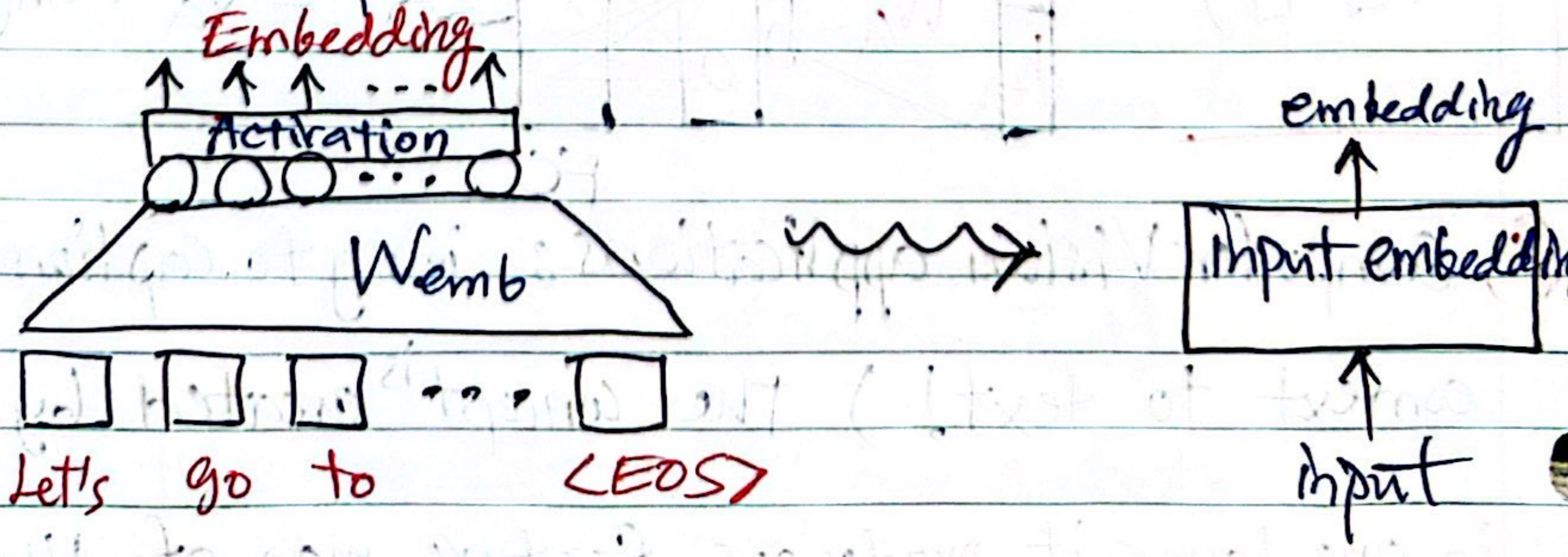
« Transformers »

Since transformers are originated in NLP, we start with NLP case for our explanations;
Since transformers are fairly complex concepts, we will explain their several building blocks seperately.

