Generative AI for business stakeholders and non-technical roles

Abdelwahid Benslimane
wahid.benslimane@gmail.com

28th September 2023

Preamble

Generative AI models, and in particular those that do **NLP** (natural language processing), generally have an **Encoder-Decoder** architecture. This architecture has been around since 2014. Even the **Transformer** model, which will be explained later in this document, has inherited this architecture from a high-level perspective, but with significant differences on a closer look.

A description of the Encoder-Decoder model as it existed before the Transformers' era will be given as a start. This will make it easier to then introduce the Transformer model.

A lot of AI-related concepts, such as recurrent neural network (RNN), or word embedding among other things will be introduced, before gradually leading the reader towards an understanding of the Transformer and the **Self-Attention** mechanism which characterizes it, and which represents a significant advance in the field of generative AI.

This document is primarily aimed at professionals who do not have a scientific education in the field of AI and who do not necessarily come from a technical background, and at anyone wishing to learn more about this subject.

The aim is to facilitate a conceptual understanding of generative AI and also to provide a solid foundation of knowledge.

Index

I) Encoder-decoder before the Transformer's era

I.1) Introduction p. 3
I.2) Let's dive a little deeper p. 4
I.3) Encoder p. 4
I.4) Decoder p. 5
I.5) RNN + Attention p. 6
I.6) Remaining challenges p. 7

II) Transformer

II.1) Introduction p. 8
II.2) A few generalities p. 9

II.3) Encoder block
II.3.1) Introduction p. 9
II.3.2) Input p. 9
II.3.3) Self-Attention (Multi-Head Attention) p. 10
II.3.4) Self-Attention algorithm p. 10
II.3.5) Multi-Head Attention p. 12
II.3.6) Add & Norm + Feed Forward layers p. 13
II.3.7) Output p. 13

II.4) Decoder block
II.4.1) Introduction p. 13
II.4.2) Input p. 14
II.4.3) Masked Multi-Head Attention p. 14
II.4.4) Encoder-Decoder Multi-Head Attention p. 15
II.4.5) Output p. 15

II.5) Transformers: advantages, drawbacks and scope p. 15

II) Encoder-decoder before the Transformer’s era

I.1) Introduction

The Encoder takes a sequence of words, processes it, and regurgitates it in the form of a mathematical object called a **vector** that the Decoder will be able to use to translate, or answer a question etc. in the case of **NLP**.

This vector is the representation of a sequence of words, in fact its semantics, in a certain reference frame called **latent space**.

It is important to remember that AI models perform mathematical calculations and that all the information they process must be represented mathematically. This is also the case for the sequence of words submitted to the input of an Encoder. Words are also represented in vector form.

Images and sounds and any kind of information that the body receives are converted into electrochemical signals that the brain then manipulates. Similarly, AI models require information to be converted into numbers arranged in vectors.

The process of going from a word to its vector representation is called **word embedding**. There are various word embedding techniques.

Once the words are in vector form, we can perform operations involving these words/vectors. For example, we can calculate the semantic proximity between 2 words using simple mathematical calculations.

It's important not to confuse word embedding, which consists in representing words (or sometimes groups of words that form a semantic unit) in vector form, with the Encoder's work, which produces a vector containing richer information about the word sequence and that can be interpreted by the Decoder.

The whole process can be modelled as follows in the case of a translation task:

“to be or not to be” -> **word embedding** -> **Encoder** -> vector in a latent space-> **Decoder** -> “être ou ne pas être”

