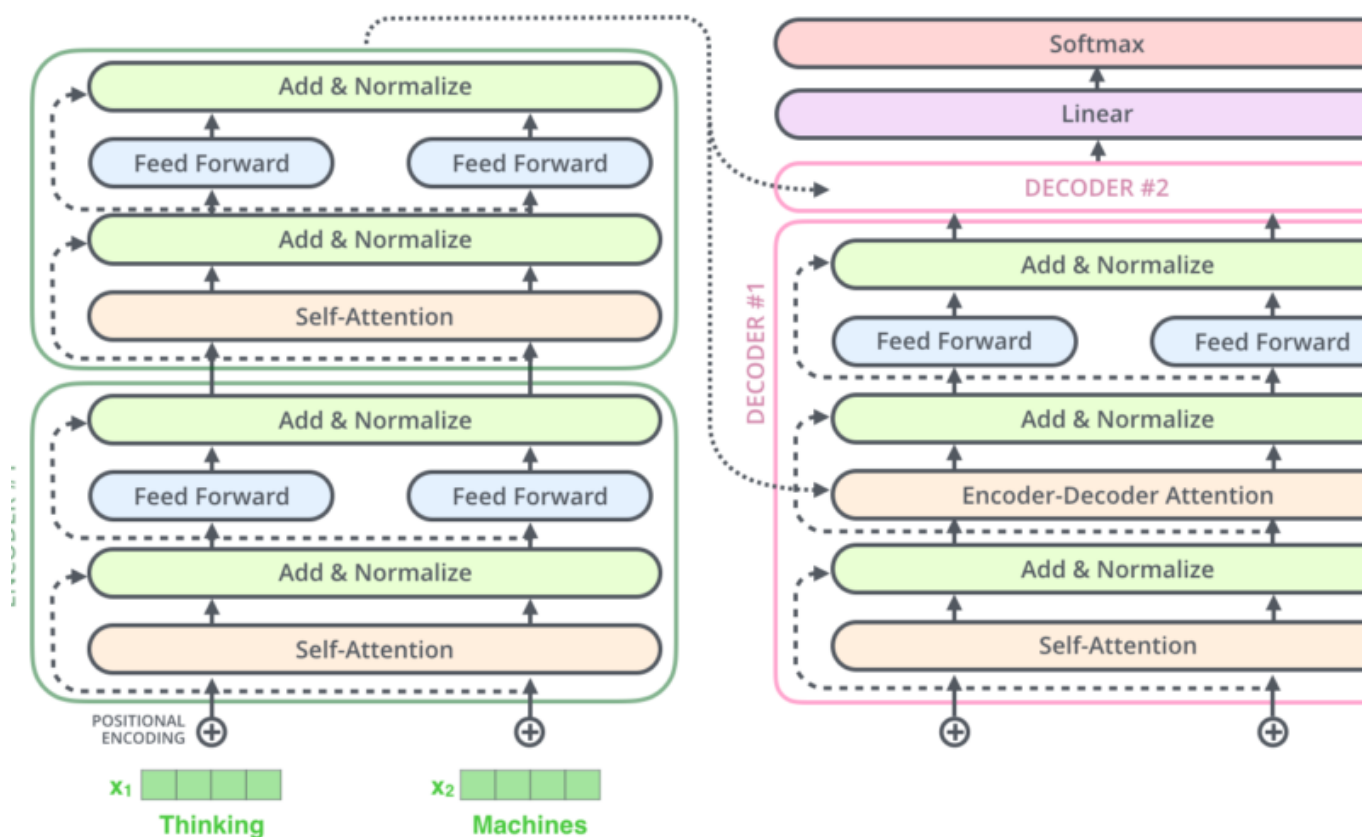


F — FOUNDATION MODELS & GEN AI

Transformers Illustrated

by **André Lopes** · August 23, 2023 · ⌚ 18 minute read

One of the most famous and educational articles on the architecture of Transformers, the basis of Generative AI models, now in Portuguese. Available on BRAINS.

we first need to understand the Attention Mechanism , presented by Google in 2017 in the article *Attention is All You Need* .

In our previous post, **Visualizing a Translation Model (Seq2seq with Attention)** , we paid more attention to the Attention mechanism itself. This concept initially improved the performance of Neural Networks in translation applications.

Now, in this post, we will talk about the architecture of Transformers. An architecture that uses Attention to accelerate the speed at which language models can be trained. The biggest benefit comes from the fact that Transformers makes use of parallelization in its training process. In fact, Google Cloud recommends using Transformers as a reference model for its Cloud TPU offering . So let's try to break down the model and analyze how it works.

A hard-coded TensorFlow implementation of Transformers is available as part of the Tensor2Tensor package . The Harvard Natural Language Processing (NLP) group created an annotated guide to the article with implementation in PyTorch .

In this post, originally called ***The Illustrated Transformer*** , the author, **Jay Alamar** , attempts to simplify the concepts and present them one at a time. The objective is to facilitate understanding for people without in-depth knowledge of Deep Learning.

Jay **Alamar** is a specialist in Natural Language Processing. He wrote some super educational articles on his blog (<https://jalamar.github.io/>). With his permission, we are bringing some of them to BRAINS, in Portuguese.

Note: All text below was translated from the original content, except for the translation notes explained.

A high-level look

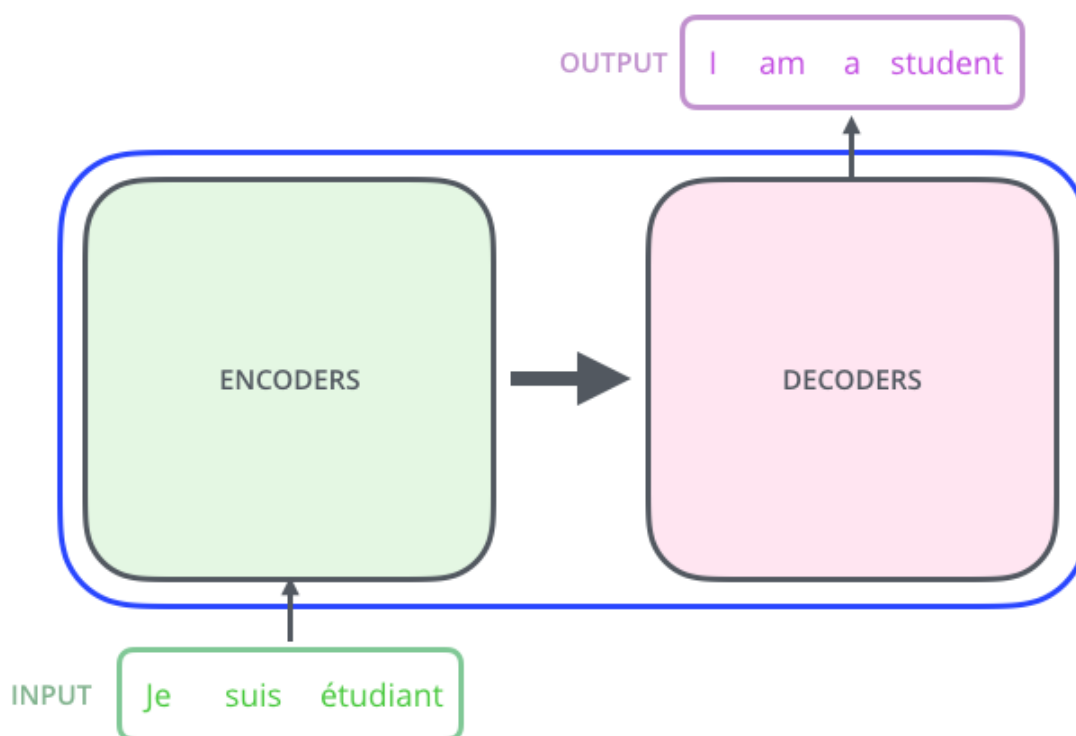


Let's start by looking at the model as a black box. In a translation application, the model would receive a sentence in one language and return its translation in another. In the case here, *"I am a student"* would be translated from French (*"Je suis étudiant"*) to English (*"I am a student"*).

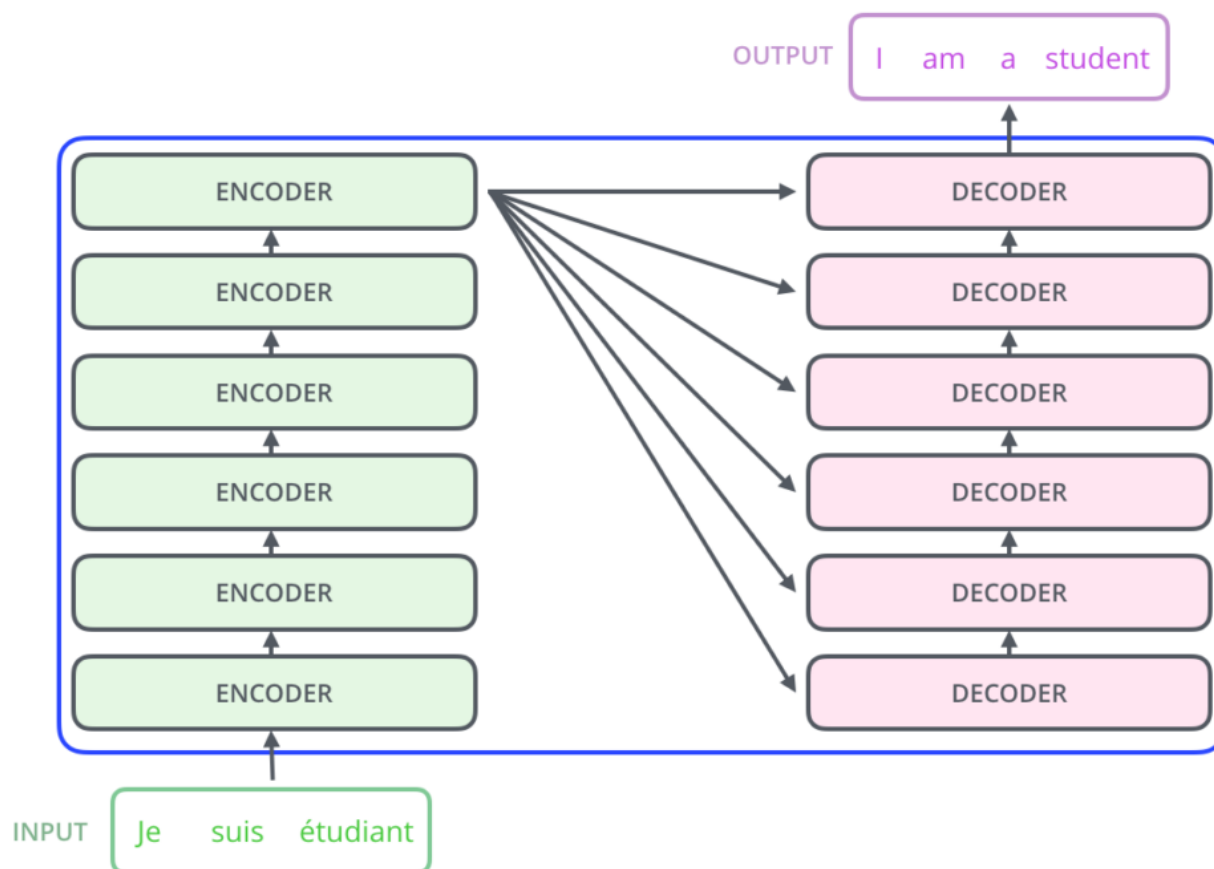


Transformers translating content.

Opening the hood of this *Optimus Prime* wonder , we can see encoding components (**Encoders**), decoding components (**Decoders**) and connections between them.

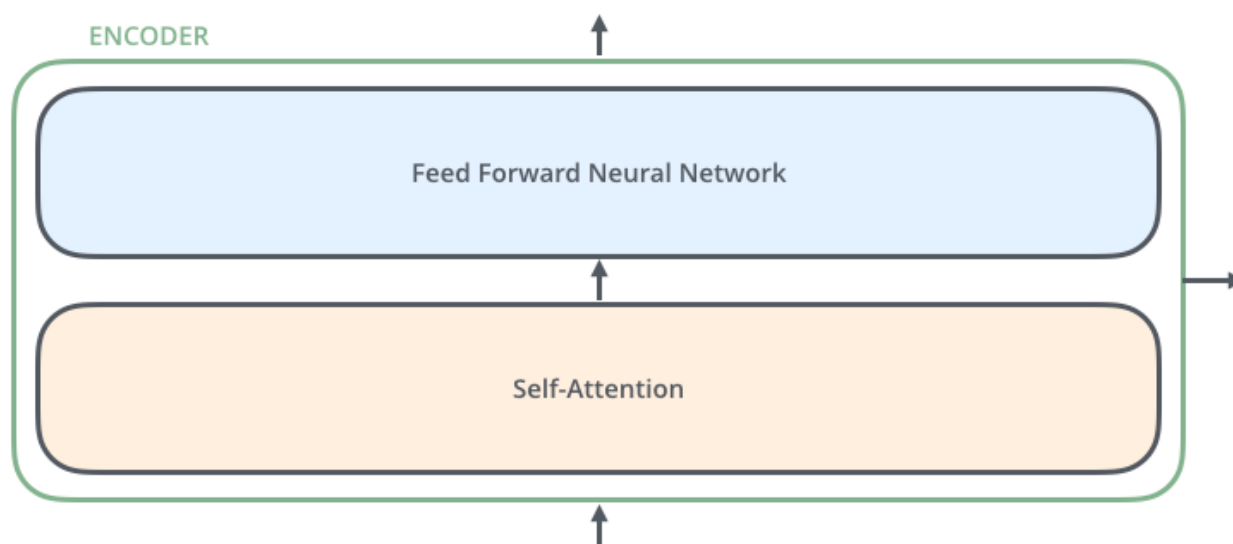


The encoding component is a stack of **Encoders** (the original article stacks six of them on top of each other – there is nothing magical about the number six, we can definitely experiment with other quantities). The decoding component is a stack of **Decoders** of the same number.



