Transformers

Motivation, Architecture

AUA, DS330 Deep Learning 2021, Hovhannes Tamoyan

Motivation

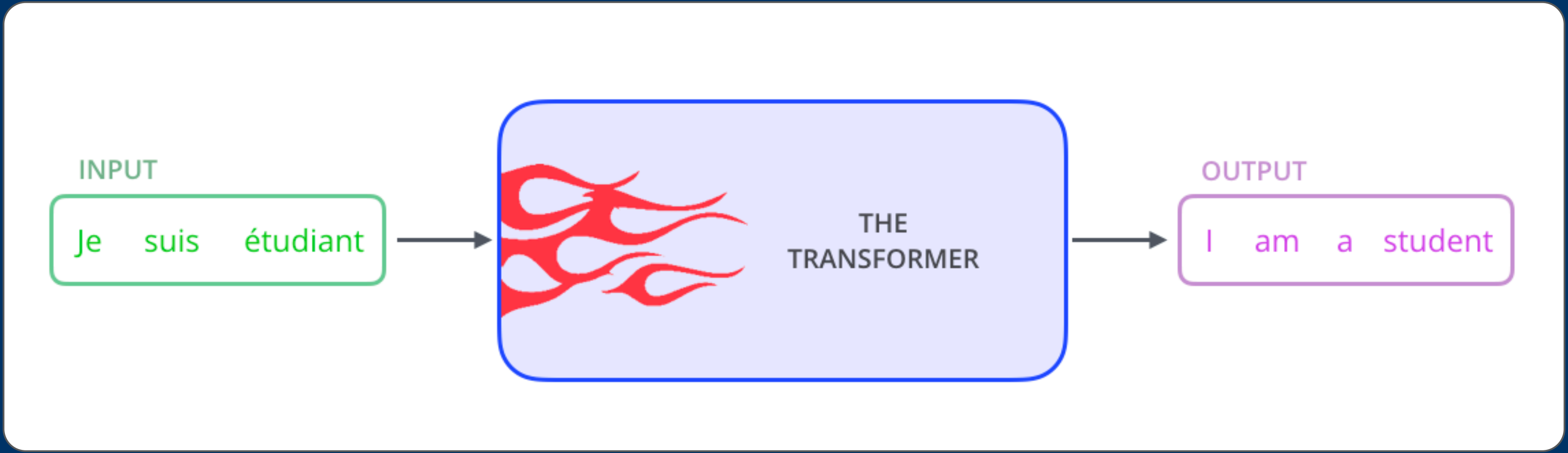
RNNs are great, but the recursive nature of RNN family not only makes models very slow but also excludes the opportunity of parallelisation: the processes need to be sequential.

Transformers are based on self-attention mechanism.
It is also possible to parallelize the computations of Transformer.

The Transformer was proposed from Vaswani et. al. in 2007, in the paper - "Attention is All You Need".
HarvardNLP implementation in PyTorch - "The Annotated Transformer"

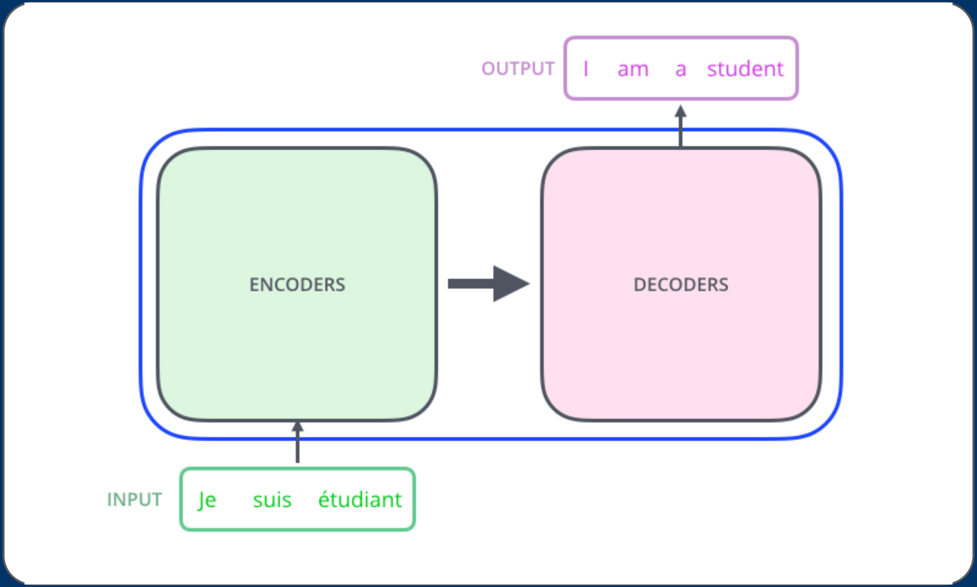
A High-Level Look

Let's take a look at this Machine Translation example.



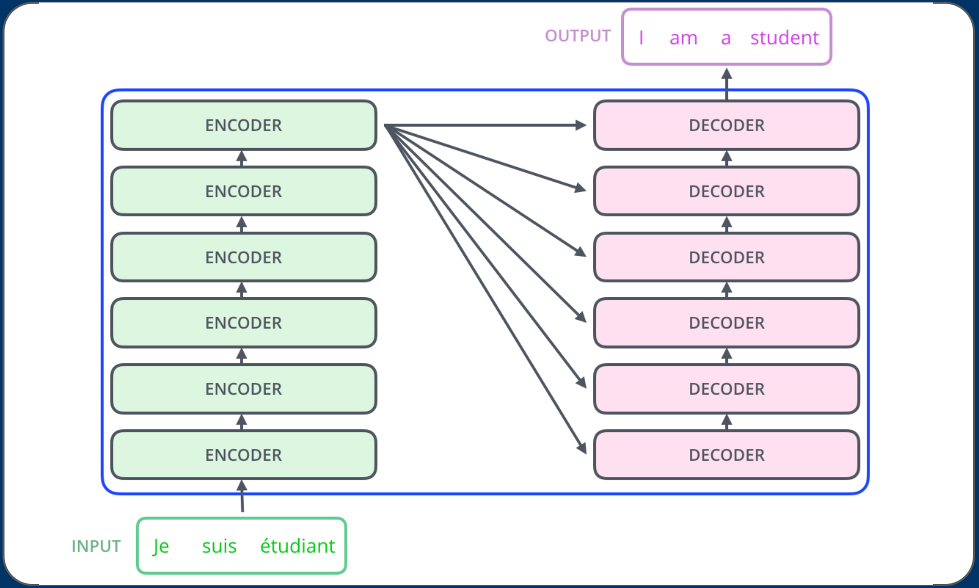
Main Components

Under the hood, we see an encoding component, a decoding component, and connections between them.

