Transformers – an in-depth tutorial
Part I: Transformer's architecture

Laura-Maria Cornei

Feb 2023

Table of Contents

1. Introduction

1.1. Motivation

Transformers were introduced [1] as neural network models that can solve transduction and sequence modelling tasks (such as machine translation, language modelling, etc.); they achieved a better performance and were more parallelizable than existent state-of-the-art models (recurrent neural networks and convolutional networks).

Recurrent neural networks process the input sequentially, so parallelization is not possible. Moreover, even improved versions, such as LSTM and GRU, struggle with modelling long term dependencies between elements from the input. Furthermore, there is no explicit modelling of the learned dependencies.

Convolutional neural networks model the local context and many layers are required to model the long-term dependencies.

Nowadays, transformers are used in various NLP tasks (text generation, text summarization, question answering, entailment, machine translation, etc.) and are also becoming more popular in fields such as computer vision [2].

2. Architecture

2.1. General overview

The classic architecture of a transformer consists of *an encoder*, represented by a stack of N encoder blocks/layers, and *a decoder*, represented by a stack of N decoder blocks/layers.

First, we will make a short remark regarding the data preprocessing step, that will be useful when explaining the encoder-decoder interaction later. As it is highlighted in Figure 1, the data received by the encoder, as well as the data given to the decoder through source (**) are preprocessed in the following way: each input is replaced by some associated embeddings (word vector representations) and then a positional encoding is inserted in the word vector representations to form the final embeddings. The preprocessing steps will be explained later in the next sections. For now, we will refer to this type of preprocessing by saying that some data is P.E.E. (positional encodings in the embeddings) processed.

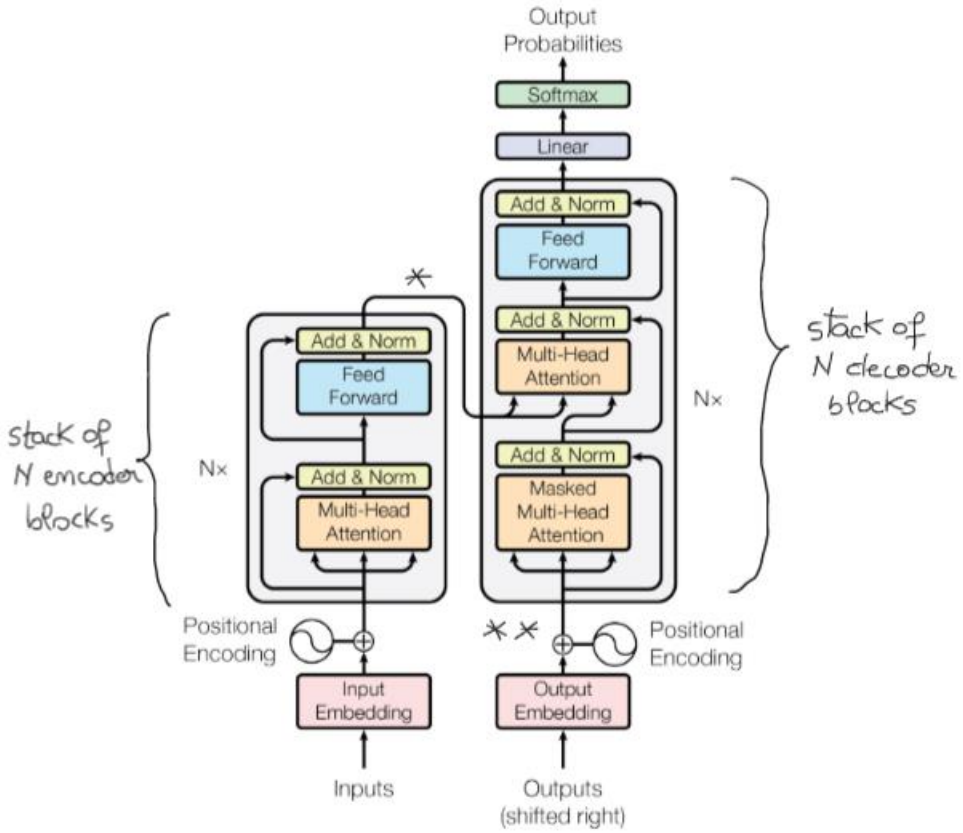

Figure 1. Transformer's architecture

Before diving into more detailed explanations, it is worth mentioning the purpose of the encoder and the decoder. To illustrate this better with an example, we will assume the transformer solves the Romanian to English machine translation task. The whole procedure is illustrated in Figures 2 and 3.

The encoder takes as input a P.E.E. processed sentence (/sequence of words) in Romanian (that needs to be translated in English) and outputs a contextualized representation of this sentence. This contextualized representation is able to capture the syntactic and semantic dependencies between the words from the given sentence. The decoding phase starts after the encoding phase finishes.

The decoder receives this contextualized representation from the encoder as input (input marked with '*' in Figures 1 and 2) and also receives a sequence of P.E.E. processed words as input (input marked with '**' in Figures 1 and 2). This sequence of words ('**') initially contains only one processed start symbol <SOS>, corresponding to the start of the sentence to be generated as a translation. At each time step, the decoder receives as input a sequence of processed words and outputs one word in English; next, this word in English is P.E.E. processed and appended to the sequence of other processed words to form the input of the decoder for the next time step. The decoder stops outputting words when it outputs an <EOS> (End Of Sentence) tag. To sum up, the decoder uses the contextualized representation of the sentence in Romanian and the representation of the translated sentence in English generated so far to predict the next word that should be added to the English translation.

