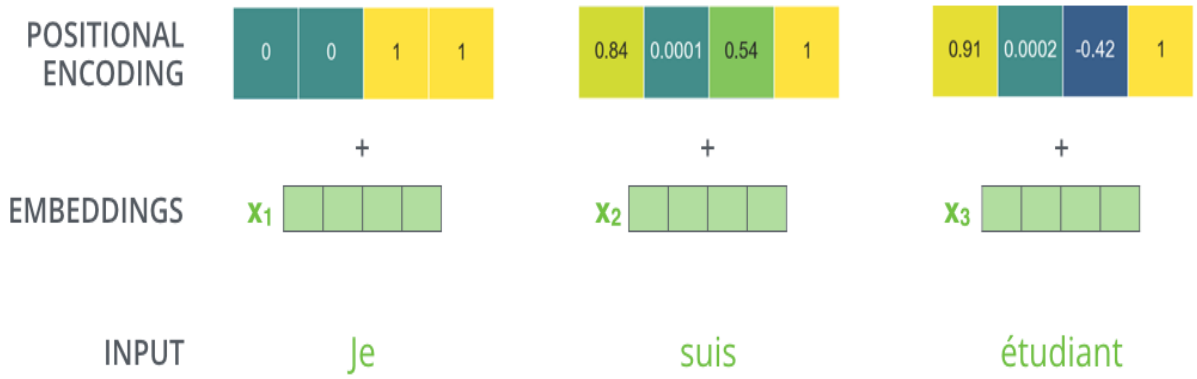

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

**Positional Encoding:**

The transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.

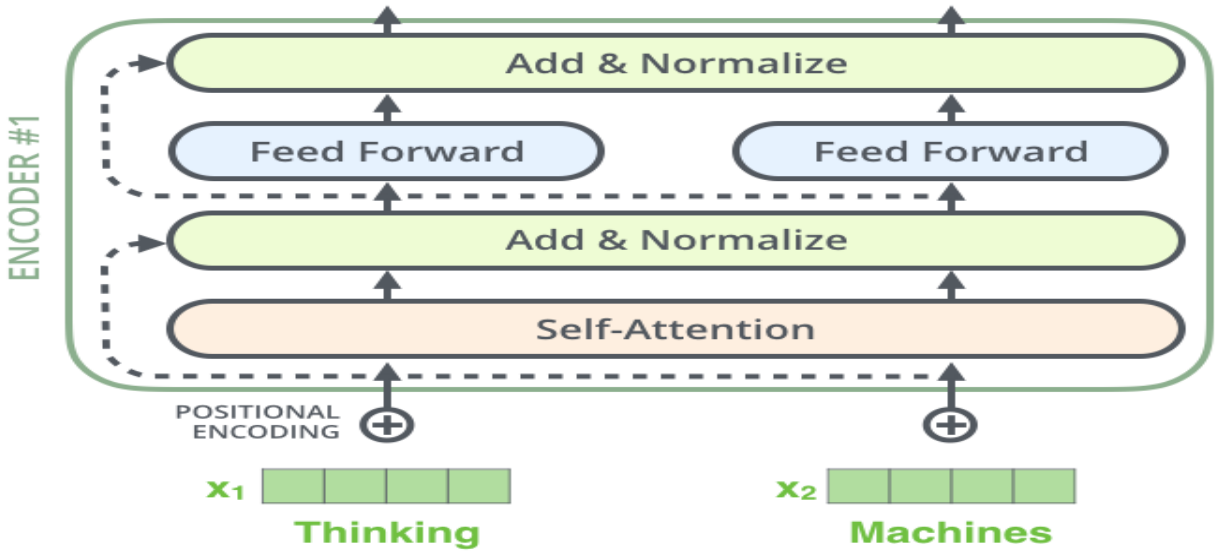



If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:

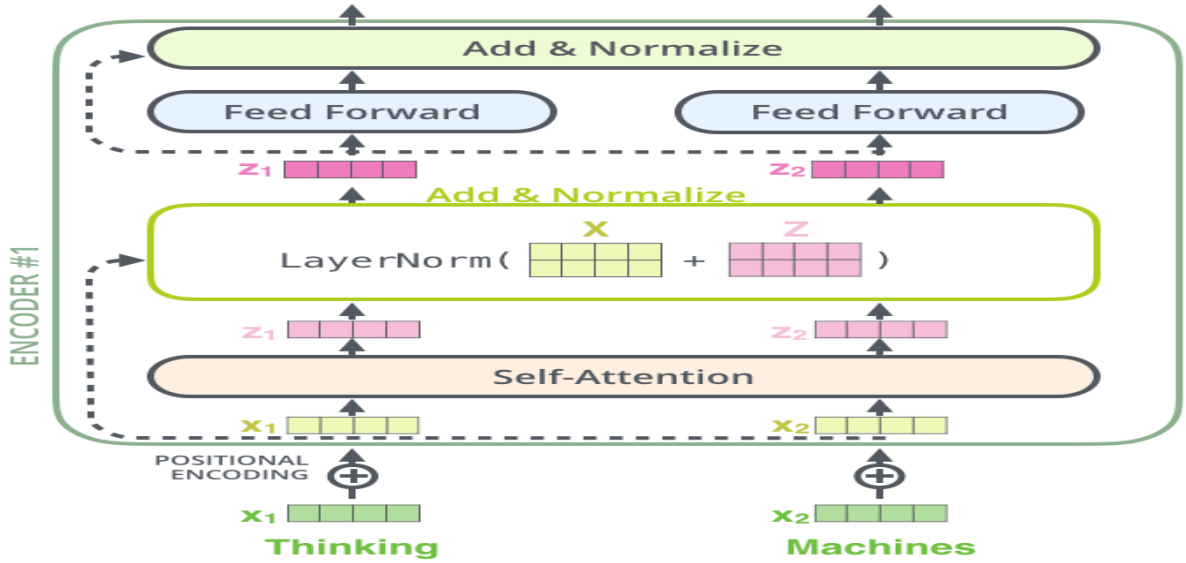


**Residuals:**

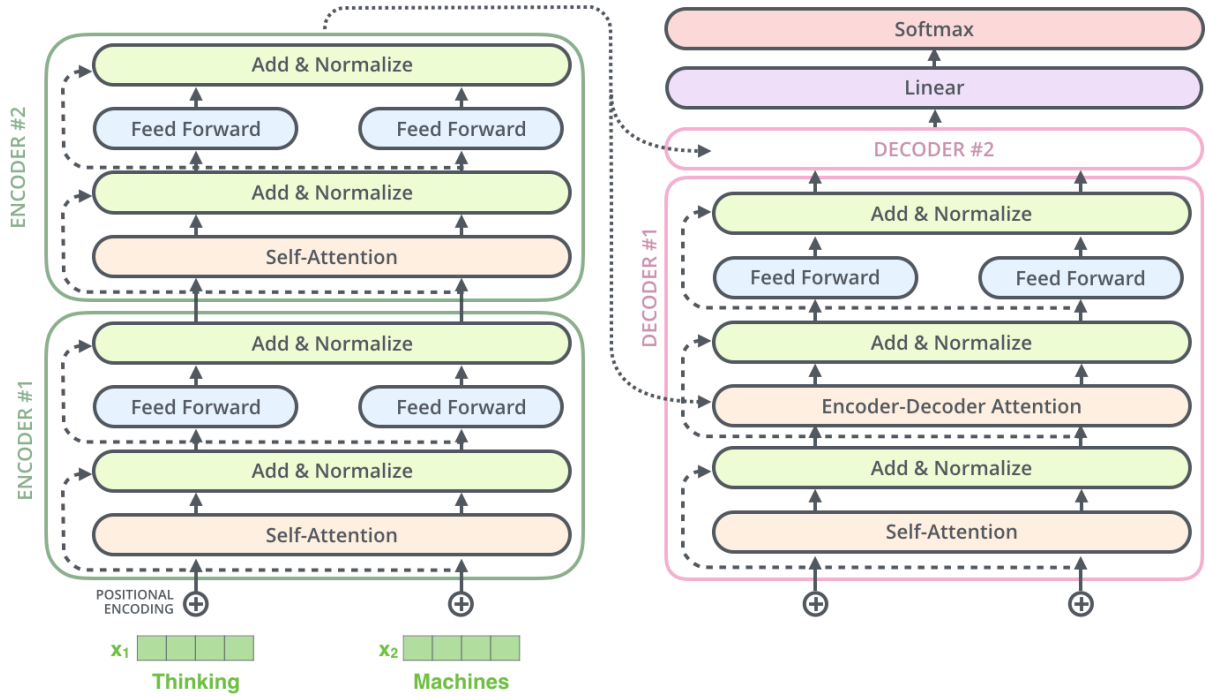
Each encoder has a residual connection around it and is followed by a layer normalization step.