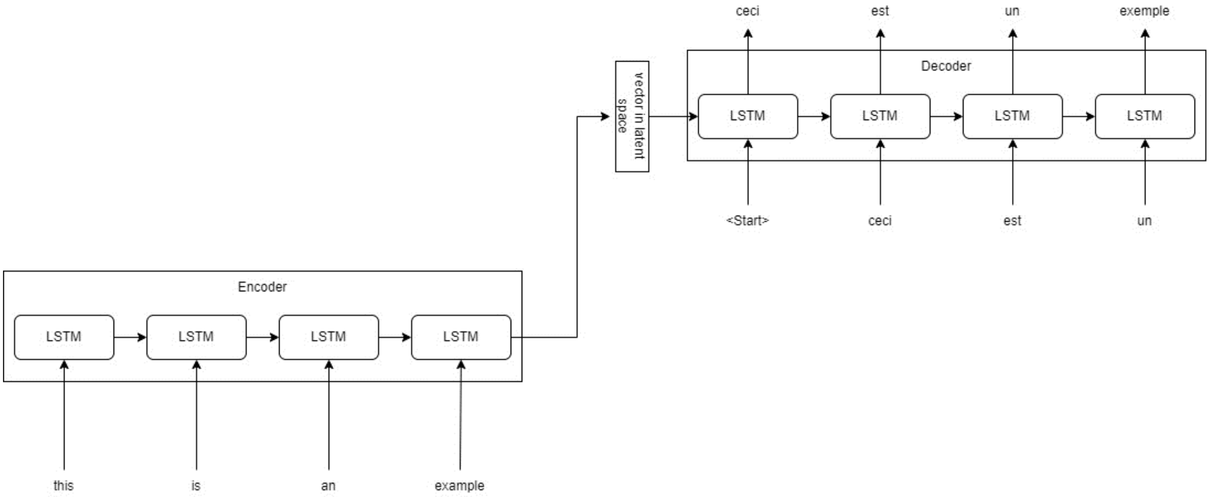
The main challenge in translation, for example, is to translate the overall meaning of a sentence, rather than doing a word-by-word translation. To do this, the model must be able to capture the meaning of the input word sequence in the encoding part.

For example, the word-by-word translation of “c’est tiré par les cheveux” would be “it’s pulled by the hair” but if we translate the meaning then it becomes “it’s far-fetched” and this is what NLP models are able to do automatically.

I.2) Let’s dive a little deeper

Let's take a deeper look at the architecture of an Encoder-Decoder model.

The diagram below is a general view of the model. All the elements which compose it will become meaningful throughout the explanations which I provide hereafter.



Both the Encoder and the Decoder are mainly composed of a recurrent neural network (**RNN**), the most advanced form of which is made up of **LSTM** (long short time memory) or **GRU** (gated recurrent unit) cells, which are nothing more than computational units. It's not necessary to know the details of the internal architecture of these cells, or even the calculations they perform, if you're not implementing a model.

We speak of RNN because the cells are organized in sequence and each cell uses the result of the preceding cell to perform its calculation and transmits its own result to the following cell.

I.3) Encoder

In the Encoder, the RNN will take as input a sequence of words (in vector form) of fixed and predefined length (number of words). The length of the sequence corresponds to the length of the RNN (number of cells).

The length of the RNN in the Encoder can be different from the length of the RNN in the Decoder.

We're talking about words here for simplicity's sake, but the correct term would be token. Tokenization is the process that splits a text into several smaller units. A token can be a word, a group of words forming a semantic unit, a subword, or even a character sometimes. In the following, the term word will continue to be used.

The first cell transmits the result of its calculation, known as **hidden state**, to the second, which uses it to perform its own calculation, and transmits its hidden state to the next cell, and so on.

The calculation at each cells involves the hidden state, which is a vector, transmitted by the previous cell and the vectorized word corresponding to the time step/stage in the input sequence that the cell represents.

The last cell then transmits its hidden state to the first cell of the RNN at the Decoder, which initializes its calculation with it. This last hidden state is in fact the latent space vector which contains a summary of the information that the RNN was able to extract from the word sequence.

The RNN can be more or less complex. For example, it can be bidirectional in the Encoder, or contain several layers. It is not necessary to go any further and learn about the different possible architectures of an RNN to understand the principle of its functioning and its implication in the Encoder-Decoder model.

I.4) Decoder

In the Decoder, each cell takes as input the hidden state produced and transmitted by the previous cell (as a reminder, the first cell receives the hidden state from the last cell of the RNN at the Encoder), as well as the word of the translation produced by the previous cell. The fist cell always uses a <Start> token instead, since there was no translation before it. Here it is translation that is being discussed, but the model can also perform other tasks.

And naturally, each cell produces as output a prediction of the next word and a hidden state. Both will be injected into the next cell.

The goal of the Decoder is to obtain the most likely output sequence overall. Translated into mathematical terms, the model will seek to maximize this value P(Y|X) (the probability of obtaining the translation Y knowing the sequence in input X) with:

X = x_1, x_2,…, x_n (a sequence of input words)