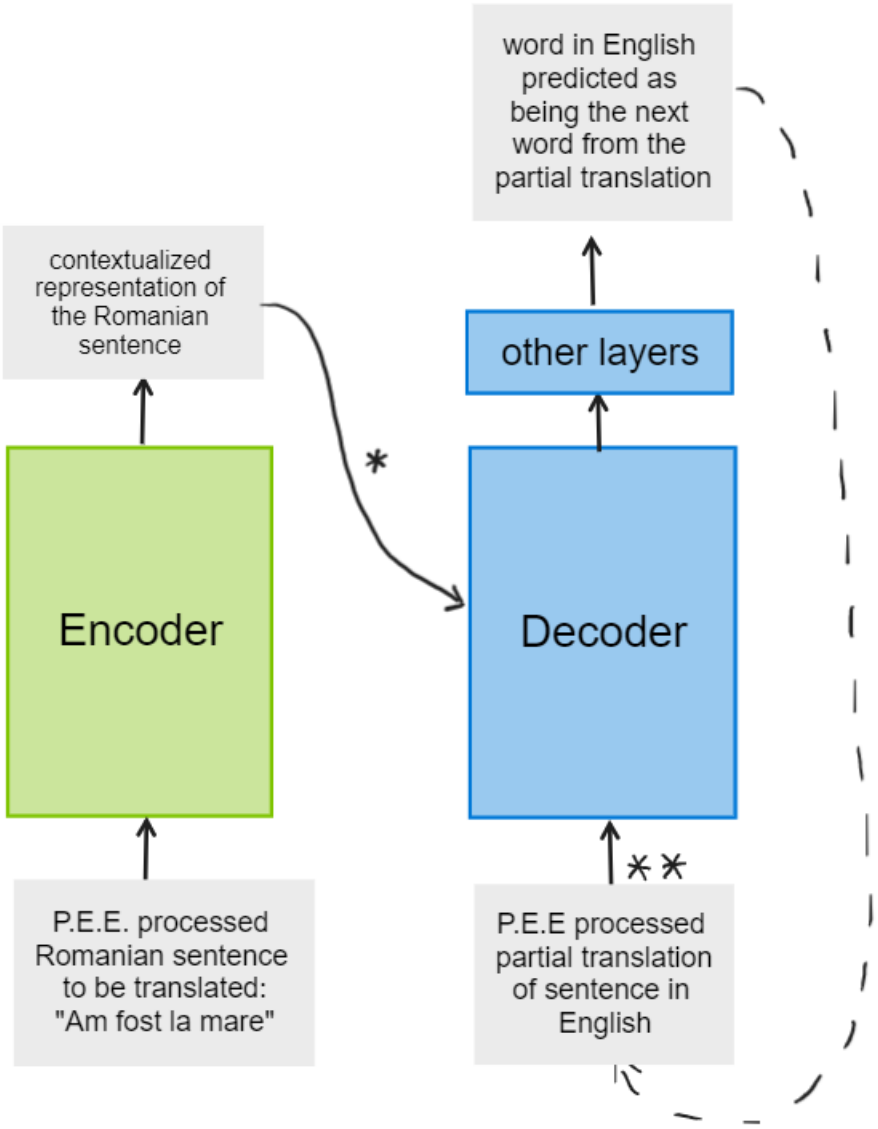

Figure 2. Transformer applied to machine translation task - high level overview

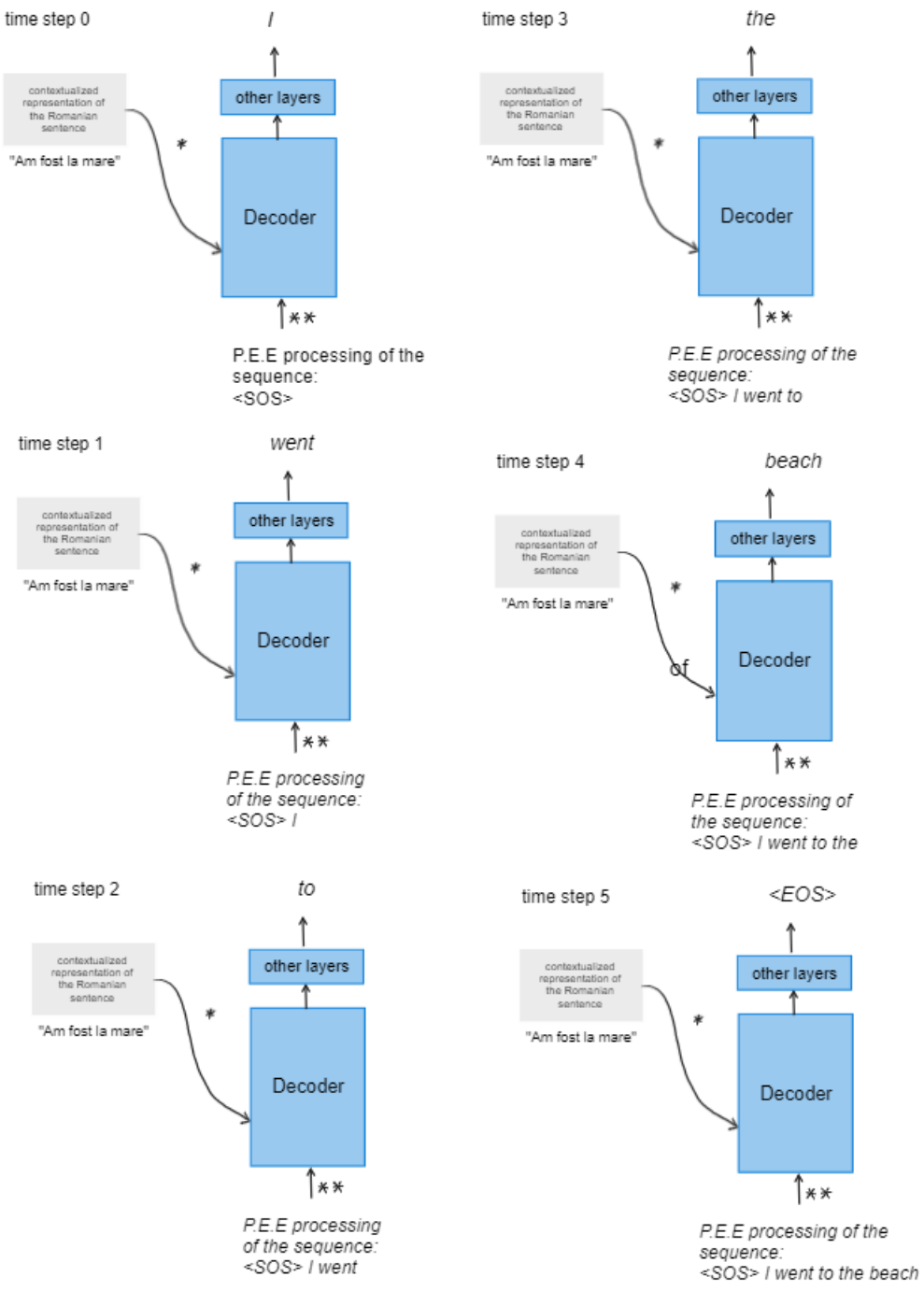

Figure 3. The process of decoding detailed for the machine learning task - high level overview

Next, we will look in more detail at how the transformer processes the inputs and generates the outputs, considering the interactions between the encoder and the decoder layers. These interaction details are also illustrated in Figure 4.

