

# Transformer

*Google Scholar: 4th most influential paper in 2020*



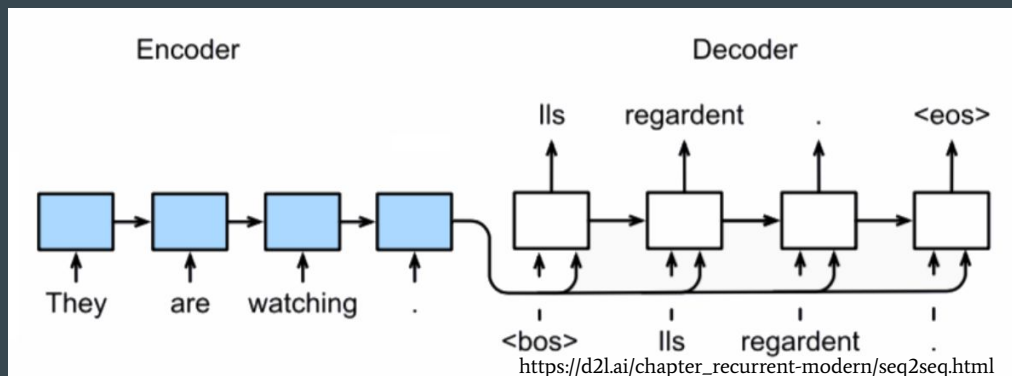
Present by: Liyan Tang

Attention Is All You Need (2017): <https://arxiv.org/abs/1706.03762>

Reference: <http://jalammr.github.io/illustrated-transformer/>

<https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>

# Recurrent Neural Network (RNN)

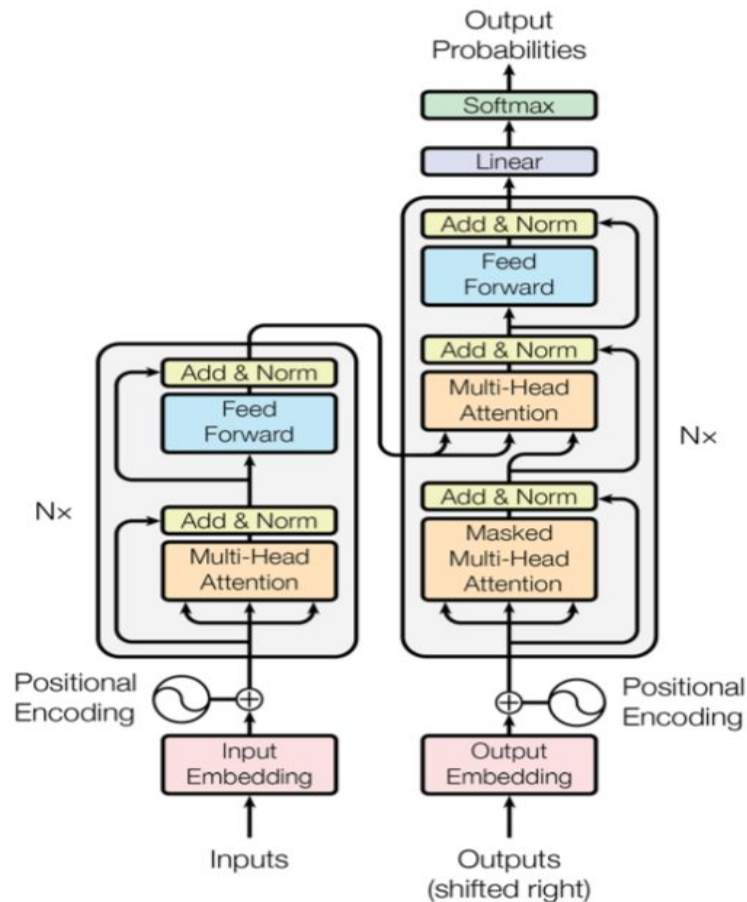


## RNN:

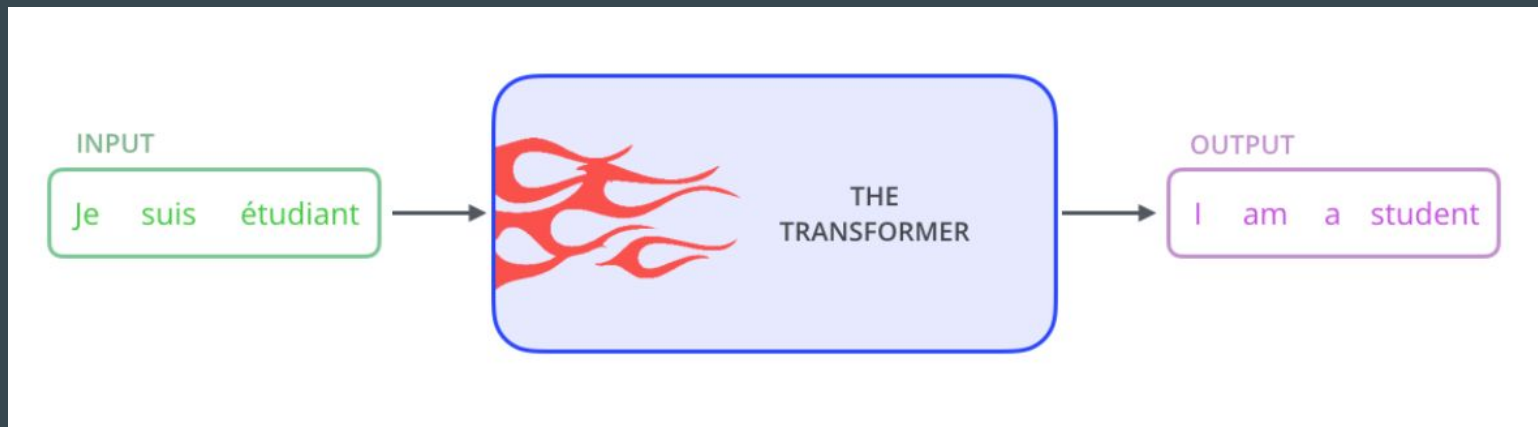
1. **Sequential processing:** *sentences must be processed words by words.* In order to encode the second word in a sentence you need the previously computed hidden states of the first word, which means you cannot train the model in parallel.
2. **Past information retained through past hidden states:** The encoding of words quickly lose their influence after a few time steps. LSTM and bi-LSTM can to some extent mitigate this problem, but nevertheless the problem is inherently related to recursion structure.

# Transformer

1. **Non sequential:** *sentences are processed as a whole rather than word by word.*
2. **Self-Attention:** Each token looks all positions of the input sequence.

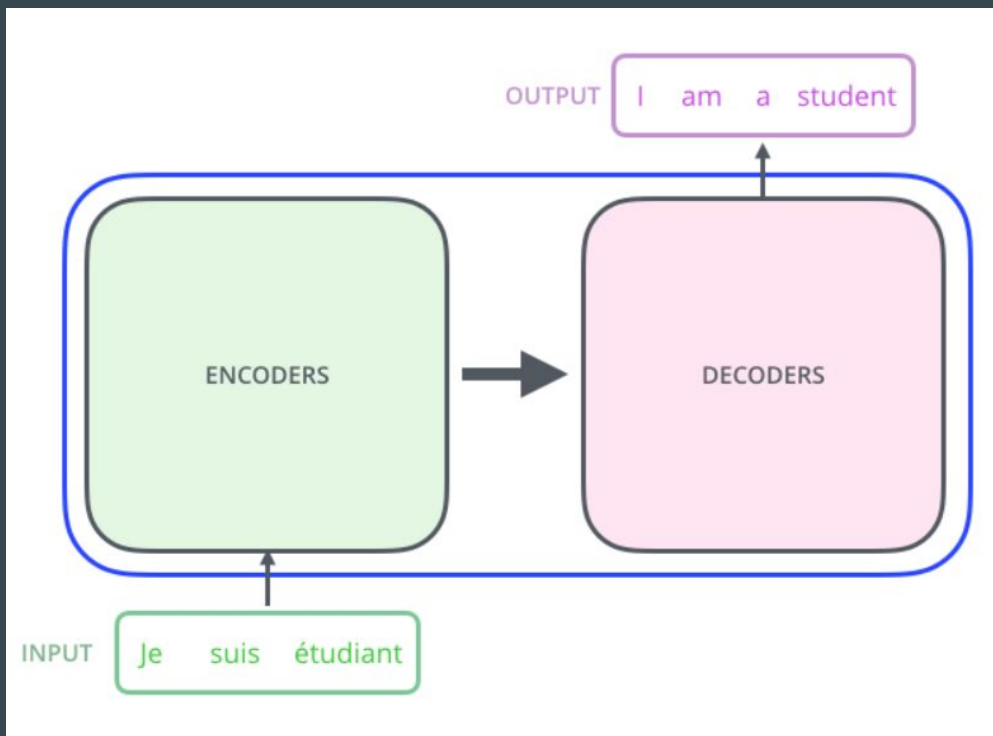


# A High-Level Look



In a machine translation application, it would take a sentence in one language, and output its translation in another.