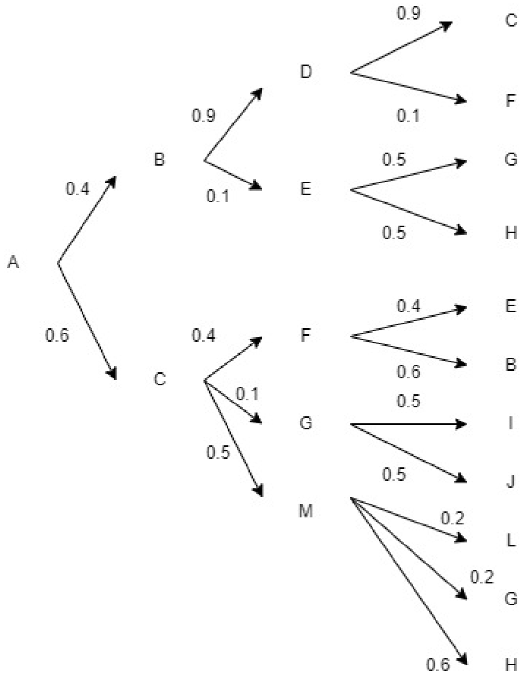
Y = y_1, y_2,…, y_n (a sequence of output words)

The translation y_n will depend on x_1, x_2,…, x_n but also on the choice made for y_1, y_2,…, y_{n-1}, this is why we speak of **autoregressive** model, because the past influences the present.

The algorithm used to find the most likely output words can for example be **Beam Search** or **Greedy Search**.

Greedy Search will select the most likely word at each step while Beam Search will keep a track of a fixed (and small) number of the most likely ones at each step and will generate the most likely sequence among all the possibilities.

Greedy Search is faster, but Beam Search produces a more accurate result, therefore using one or the other method depends on the use case and constraints.



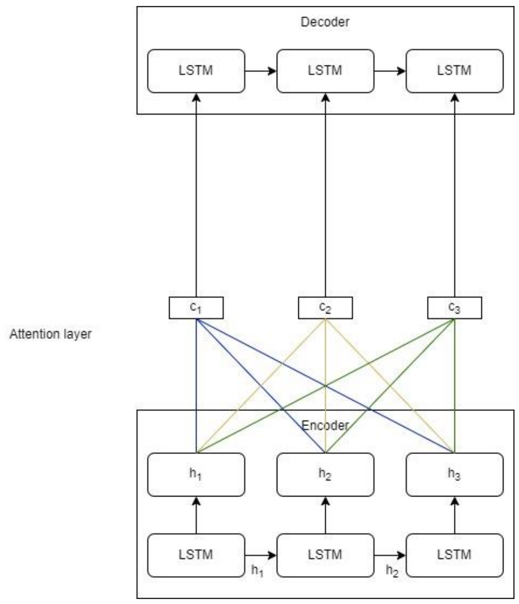
Beam search output: ABDC

Greedy search output: ACMH

I.5) <u>RNN + Attention</u>

Experience has shown that the quality of the output does deteriorate as the length of the input sequence increases because its semantic content is more difficult to encapsulate in the hidden state produced by the last LSTM cell of the RNN at the Encoder, and this is where the Attention mechanism comes into play.

The idea behind the Attention mechanism is that instead of providing a unique vector (the hidden state produced by the last cell of the RNN at the Encoder) to the Decoder, we provide a dynamic vector which changes according to the outputting stage, and which is a weighted sum of the hidden states of all the LSTM cells of the Encoder.



- h_1, h_2, h_3 are the hidden states generated by the first, second and third cells at the Encoder respectively
- c_1, c_2, c_3 are 3 vectors which are in fact 3 different weighted sums of the hidden states h_1, h_2, h_3 and which are transmitted respectively to the first, second and third cell of the RNN at the Decoder

The cells at the Decoder continue to use for their calculation the word generated in the previous step. From this standpoint, nothing changes

I.6) Remaining challenges

Although the Attention mechanism improves the performance of the Encoder-Decoder model, it is still just another brick on top of the RNN layer.

The RNN gives rise to several challenges. Firstly, because of its design and the sequential nature of the calculations (each cell needs an output of the preceding cell to perform its calculation), the training cannot be parallelized.

Another problem is that the longer the sequence of input words, the more the contribution of the first words in the sequence tends to decrease, and therefore the less the model's output is compelling.

These challenges are overcome by the **Transformer** model, which no longer uses RNN but the **Self-Attention** mechanism instead.