As it was mentioned before, the encoder receives multiple P.E.E. processed inputs at once. In case of our task, the inputs are represented by the words corresponding to the Romanian sentence that we aim to translate in English. Let's assume the input size (in our case, the number of words from the input) is **n**. A P.E.E. processed input is represented by an embedding that incorporates information about that input and its position in the sequence.

First, the processed inputs are passed through the stack of encoder layers as follows: each encoder block receives **n** inputs and outputs **n** elements which represent the contextual representations (/contextual embeddings) of the given inputs; the elements outputted by the i_{th} encoder block are given as inputs to the $i + 1_{th}$ encoder block. The purpose of an encoder layer is to extract a new representation of its given inputs, such that any new representation of an input combines the previous input representation with contextual information relevant to that input. The first encoder layers model less complex aspects of the language (for example trivial dependencies between words), while the final encoder layers model more sophisticated aspects of the language (for example semantics).

The decoding phase starts only when the encoding phase is over. The outputs from the final encoder block (which represent the final contextual representations of the inputs) will be given as input to the Mult-Head Attention subcomponent of each decoder block. This type of input is marked with '*' in Figures 1, 2, 3 and 4.

The decoder also receives a sequence of P.E.E. processed elements as input. This type of input is marked with '**' in Figures 1, 2, 3 and 4. The sequence of elements initially contains only one element: the embedding of a special tag, <SOS> (Start of Sentence). At each time step, a sequence of elements is passed through the stack of decoder blocks and next through a linear layer and a Softmax layer, resulting in the decoder generating an element (in our case, a word in English). To summarize, at one time step, the decoder receives as input a sequence of elements and outputs one element.

Next, this outputted element is P.E.E. processed and will be appended to the input (**) given to the decoder in the following time step. The decoder is autoregressive, meaning that it uses previous outputted information as input for the current step. More exactly, it appends the element generated as output at the i_{th} time step (after processing it) to the sequence of processed elements given as input for the $i + 1_{th}$ time step.

Next, we will discuss about how the information is passed through the decoder blocks and the purpose of the linear and Softmax layers that follow the stack of decoders. Passing the processed sequence of elements (**) through the stack of decoder blocks results in obtaining contextualized representations (/contextualized embeddings) of them, that also incorporate information related to the inputs from the encoder (*). The linear layer, which is a simple fully connected neural network, projects these contextualized representations into a large vector of scores with the size of the vocabulary (in our case, the size of the vocabulary is equal to the number of English words in the vocabulary). The Softmax layer transforms the scores from this vector into probabilities. Thus, the probability from the i_{th} position in the final obtained vector corresponds to the probability that the i_{th} element from the vocabulary is generated by the decoder.

If a greedy approach is used, the decoder generates as output the element (the English word) corresponding to the position in the vector which has associated the highest probability. This classical greedy strategy of choosing the generated word is a simple one, however the final outputted sequence might not have the highest overall probability, even if the individual words are chosen as having maximal probabilities. Authors of the original paper [1] use a beam search strategy to select the outputted word at each time step. This strategy takes into account multiple hypothesis (multiple partial translations that could be generated). To avoid a complexity explosion, the least likely hypothesises are pruned. Beam search does not guarantee that the decoder generates the most likely sequence, but it improves a lot the performance of the model.

Figure 4 describes the discussed process (the greedy approach for selecting the final word is used).