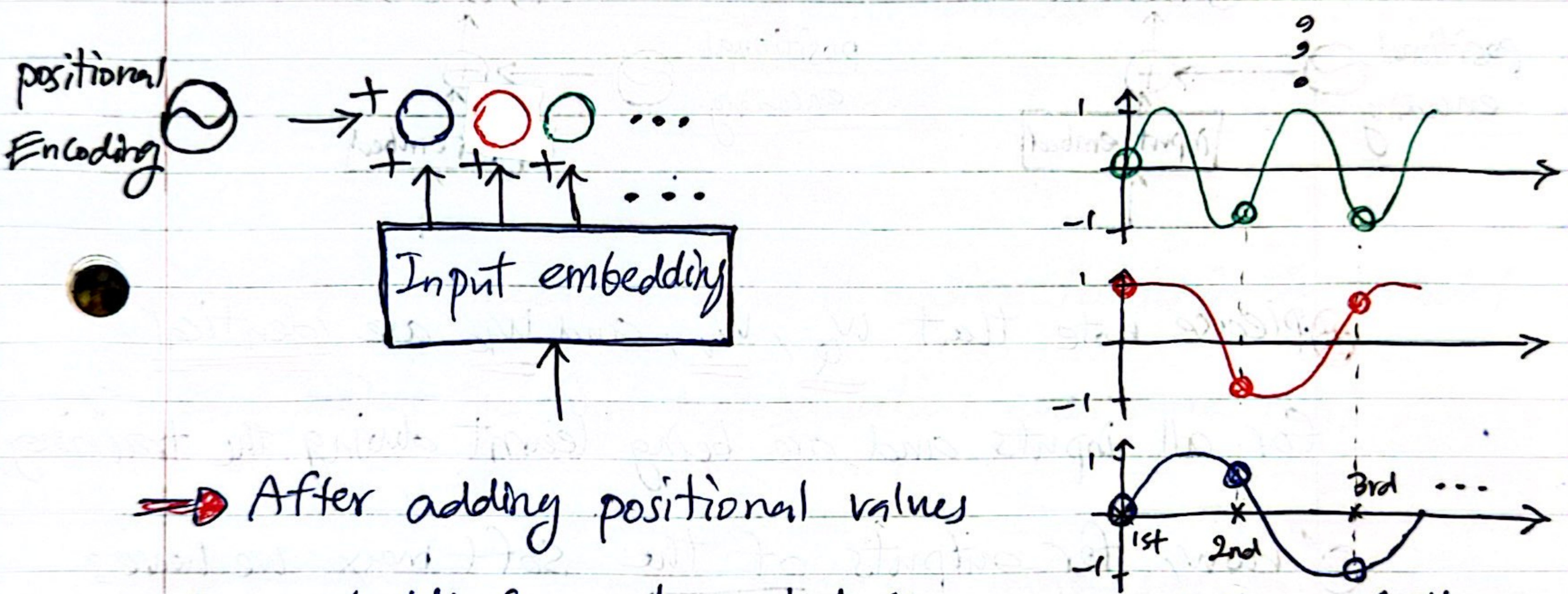
1// Word Embedding layer:

Input sentences are composed as word fragments and we call them tokens. To encode or embedd words, we pass them to a relatively light network and then an activation function. This network will be trained during the training phase of the entire network → Note: we apply the same weights on all word inputs + <EOS>



2/ Positional encoding stage:

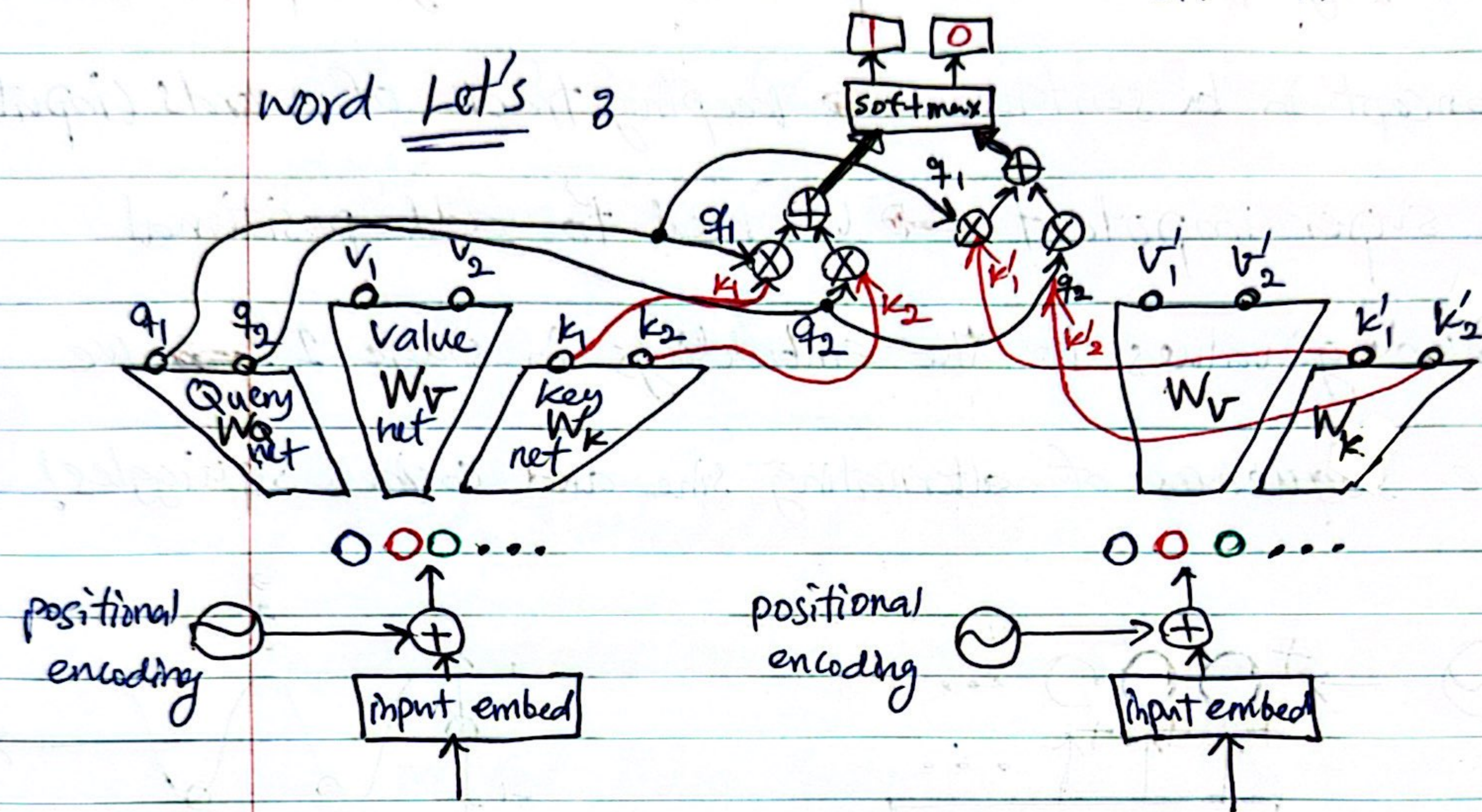
Changing position of inputs make a huge impact on the Concept in a sentence → keeping track of words (inputs) is super impotant → We need to add positional encoding values to the embeddings in stage 1. → We use sequences of alternating sine and cosine squiggles!



→ After adding positional values to embeddings, we have a better representation of the input sequence → We have the track of input orders.