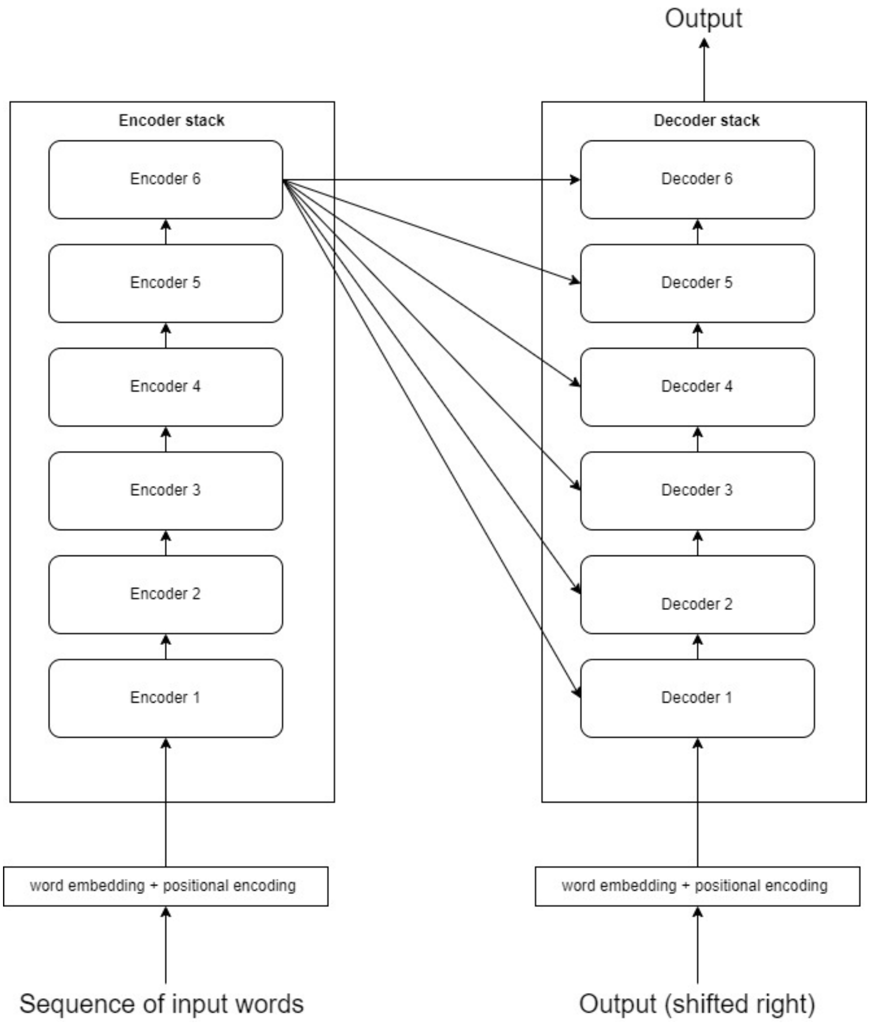
II) Transformer

II.1) Introduction

In this section, I'm going to explain the Transformer model, which is a very popular model developed by researchers at Google and the University of Toronto and having been presented in 2017 in a now famous paper entitled "Attention is all you need". For example, behind the GPT models and ChatGPT in particular, there is a Transformer.

The diagram below shows the architecture of a Transformer from a high-level perspective. Further details will be provided on the composition of each block and how it fits within the model, but always in a very conceptual manner.

For the sake of simplicity, **certain stages will not be shown nor explained**, the idea being to give a broad, albeit fairly elaborate, idea of how a Transformer works.



II.2) A few generalities

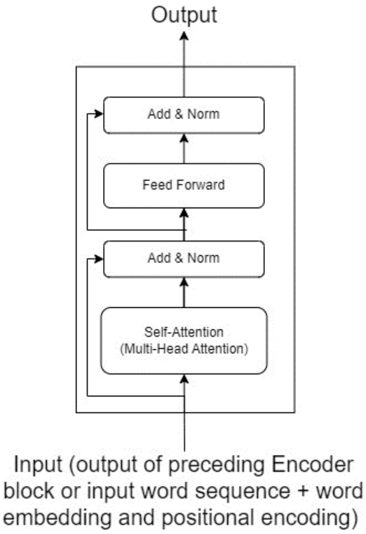
The Encoder, and respectively the Decoder, are now divided into a multitude of Encoder layers, and respectively of Decoder layers. In the original model, the Encoder contains a stack of 6 Encoder layers, and there is the same number of Decoder layers in the Decoder stack.

As Transformers do not use RNNs, they are not capable of identifying the order of the words they are submitted with. for this reason, in addition to traditional word embedding, another step has been added to encode in its mathematical representation the position of a word in the sequence to which it belongs. This is referred to as **positional encoding**.

II.3) Encoder block

II.3.1) Introduction

The diagram below shows the anatomy of an Encoder block (layer). The explanations will focus mainly on the Self-Attention layer.