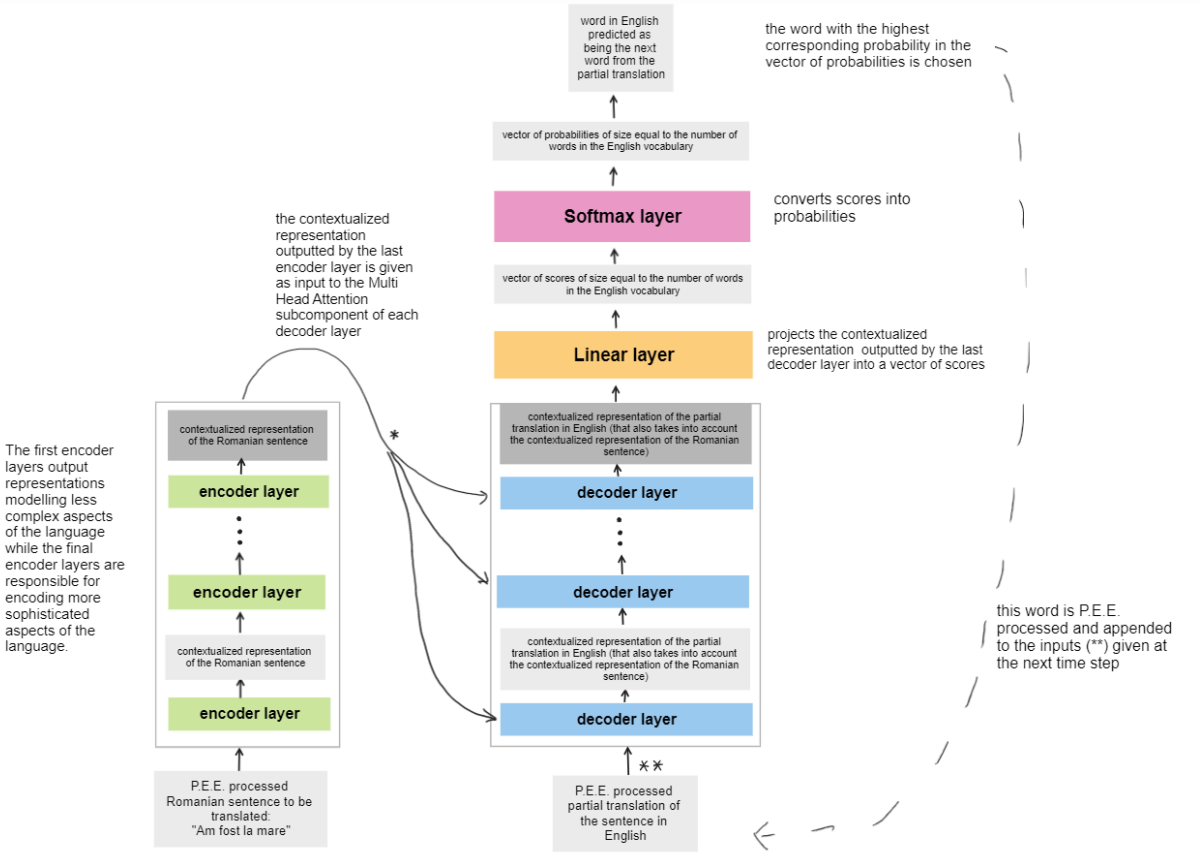

Figure 4. Transformer applied to machine translation task - lower-level overview

2.2. Input preprocessing

2.2.1. *Overview*

We previously mentioned that transformers take as input a P.E.E. processed sequence of tokens. We will highlight the steps used to transform the input served to the transformer considering the machine translation example. More details regarding the preprocessing steps can be found for example in the Hugging Face (library for working with transformers) documentation [3].

In case of the machine translation task, the input given to the transformer at one time step is a preprocessed sentence/paragraph in Romanian that needs to be translated. The sentence/paragraph is passed through the following steps to yield the final input served to the transformer:

- *Tokenization*. The paragraph is split into tokens using a tokenizer. Subsection 2.2.2. details types of used tokenizers (word-based, subword-based, character-based).

- *Padding and/or truncation*. The input length (number of tokens) that can be given at one step to the transformer is bounded to a maximum length (for e.g., 512, 1024, etc.) to avoid memory and efficiency issues. However, from a theoretical standpoint, transformers can work with unlimited sequence lengths. If a sequence of tokens is longer than the maximum admitted length it can be truncated.

  The transformer architecture can receive as input tokenized sequences of varying lengths, one sequence at each time step. However, if we want to increase the computational efficiency by processing multiple sequences at once (batch processing), sequences should be padded with a special token to reach the length of the longest sequence in the batch; this is due to implementation and optimization considerations.
- *Replacing each token with its embeddings*. Each resulted token from the sequence is replaced with a vector representing an embedding for that token. This embedding can be a static, previously learned one (for example a Word2Vec or a GloVe embedding), but more commonly the embedding is learned through the training procedure. All embedding vectors have the same size, let it be d_{model}. The purpose of this step is to convert each token into a word vector representation that captures the meaning of the token.
- *Positional encoding integrated in the embeddings*. Transformers do not inherently take into account the position of the tokens in a sequence given as input (as opposed to recurrent neural networks, which encapsulate information about tokens' order by processing one token from a sequence at a time). Thus, positional encoding is used as a mechanism to embed the tokens' order in the final input given to the transformer. Section 2.2.3 describes in more detail the positional encoding step. Simply put, each token is associated a positional encoding describing the position of the token in the sequence. The positional encoding of a token is a vector of size d_{model}. The final embedding associated to a token is the summation of two vectors: the positional encoding vector and the token's embedding; therefore, it also has size d_{model}.

In the end, after passing the input (in our case the sentence/paragraph in Romanian) through the preprocessing steps described above, we get a sequence of final embeddings. Each final embedding corresponds to a token and is calculated as the sum between the positional encoding of the token and the token's embedding. These

final embeddings will be served as input to the transformer (all at one time step). The preprocessing flow is highlighted in Figure 5.

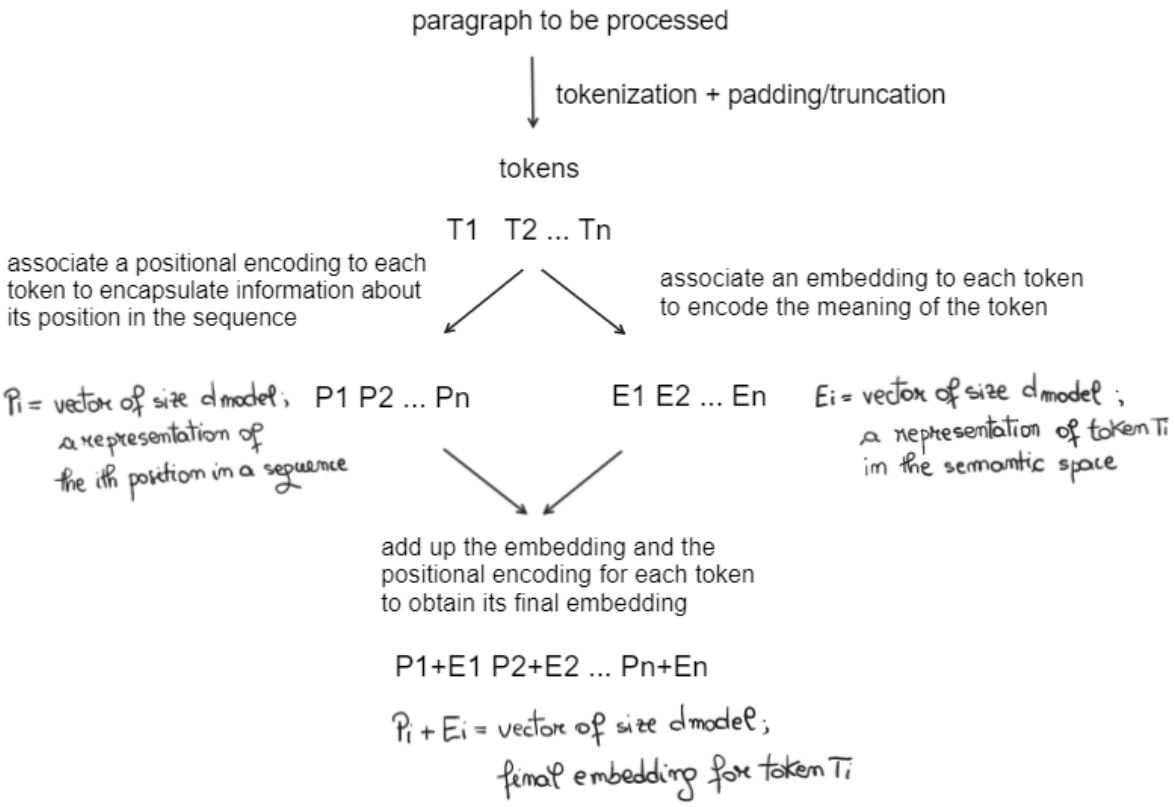

Figure 6. Summary of the preprocessing steps for an NLP task to obtain the input for a transformer

2.2.2 Tokenization

The tokenization can be word-based (split text to obtain words), character-based (split text to obtain characters) or subword-based (split text to obtain subwords). The word-based tokenization produces a very big vocabulary when used on a large text corpus; this may lead to increased time complexity and memory issues. The character-based tokenization reduces greatly the vocabulary size, but comes with a loss of performance, making it harder for the model to learn meaningful representations.