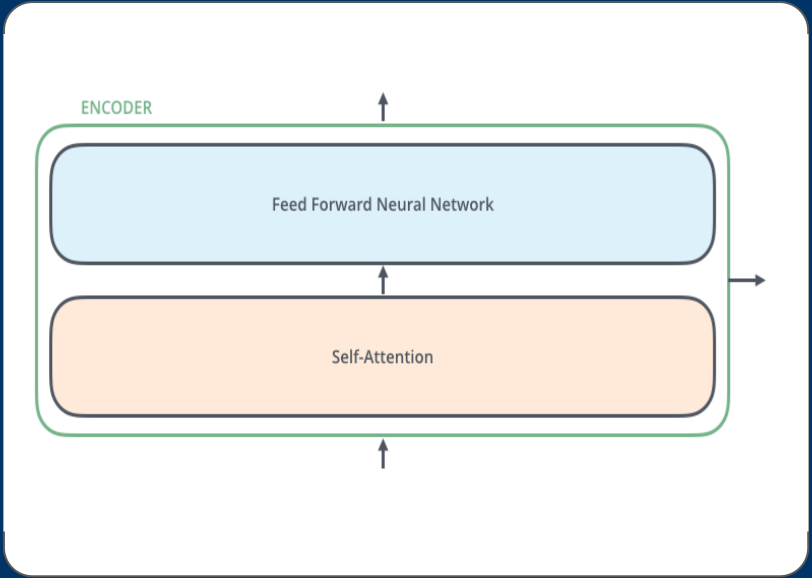


Architecture

The encoding component is a stack of encoders, there’s nothing magical about the number six, one can definitely experiment with other arrangements.
The decoding component is a stack of decoders of the same number.

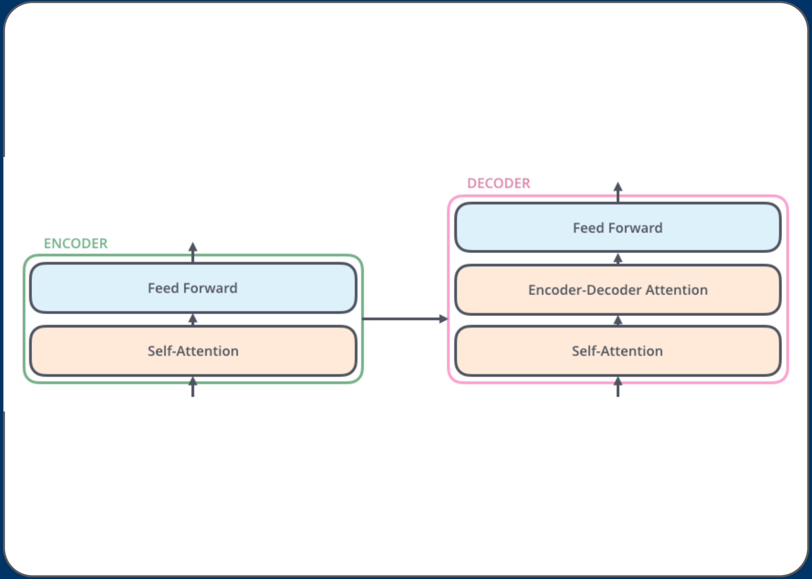


Architecture



The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sublayers.

Architecture



The encoder inputs first flow through a **self-attention** layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.

The outputs of the self-attention layer are fed to a feed-forward neural network (**FFN**). The exact same **FFN** is independently applied to each position.

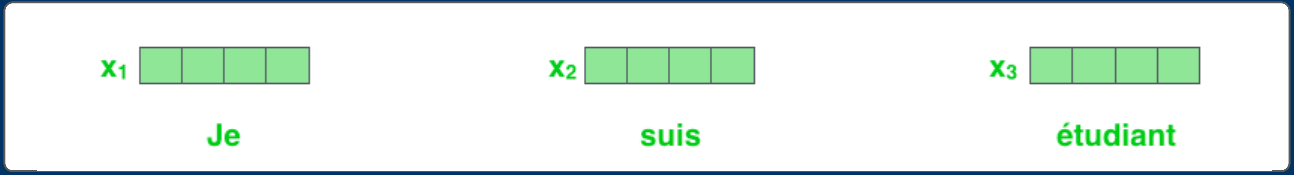
The decoder has both those layers, but between them is an **attention** layer that helps the decoder focus on relevant parts of the input sentence.

Input

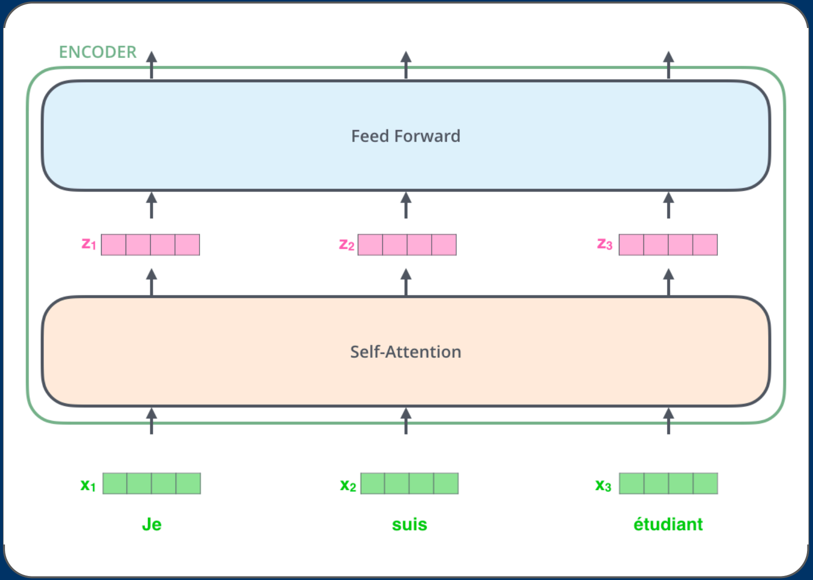
Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size E – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.