Stacks of Transformers Encoders and Decoders.

Encoders are all identical in structure (although they do not share weights, *like* neural networks). Each is broken into two sublayers:

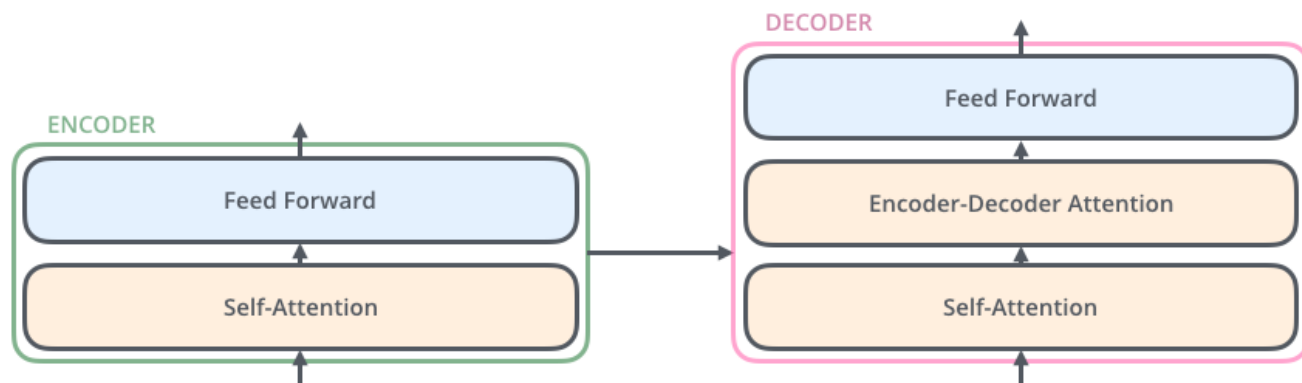


Two layers of the Transformers Encoder.

Encoder inputs first pass through a ***Self-Attention*** layer – a layer that helps the Encoder examine other words in the sentence while encoding a specific word. We will take a closer look at *Self-Attention* later in this post.

The outputs of the *Self-Attention* layer are fed into a ***Feed Forward*** Neural Network , or “Direct Feed”. Exactly the same *Feed Forward Neural Network* is applied independently to each position.

The Decoder has these two layers, but between them, there is an ***Encoder-Decoder Attention*** layer . This helps Decoder focus on the most relevant parts of the input sentence (similar to what **Attention** does in [sequence-to-sequence models – seq2seq](#)).



Three layers of the Transformers Decoder.

Bringing Tensors into the picture

Now that we've seen the main components of the Transformers model, let's start looking at the various vectors/tensors and how they flow between these components to transform the input of a trained model into an output that makes sense.

As is the case in Natural Language Processing (NLP) applications *in* general, we start by converting each input word into a vector using an **Embedding** algorithm .

x_1 
Je

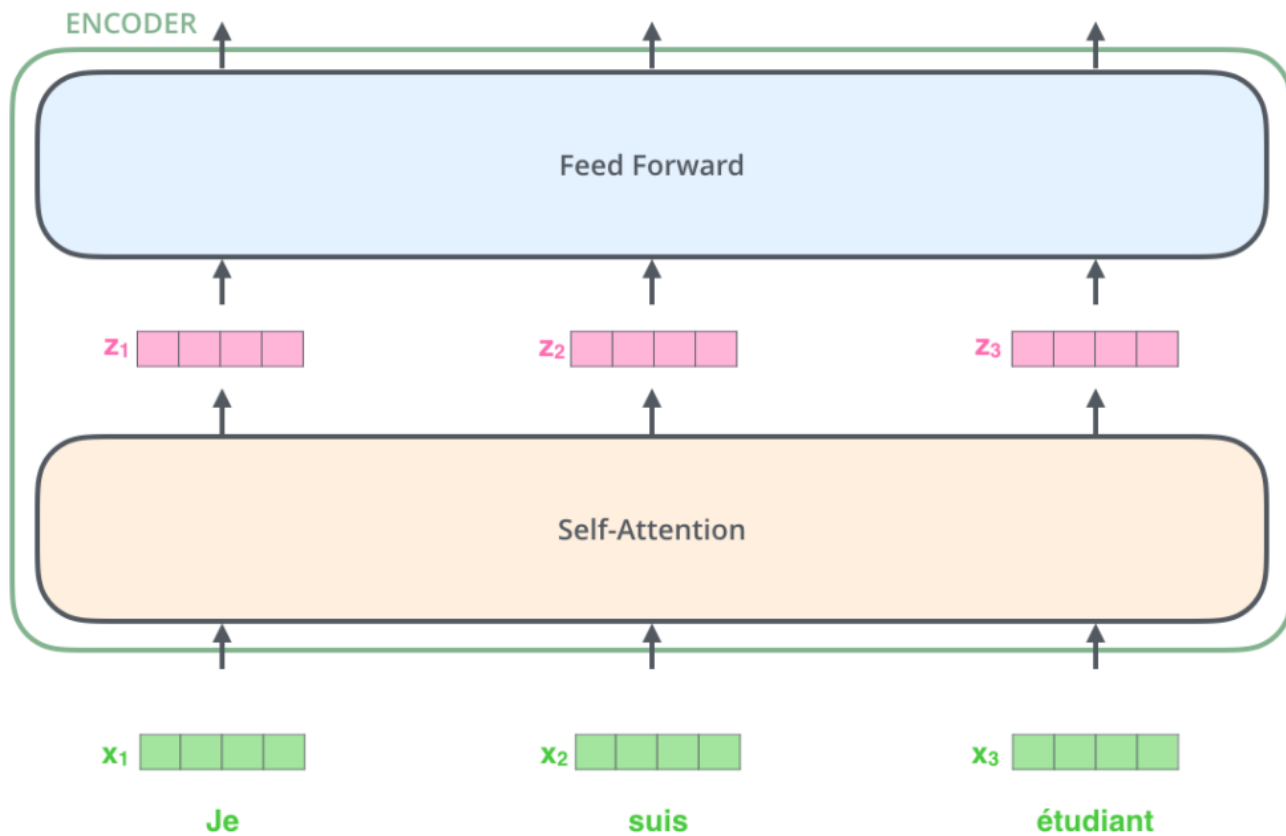
x_2 
suis

x_3 
étudiant

Each word is "embedded" in a vector of size 512. Let's represent these vectors with these simple green boxes.

Embeddings are only generated in the first Encoder, the lowest. The abstraction common to all Encoders is that they receive a list of vectors, each of size 512. In the first Encoder, these vectors would be the Embeddings of the words, but in the other Encoders, they would be the outputs of the Encoder directly below. The size of this list is a hyperparameter that we can define: basically, it would be the length of the longest sentence in our training dataset.

After generating the Embeddings of the words in our input sequence, each one of them passes through each of the Encoder layers.



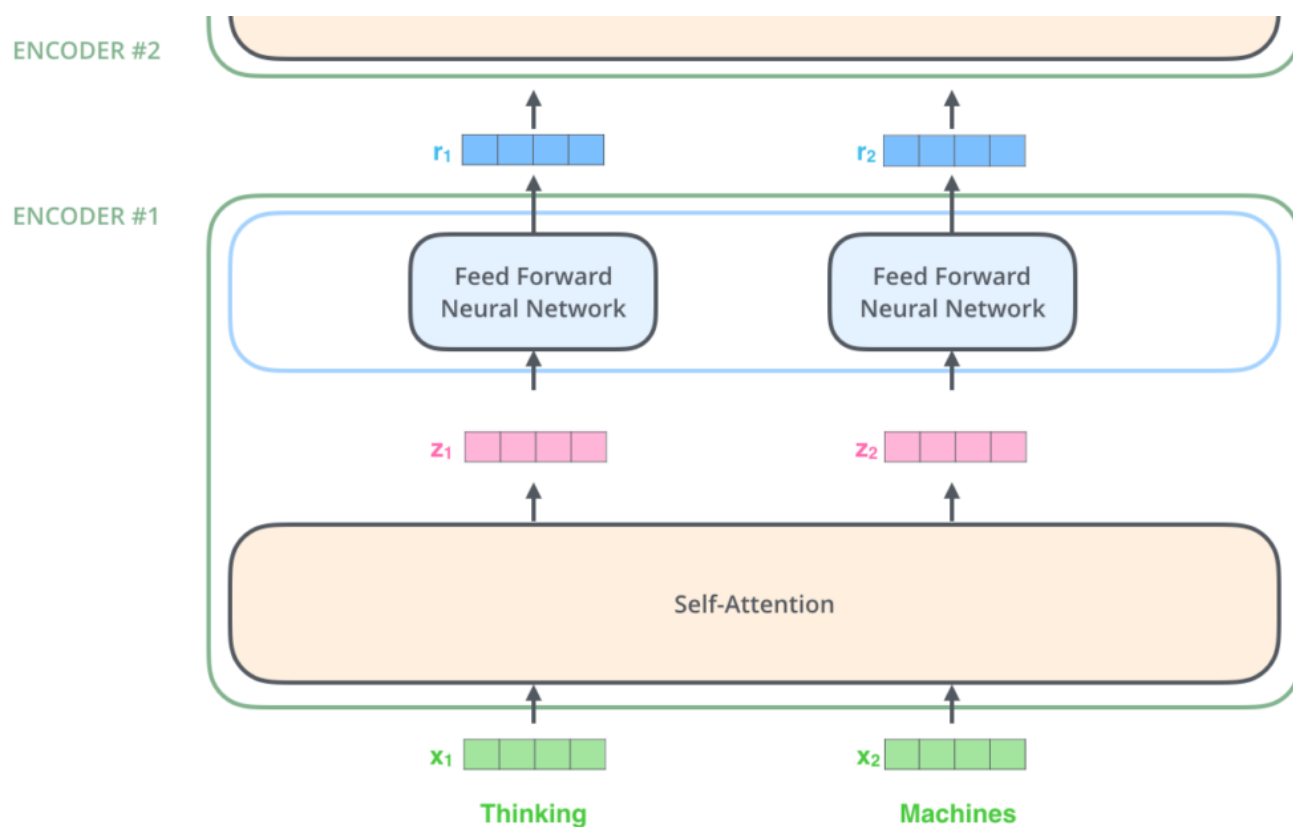
Words transforming into Embeddings and flowing through Encoders.

Here we begin to see a fundamental property of Transformers, which is that the word in each position flows along its own path in the Encoder. There are dependencies between these paths in the *Self-Attention* layer. However, the *Feed-Forward* layer does not have these dependencies, and so multiple paths can run in parallel as they flow through the *Feed-Forward* layer.

Next, let's change the example to a shorter sentence. We will examine what happens in each sublayer of the Encoder.

Now we are doing Encoding!

through the *Self-Attention* layer . It then passes through the *Feed-Forward* Neural Network . Only then does it send the output up to the next Encoder.



The word in each position goes through a *Self-Attention* process . Then, they pass through a *Feed-Forward* Neural Network – exactly the same network in which each vector passes separately through it.

***Self-Attention* at a high level**

Don't be fooled by the fact that we are passing around the term "*Self-Attention*" as if it were a concept that everyone should know. I personally had never come across this concept until I read the article *Attention is All You Need* . Let's simplify how this works.