The subword-based tokenization brings the advantages of the two other types together, meaning that it produces a reasonable vocabulary size and it also enables the model to learn meaningful representations; infrequent words are decomposed

into more meaningful, frequent subwords (for e.g., “learning” -> “learn” + “ing”). Example of commonly used subword-based tokenizers are WordPiece, Unigram and Byte-Pair Encoding. It is important to notice that the results from the subword tokenization vary depending on the corpus, as they are influence by the words and subwords frequencies.

Figure 6 illustrates an example of the three types of tokenizers applied over a custom sentence: “I am learning about transformers!”.

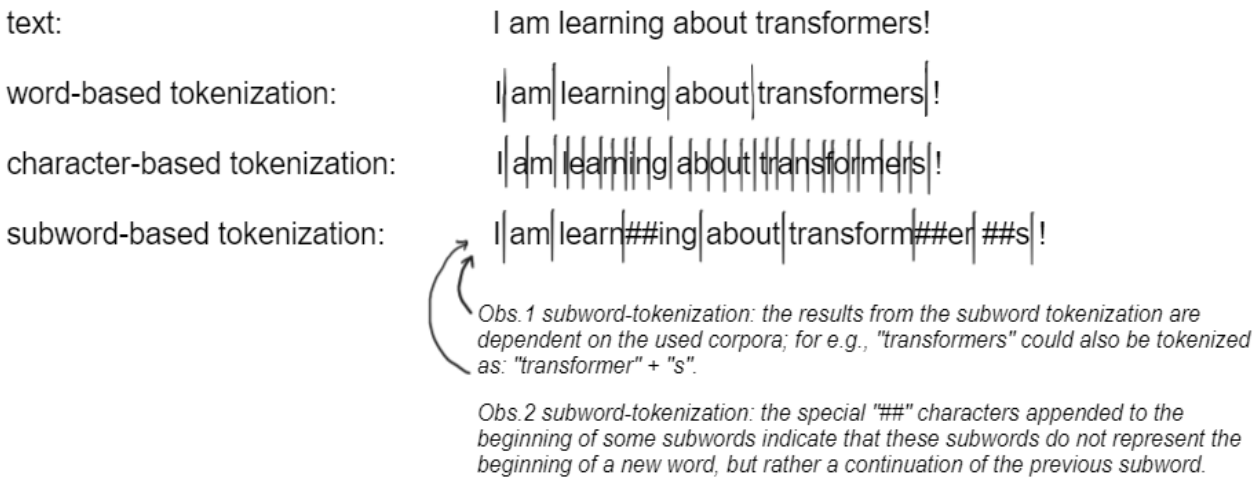

Figure 6. Example of applying various types of tokenizers as a preprocessing step

2.2.3. Positional encoding

The positional encoding models the position of each token in the sequence, such that the transformer model can have access to the tokens’ order information. The positional encoding incorporates only information regarding the position of the token in the sequence and doesn’t take into account any information regarding the token itself.

The positional encoding proposed in the original paper [1] satisfies two requirements:

1. Tokens on different positions in a sequence should have different positional encodings and tokens having the same position in different sequences should have the same positional encoding. This is a natural requirement, as a positional encoding should uniquely identify a position.

2. The positional encoding should be bounded, more exactly the value(s) representing the positional encoding shouldn't get very large (regardless of the processed sequence length!).

Next, we will justify why requirement 2 is necessary by discussing about the relationship between semantic embeddings (noted with E) and positional encodings (noted with P).

As it was previously mentioned, each token is associated with a semantic embedding (E_i) that models the position of the token in a d_{model} dimensional semantic space. Thus, tokens indicating words which are semantically related (for e.g., "sea" and "sand") appear closer to one another in this space than unrelated tokens (for e.g., "sea" and "cat").

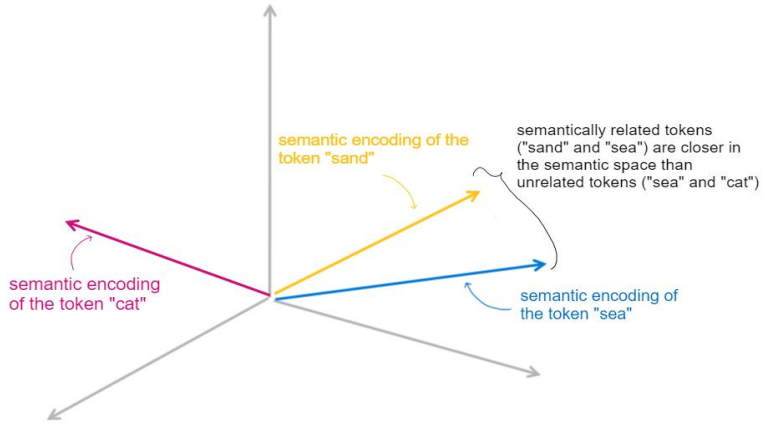

Figure 7. Example of visualization of the semantic embeddings in a 3D space

Each token is also associated with a positional encoding (P_i) that models the position of the token in a d_{model} dimensional space, such that all tokens appearing first in a sequence will have the same positional encoding, all tokens appearing second in a sequence will also have the same positional encoding, etc.

By adding up the semantic embedding of a token with its positional encoding we create a new vector in the d_{model} dimensional space, that takes into account both the meaning and the position of the token in the sequence.

If the vectors denoting the positional encodings have a very large modulus (i.e., the values in the vectors are too large), then the influence of the positional encoding vectors on the final embedding vectors will be too big, overshadowing the importance of the semantic embeddings. Thus, unrelated words might end up having

similar final embeddings, which could hurt the model's ability to understand the semantics.