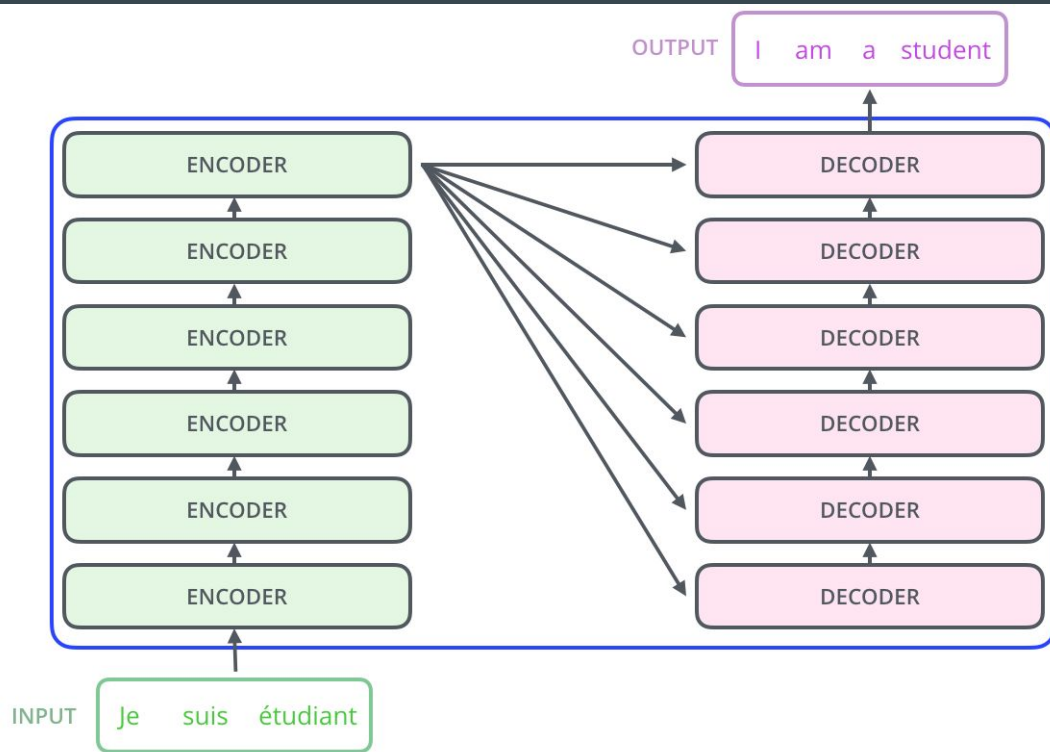
# A High-Level Look



we see

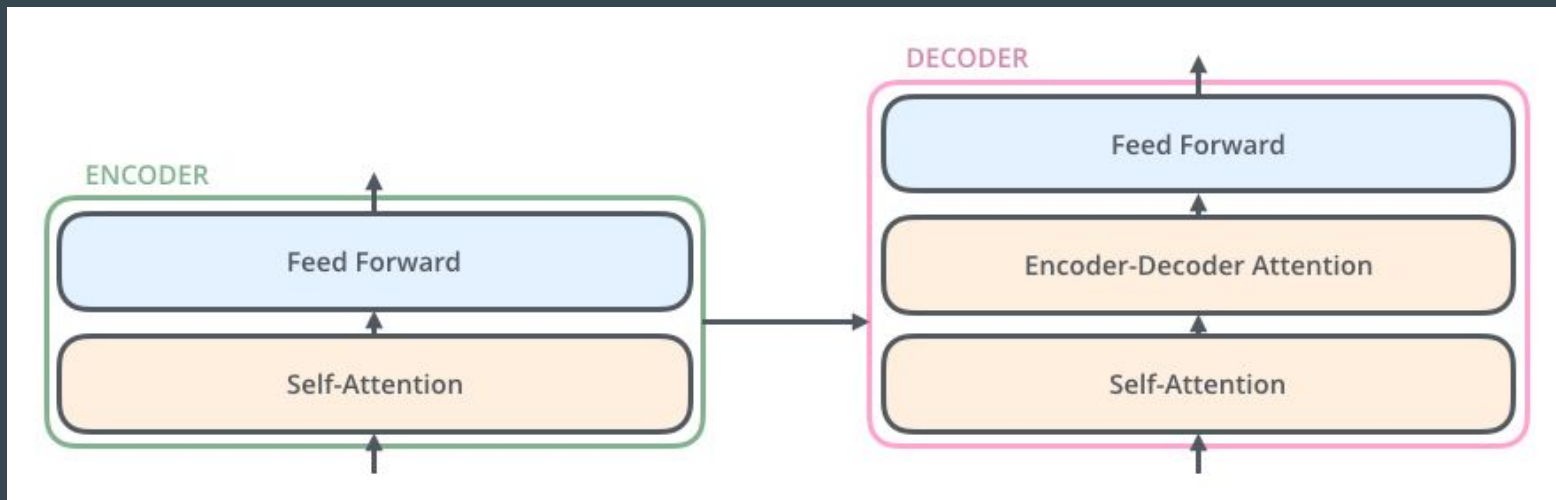
1. An encoding component;
2. A decoding component;
3. And connections between them.

# A High-Level Look



The encoding component is a stack of encoders (the paper stacks six of them). The decoding component is a stack of decoders of the same number. The encoders are all identical in structure, yet they do not share weights..

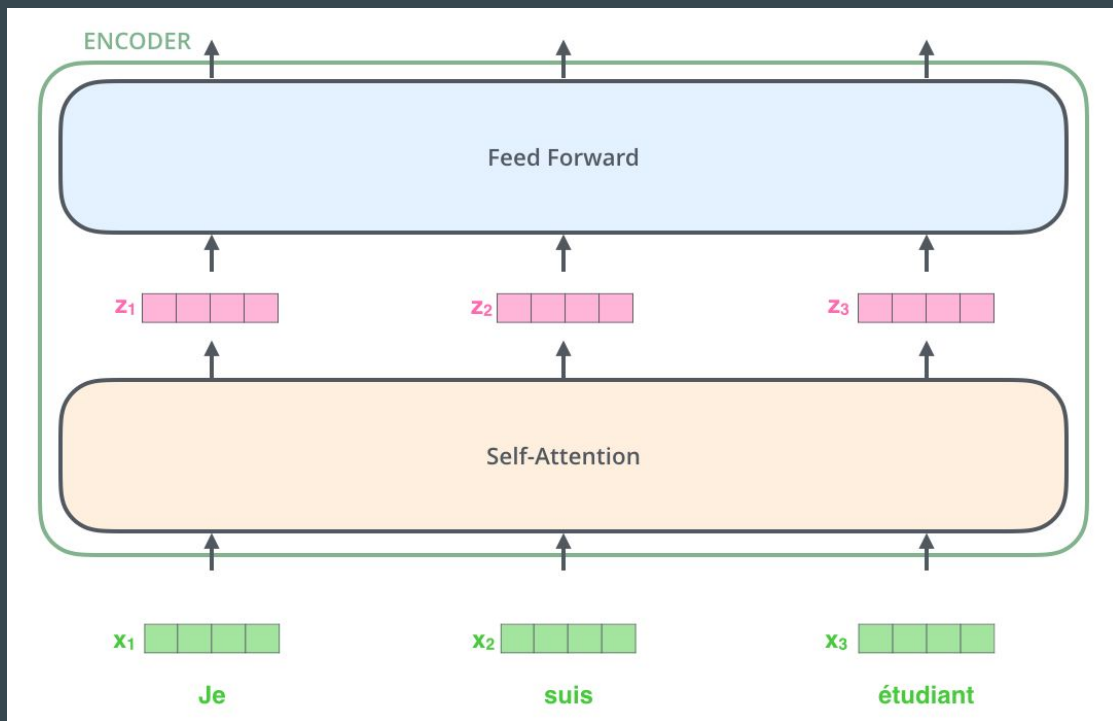
# A High-Level Look



Each encoder is broken down into two sub-layers.

1. The encoder's inputs first flow through a self-attention layer.
2. The outputs of the self-attention layer are fed to a feed-forward neural network.
3. The decoder has an extra attention layer that helps the decoder focus on relevant parts of the input sentence.

# Take a Closer Look



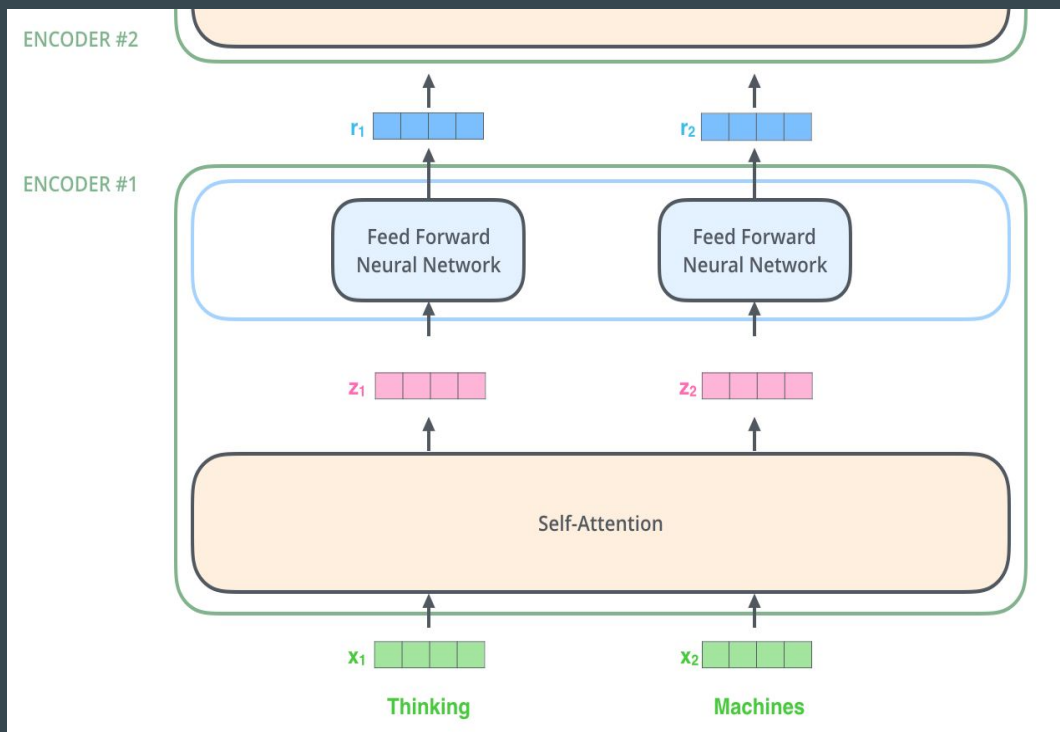
Each word embedding of the input sequence flows through each of the two layers of the encoder (vector size: 512 by default).