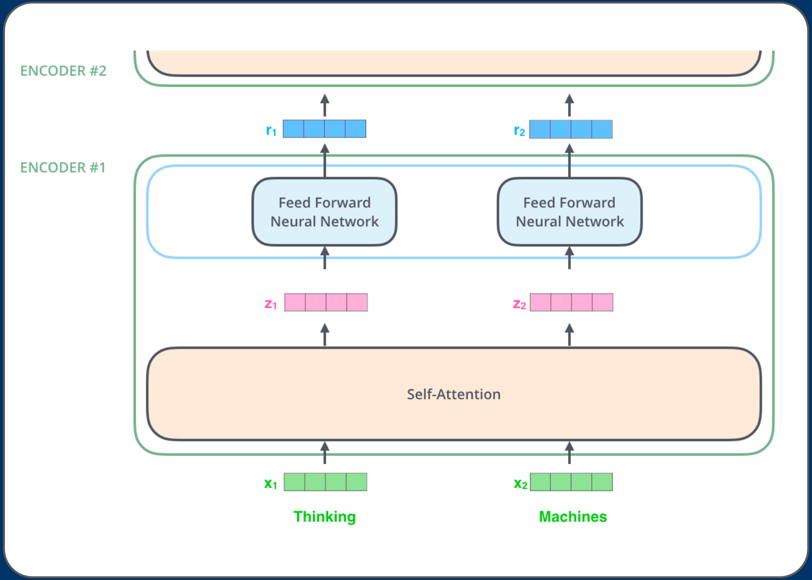
Encoder



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.

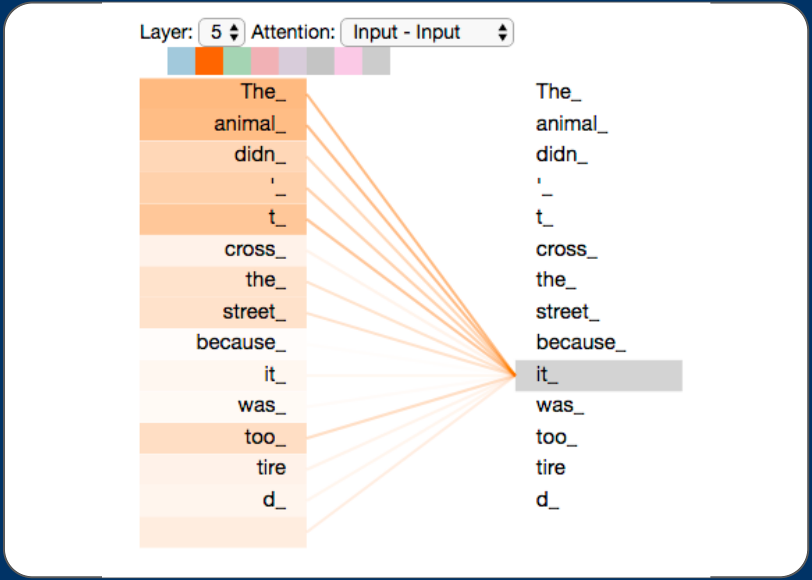
There are dependencies between these paths in the **self-attention** layer. The **FF** layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the **FF** layer.

Encoder



From now on we will take a look at a shorter example of a sentence to see what happens in each sub-layer of the encoder.

Self-Attention



Say the following sentence is an input sentence we want to translate:

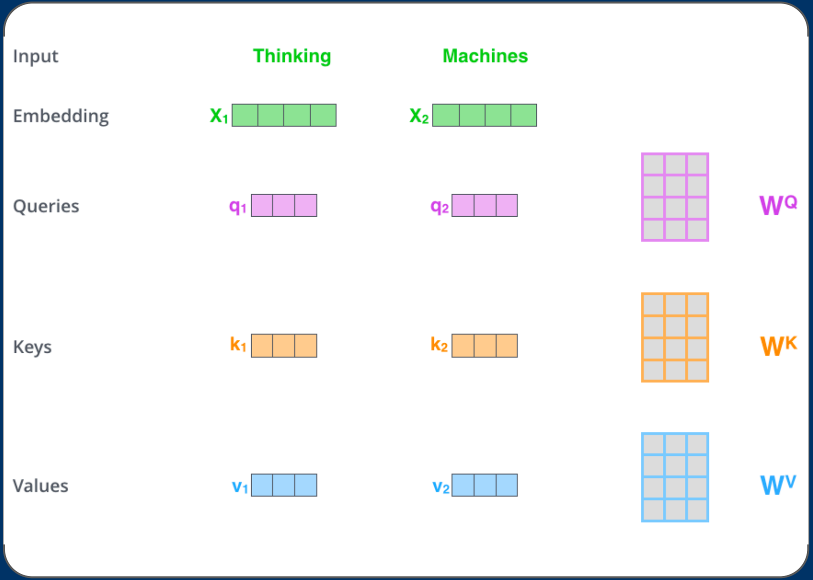
"The animal didn't cross the street because it was too tired"

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.
When the model is processing the word "it", **self-attention** allows it to associate "it" with "animal".

As the model processes each word (each position in the input sequence), **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.

Think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing. Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.