To illustrate this with an example (Figure 8), suppose the token "sand" appears in two sequences processed by the transformer. In the first sequence it appears on position i and in the second sequence it appears on position j. If we note with $E1$ tokens in the first sequence and with $E2$ tokens in the second sequence, then the semantic embedding for the token "sand" is $E1_i = E2_j$ (we have equality because it is the semantic embedding for the same token, but in different sequences). The positional encoding for the token "sand" is P_i for the first sequence, as the token sand is the i_{th} token in the sequence and P_j for the second sequence. Therefore, the final embedding for the token "sand" will be $E1_i + P_i$ for the first sequence and $E2_j + P_j$ for the second sequence. Figure 8 highlights the fact that the same token ("sand") can have different final embeddings based on its position in a text.

Suppose now that the modulus of the positional encoding for the i_{th} position in a sequence is very large. In Figure 8, this new positional encoding vector is called "hypothetical P_i". We can see that the final embedding of the token "sand" in sequence 1 (called "hypothetical $\mathrm{E1_i} + P_i$") is highly skewed away from the initial semantic embedding ($E1_i$), becoming too close to the semantic embedding of an unrelated token ("cat").

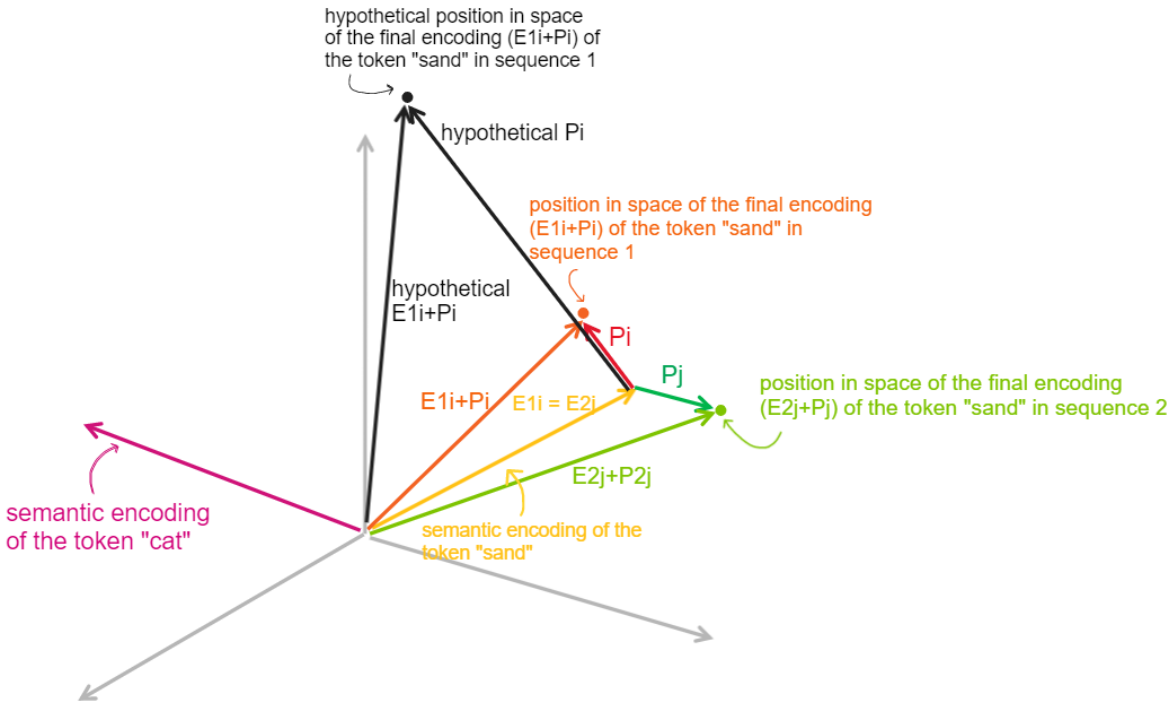

Figure 7. Example of visualization of the semantic embeddings and positional encodings (to demonstrate the need for bounded positional encodings)

To satisfy requirements 1 and 2, authors of [1], proposed a positional encoding that is represented by a vector of size d_{model} (where d_{model} is an even number). The formula used to calculate the positional encoding for the i_{th} token in a sequence is:

$$p_i = \begin{bmatrix} \sin(w_1 * i) \\ \cos(w_1 * i) \\ \sin(w_2 * i) \\ \cos(w_2 * i) \\ \vdots \\ \sin\left(w_{\frac{d_{model}}{2}} * i\right) \\ \cos\left(w_{\frac{d_{model}}{2}} * i\right) \end{bmatrix} \quad (1)$$

where

$$w_k = \frac{1}{10000^{\frac{2k}{d_{model}}}} \quad (2)$$

Working with a single bounded non-periodical function would have led to different positional encodings for the same position for different sequence lengths, thus violating requirement 1.

Working with a single bounded periodical function would have led in some cases to the same positional encoding for different positions in a sequence, again breaking requirement 1.

Therefore, the solution is to use multiple periodical functions to define the positional encodings.

The authors chose to use the sine and cosine functions as they are bounded (thus, satisfy requirement 2) and have high variability (this is used to obtain unique positional encodings for each position). To ensure that the positional encodings are unique, authors proposed the use of sine and cosine functions with different frequencies. Lower frequencies are used to describe broadly the position of the token in the sequence, while higher frequencies give more details regarding the exact position in the sequence. Author of [4] also compares these frequencies with the binary encoding, describing that high frequencies could be associated with less significant bits, while low frequencies could be associated with more significant bits.

[4] contains other interesting details regarding the properties of the positional encoding chosen by authors of [1], such as the ability to attend relative positions.

2.3. The attention mechanism

2.3.1. Intuitive explanation of the (self-)attention mechanism

One of the fist neural network models in which the attention mechanism was integrated (to improve the model and to avoid an information bottleneck) were the Sequence-to-sequence models. We will not go into details about these models, however, for those interested, [5] represents a good lecture about them.

The *attention mechanism* enables the transformer to model dependencies between the tokens in a sequence A and tokens in a sequence B; thus, it helps the transformer *to focus* on important relations, while ignoring other irrelevant things. The detected dependencies have associated attention scores that indicate the strength of the

relation/the intensity of the "focus". If sequence *A* is the same as sequence *B*, then the mechanism is called *self-attention*.