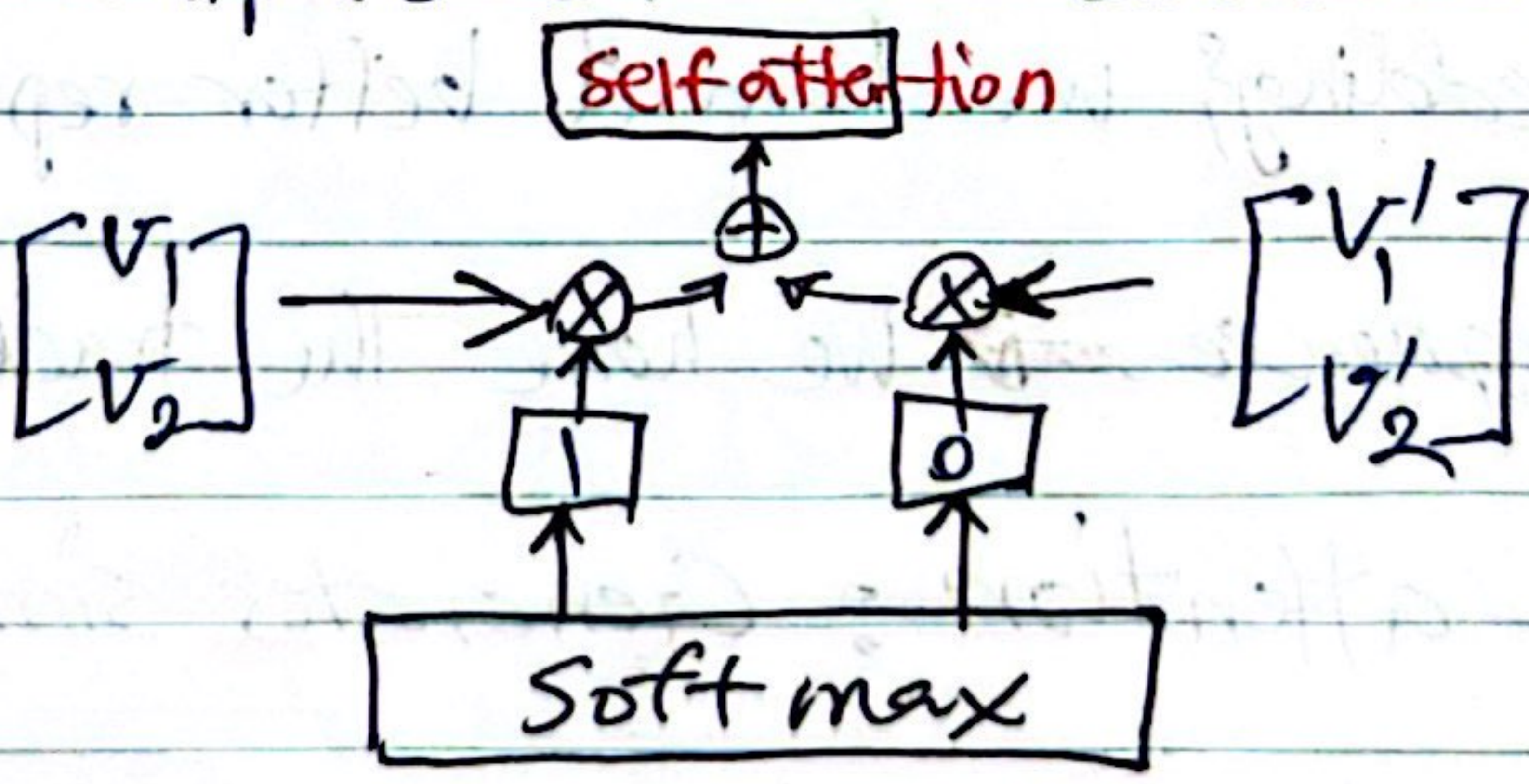
3/ Self-attention: Generates similarity between each word in the sentence and all other words, including itself! E.g., The red pizza tasted good!

Now, lets consider we have this sentence: "Let's go!" and we want to calculate the self attention for word Let's:



please note that W_Q, W_V, and W_K are identical for all inputs and are being learnt during the training!