If we're to visualize the vectors and the layer-norm operation associated with self-attention, it would look like this:

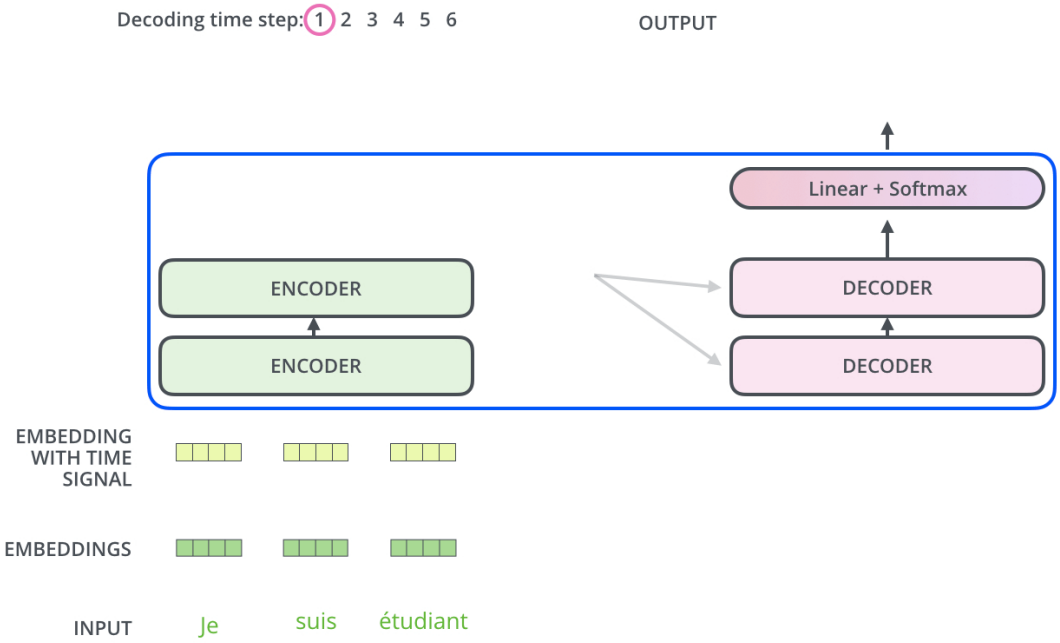


The Transformer of 2 stacked encoder and decoder is as follows:

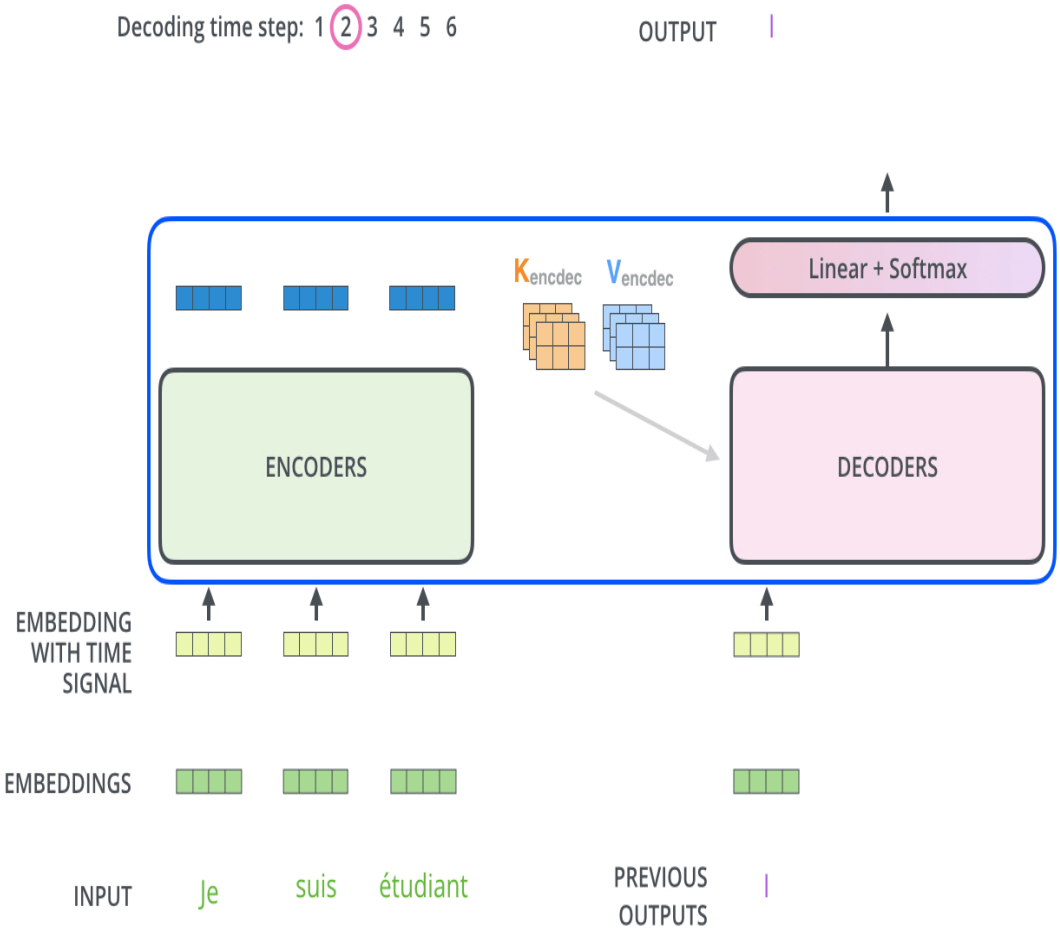



**Decoder:**

Decoder starts by processing 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:



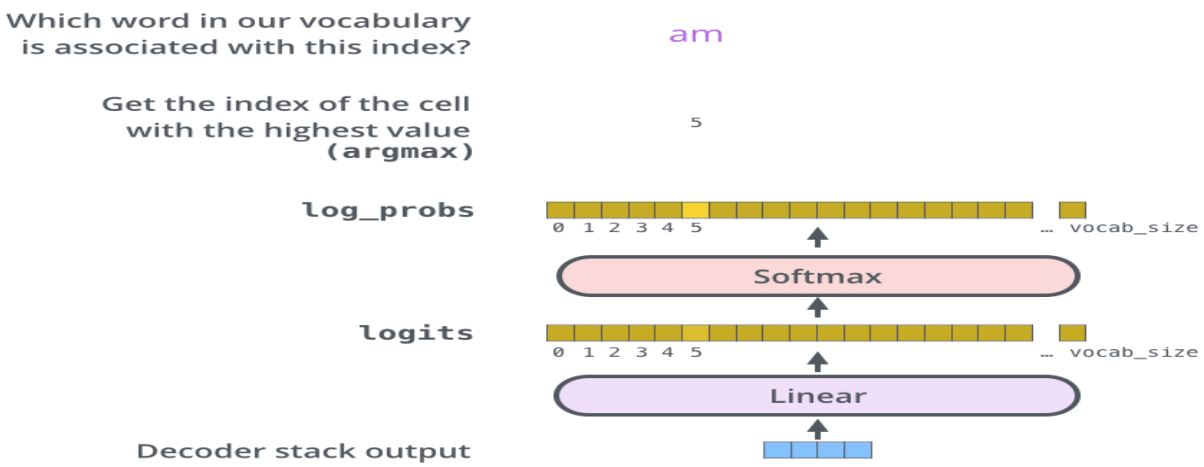
The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step.




**Final Linear and 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.

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.




**Conclusion:**

Encoder-Decoder and Attention mechanism are the integral parts of the Transformer. It solves the problem of Sequence pattern and Word formation. Large amount of data can be trained, and the error is minimized with the help of Transformer model.

The images in this paper is taken from **Illustrated Bert** as reference.