Self-Attention

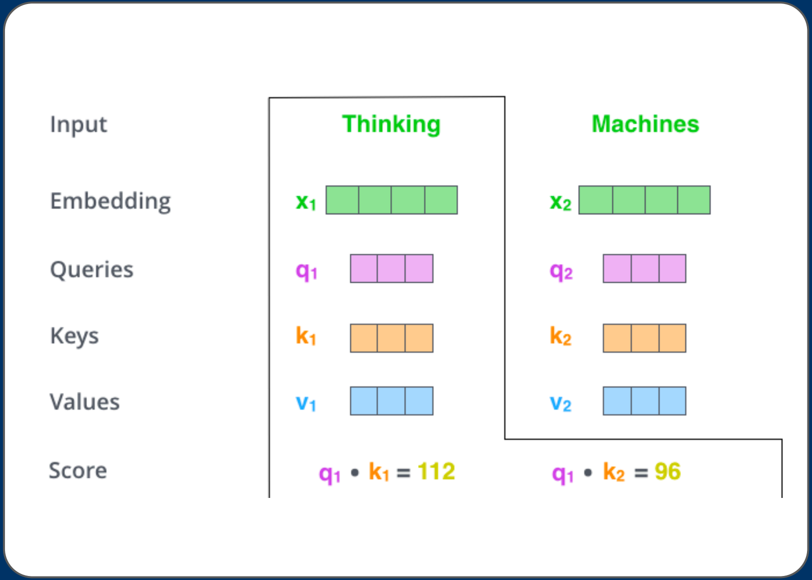


Let’s first look at how to calculate **self-attention** using vectors, then proceed to look at how it’s actually implemented – using matrices.

The first step in calculating **self-attention** is to create three vectors from each of the encoder input vectors (in this case, the embedding of each word). So for each word, we create a **Query** vector, a **Key** vector, and a **Value** vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. They don’t HAVE to be smaller, this is an architecture choice to make the computation of **multiheaded-attention** (mostly) constant.

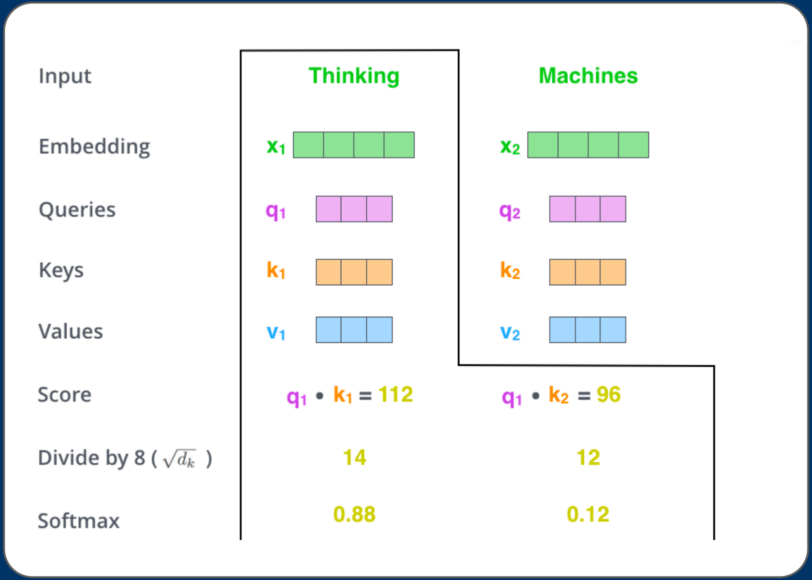
Self-Attention



The second step in calculating **self-attention** is to calculate a **score**. Say we're calculating the **self-attention** for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence 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. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of $\mathbf{q_1}$ and $\mathbf{k_1}$. The second score would be the dot product of $\mathbf{q_1}$ and $\mathbf{k_1}$.

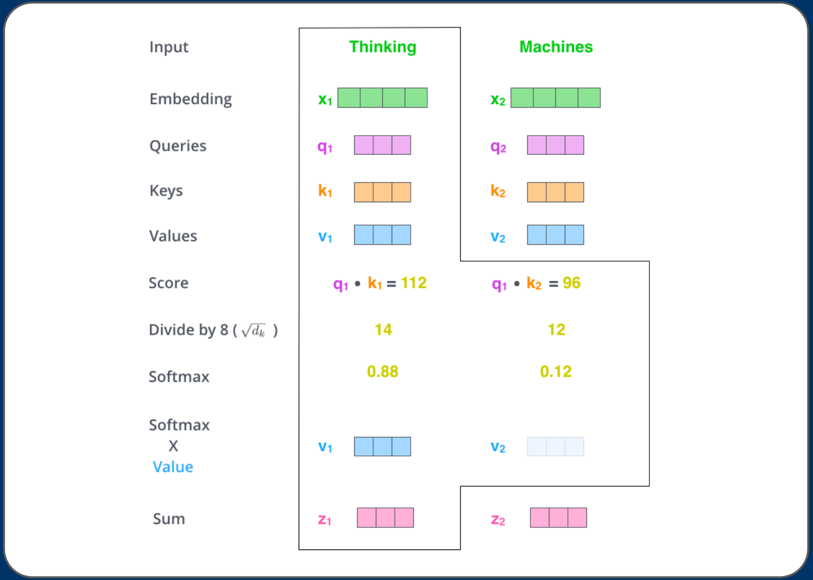
Self-Attention



The third and fourth steps are to divide the **scores** by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a **softmax** operation. **Softmax** normalizes the scores so they're all positive and add up to 1.