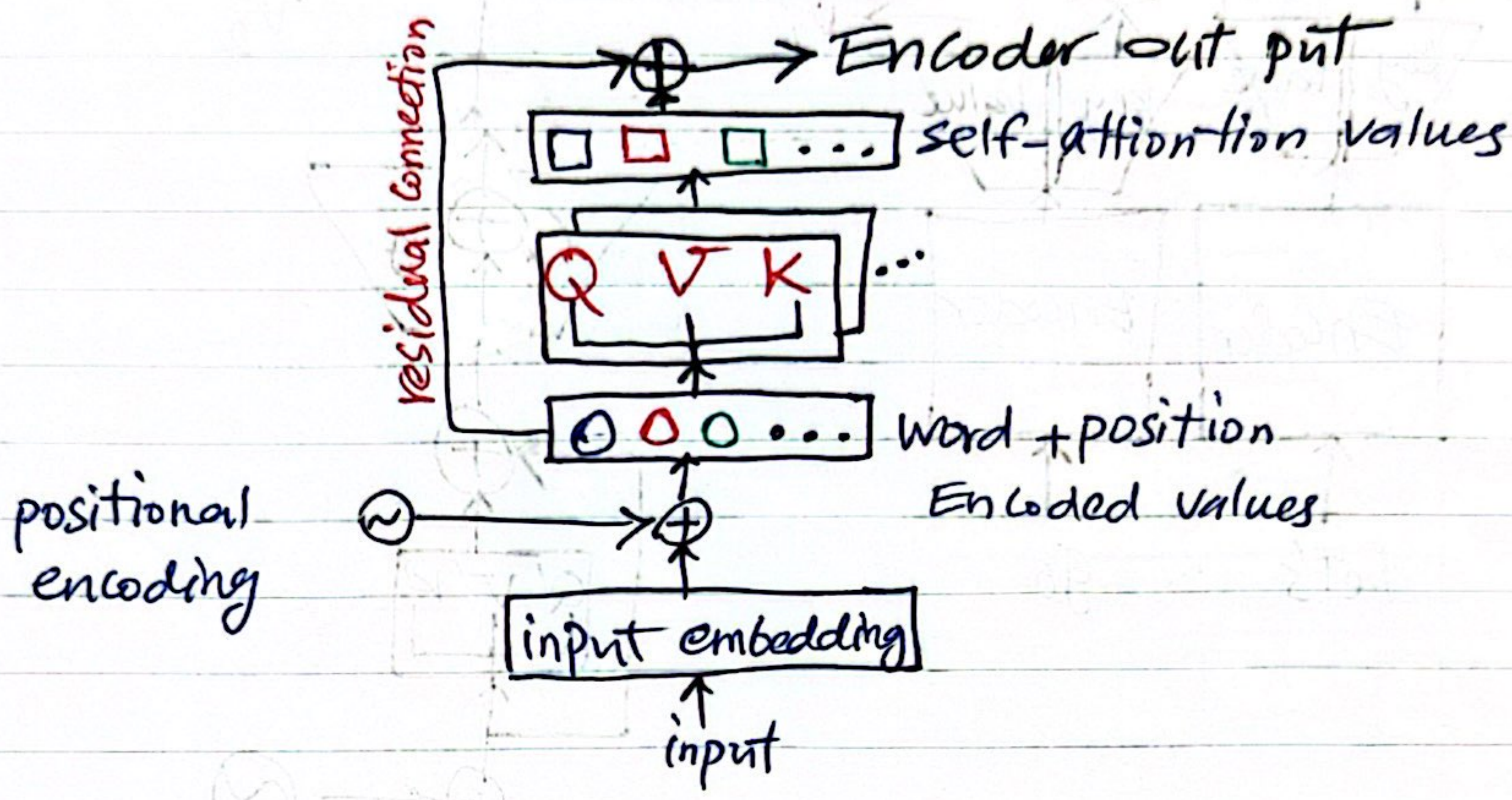
now, for outputs of the soft max we have:



We do the same thing for all pairs of the inputs

since weights are shared over different input instance, Transformer is ideal for parallel computing.