The maximum input sequence length is 512. It's a hyper-parameter we can set. Basically it would be the length of the longest sentence in our training dataset; or we can take a length at 90th percentile for faster performance.

#pad if a input sequence is not max len.



# Take a Closer Look



An encoder receives a list of vectors as input. It processes this list by

1. passing these vectors into a 'self-attention' layer, where tokens interact with each other.
2. then into a feed-forward neural network, where each token pass the exact same network with each vector flowing through it separately (we can parallelize  $\wedge$   $\wedge$ .)
3. then sends out the output upwards to the next encoder.

# Self-Attention at a High Level

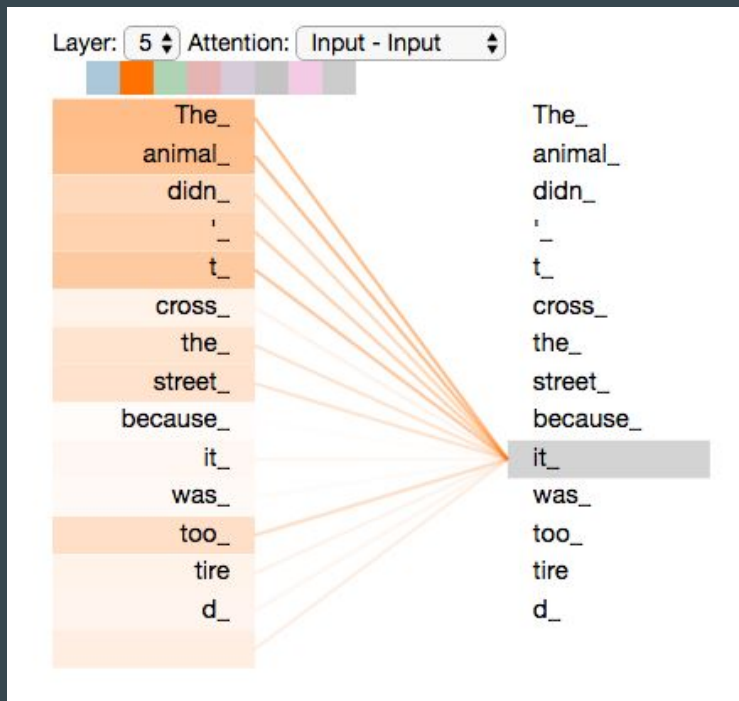
As the model processes each word/token, self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word/token.

In RNN, a hidden state allows RNN to incorporate its representation of previous words/token it has processed with the current one it's processing. You cannot look words after the current position.

# Self-Attention at a High Level (an example)

Suppose we are going to translate the following sentence:

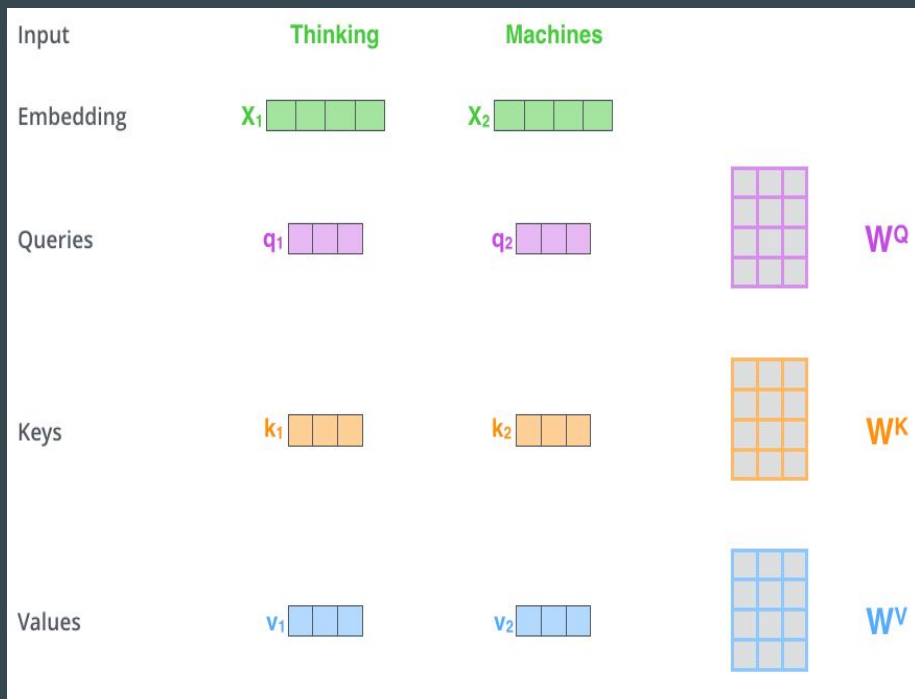
*The animal didn't cross the street because it was too tired*



When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

As we are encoding the word "it" in encoder, part of the attention mechanism was focusing on "The Animal".

# Self-Attention in Detail (Step 1)



The first step is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word).

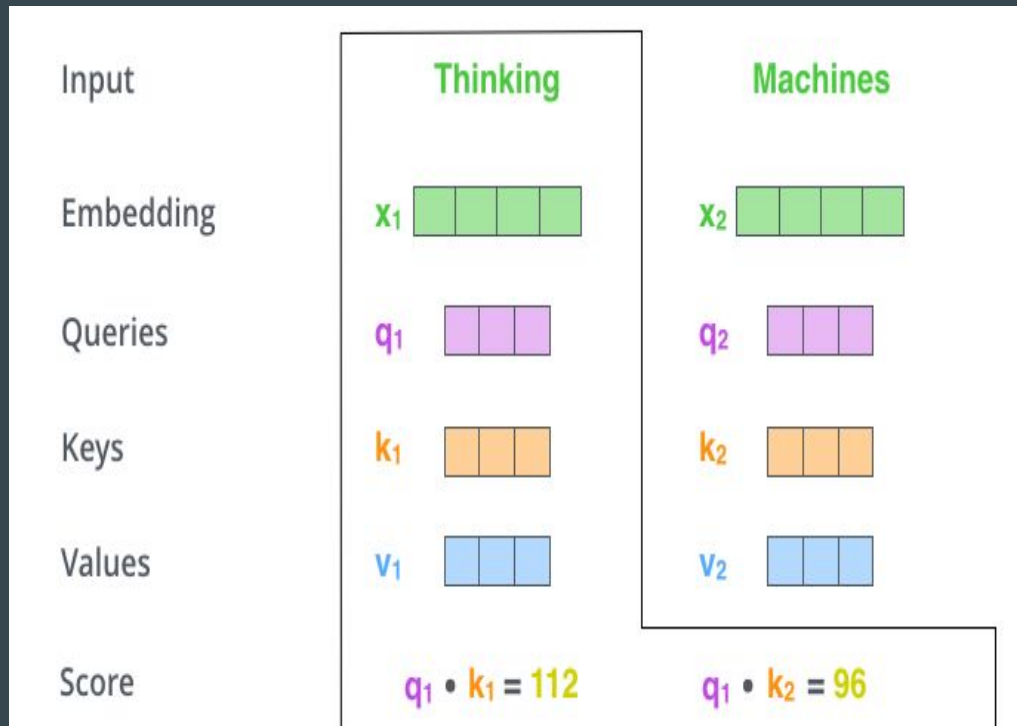
So for each word, we create

1. a Query vector,
2. a Key vector,
3. a Value vector.

These vectors are created by multiplying the embedding by three trainable matrices.

Notice that these new vectors are smaller in dimension than the embedding vector (they don't have to be).

# Self-Attention in Detail (Step 2)



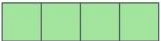
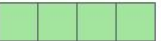
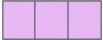
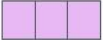
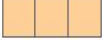
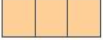
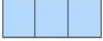
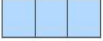
The second step is to calculate a score.

Suppose we calculate the self-attention for the first word, “Thinking”.

We need to determine how much focus to place on other inputs as we encode a word at a certain position.

The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we’re scoring.