Let's assume we have the following sentence as input and we want to translate it:

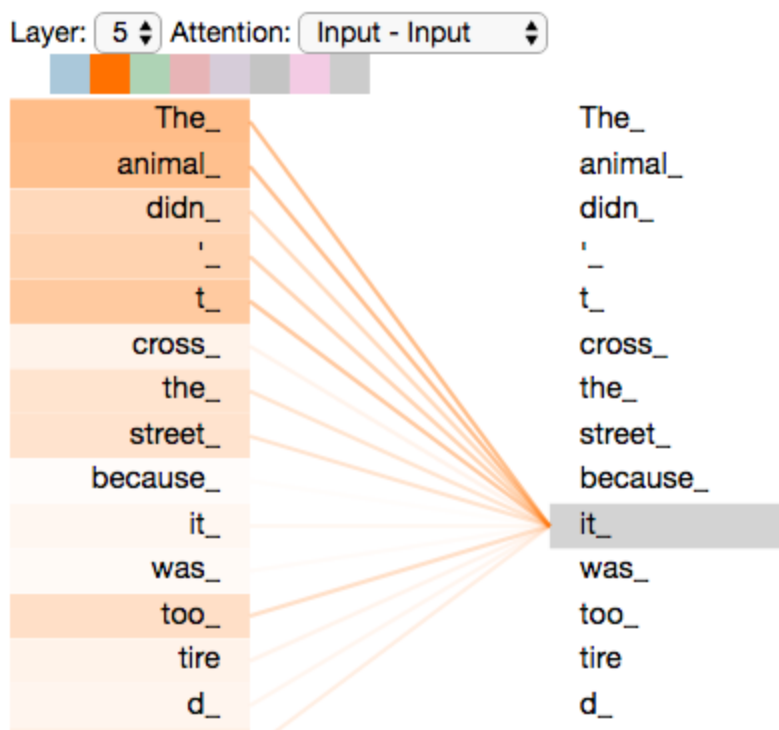
The animal didn't cross the street because it was too tired

But what does the word “it” in this sentence refer to? Are you referring to the street or the animal? Was the street tired? Or was the animal tired? It's a simple question for a human. But not that simple for 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 it better understand this word. This way we achieve better *Encoding* of the word.

If you are familiar with Recurrent Neural Networks (RNNs), think about how maintaining a *Hidden State* allows the RNN to incorporate its representation of previous words/vectors it has already processed with the current word it is processing. *Self-Attention* is the method Transformers use to incorporate the “understanding” of other relevant words into the word we are currently processing.



While encoding *the* word "it" in Encoder #5 (the top Encoder in the stack), part of the **Attention** mechanism was focusing on "The animal" and incorporated a part of its representation in the encoding of "it".

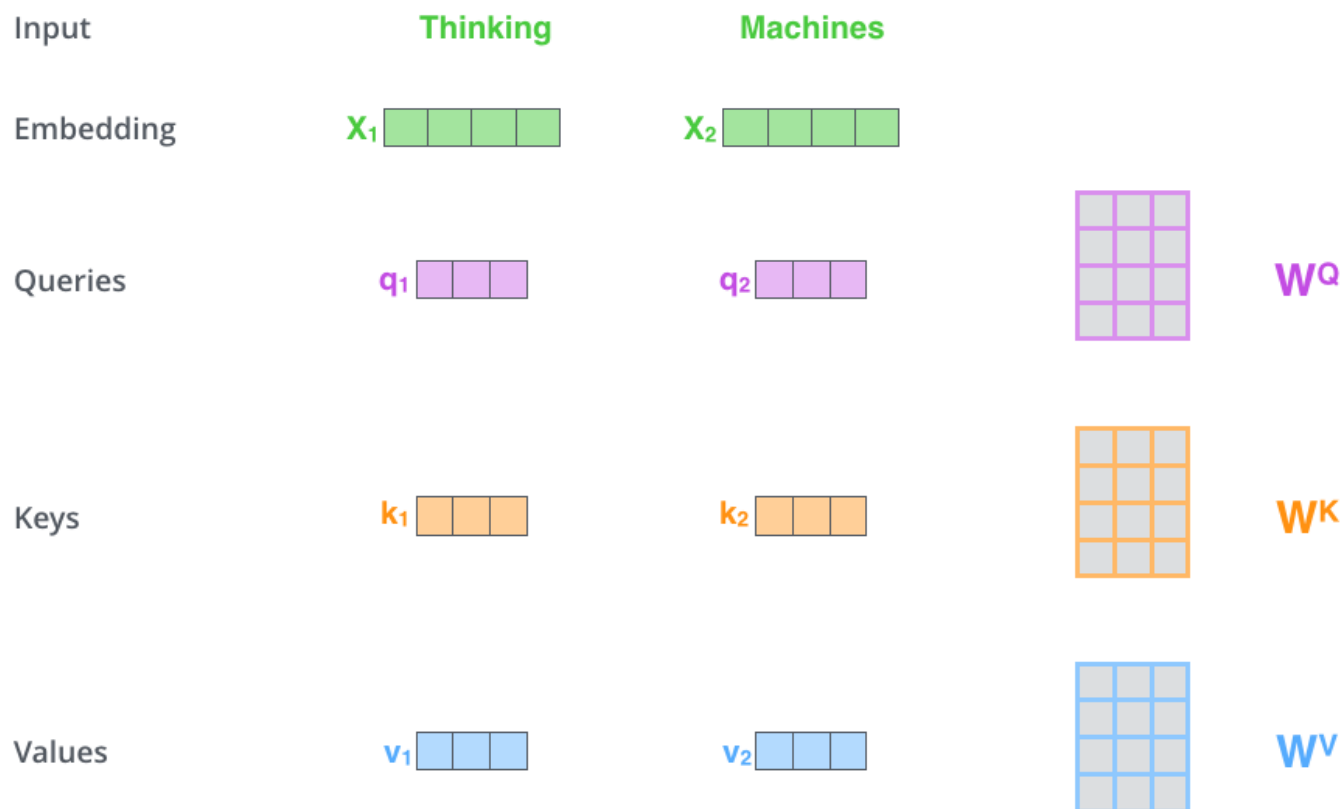
Be sure to check out the [Tensor2Tensor notebook](#) , where you can load a Transformers model and examine it using an interactive visualization.

Self-Attention in detail

Let's first see how to calculate *Self-Attention* using vectors. So we will continue to see how it is actually implemented – using matrices.

The **first step** to calculate *Self-Attention* is to create three vectors from each Encoder input vector. 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 *Embeddings* by three matrices that were learned during the training process.

Note that these new vectors are smaller in dimensions than the Embeddings vector . The dimensionality is 64. The *Embeddings* and Encoder input/output vectors have a dimensionality of 512. They do not NEED to be smaller, this is an architectural choice to make the **Multiheaded Attention** calculation (largely) constant.



Multiplying x_1 by weight matrix W^Q produces q_1 , the "Query" vector associated with this word. We ended up creating a "Query", "Key" and "Value" projection for each word in the input sentence.

What are the *Query*, *Key* and *Value* vectors ?

They are abstractions that are useful for calculating and thinking about Attention. Once you read on about how Attention is calculated below, you'll know pretty much everything you need to know about the roles each of these vectors plays.

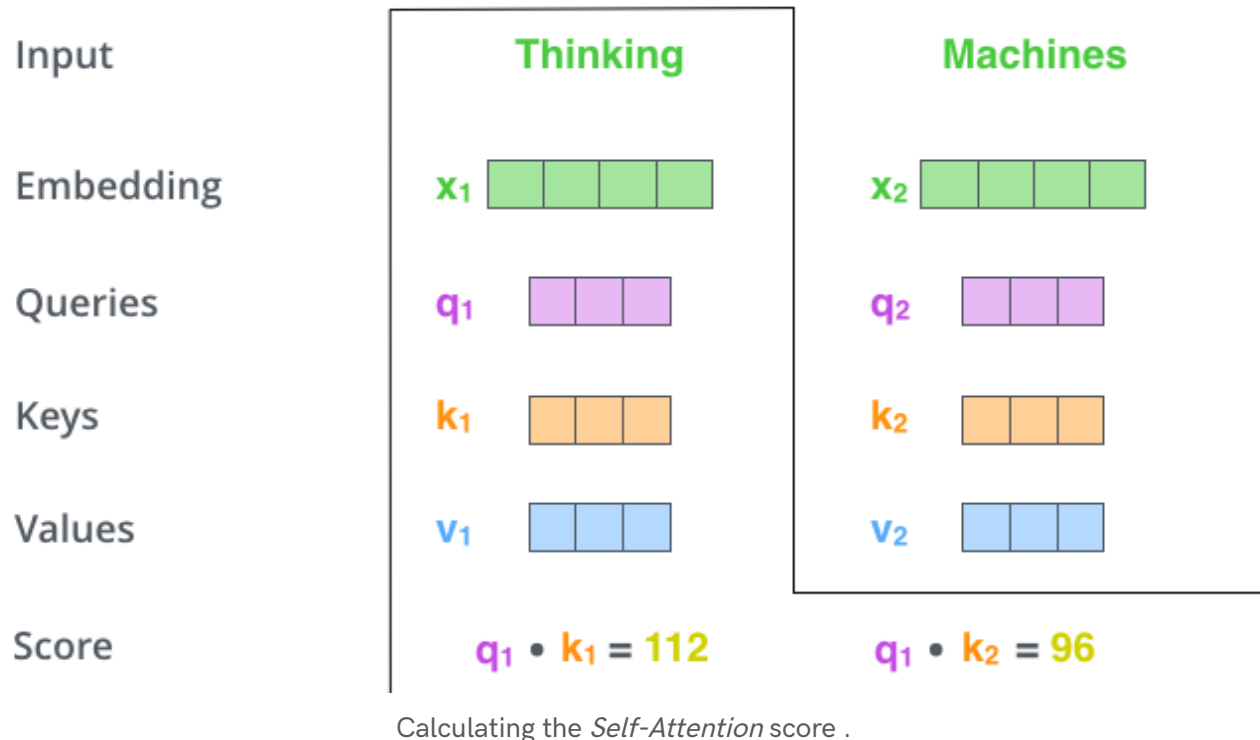
The **second step** to calculating *Self-Attention* is to calculate a score. Let's say we are calculating the *Self-Attention* for the first word in this example, "Thinking". We need to assign a score to each word in the input sentence in relation to this word. The score determines how much we should focus on other parts of the sentence while encoding a word in a certain position.

BRAINS 



are processing *Self-Attention* for the word in position 1 , the first score would be

the scalar product of **q1** with **k1**. The second score would be the scalar product of **q1** with **k2**.



The **third and fourth steps** are to divide the scores by 8. We divide by the square root of the dimensions of the vectors used in the article – 64. This leads us to have more stable gradients. We could have other possible values here, but this is the default. After that, we pass the result through a **Softmax** operation . **The Softmax** function normalizes the scores so that they are all positive and add up to 1.

Input

Embedding

Queries

Keys

Values

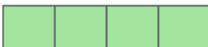
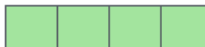
Score

Divide by 8 ($\sqrt{d_k}$)

Softmax

Thinking

Machines

 x_1  x_2  q_1  q_2  k_1  k_2  v_1  v_2  $q_1 \cdot k_1 = 112$ $q_1 \cdot k_2 = 96$

14

12

0.88

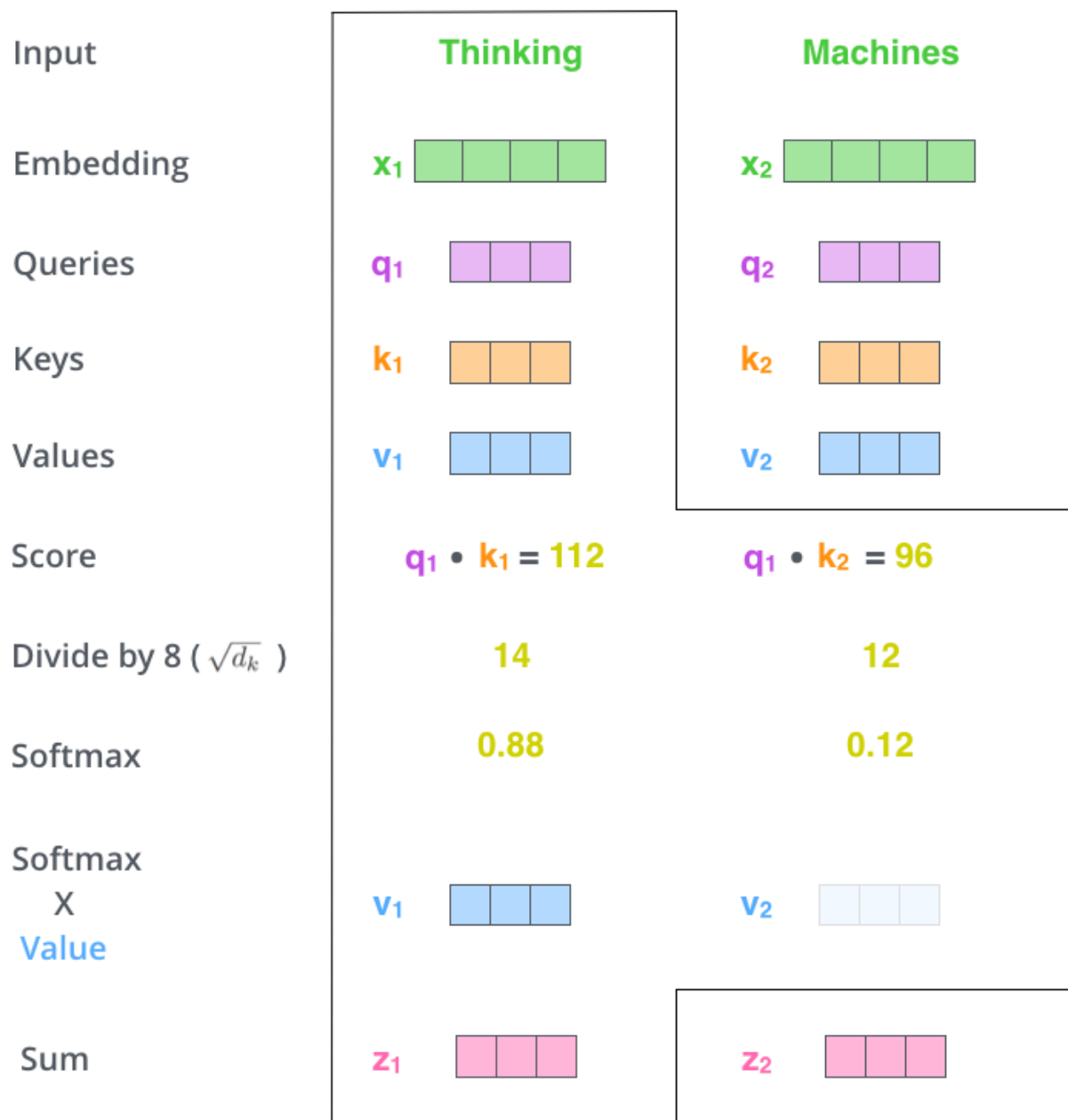
0.12

Normalizing the scores with a Softmax function.