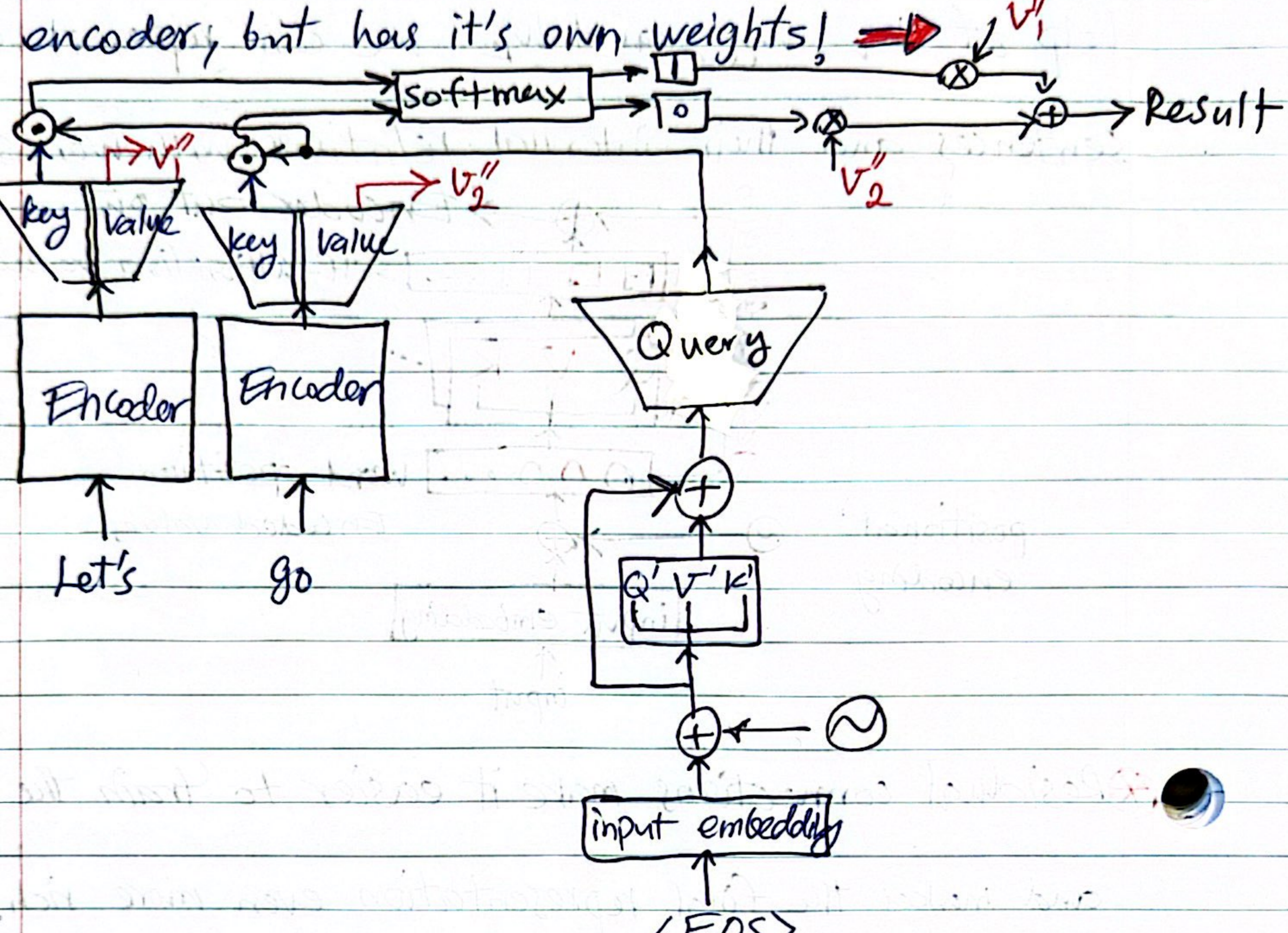
We also can stack self attention block and with the help of W_V, W_Q, and W_K we can represent complicated sentences and their internal relations with words.



⊛ Residual connections make it easier to train the models and makes the final representation even more rich!