Output
Add & Norm
Feed Forward
Add & Norm
Self-Attention
(Multi-Head Attention)
Input (output of preceding Encoder block or input word sequence + word embedding and positional encoding)

II.3.2) Input

An Encoder block receives its input from the previous layer, always in the form of a mathematical object called a tensor. However, the first Encoder block receives a text as input after it has passed through the word embedding and positional encoding stages, which also produce a tensor.

It is not necessary to know in detail what a tensor is, just remember that it is a mathematical object in the form of which information is transmitted between the different blocks. Throughout this document you can just think of a sequence of vectors instead.

II.3.3) Self-Attention (Multi-Head Attention)

The really original part of the architecture of an Encoder block is the Self-Attention layer, or more precisely the **Multi-Head Attention** layer, but I'll come back to this nuance later.

Before I explain the Self-Attention mechanism, I'd like to raise a problem that's important to resolve when using NLP: removing ambiguities.

For example, the term “apple” taken out of context could just as easily refer to the fruit as to the brand of electronic products. Another example is the sentence “Karim asked Michael to tell his colleague that they would come back late”. Who should have been informed, Michael’s colleague or Karim’s colleague?

Grammar alone cannot always resolve ambiguities, and for a human being it is the context in which a term or phrase is used that will indicate the meaning it takes on. Contextualisation is therefore an important notion in human language. It is equally important in NLP.

Intuitively, Self-Attention is in fact a way of finding some kind of correlations between the words that make up a sequence, or in some way, of automatically finding the best approximation of the relationship that links them all together. It is a manner of formalising the notion of context so that it can be learned by a model.

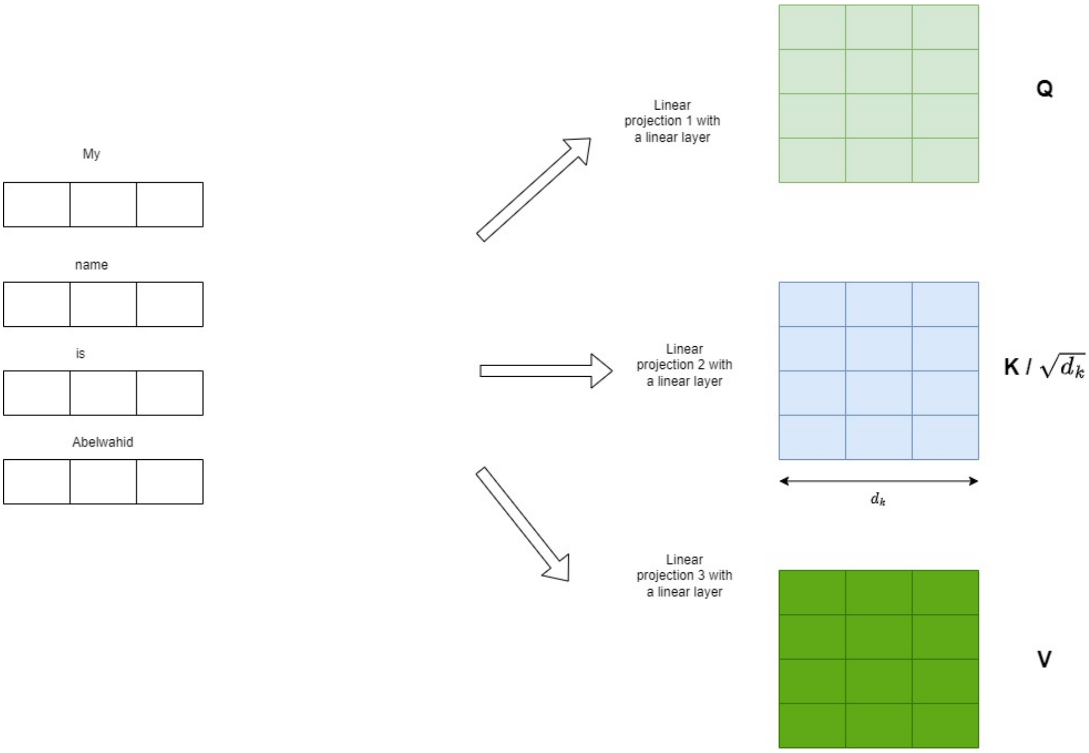
For example, in the sentence “the cat has blue eyes”, the word “blue” relates more to the word “eyes” than to the word “cat” for example, and this is what a model will try to learn thanks to the Self-Attention mechanism.

II.3.4) Self-Attention algorithm

The first part consists of linearly projecting the representations of each word into 3 new distinct spaces using 3 distinct **linear layers**.