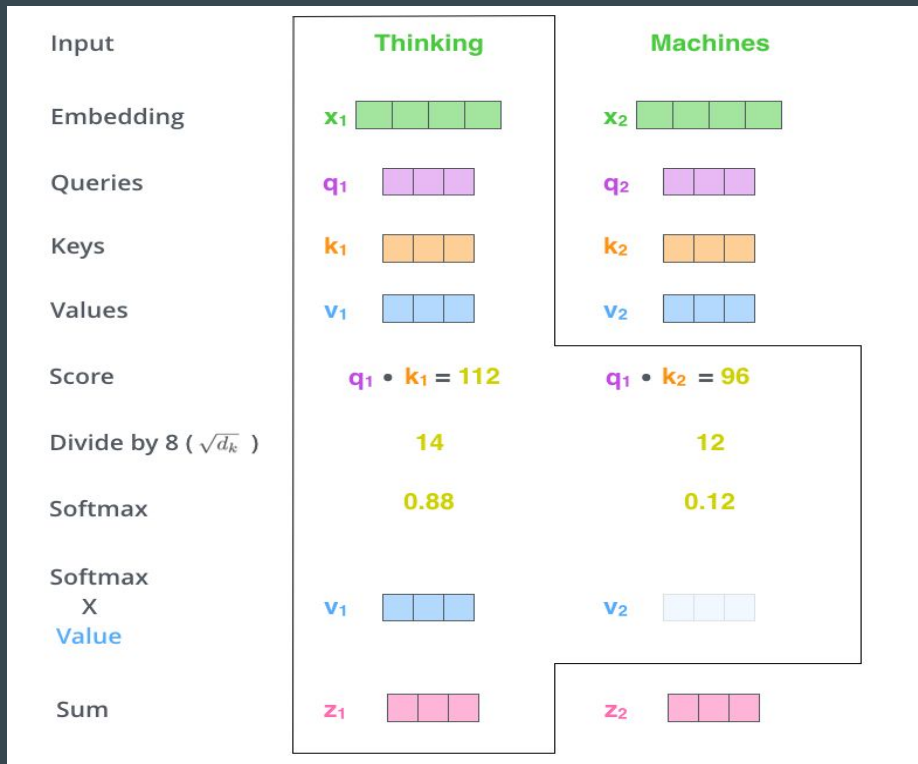
# Self-Attention in Detail (Step 3, 4)

Input	Thinking	Machines
Embedding	$x_1$ 	$x_2$ 
Queries	$q_1$ 	$q_2$ 
Keys	$k_1$ 	$k_2$ 
Values	$v_1$ 	$v_2$ 
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by $8 (\sqrt{d_k})$	14	12
Softmax	0.88	0.12

The third step is to divide the scores by the square root of the dimension of the key vectors (64 is used in the paper). This leads to having more stable gradients. There could be other possible values here).

Then the fourth step is to pass the result through a softmax operation, which determines to what extent each word will be attended to at this position.

# Self-Attention in Detail (Step 5, 6)

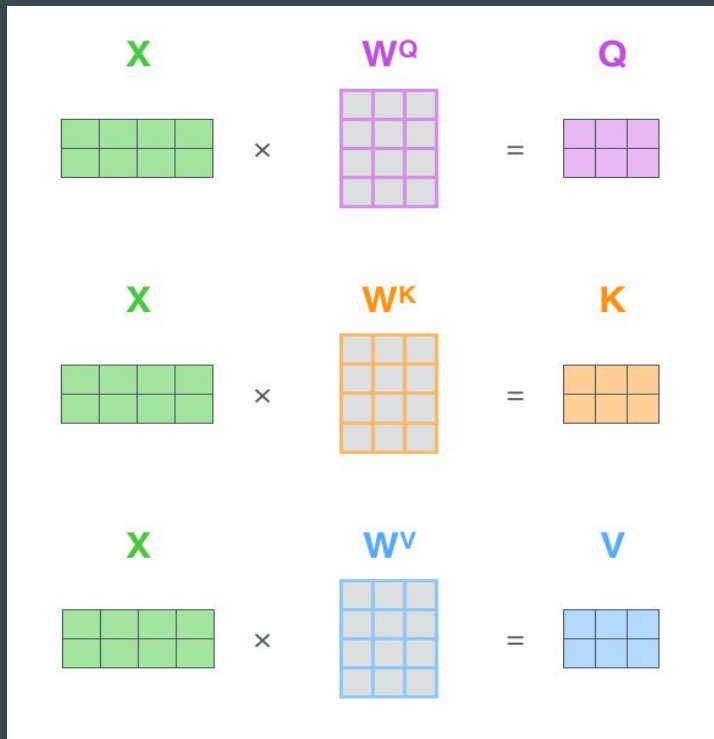


The fifth step is to multiply each **value vector** by the softmax score (in preparation to sum them up).

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 resulting **vector** is one we can send along to the feed-forward neural network.

# Matrix Calculation of Self-Attention (Step 1)

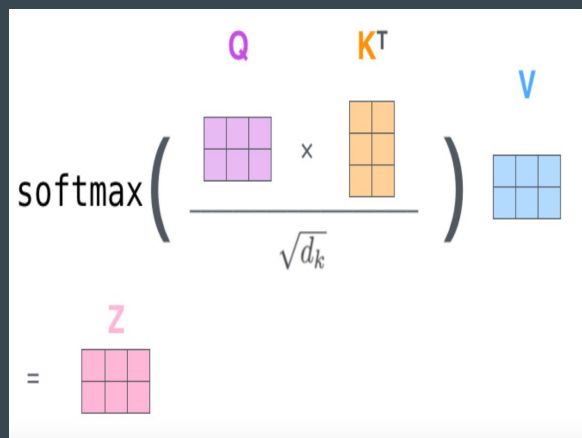


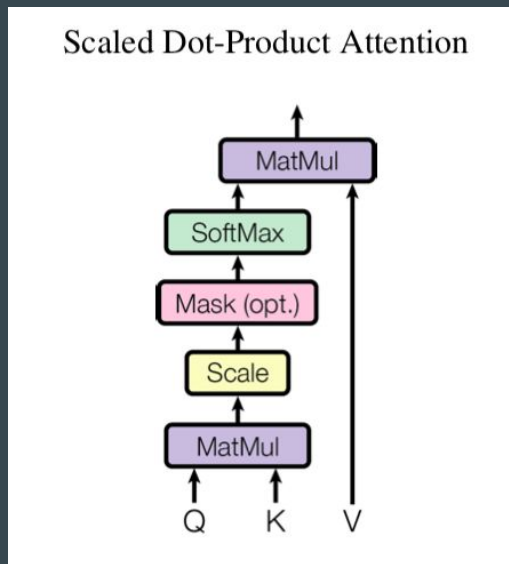
The first step is to calculate the **Query**, **Key**, and **Value** matrices. We do that by packing our embeddings into a matrix  $X$ , and multiplying it by the weight matrices we've trained ( $W^Q$ ,  $W^K$ ,  $W^V$ ).

Every row in the  $X$  matrix corresponds to a word in the input sentence.



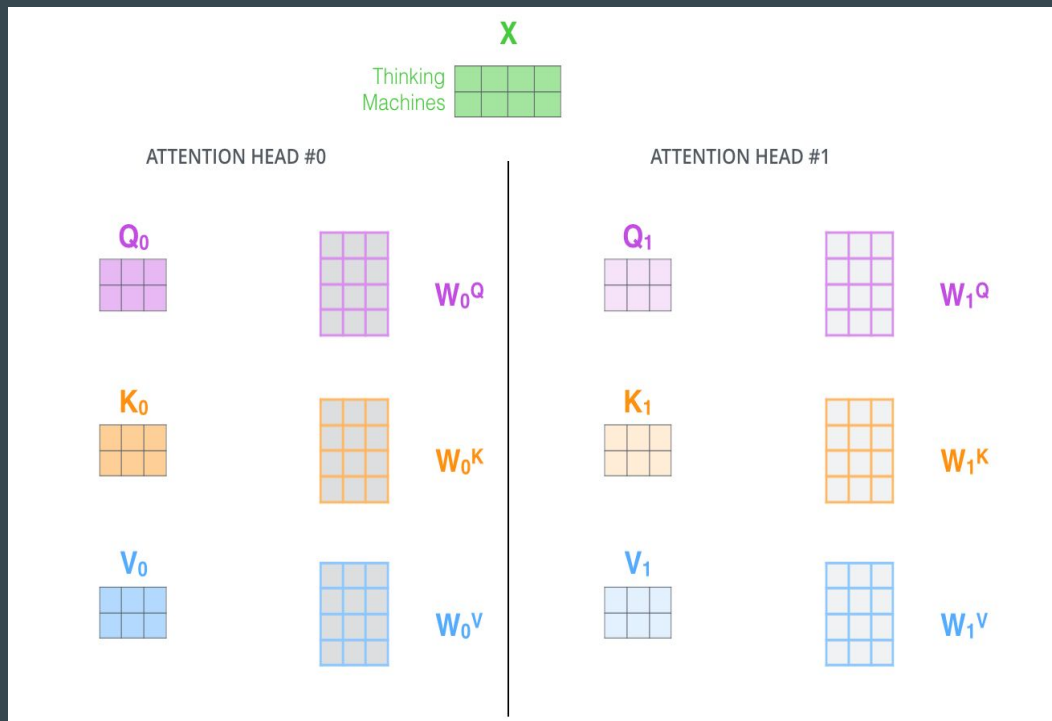
# Matrix Calculation of Self-Attention (Step 2)