Figure 8 is an example taken from [5], which indicates the attention scores assigned using the attention mechanism in case of a French (*il m'a entarté*) to English (*he hit me with a pie*) machine translation task. Dark colours indicate higher attention scores. We can clearly see that the word "*il*" is strongly associated with "*he*", while the word "*entarté"* is strongly associated with "with a pie", fairly associated with "hit" and mildly associated with "me".

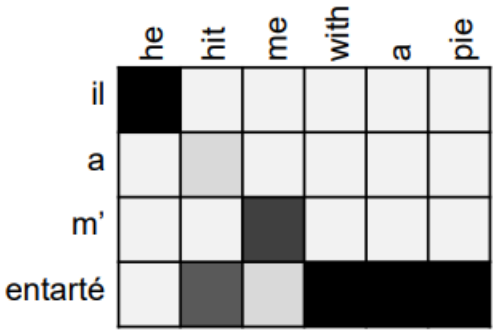

Figure 8. Example of attention matrix for the French to English MT task

Figure 9 describes an example of self-attention, used to detect dependencies in the sentence: "She unwrapped the birthday gift":

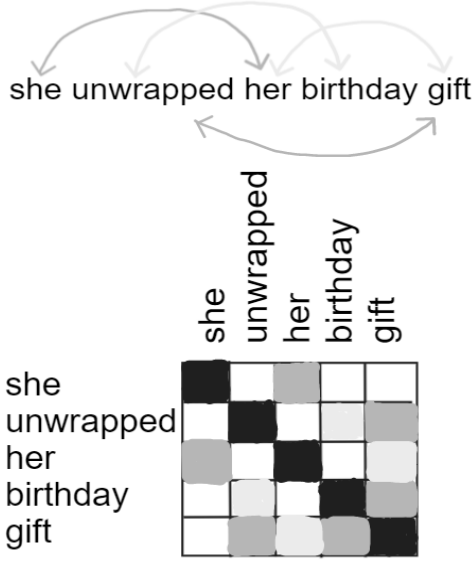

Figure 9. Example of self-attention dependencies and associated matrix

One important consideration about attention is that it can help modelling relations between tokens regardless of the distances between them. Searching for dependencies only in a well-defined neighbourhood would have brought several drawbacks:

- *Proximity doesn't guarantee syntactic or semantic relatedness.* For example, in the sentence "She started painting the walls in blue" there is no strong connection between the word "walls" and the word "blue" or between the word "She" and the word "the", even though they are in proximity.
- *Some languages have free word order*, meaning that there are many possibilities of reordering the words such that the meaning remains the same, so in this case the neighbourhood would vary depending on the reordering.

Next, we will explain how to determine the attention scores that indicate the strength of the dependency between different tokens.

First, we need a way to measure the relatedness between the tokens' representations. Let's consider that each token is represented through its embedding resulted from the P.E.E. processing. We already know that these representations incorporate information related to the meaning and the position of the tokens. Thus, we could think of the dimensions of these embeddings as describing semantic features and/or positional information. Words with a strong dependency on one another will have embeddings with close values for multiple dimensions.

To give an oversimplified example, the dependency between "university" and "student" is stronger than the dependency between "university" and "cat"; this happens because these words have more resembling values for one or multiple dimensions that encode features related to semantical aspects, such as: education, career, academia, etc.; moreover, these words might occur on consecutive ("university student") or very close positions and this also contributes to making the embeddings of these two words more alike. Figure 10 intuitively describes this idea:

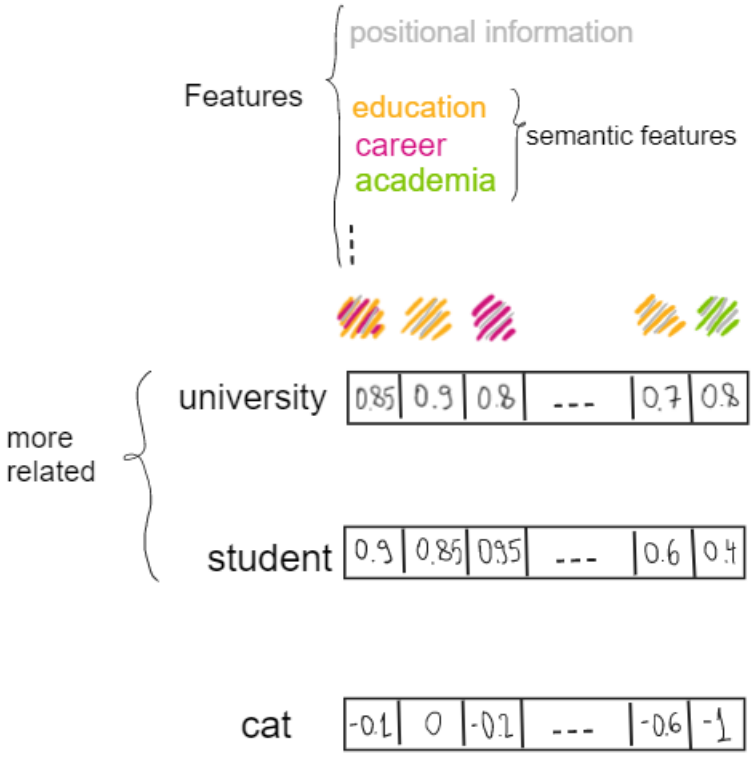

Figure 10. Dummy example to explain the meaning of the embeddings' dimensions

One of the most classic types of attention mechanisms is the *dot-product based attention* which uses the *dot product* to calculate the attention scores. The attention score s_{AB} that measures the degree of dependence between two embeddings A and B is equal to the dot product between the two vectors. The dot product between two embeddings is defined as the sum of the products of their components and also as the cosine of the angle θ between the vectors multiplied by their norms:

$$s_{AB} = A \cdot B = \sum_{k=1}^{n} A_k \cdot B_k = \cos\theta \cdot \|A\| \cdot \|B\| \quad (3)$$

Notice that there is a strong connection between the dot product and the cosine similarity (the last being equal to $\cos\theta$). A high dot product value between A and B indicates a strong dependency between the tokens, while a small/negative value indicates little to no dependency.