This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

Self-Attention

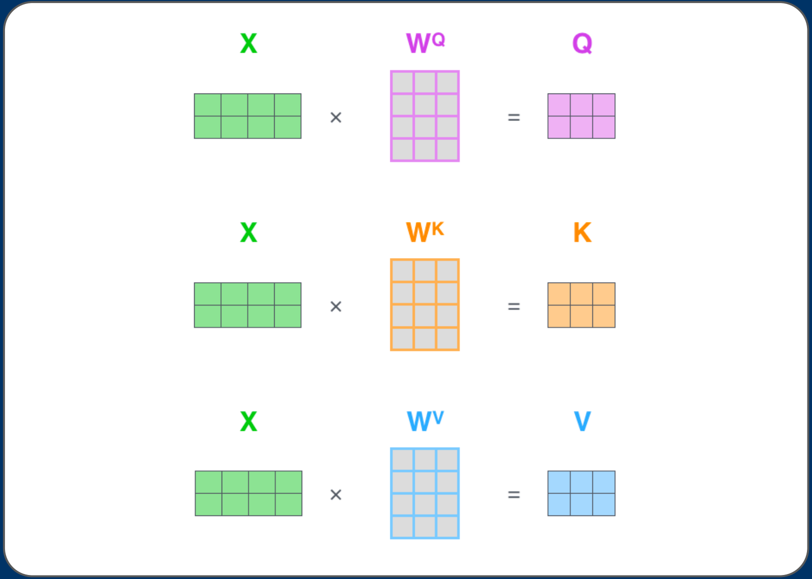


The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

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).

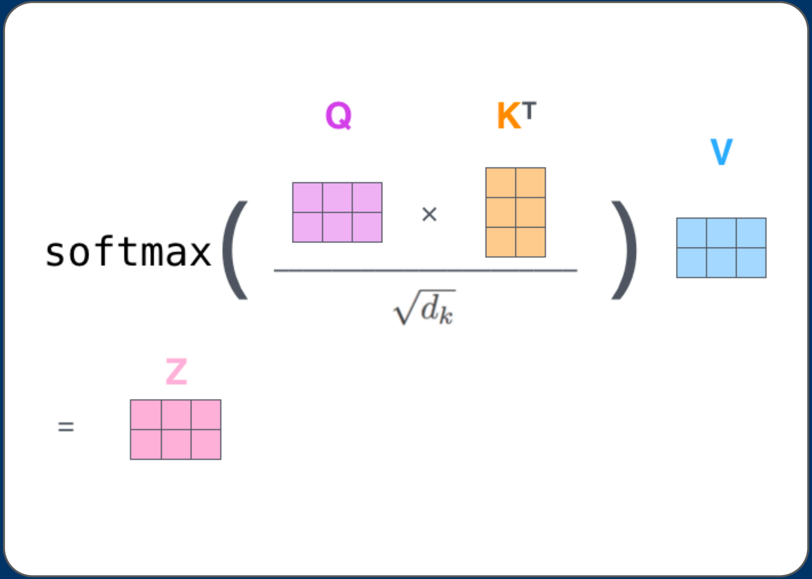
The resulting vector is one we can send along to the FFN. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

Matrix Calculation of Self-Attention



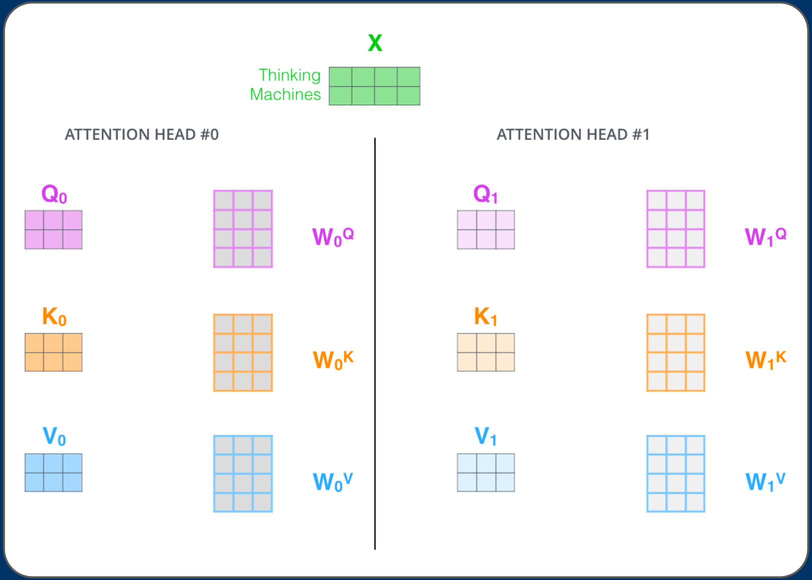
The first step is to calculate the **Query**, **Key**, and **Value** matrices. We do that by packing our embeddings into a matrix **X**, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V).

Matrix Calculation of Self-Attention



Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

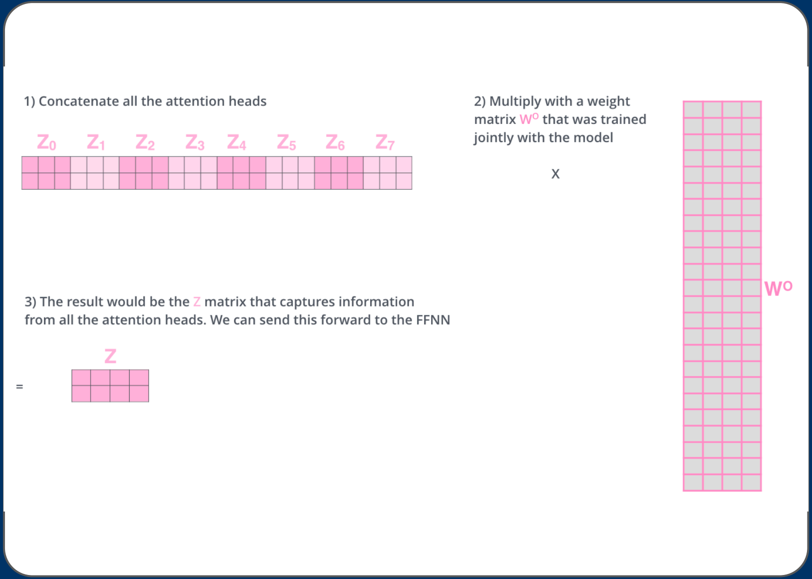
Multi-Head Attention



This improves the performance of the attention layer in two ways:

- It expands the model's ability to focus on different positions. Yes, in the image, $\mathbf{z_1}$ contained a little bit of every other encoding, but it could be dominated by the the actual word itself.
- It gives the **attention** layer **multiple** "**representation subspaces**". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized.