$$\text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right) V = Z$$




Finally, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

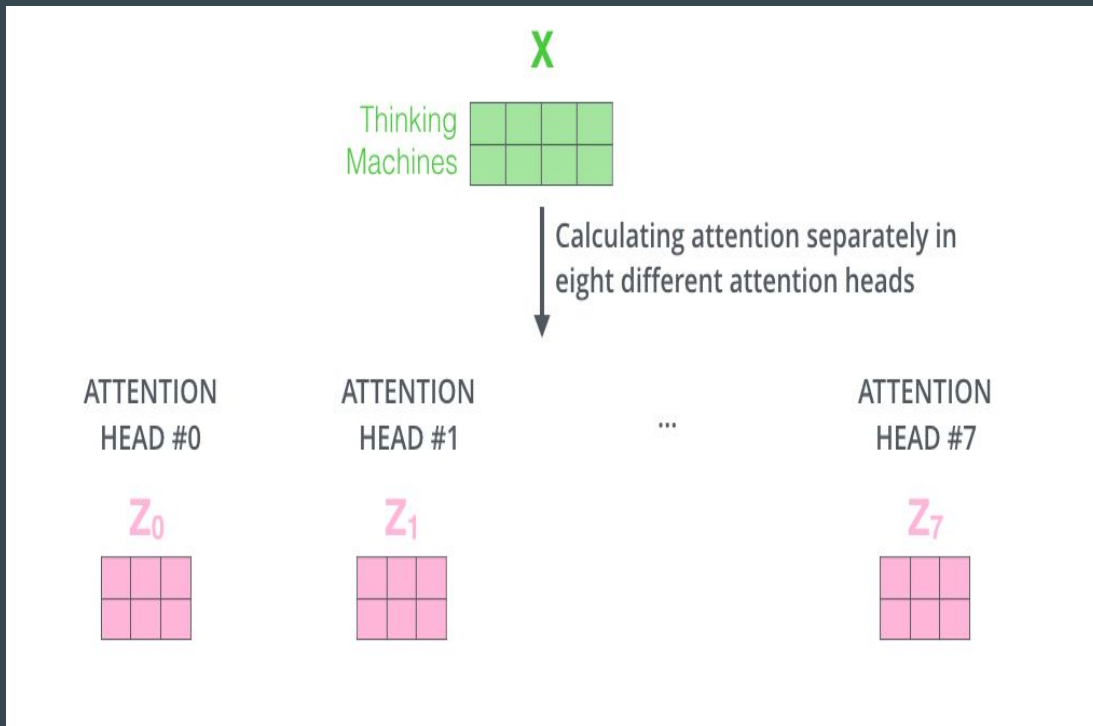
# Multi-Head Self-Attention



The paper further refined the self-attention layer by adding a mechanism called “multi-headed” attention (the Transformer use 8 attention heads).

It gives the attention layer multiple “representation subspaces”. With multi-headed attention, we maintain separate  $Q/K/V$  weight matrices for each head resulting in different  $Q/K/V$  matrices. As we did before, we multiply  $X$  by the  $WQ/WK/WV$  matrices to produce  $Q/K/V$  matrices.

# Multi-Head Self-Attention



Problem: The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

Solution: We concat the matrices then multiply them by an additional weights matrix  $W_O$ .

# Multi-Head Self-Attention

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. Concatenate all the attention heads.
2. Multiply with trainable matrix  $W^O$ .
3. Then we get a matrix  $X$  which captures information from all attention heads.

# Put All Things Together

1) This is our input sentence\*

Thinking  
Machines

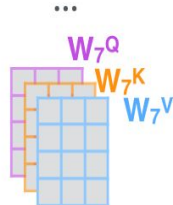
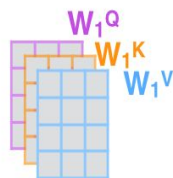
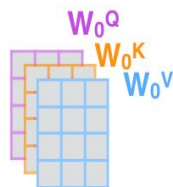


2) We embed each word\*

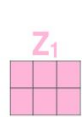
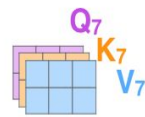
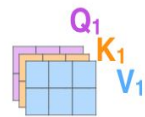
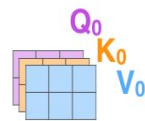
\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



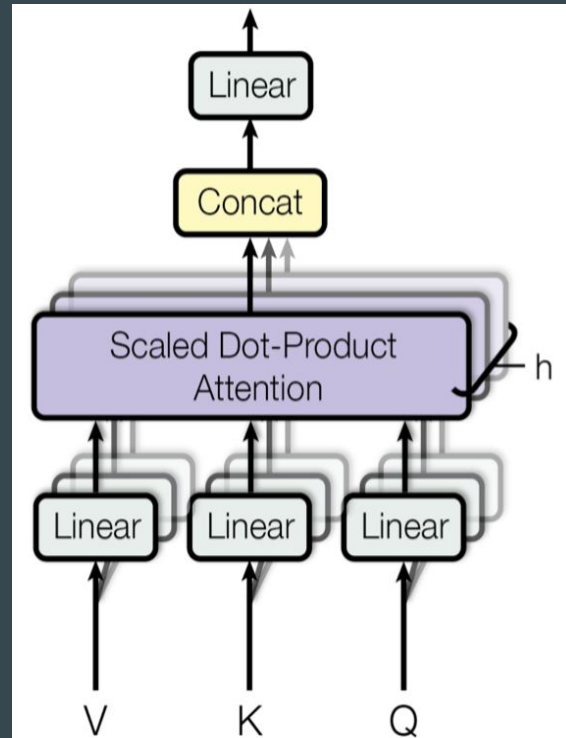
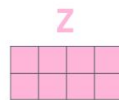
3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices



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

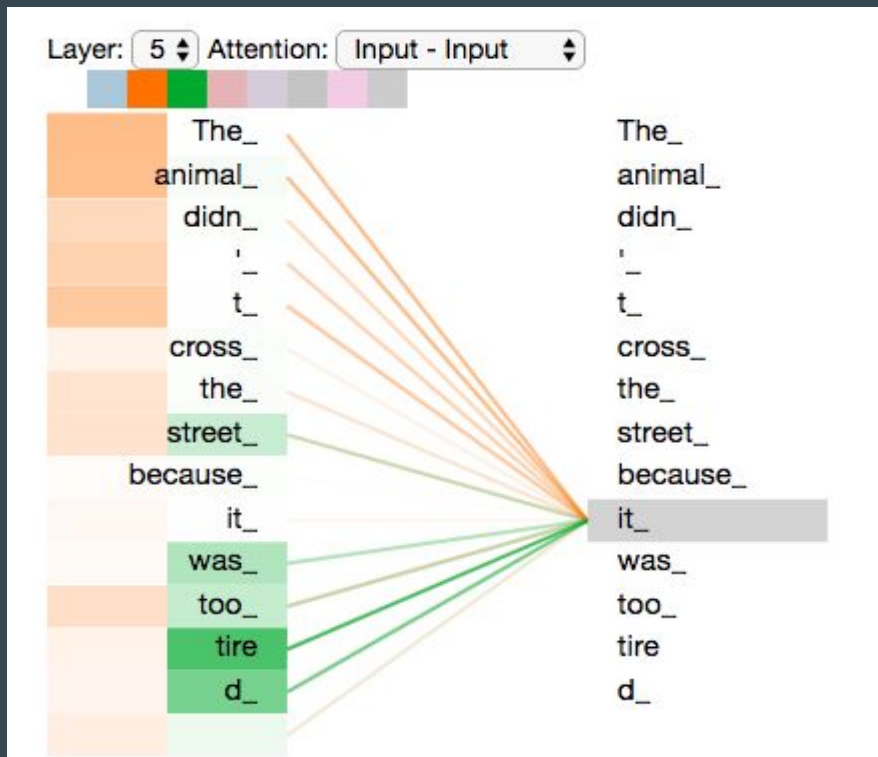


$W^O$



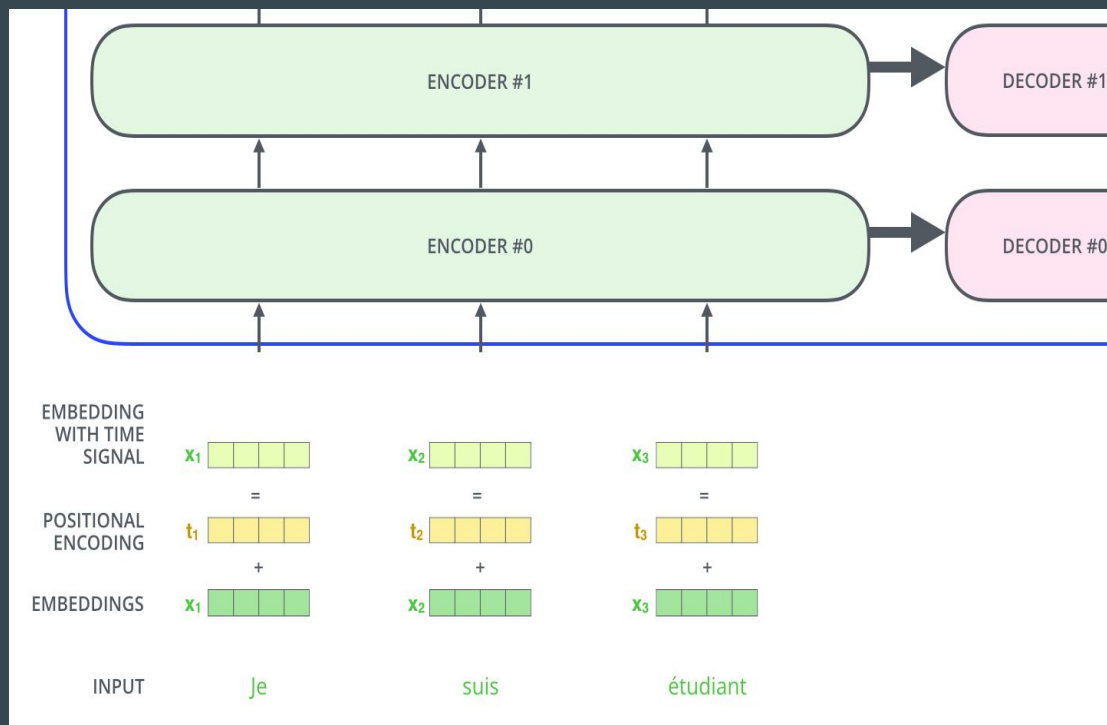
# Re-visit Self-Attention example

*The animal didn't cross the street because it was too tired*



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" consists of some of the representation of both "animal" and "tired".