What you need to know about a linear layer is that it is simply characterised by weights. These weights are learned when the model is trained.

These projections produce the matrices K, Q and V which are represented in the graph below.

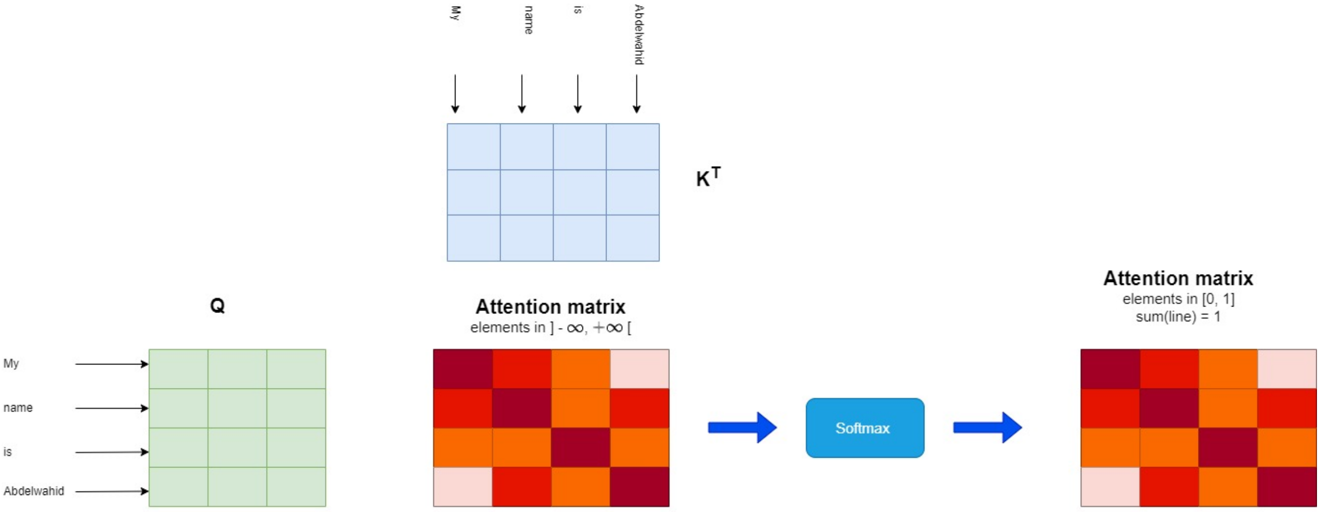

My
name
is
Abelwahid
Linear projection 1 with a linear layer
Linear projection 2 with a linear layer
Linear projection 3 with a linear layer
Q
K / $\sqrt{d_k}$
d_k
V

In order to obtain better results in the next stages of the process, the matrix K is divided by a certain coefficient which is in fact the square root of the size of a row vector of the matrix. However, this operation is not essential to perform Self-Attention.

The second step is to do the dot product of the matrix Q and the transpose of the matrix K. Each element of the matrix thus produced, which is called **Attention matrix**, is in fact the scalar product of a row vector of Q and a row vector of K, and the values lie between $-\infty$ and $+\infty$.

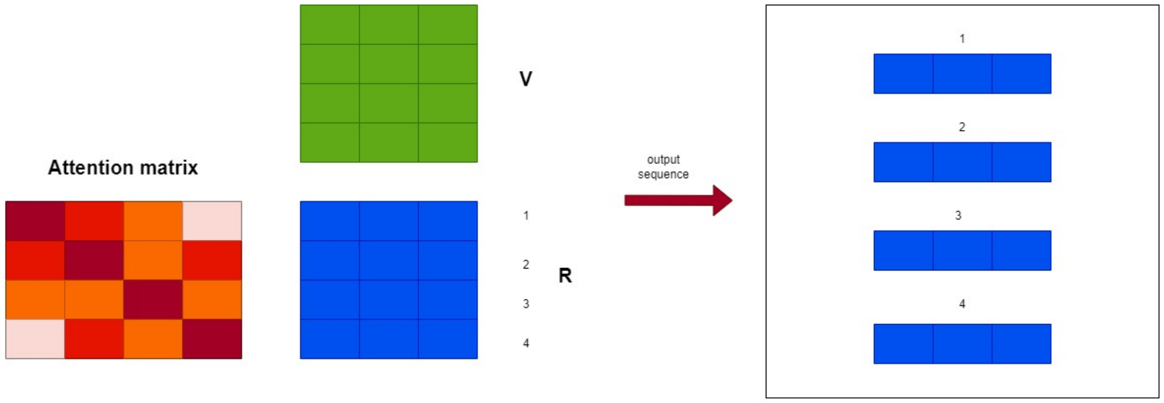
We then apply the softmax function so that the elements of the Attention matrix lie between 0 and 1 and the sum of the values in the same row is equal to 1 (we thus obtain probabilities).