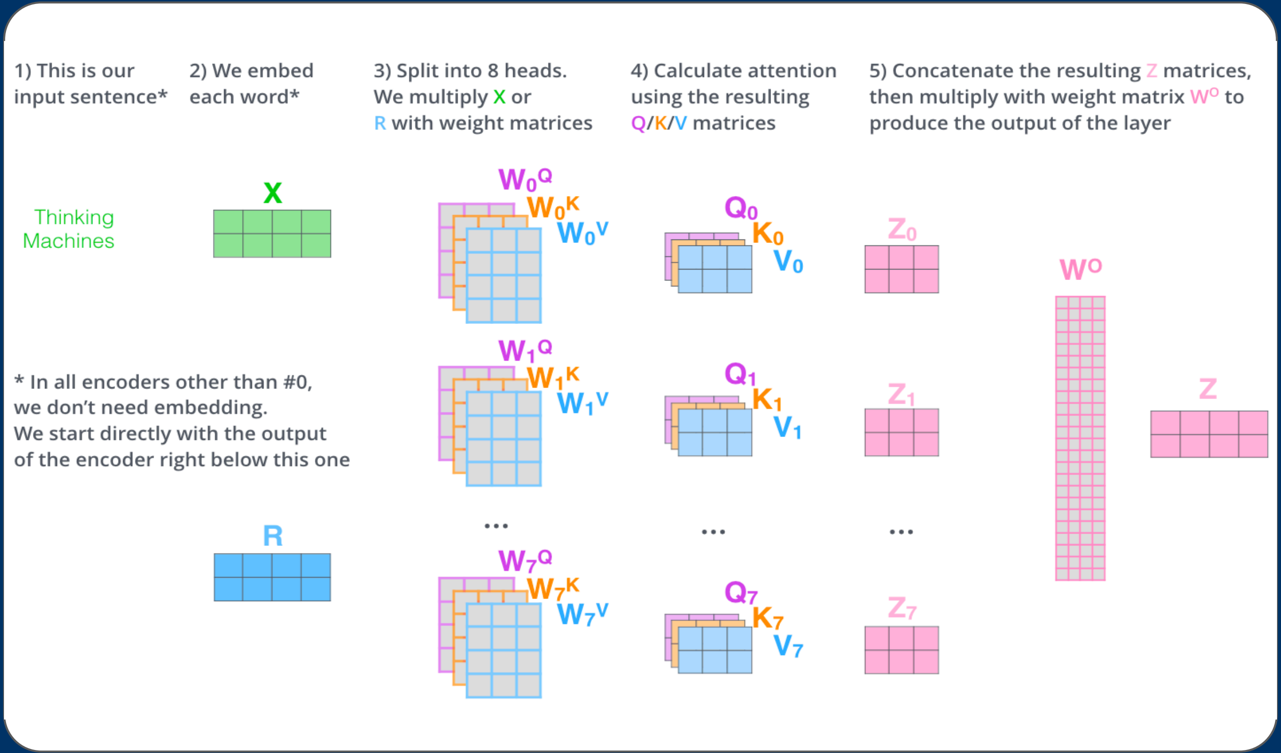
Multi-Head Attention


1) Concatenate all the attention heads
Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7
2) Multiply with a weight matrix W^O that was trained jointly with the model
x
W^O
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

If we do the same **self-attention** calculation we outlined before, just eight different times with different weight matrices, we end up with eight different **Z** matrices, this leaves us with a bit of a challenge. The **FFN** 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.

How do we do that? We concat the matrices then multiply them by an additional weights matrix **W^O**.

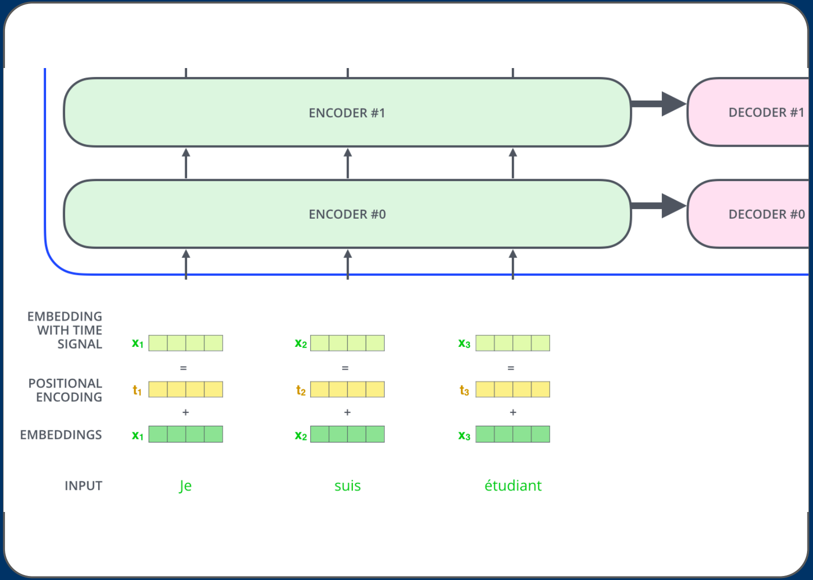
Multi-Head Attention



1) This is our input sentence*
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
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

Let's try to put them all in one visual so we can look at them in one place.