This *Softmax* score determines how expressive each word is in this position. Clearly the word in this position has the highest *Softmax* score . But sometimes it's useful to pay attention to another word that is relevant to the current word.

The **fifth step** is to multiply each *Value vector by the Softmax* score (in preparation for adding them together). The idea here is to keep the values of the word(s) we want to focus on intact, and reduce the importance of irrelevant words (by multiplying them by very small numbers, like 0.001 for example).

The **sixth step is to add the resulting weighted** *Value* vectors . This produces the output of the *Self-Attention* layer at this position (for the first word).



When adding v_1 with v_2 , in this first position, we will have z_1 as the output of the Self-Attention layer.

*This completes the Self-Attention calculation . The resulting vector is one that we can send to the *Feed-Forward* neural network . In the actual implementation, however, this calculation is done in the form of matrices for faster processing. So let's see how this works now, since we've had an introduction to calculation at a word level.*

Self Attention matrix calculation

BRAINS 



The **first step** is to calculate the **Query**, **Key** and **Value** matrices. We do this by stacking our *Embeddings* in an array **X** and multiplying it by the matrices we trained: **W_Q**, **W_K**. It is **W_V**.



Each row in the matrix **X** matches a word from the input sentence. Again we see the difference in the size of the *Embedding* vectors (512, or 4 boxes in the figure) and the q/k/v vectors (64, or 3 boxes in the figure).

Finally, given that we are dealing with matrices, we can condense steps two

BRAINS 



$$\text{softmax} \left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \square & \square \\ \square & \square \\ \square & \square \end{matrix} \end{matrix}}{\sqrt{d_k}} \right) \begin{matrix} \text{V} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix}$$

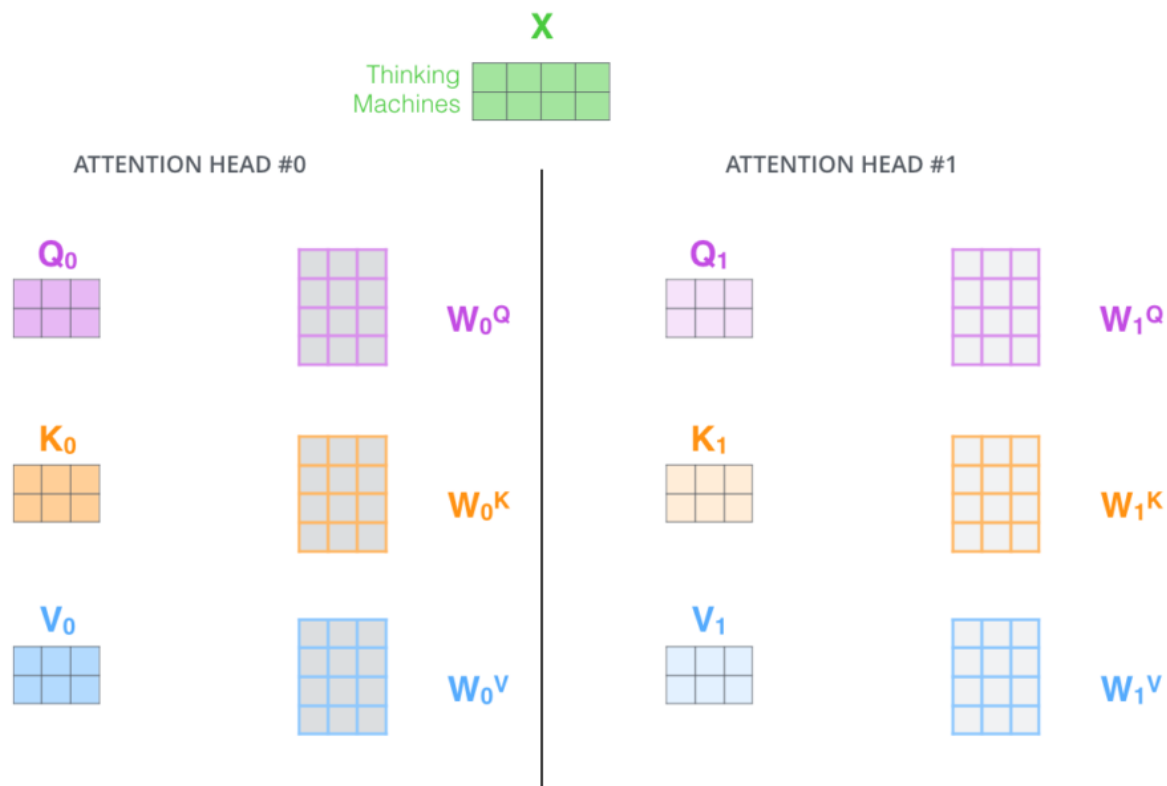
$$= \begin{matrix} \text{Z} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix}$$

The calculation of *Self-Attention* in the form of matrices.

A multi-headed monster

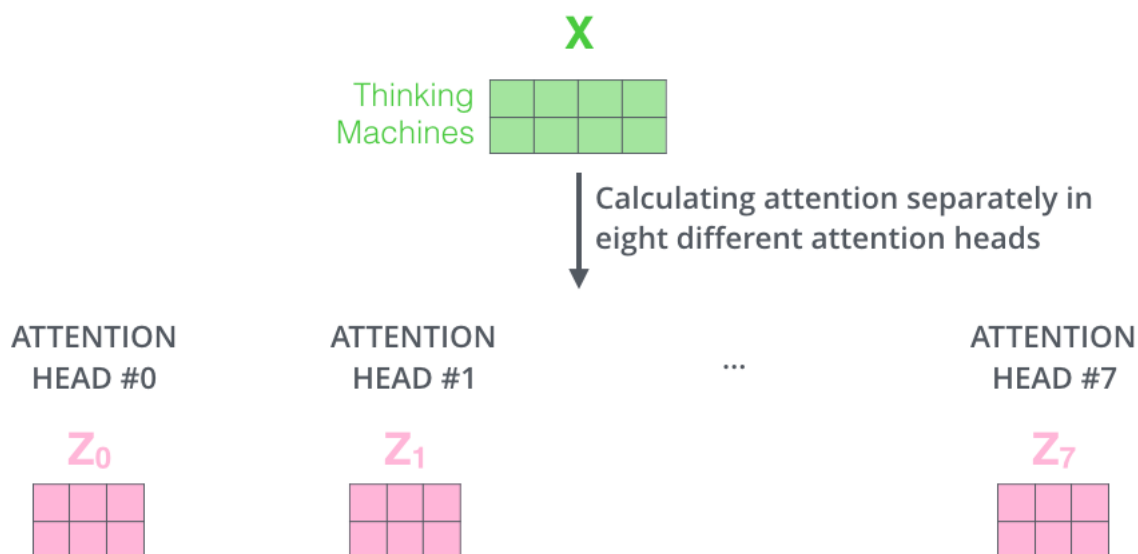
The paper further refined the *Self-Attention* layer by adding a mechanism called “*Multi-Headed*” *Attention* . This improves the performance of the Attention layer in two ways:

1. Expands the model's ability to fork in different positions. Yes, in the example above, z_1 contains a bit of all the other Encodings, but it can be called the word itself. If we are translating a sentence like “*The animal didn't cross the street because it was too tired*” , it would be useful to know which other word “*it*” refers to.
2. It gives the Attention layer multiple “representation subspaces”. As we will see below, with “Multi-Head Attention” we have not just one, but several Query/Key/Value weight matrices (Transformers use 8 Attention “heads”, so we end up having 8 sets for each Encoder/Decoder). Each of these sets is initialized randomly. Then, after training, each set is used to project the input *Embeddings* (or *vectors of lower Encoders/Decoders*) into a *different representative subspace*.



With Multi-Headed Attention, we keep the Q/K/V weight matrices for each *Head* separate, resulting in different Q/K/V matrices. As we did before, we multiply X by $WQ/WK/WV$ to produce the matrices Q/K/V.

If we do the same *Self-Attention* calculations that we highlighted above, only eight different times with different weight matrices, we end up with eight different Z matrices.



This leaves us with a challenge. *The Feed-Forward* layer is not expecting 8 matrices. It's expecting a single array (one vector for each word). So we need a way to condense these eight matrices into one.

How do we do it? We concatenate the matrices and then multiply them by an additional weight matrix **W_O**.

1) Concatenate all the attention heads

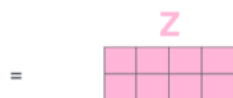


2) Multiply with a weight matrix **W_O** that was trained jointly with the model

x



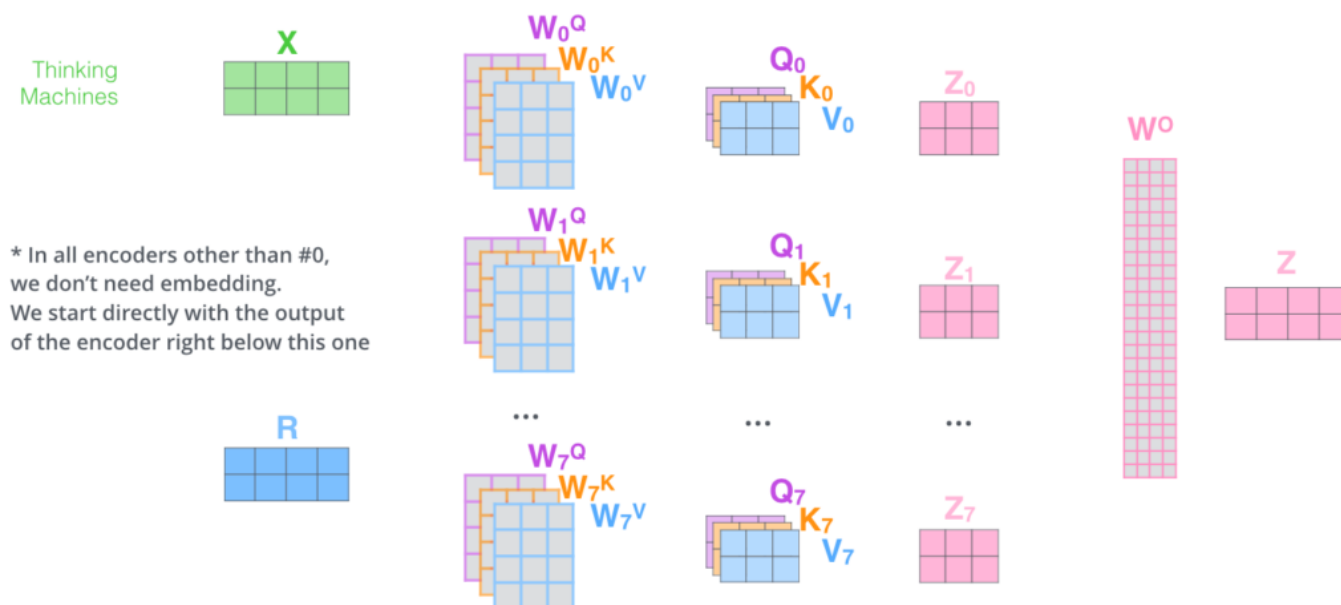
3) The result would be the **Z** matrix that captures information from all the attention heads. We can send this forward to the FFNN



We concatenate all the "heads" of Attention. We multiply with a W^O matrix that was trained along with the model. The result is the matrix **Z** which captures information from all *Attention Heads*. We can then send this information to *Feed-Forward*.