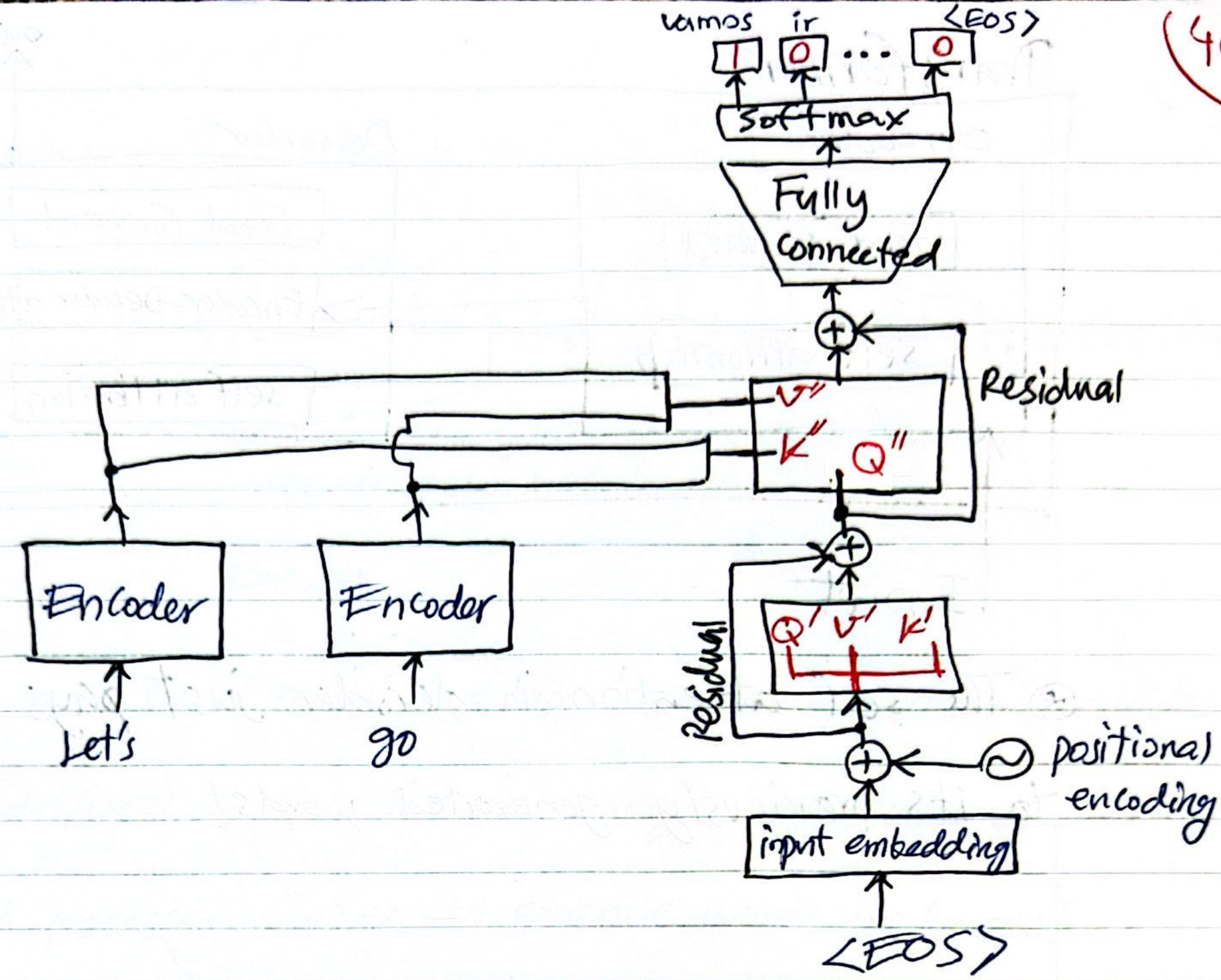
➡ Encoder: encodes words into numbers ⊕ encodes the position of each word ⊕ encodes the relationships among the words ⊕ make it easy to train quickly in parallel.

⊛ Now, we start to build the decoder network. we use another network for word embedding, add positional encoding, Calculate the same method for self attention with decoder tokens, and add residual layer. Please not that the architecture for decoder is exactly similar to encoder, but has it's own weights!



next, we creat another key and value networks (with differ weights) upon encoder and calculate keys dot product with the new Query network upon decoder value! we pass these dot products to a softmax and then, multiply and add softmax outputs with the outputs of Value networks: v_1'' and v_2'' and add them together.

Next illustration, makes the whole procedure more clear!