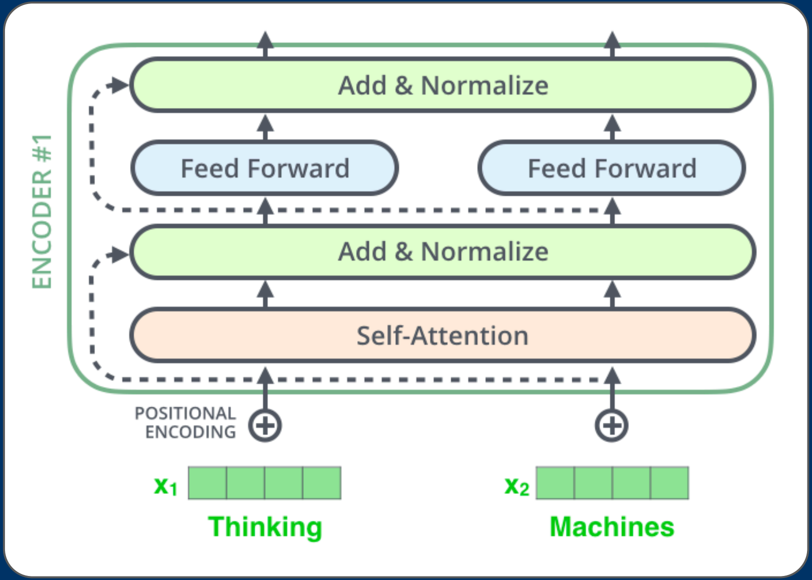
Positional Encoding



One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

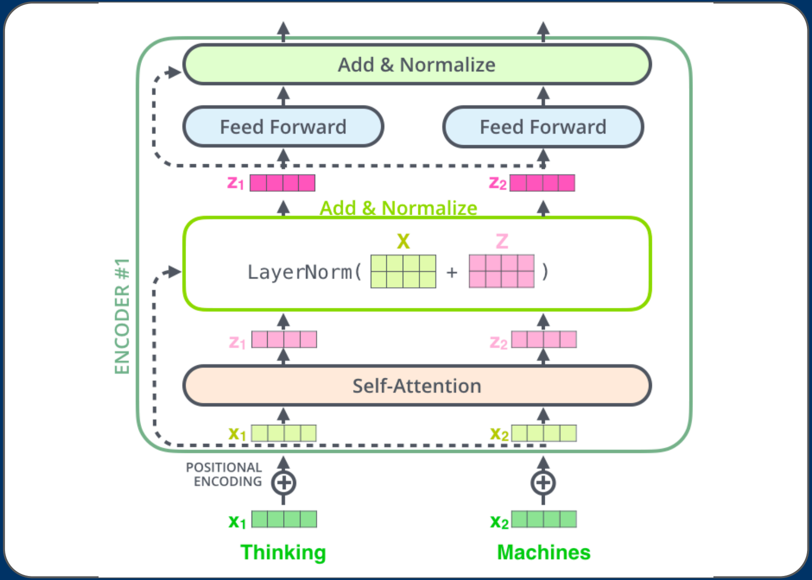
To address this, the Transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the **position** of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the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

The Residuals



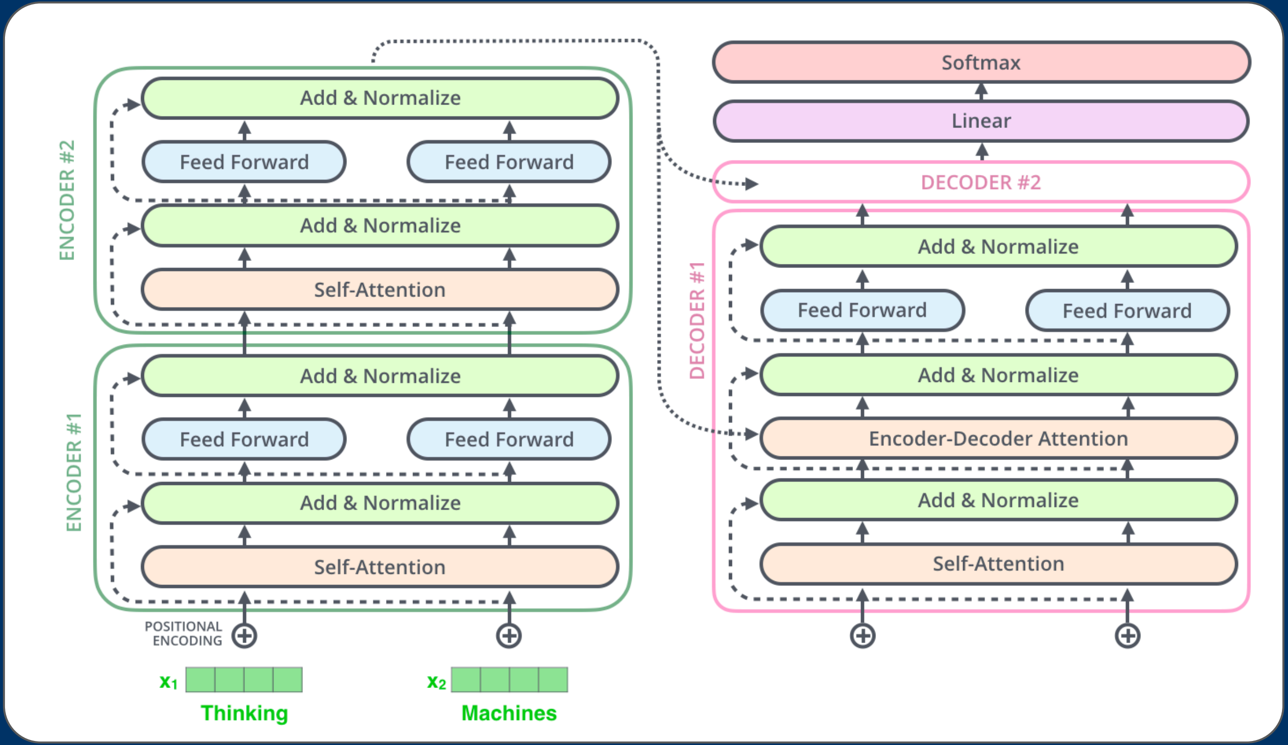
One detail in the architecture of the encoder, is that each sublayer (**self-attention**, **FFNN**) in each encoder has a residual connection around it, and is followed by a **layer-normalization** step/operation (no learning).