This diagram was inspired by the one used in this video: https://www.youtube.com/watch?v=L3DGgzIbKz4.



It must be noted that the Attention matrix is not symmetrical.

Finally, the dot product of the Attention matrix and the matrix V is performed to obtain the matrix R. The rows of the matrix R are the vectors of the output sequence of the Self-Attention layer.



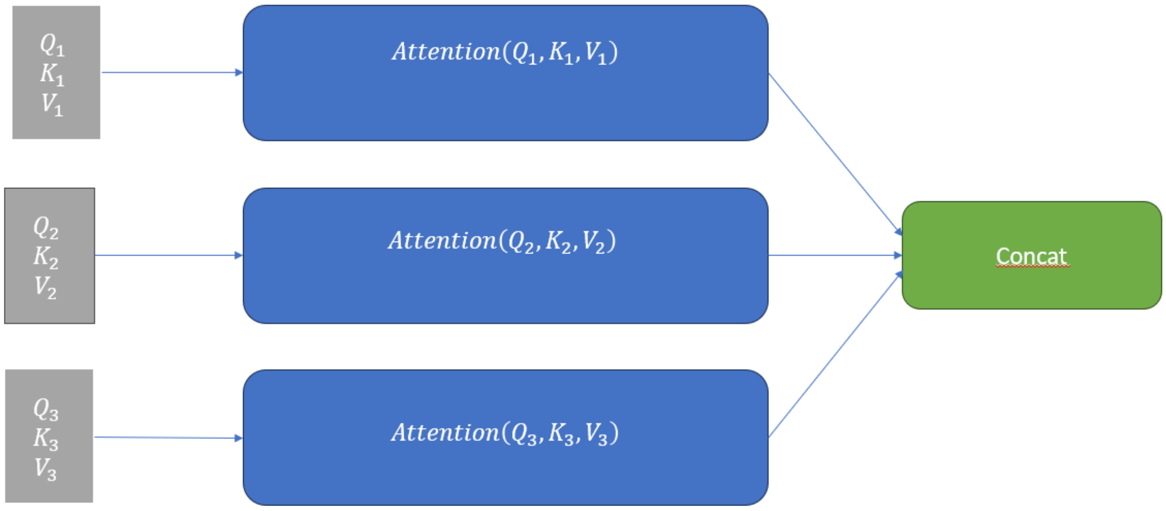
The Self-Attention mechanism can be summarised by the following equation:

$$Attention(Q,K,V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Of course, after the data has been through the first Self-Attention layer of the Encoder stack, a vector does not correspond anymore to a single word and the information of all the sequence has been dispatched among the vectors.

II.3.5) Multi-Head Attention

We can describe Multi-Head Attention simply by saying that instead of using a single Self-Attention mechanism, we will use it multiple times in parallel, with different projections to obtain several K matrices, several Q matrices and several V matrices, and club the results.


Q_1 K_1 V_1
Q_2 K_2 V_2
Q_3 K_3 V_3
$Attention(Q_1, K_1, V_1)$
$Attention(Q_2, K_2, V_2)$
$Attention(Q_3, K_3, V_3)$
Concat

The general idea is to allow the model to learn different characteristics from the data it is presented with, a bit like using different cameras with different points of view to film a scene.

II.3.6) Add & Norm + Feed Forward layers

I won't go into detail about the mechanisms involved in these layers, although they're not complex, because the part that is essential to understand when it comes to Transformers is the Self-Attention mechanism. You can simply think of the Add & Norm and Feed Forward layers as a way of improving the model.