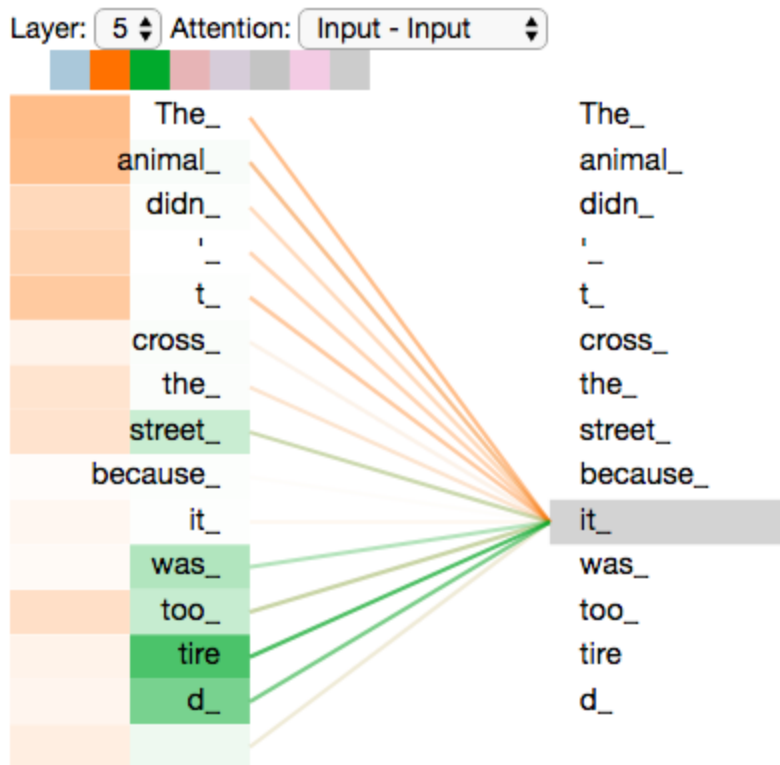
That's basically all we have about *Multi-Headed Self-Attention*. It's a bit of a matrix, I admit. Let us try to put them all in a visual way so we can see them all in one place.

- 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



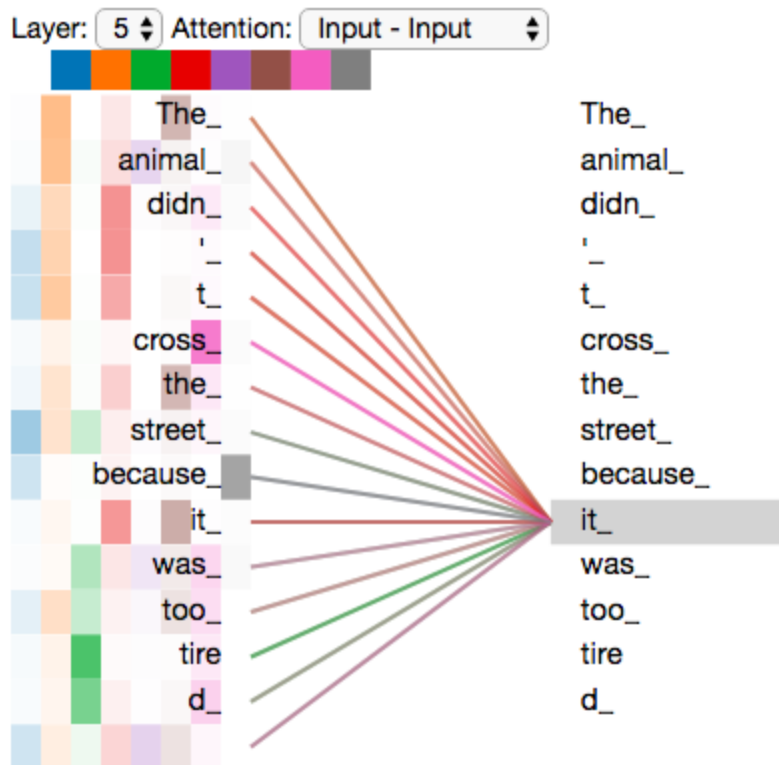
We take the input phrase. We create *Embeddings* X of each word. We divide it into 8 *Heads*. We multiply X or R by weight matrices. We calculate Attention with the resulting matrices $Q/K/V$. We concatenate the resulting matrices Z , then we multiply by the weight matrix W^O to produce the layer's output. And note that for all Encoders after #0, we don't need *Embeddings*. We start directly with the output R of the Encoder below.

Now that we've talked about *Attention Heads*, let's revisit our example from before to see how the different Attention *heads* are focusing when we Encode the word "it" in our example sentence.



When Encoding the word "it", one *Attention Head* is focusing more on "the animal", while another is focusing more on "tired". In a way, the model's representation of the word "it" incorporates some representation of both "animal" and "tired".

If we add all the *Attention Heads* into the picture, however, things start to get more difficult to understand.

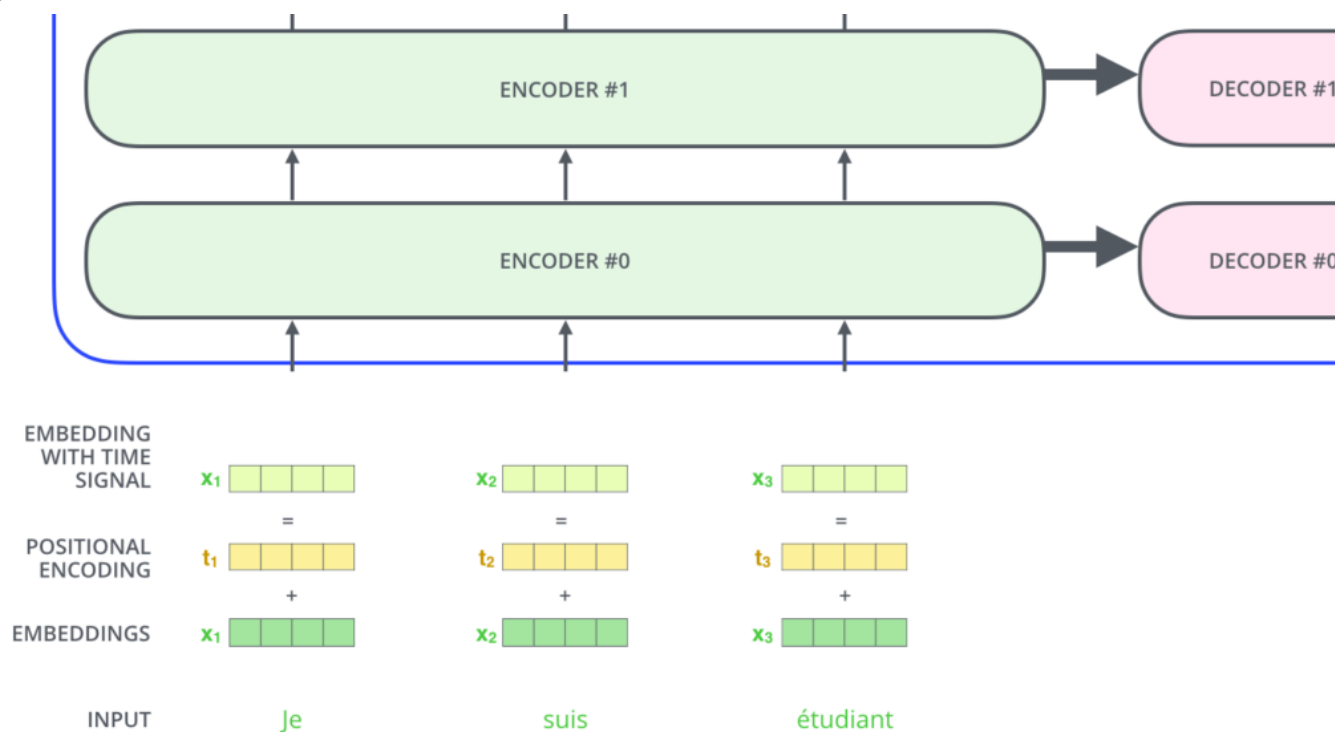


Now it is more difficult to visualize the representation of the word
"it".

Representing sequence order using Positional Encoding

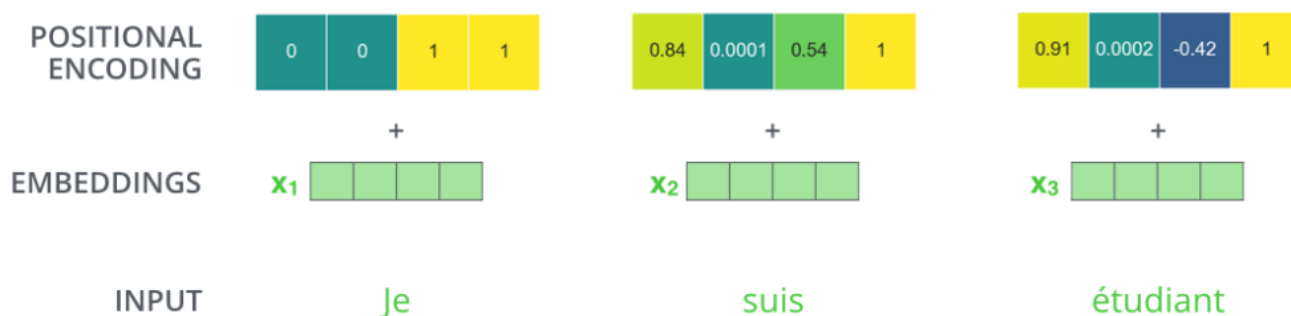
One part that is missing from the model as we have described it so far is a way to take into account the word order of the input sequence.

To address this, Transformers add one more vector to each input *Embedding*. These vectors follow a specific pattern that the model learns, which helps determine the position of each word, or the distance between different words in the sentence. The idea here is that by adding these values to the *Embeddings* we have significant distances between the *Embeddings* vectors when they are projected into Q/K/V vectors during the Attention scalar product.



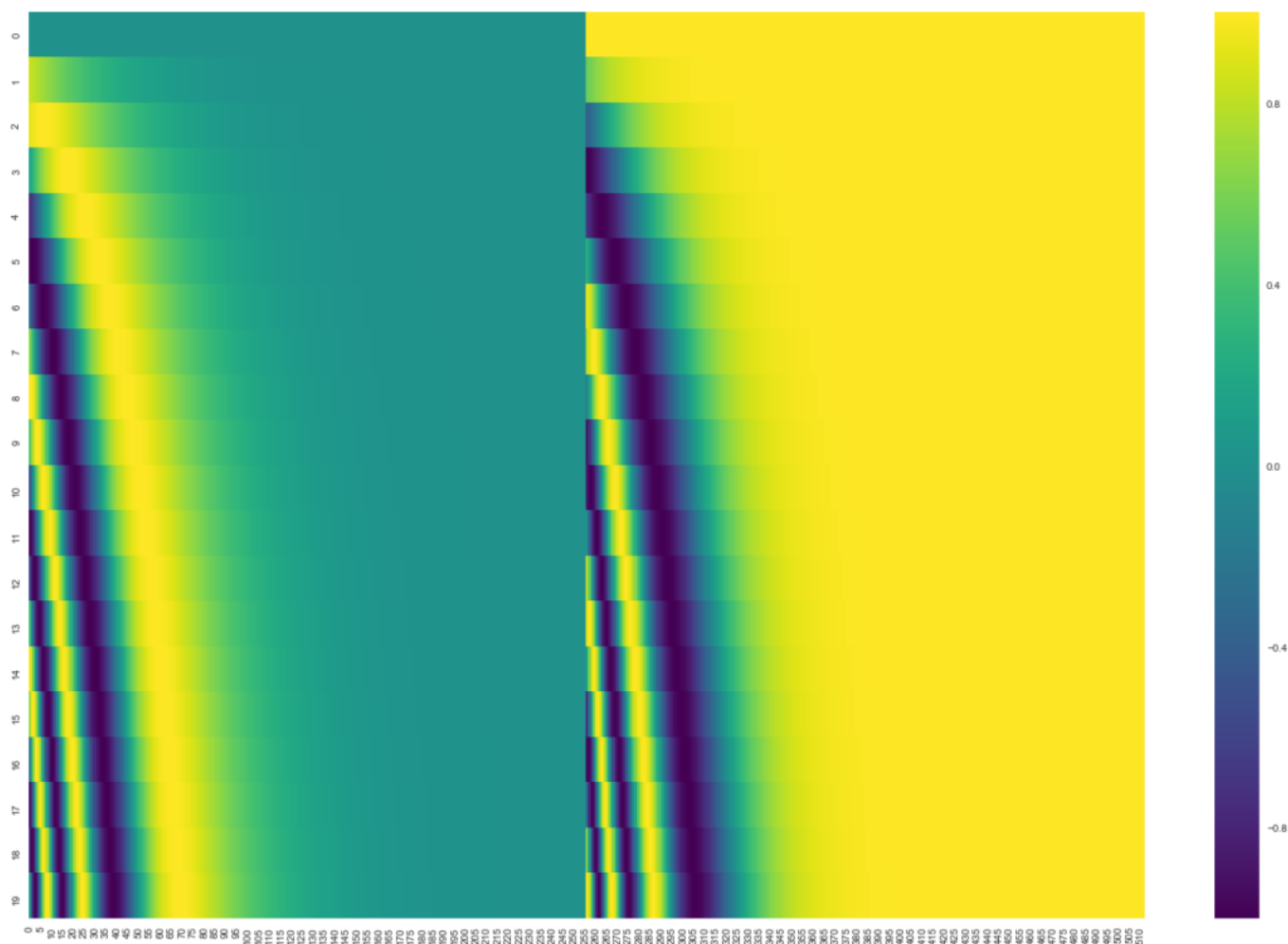
To give the model a sense of word order, we add Positional Encoding vectors – whose values follow a specific pattern.