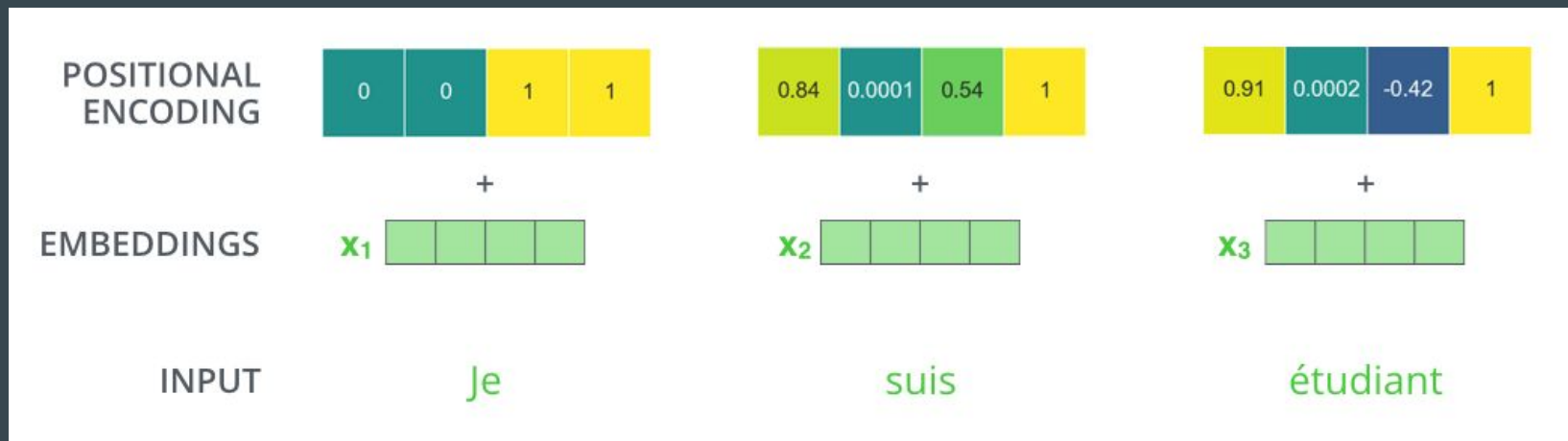
# Representing The Order of The Sequence



One thing that's missing is a way to account for the order of the words in the input sequence.

To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

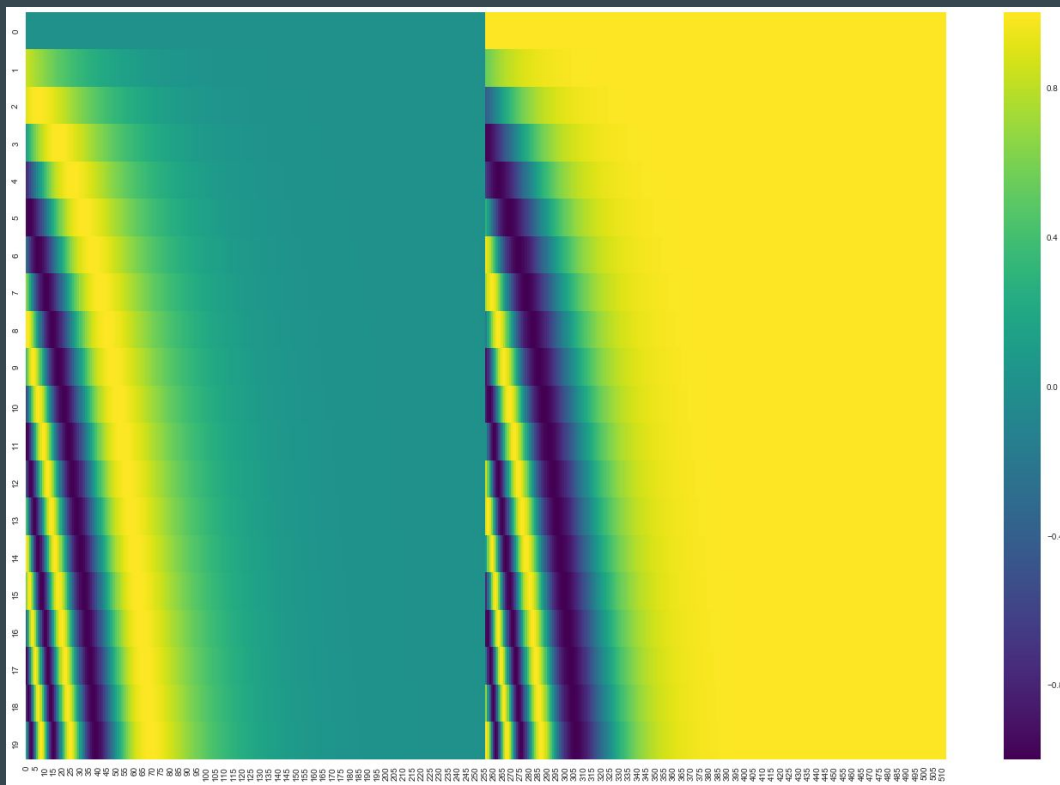
# Representing The Order of The Sequence



If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like the figure above.

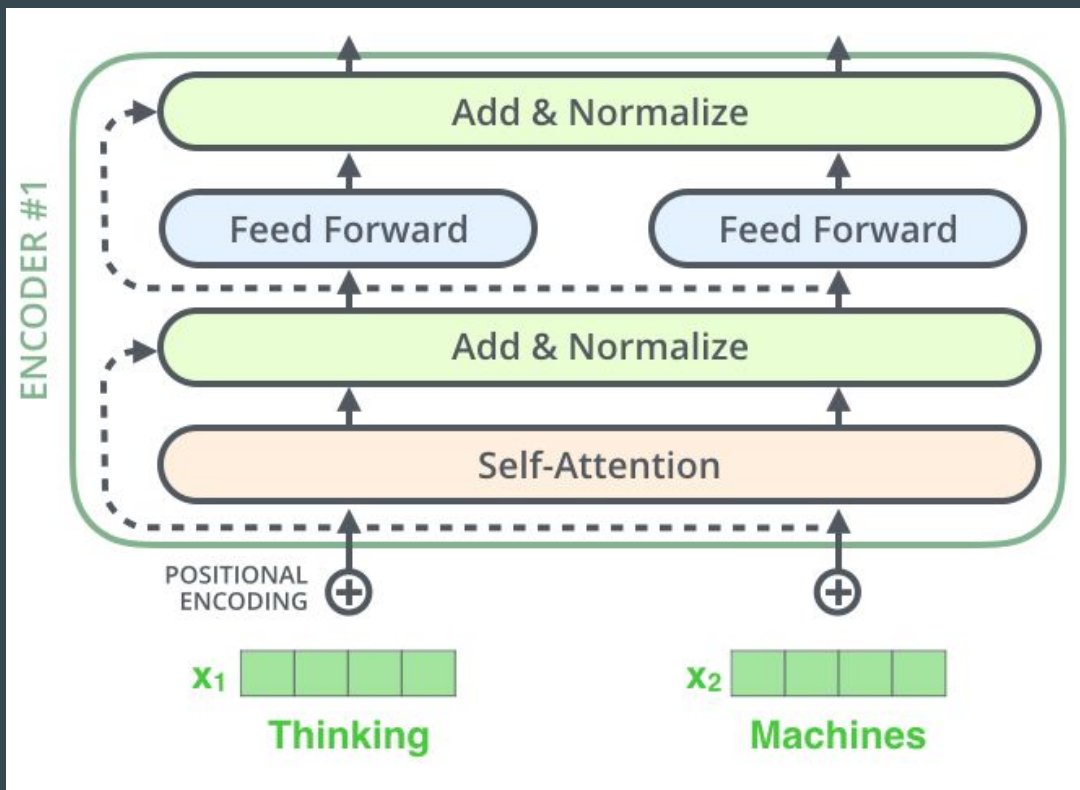


# Representing The Order of The Sequence



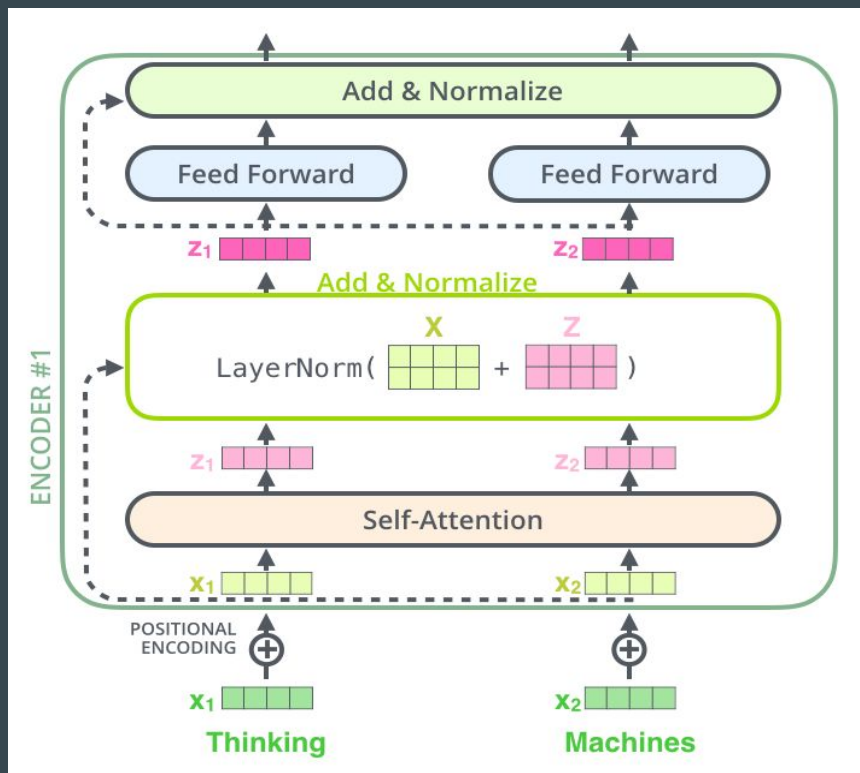
A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by a sine function, and the right half is generated by a cosine function. They're then concatenated to form each of the positional encoding vectors.