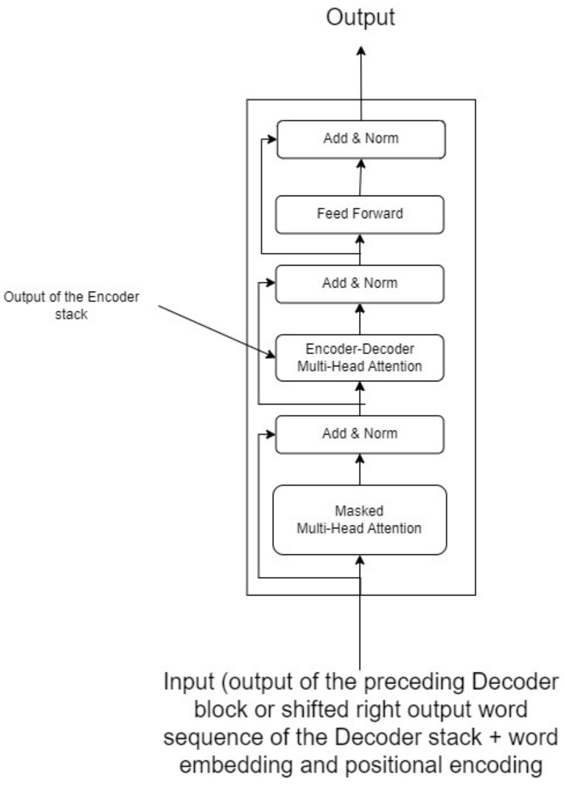
II.3.7) Output

Each Encoder block transmits its output to the next block, which performs equivalent processing. The output of the last Encoder block in the Encoder stack is transmitted to each of the Decoder blocks in the Decoder stack.

II.4) Decoder block

II.4.1) Introduction

Below is the simplified diagram of a Decoder block. Main parts (Masked Multi-Head Attention and Encoder-Decoder Multi-Head Attention) will be described.


Output
Add & Norm
Feed Forward
Add & Norm
Output of the Encoder stack
Encoder-Decoder Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Input (output of the preceding Decoder block or shifted right output word sequence of the Decoder stack + word embedding and positional encoding

II.4.2) Input

The input of the first layer (**Masked Multi-Head Attention**) of each Decoder block is the output of the preceding Decoder block, except for the first Decoder block which takes as input the sequence of words generated by the previous passes in the Decoder stack (this is roughly what shifted right output means) after it has been through the word embedding and positional encoding stages.

Just as the Encoder-Decoder model involving RNNs, the Transformer is autoregressive.

It should also be noted that the **Encoder-Decoder Multi-Head Attention** layer takes as input not only the output of the previous layer but also the output of the Encoder stack. The process used in this layer will be explained later in this document.

II.4.3) Masked Multi-Head Attention

In this layer the Multi-Head Attention explained previously is performed, with the difference that it is the relationship between each word and the words that precede it only that is evaluated, and not the relationship of all the words to each other.

For example, in the sentence "Attention is all you need", for the word "all" the Multi-Head Attention will be calculated only with the words “Attention” and “is”.

To clarify, still on the conceptual level, the Multi-Head Attention/Attention between word A and word B is different from the attention between word B and word A, so the Attention between the word "all" and the words that come after it would have been different from that which will be calculated in the opposite direction.

II.4.4) Encoder-Decoder Multi-Head Attention

This layer applies the Multi-Head Attention between the output vector sequence of the Masked Multi-Head layer (after it has passed through the Add & Norm layer "on top" of it) and the output vector sequence of the Encoder stack.

Matrixes K and V are created from the output of the encoder stack and matrixes Q are created from the output of the Masked Multi-Head Attention layer (after it has passed through the Add & Norm layer).

This layer allows the decoder to concentrate on the most important parts of the input word sequence of the model.

II.4.5) Output

Each Decoder block transmits its output to the next block, which performs equivalent processing.

During a pass, once the data has traversed all the Decoder stack, the model, after a few additional operations, will generate a new word which will be added to the input sequence for the next pass in the stack, and so on until the model has completed the generation process.

II.5) Transformers: advantages, drawbacks and scope

First, transformers are highly parallelizable, which enables GPUs computing power to be leveraged.

Transformers also make all the words in a text interact with all the others, through Self-Attention mechanism. This makes it possible to model long-term dependencies, i.e. between very distant words, whereas when RNNs are used, this is not possible.

Transformers are very large architectures (sometimes several hundred million parameters), which means that a gigantic amount of data needs to be collected in order to train them.

However, in most cases, pre-trained Transformers are used. These are trained using data available on the Internet. This makes it possible to start with models that have already acquired certain fundamentals, such as the structure of a language, for example. It is then sufficient to fine-tune a pre-trained model in order to train it on a particular task in a specific context with a smaller amount of data. This will reduce the computation costs.

Transformers have become very popular, and their use is not limited to NLP. The world of computer vision and even scientific research have also adopted them. For example, the AlphaFold 2 model released by DeepMind uses a Transformer to predict the 3D structure of proteins. To learn more about it, you can for example consult this page: https://daleonai.com/how-alphafold-works.