If we assume that *Embeddings* have a dimensionality of 4, *Positional* Encodings would look like this:



But what does this pattern look like?

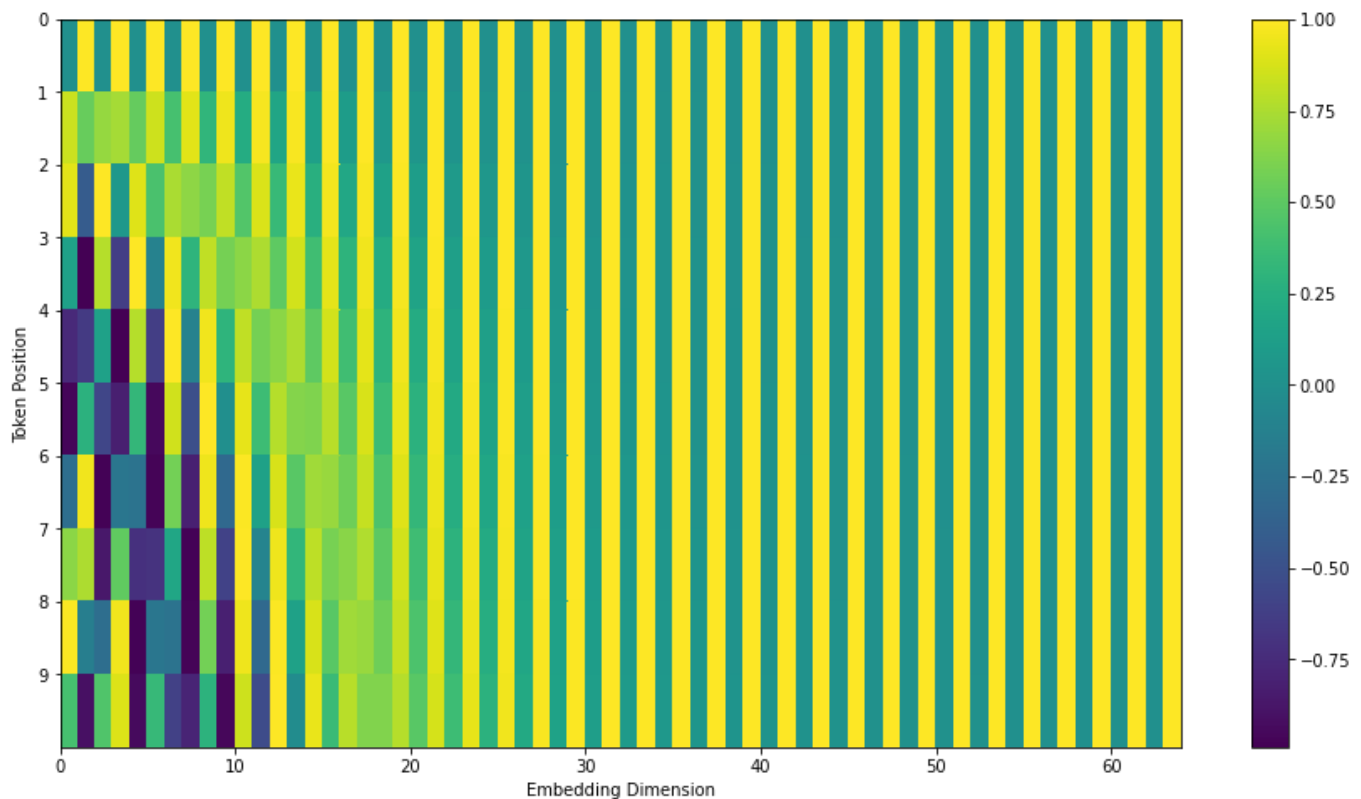
In the following figure, each line corresponds to a Positional Encoding of a vector. So the first line would be the vector that we would add to the *Embedding* of the first word of the input sentence. Each line contains 512 values – each with a value



A real example of Positional Encoding for 20 words (lines) with an *Embedding* of size 512 (columns). You can see that it appears to be split in the middle. This is because the values on the left are generated by one function (which uses Sine), and on the right half are generated by another function (which uses Cosine). They are concatenated to form each of the Positional Encoding vectors.

The formula for Positional Encoding is described in the article (section 3.5). We can see the code to generate them in the `get_timing_signal_1d()`. This is not the only method for Positional Encoding. However, it gives the advantage of being able to scale and generalize to previously unseen sequence lengths (for example if our trained model is asked to translate a sentence longer than any other in the training database).

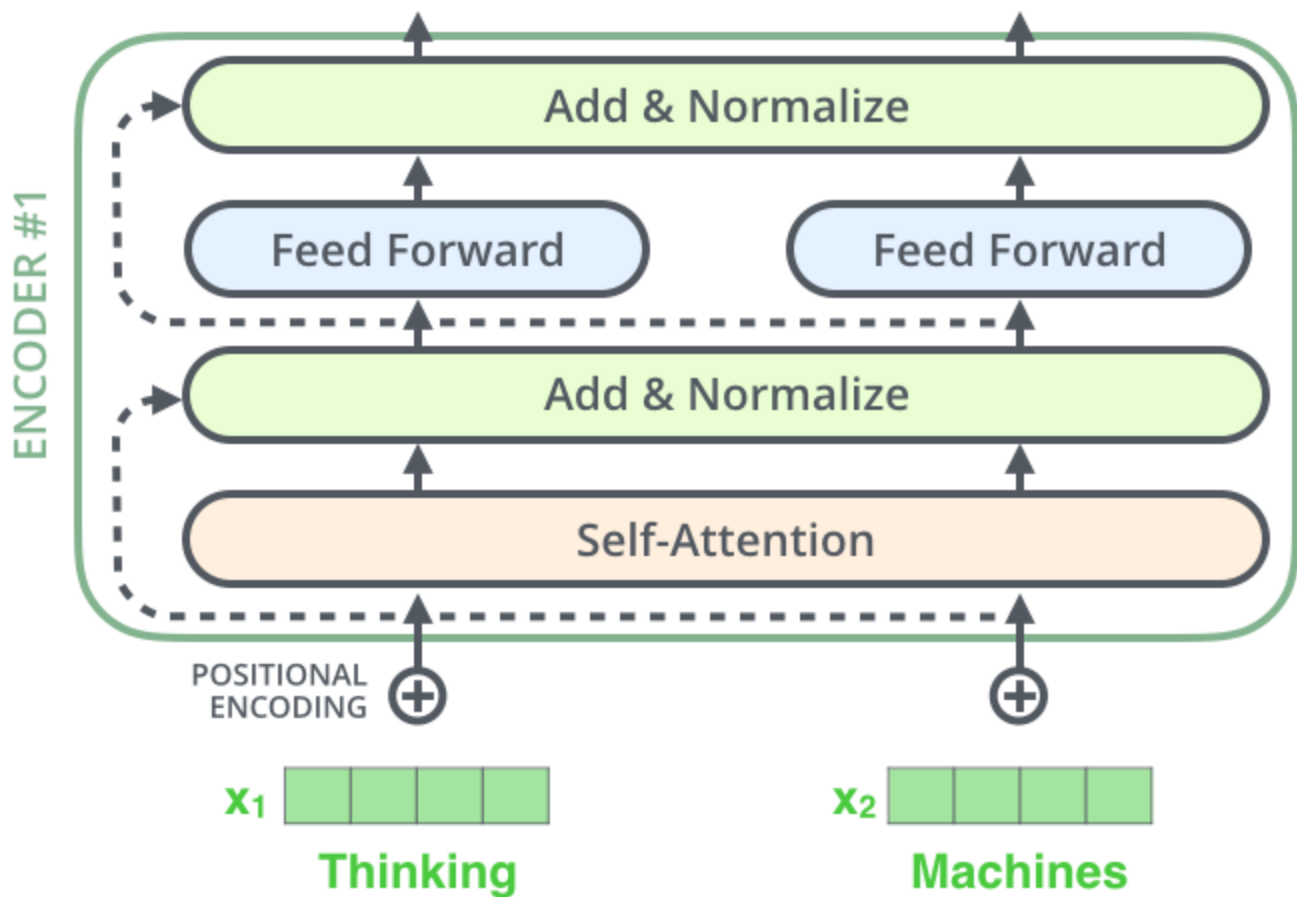
July 2020 Update: The Positional Encoding shown above is from the Transformers implementation of Tensor2Tensor. The method demonstrated in the article is a little different and does not concatenate, but interleave the two



Visualization of the Positional Encodings of the original article.

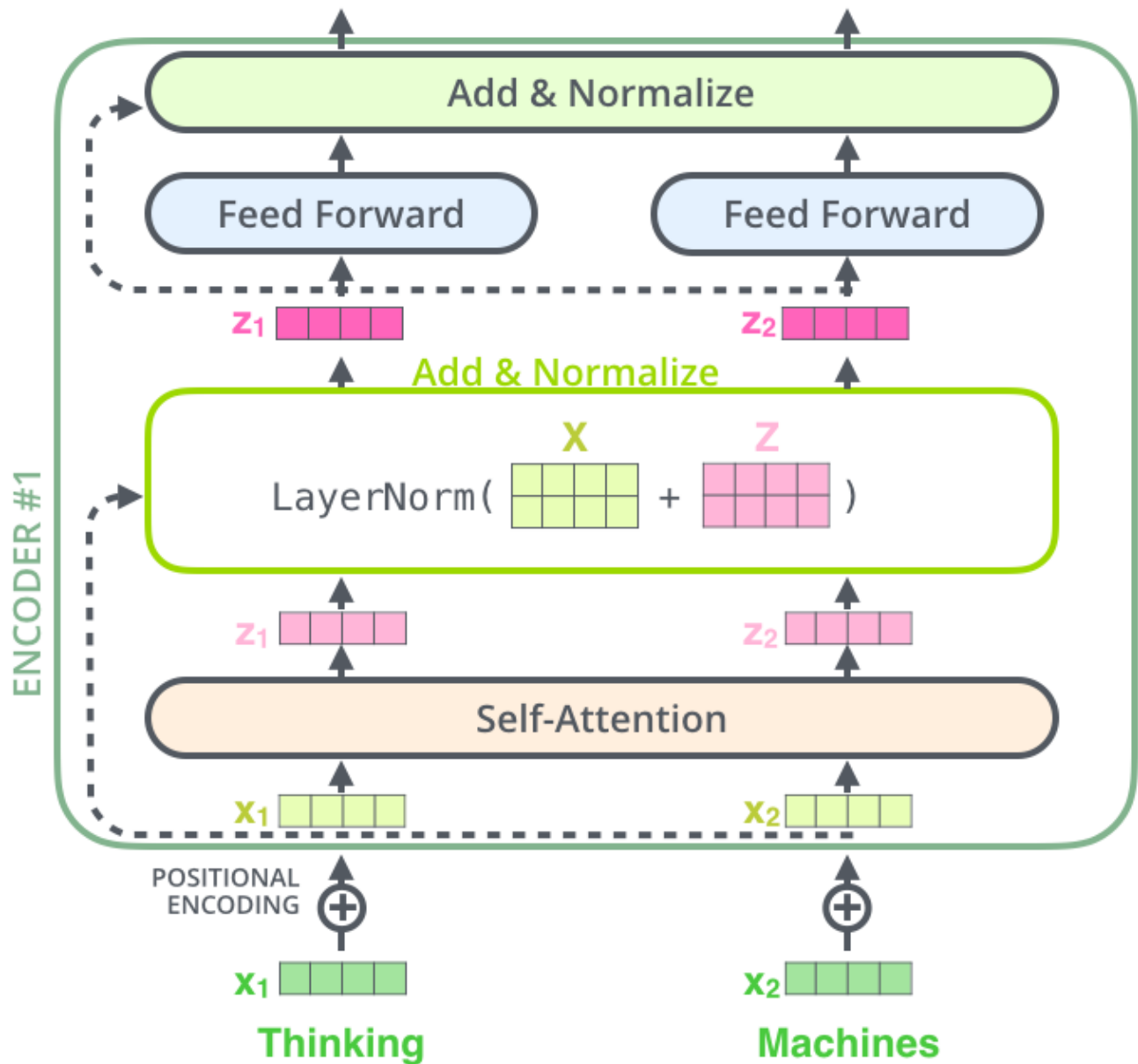
The Residuals

One detail in the Encoder architecture that we have to mention before moving on, is that each sublayer (*Self-Attention* , *Feed-Forward*) in each Encoder has a residual connection around it, and is followed by a [normalization step of the layer](#) .