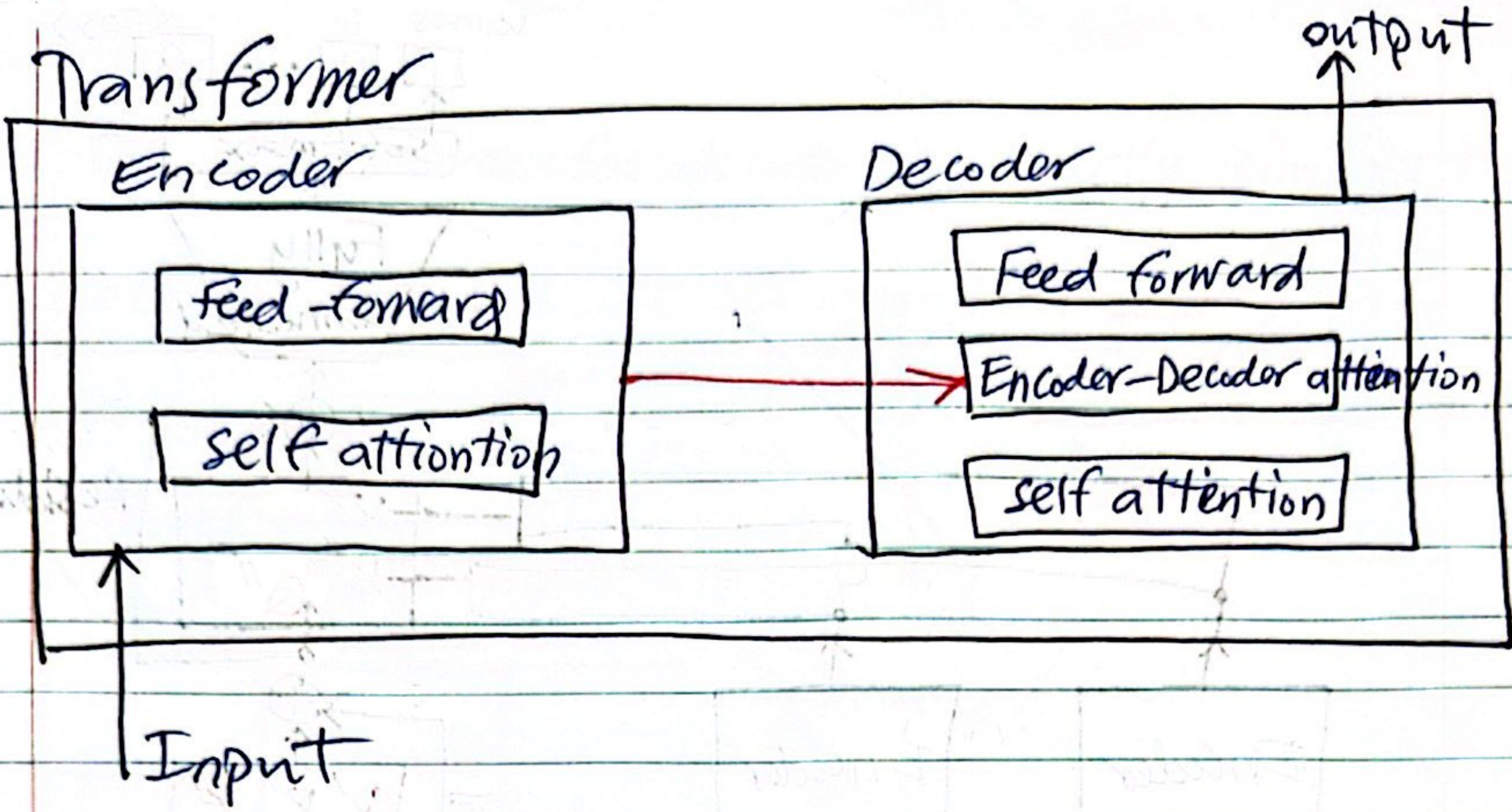
- To get large Transformers to work on large datasets:

1/ we have to normalize the values after every step (e.g., after positional encoding, after self attention for both encoder and decoder)

2/ we have to increase the # of tokens and complexity of networks.

3/ Similarity in attention $= \dfrac{\text{dot product}}{\sqrt{\#\text{ embedding values}}}$

- This is the high level overview of Transformers



⊛ The self attention in decoder just pays attention to its previously generated words!