Layer Normalization Operation



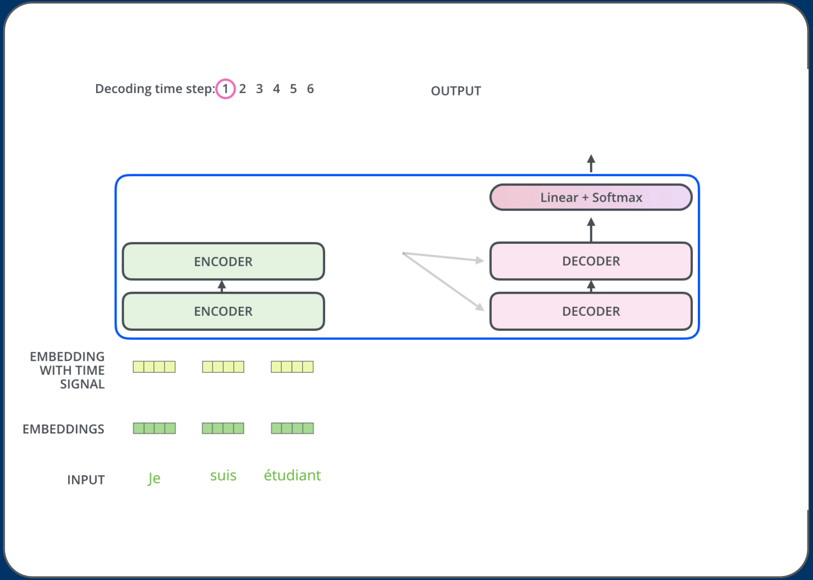
If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

The Architecture



This goes for the sublayers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this.

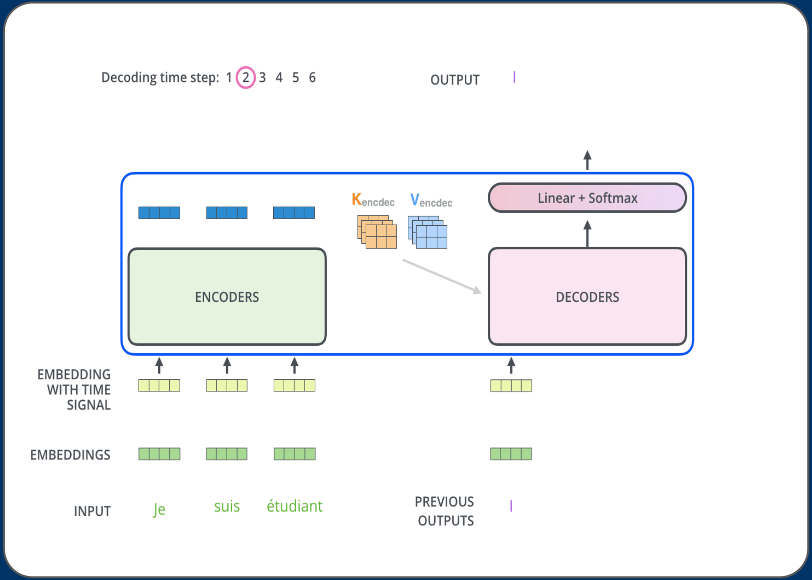
Decoder



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 starts 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.

Decoder



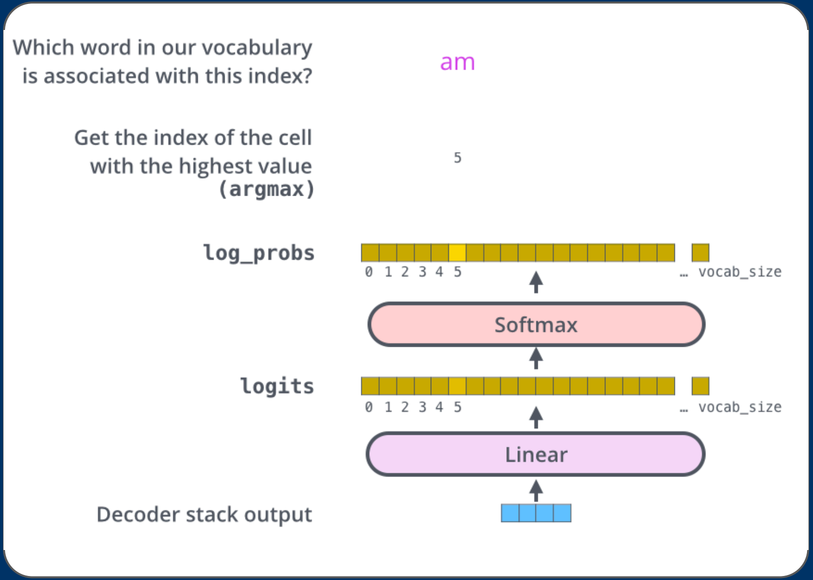
The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like 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 Attentio**n" 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?
am
Get the index of the cell with the highest value (argmax)
5
log_probs
0 1 2 3 4 5 … vocab_size
Softmax
logits
0 1 2 3 4 5 … vocab_size
Linear
Decoder stack output

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 **FFNN** that projects the vector produced by the stack of decoders, into a much, much larger vector called a **logits** vector.

Each cell in the **logits** vector 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.

Conclusion

Significantly improves NN models' performance:

It's very useful to allow decoder to focus on certain parts of the source sentence.

Solved the bottleneck problem:

Attention allows decoder to look directly to source; bypass bottleneck

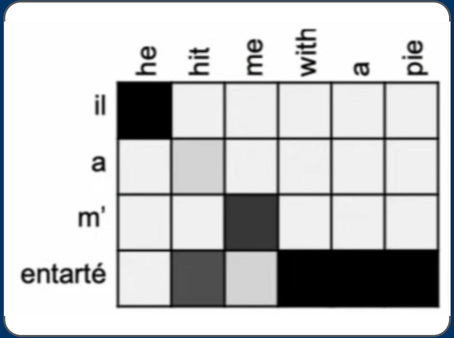
Helps with vanishing gradients problem:

Provides shortcut to farway states - ski-connections

Provides some interpretability:

By inspecting Attention distribution, we can see what decoder was focused on

We get (soft) alignment for free - this is cool because we never explicitly trained an alignment system. The network just learned alignment by itself.



Thank You