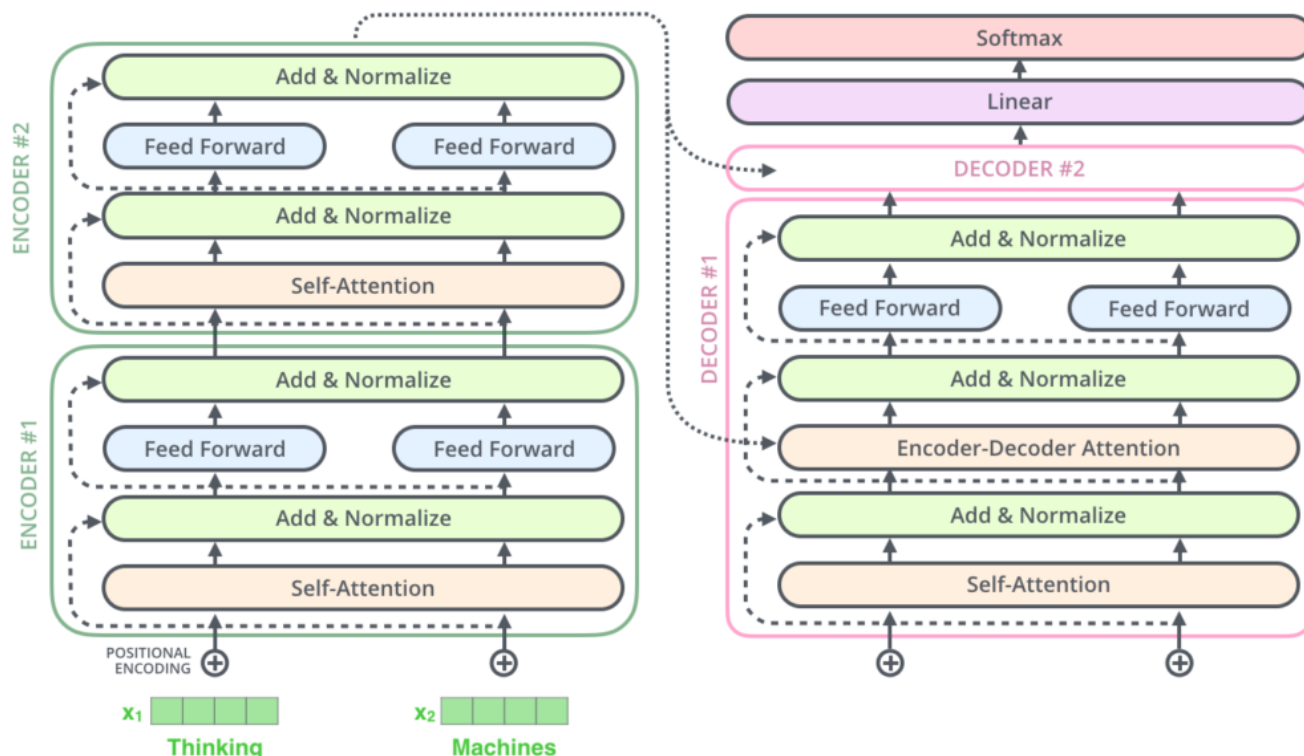
Details of the Encoder sublayers in the Transformers architecture.

If we were to visualize the vectors and layer normalization operation associated with *Self-Attention*, it would look like this:



Details of the Encoder sublayers, with vector representation.

And this applies to the Decoder sublayers as well. If we think of a Transformer with 2 stacked Encoders and Decoders, it would be something like this:

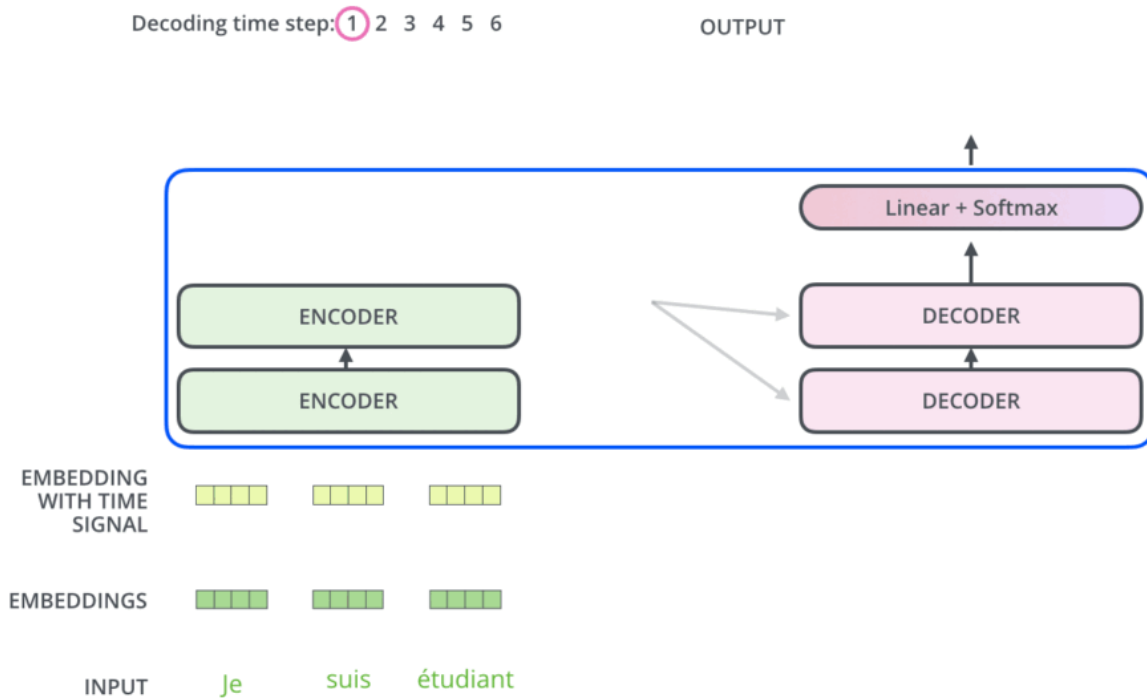


Transformers architecture with two Encoders and two Decoders.

The Decoder side

Now that we've covered most of the concepts on the Encoder side, we basically know how the Decoders components work as well. But now let's see how they work together.

The Encoder starts by processing the input sequence. The output of the upper Encoder is then transformed into a set of Attention vectors \mathbf{K} and \mathbf{V} . These will be used by each Decoder in the "Encoder-Decoder Attention" layer, which helps the Decoder to focus on the appropriate places in the input sequence.

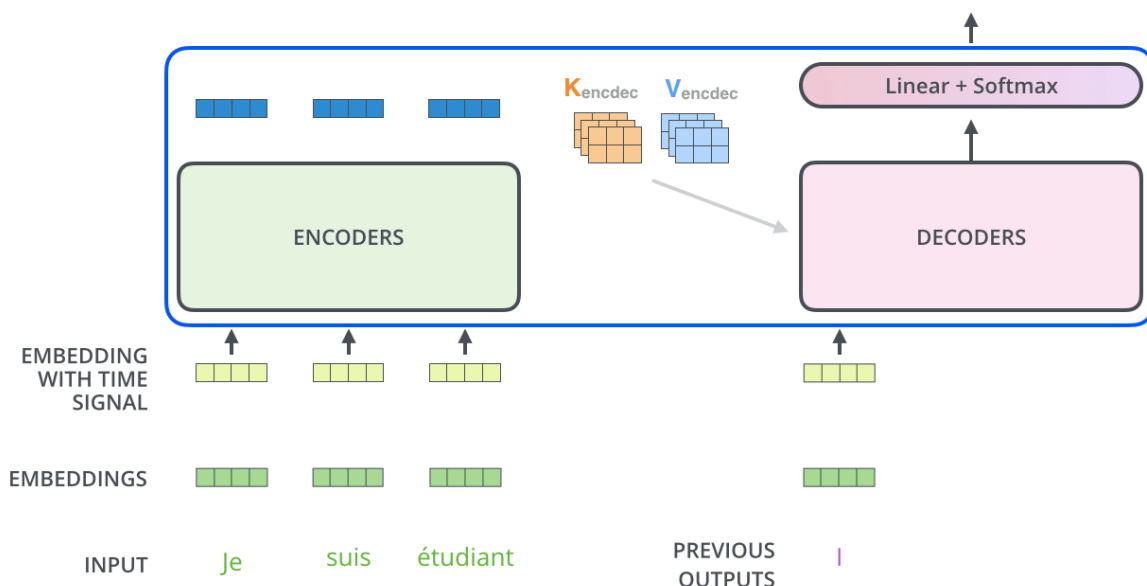


After the Encoding phase ends, we begin the Decoding phase. Each step of the Decoding phase generates an element of the output sequence (in this case, each word translated into English).

The next steps repeat the process until a special symbol is reached, indicating that the Transformer Decoder has finished its output. The output of each step is fed to the lower Decoder in the next step, and the Decoders propagate their decoding results just as the Encoders did. And just as we did with the Encoder inputs, we generate the *Embeddings* and add the Positional Encodings to the inputs of these Decoders to indicate the position of each word.

Decoding time step: 1 2 3 4 5 6

OUTPUT |



Complete Transformers translation process, which upon completion generates an **<end of sentence>** token .

The *Self-Attention* layers in the Decoder operate slightly differently than those in the Encoder.

In Decoder, the *Self-Attention* layer can only pay attention to previous positions in the output sequence. This is done by masking future positions (setting them to $-\infty$) before the Softmax step in the *Self-Attention* calculation .

The "Encoder-Decoder Attention" layer works exactly like the *Multi-Head Self-Attention* layer, except that it creates its **Query** matrix from the layer below it, and takes the **Key** and **Value** matrices from the output of the Encoder stack.

The final Linear and Softmax layer

The Decoders stack generates a vector of decimal numbers (*floats*). How to turn this into a word? This is the work of the last Linear layer, which is followed by a Softmax layer

The *Linear layer* is a simple *Fully Connected* neural network that projects the vector produced by the Decoder stack into a much, much larger vector, called the **Logits** vector.

Let's assume that our model knows 10,000 different English words (our model's "output vocabulary") that were learned from the training dataset. This would make our *Logits* vector 10,000 cells wide – each cell corresponding to the score of a single word. This is how we interpret the model output after the Linear layer.

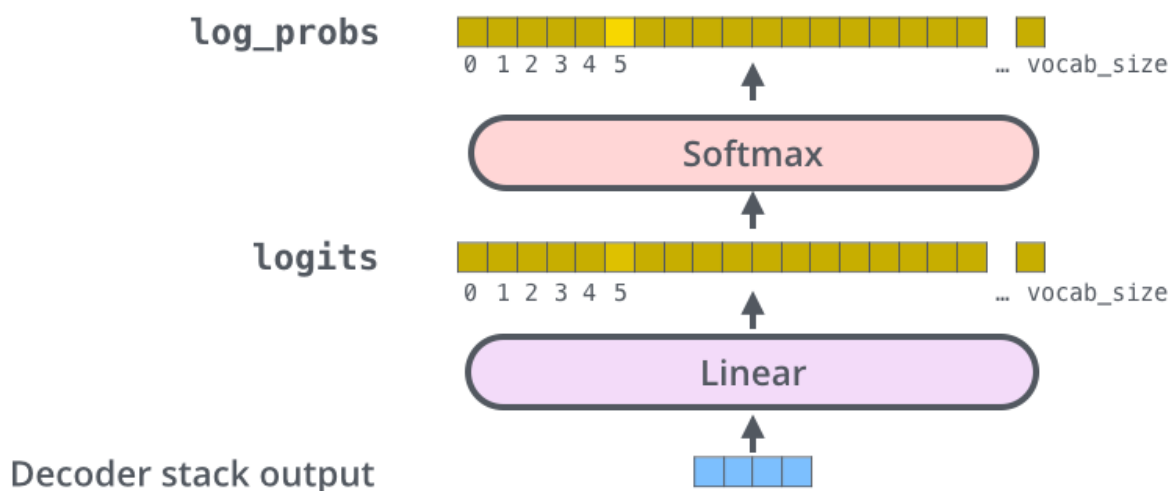
The Softmax layer then transforms these scores into probability scores (all positive, adding up to a total of 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as output for this step.

Which word in our vocabulary
is associated with this index?

am

5

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



This image starts from the bottom, with the vector produced with output from the Decoder stack. It is then transformed into an output word.

Now that we've covered the entire process of *Forward-Pass* , or “forward passing,” through a trained Transformer, it would be helpful to take a look at the idea behind model training.