# The Residual Connection



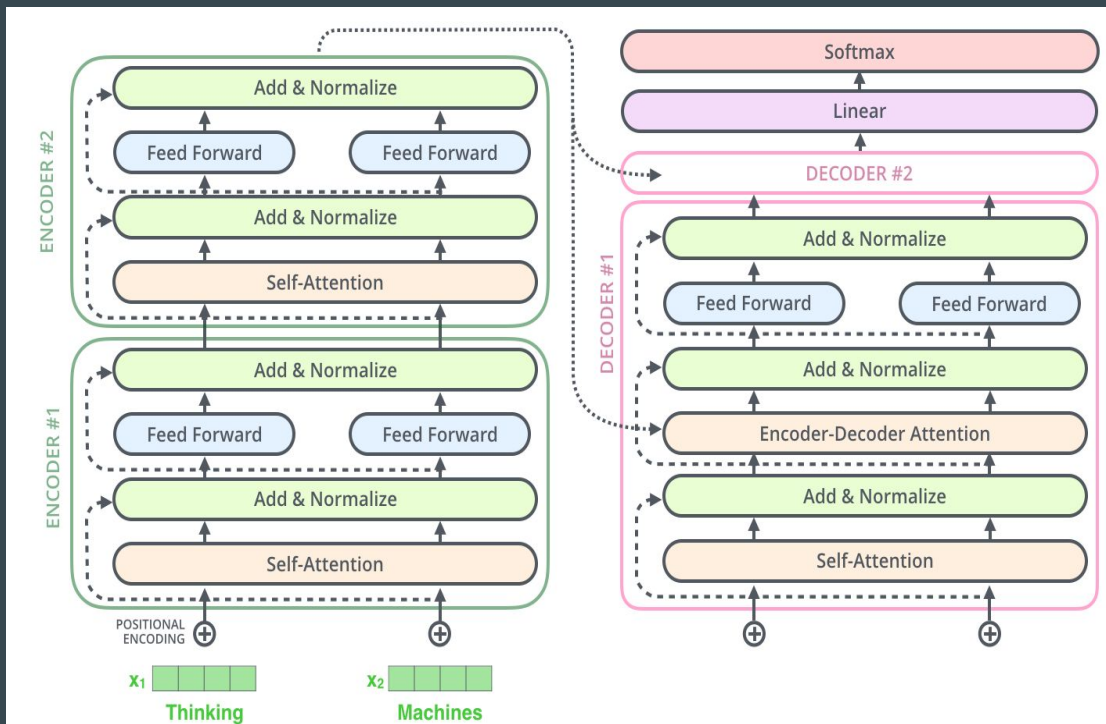
Each sub-layer (self-attention, FFNN) in each encoder has a residual connection around it, and is followed by a layer-normalization step.

# The Residual Connection



This figure shows the visualization of the vectors and the layer-norm operation associated with self attention.

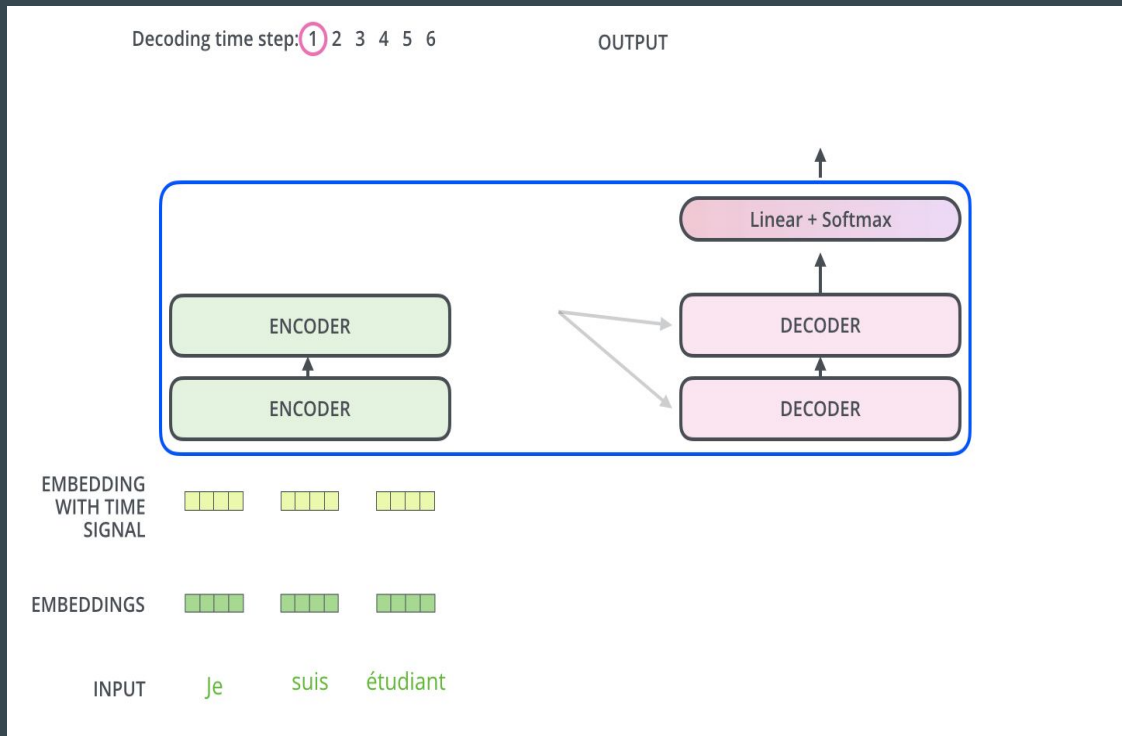
# The Residual Connection



The residual connection applies in the decoder as well.

If we think of a Transformer as 2 stacked encoders and decoders, it would look something like this figure.

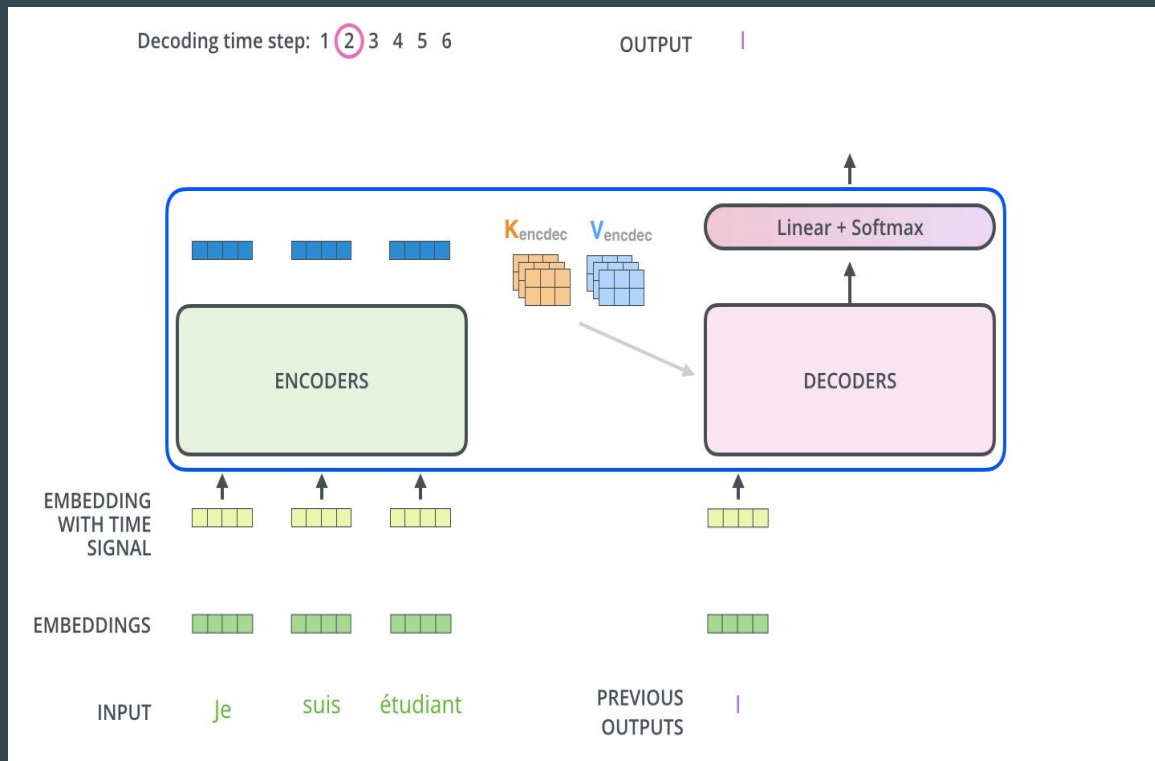
# The Decoder



Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well.

The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention matrices  $K$  and  $V$ . These are to be used by each decoder, which helps the decoder focus on appropriate places in the input sequence.