During training, an untrained model would go through the exact same *Forward-Pass* process . However, since we are training the model with a set of labeled training data, we can compare its output with the actual correct output.

To visualize this, let's assume that our output vocabulary contains just six words: “a” , “am” , “i” , “thanks” , “student” , and “ <eos>” - an abbreviation for <end of sentence>, or “sentence end”.

Output Vocabulary

WORD	a	am	i	thanks	student	<eos>
INDEX	0	1	2	3	4	5

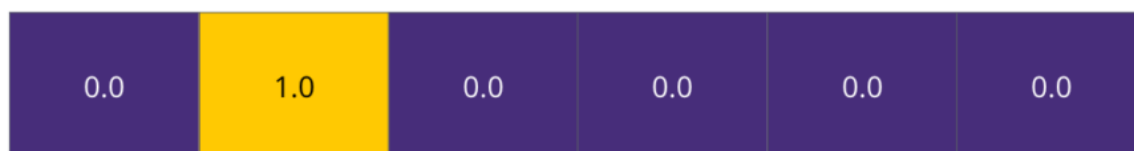
Our model's output vocabulary is created in the pre-processing phase, before we even start training the model.

Once we define our output vocabulary, we can use a vector with the same width to indicate each word in our vocabulary. This is also known as *One-Hot Encoding* . So for example, we can indicate the word “am” using the following vector.

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word "am"



Example of One-Hot Encoding of a word from the output vocabulary. The word "am" is represented by 1.0 and all others by 0.0 in this vector.

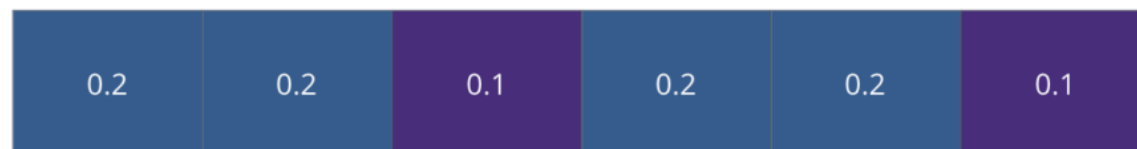
Following this summary, let's discuss our model's Loss Function, also called **Loss Function**. This is the metric we are optimizing during the training phase to arrive at a trained and, hopefully, incredibly accurate model.

The Loss Function

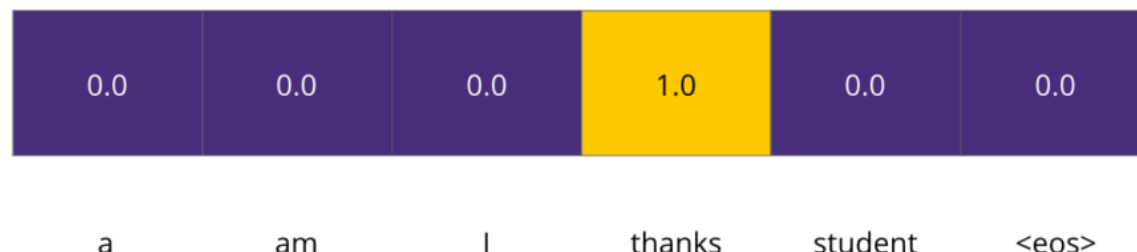
Let's say we are training our model. Let's also say this is our first step in the training phase, and we are training our model on a simple example: translating "*merci*" into "*thanks*".

What this means is that we want the output to indicate that the word "*thanks*" is the most likely. But given that the model hasn't been trained yet, that's unlikely to happen for now.

Untrained Model Output



Correct and desired output



Given that the model's parameters (weights) are randomly initialized, the untrained model produces a probability distribution with arbitrary values for each cell/word. We can compare with the real output, and then adjust the model weights using *Backpropagation* to make the model output close to the real output.

How do we compare two probability distributions? We simply subtract one from the other. For more details, read about [Cross-Entropy](#) and [Kullback-Leibler Divergence](#).

However, note that this is an oversimplified example. More realistically, we will have sentences longer than a single word. For example – input: “*je suis étudiant*” and expected output: “*i am a student*”. What this actually means is that we want our model to successively generate outputs from probability distributions where:

- Each probability distribution is represented by a vector of size `vocab_size`, the size of our vocabulary (6 in our simplified example, but more realistically, a number like 30,000 or 50,000).
- The first probability distribution has the highest probability in the cell associated with the word “*i*”.
- The second probability distribution has the highest probability in the cell

- And so on, until the fifth output distribution identifies the symbol <end of sentence>, which also has a cell associated with it.

Target Model Outputs

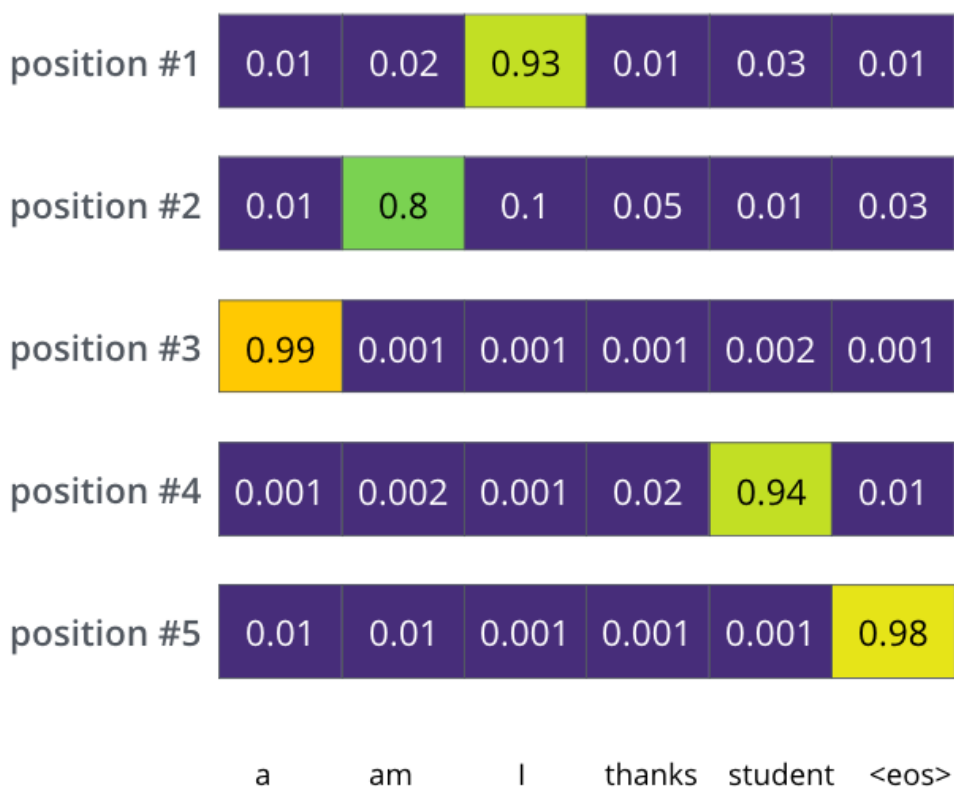


The target probability distributions for training our Transformers model for training with a sentence sample.

After training the Transformers model long enough on a large enough training base, we expect the probability distributions to look something like this:

Trained Model Outputs

Output Vocabulary: a am I thanks student <eos>



We hope that, after training, the Transformers model generates the correct translation we expect. Of course, this is no real indication of whether this sentence was part of the training dataset (see: [Cross Validation](#)). Note that each position is assigned a bit of probability. Even if it is unlikely to be the way out of this stage. This is a very useful property of Softmax that helps in the training process.

Agora, porque o modelo de Transformers produz as saídas uma de cada vez, podemos supor que o modelo está selecionando a palavra com maior probabilidade dessa distribuição. E descartando as demais. Essa é uma maneira de fazer isso, chamada de **Greedy Decoding**, ou “decodificação gananciosa”.

Outra maneira de fazer isso seria manter, por exemplo, as duas palavras com maior probabilidade (digamos que “I” e “a” por exemplo). Então, no próximo passo, executar o modelo duas vezes. Uma assumindo que a primeira palavra de saída era “I”. E outra vez assumindo que era a palavra “a”. E a versão que

Esse método é chamado de **Beam Search**, ou “busca em feixe”. No nosso exemplo, o `beam_size` é 2. Isso significa que o tempo todo, duas hipóteses parciais (traduções não concluídas) são mantidas em memória. E as principais opções, os `top_beams`, também são 2. Significando que retornaremos duas traduções. Ambos são hiperparâmetros com os quais podemos experimentar.

Fonte

Esta foi uma tradução. O conteúdo original está disponível em [The Illustrated Transformer](https://alammar.github.io/). Este e outros ótimos posts podem ser encontrados no blog do Jay Alammar (<https://jalammar.github.io/>).

Com a autorização do autor original, trouxemos este valioso conteúdo para o BRAINS, em Português. 🇧🇷

Conclusão

A arquitetura de **Transformers**, e todos os mecanismos que a possibilitam, estão em grande discussão no momento. É muito importante para todos que buscam compreender a forma como encaramos a Inteligência Artificial atualmente, entender estes conceitos. Touxemos um dos posts mais famosos do mundo sobre o tema para ajudar nossos colegas brasileiros que não falam Inglês.

Se você quiser também colaborar trazendo conteúdo sobre ML, IA e Dados para o Brasil, em Português, conheça nossa comunidade. E para entender melhor nosso propósito e saber como colaborar, leia nosso post de introdução do **[BRAINS – Brazilian AI Networks](#)**.

Caso tenha ficado com alguma dúvida, entre em contato com a gente. Se conhecer outros conteúdos bons como este, nos indique para tradução. Estamos

Contamos com vocês para crescer nossa comunidade. Até porque...

#NoBrains #NoGains

 COMPARTILHE

 TWEET



André Lopes

Fundador do BRAINS. Engenheiro de Inteligência Artificial na IBM.



VER COMENTÁRIOS (1) ▾

< — ARTIGO ANTERIOR

**Visualizando um Modelo
de Tradução (Seq2seq
com Attention)**

PRÓXIMO ARTIGO — >

**Introdução aos LLMs e à
IA Generativa**

VOCÊ TAMBÉM PODE GOSTAR

BRAINS 



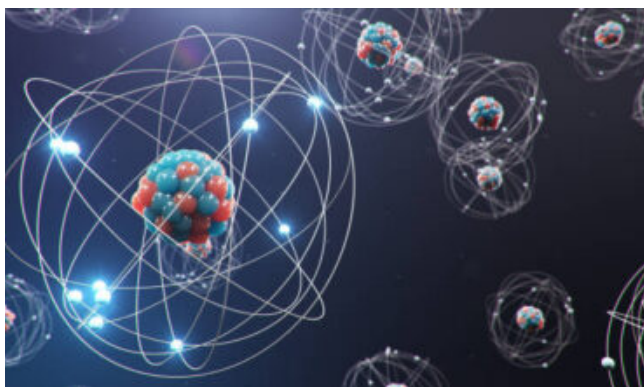


F — FOUNDATION MODELS & GEN AI

Introduction to LLMs and Generative AI

by **André Lopes** · November 22, 2023

Discover the language models (LLMs) that revolutionized how we deal with Generative AI, their main capabilities and how to extract maximum value.



F — FOUNDATION MODELS & GEN AI

Foundation Models: Models that Revolutionized AI

by **André Lopes** · June 17, 2023

Do you know Foundation Models? Learn more about the AI models that are revolutionizing the way we look at business and the world.



ABOUT BRAINS

Brazilian AI Networks

BRAINS (Brazilian AI Networks) is a community of AI students and enthusiasts that aims to bring quality content to Brazilians, in Portuguese. 🇧🇷

BRAINS 



SUPPORTERS



SEARCH

To look for

TO LOOK FOR

CATEGORIES

DATA ANALYSIS	(two)
DATABASES	(3)
BRAINS	(1)
CLOUD COMPUTING	(1)
DEEP LEARNING	(two)
DATA ENGINEERING	(7)
SOFTWARE ENGINEERING	(3)
FOUNDATION MODELS & GEN AI	(4)
JUPYTER NOTEBOOKS	(two)
MACHINE LEARNING	(10)
PYTHON	(4)
SQL	(1)

F — FOUNDATION MODELS &
GEN AI

Introduction to LLMs and Generative AI

by André Lopes



D — DEEP LEARNING

Visualizing a Translation Model (Seq2seq with Attention)

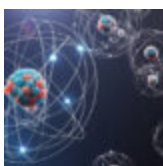
by André Lopes



F — FOUNDATION MODELS &
GEN AI

Foundation Models: Models that Revolutionized AI

by André Lopes



Subscribe to our Newsletter

Receive news about our posts, bootcamps and events.

<input type="text" value="Enter your name"/>	<input type="text" value="Enter your email"/>	<input type="button" value="SUBSCRIBE"/>
--	---	--

☐ By checking this box, you confirm that you have read and agree to our terms of use regarding the storage of data submitted via this form.

BRAINS - Brazilian AI Networks