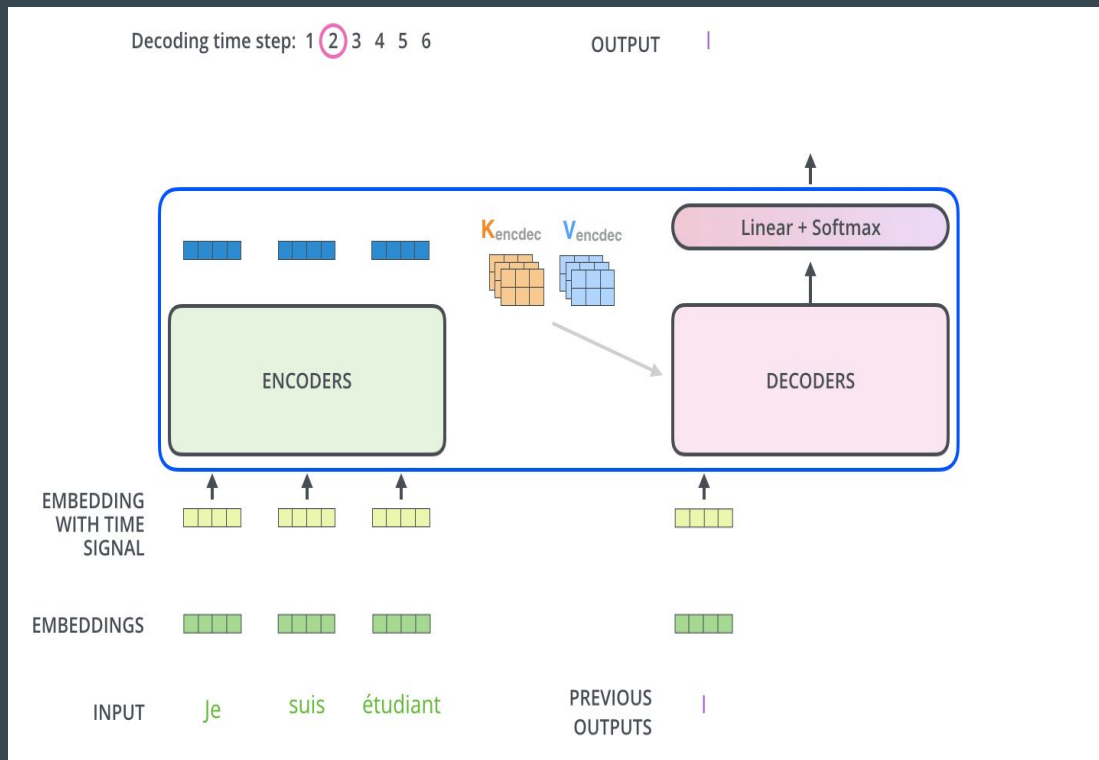
# The Decoder



The following steps repeat the process until a <EOS> token is reached. The output of each step is fed to the bottom decoder in the next time step.

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 Decoder



In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence (masking future positions 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

Which word in our vocabulary  
is associated with this index?

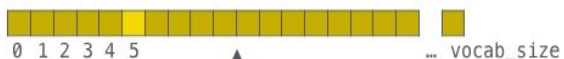
Get the index of the cell  
with the highest value  
(argmax)

Decoder stack output

am

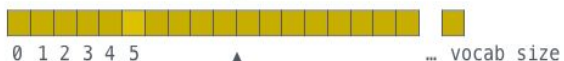
5

log\_probs



Softmax

logits



Linear

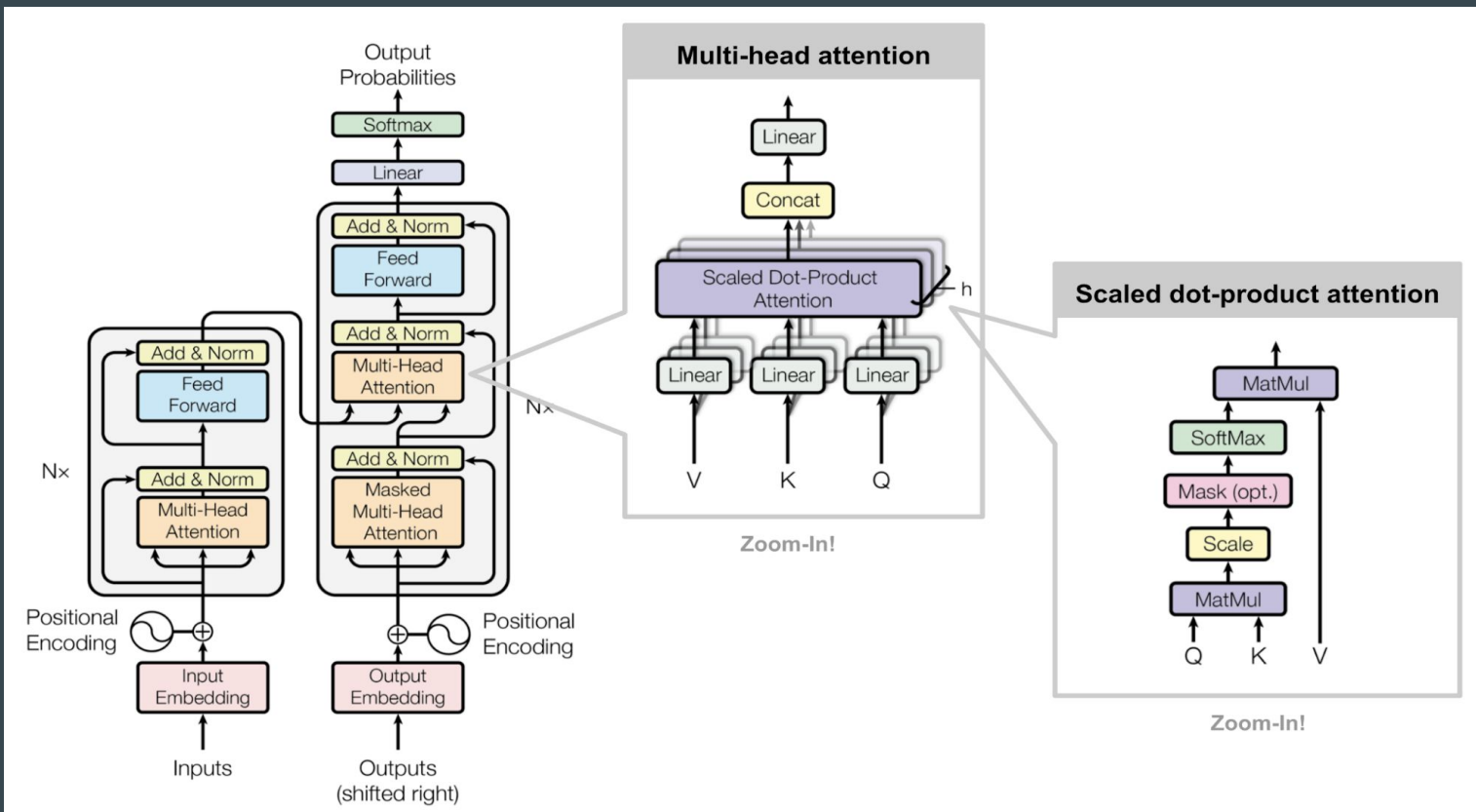


The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders into a logit vector (dim=vocab\_size).

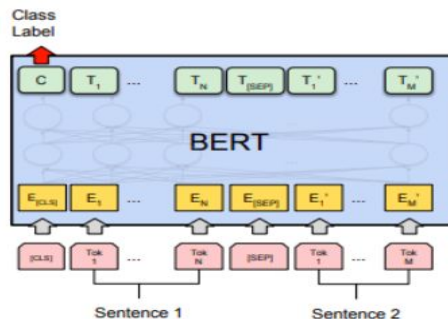
The softmax layer then turns the logit vector into probabilities. The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



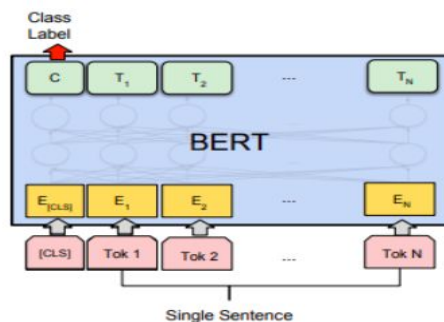
# Full Architecture Re-Visit



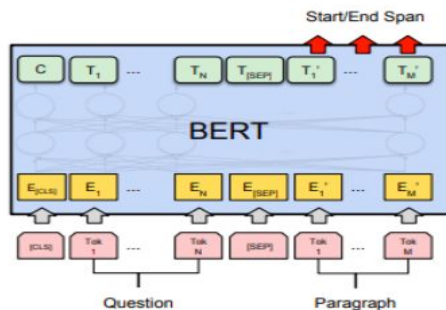
# Before We End - BERT (Bidirectional Encoder Representations from Transformer)



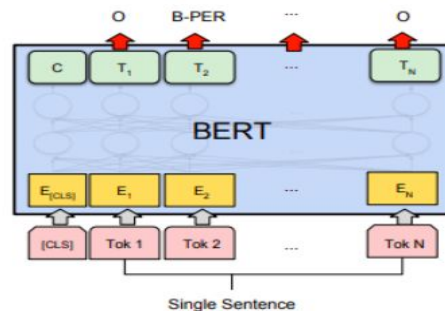
(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG



(b) Single Sentence Classification Tasks:  
SST-2, CoLA



(c) Question Answering Tasks:  
SQuAD v1.1



(d) Single Sentence Tagging Tasks:  
CoNLL-2003 NER

## Attention is All you Need - NIPS Proceedings

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best.

by A Vaswani · 2017 · [Cited by 13415](#) · [Related articles](#)

## BERT: Pre-training of Deep Bidirectional Transformers for ...

Oct 11, 2018 — Abstract: We introduce a new language representation model called **BERT**, which stands for **Bidirectional Encoder Representations from ...**

by J Devlin · 2018 · [Cited by 11138](#) · [Related articles](#)

Questions?