Let's suppose we have an embedding Q_i and a set of embeddings $X_1, X_2, \dots, X_n$. To determine the degree to which Q_i is syntactically and semantically related to the other embeddings, we can determine the attention scores $s_{Q_i X_j}$ with formula (3).

In order to interpret better the attention scores given by the dot product, normalization using the Softmax function can be used to bring the values of these scores in range [0,1]. The normalized attention scores represent the attention distribution α_{Q_i}:

$$\alpha_{Q_i X_j} = softmax\left(s_{Q_i X_j}\right) = softmax(Q_i \cdot X_j) \text{ , for j=}\overline{1,n} \quad (4)$$

where

$$softmax(v) = [softmax(v_1), \dots, softmax(v_n)] \quad (5)$$

$$softmax(v_i) = \frac{e^{v_i}}{\sum_{k=1}^{n} v_k} \quad (6)$$

The normalized attention score $\alpha_{Q_i X_j}$ can be interpreted as the probability/ the degree to which there is a dependency between Q_i and X_j. We can build a contextualized representation (/contextualized embedding) of Q_i by summing the products between each embedding X_j and its associated normalized attention score:

$$CQ_i = \sum_{j=1}^{n} X_j \cdot \alpha_{Q_i X_j} \quad (7)$$

This contextualized representation CQ_i is a way of expressing Q_i based on the vectors $X_1, X_2, \dots, X_n$, by giving higher weights ($\alpha_{Q_i X_j}$) to embeddings X_j that are strongly related (syntactically and semantically) to Q_i.

Let's summarizes the steps for determining a contextualized representation for the vector Q_i, given $X_1, X_2, \dots, X_n$:

i. Calculate the attention scores $s_{Q_i X_j}$ by multiplying each vector X_j to Q_i. In this context, embeddings X_j are called keys, while Q_i is called query.
ii. Determine the attention distribution α_{Q_i} by applying Softmax over the attention scores

iii. Determine the contextualized embedding for Q_i, CQ_i, by summing the products between each pair of normalized attention score $\alpha_{Q_i X_j}$ and vector X_j. In this context, the embeddings X_j are called values.

Please note that the final contextualized embedding CQ_i has the same size as the embeddings representing the values.

Figure 11 describes this process of determining a contextualized representation for an embedding Q_i:

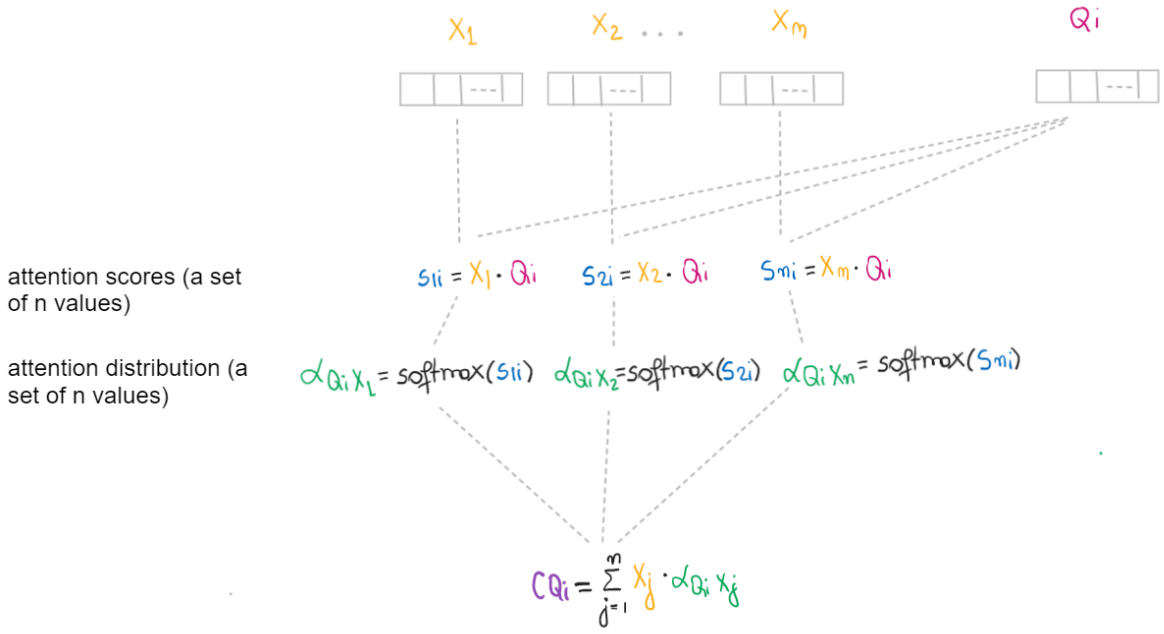

Figure 11. Process of determining the contextualized representation for an embedding Q_i

The above-described procedure can be done for multiple queries: $Q_1 Q_2, \dots, Q_m$, resulting m contextualized embeddings $CQ_1 CQ_2, \dots, CQ_m$. Please note that the process for calculating the contextualized representation for each query can be done completely independently of the other queries and thus can be parallelized.

The "key, value, query" names describe different functions of the embeddings. Usually, the values and the keys are represented by the same embeddings $(X_1, X_2, \dots, X_n)$, as in the presented case. In case of attention mechanism, the queries $(Q_1 Q_2, \dots, Q_m)$ are different from the keys $(X_1, X_2, \dots, X_n)$, while in self-attention the queries and the keys are represented by the same embeddings. Therefore, in case of self-attention, the same embeddings $(X_1, X_2, \dots, X_n)$ play three different roles: keys, queries and values.

[6] provides a very good analogy between the usage of keys, queries and values in the attention mechanism and the retrieval process for recommendation systems: "when you search for videos on Youtube, the search engine will map your **query** (text in the search bar) against a set of **keys** (video title, description, etc.) associated with candidate videos in their database, then present you the best matched videos (**values**)." It is also mentioned that in a search in which only one instance (one value) is retrieved, the attention distribution α is a one hot vector. In general, the attention distribution "proportionally retrieves" values based on their resemblance to the query (Q_i). The resemblance to the query is given by the attention scores.

2.3.2. Self-attention and attention in the transformer architecture

Figure 12 summarizes the three attention mechanisms used inside the transformer's architecture:

- The self-attention mechanism from the encoder
- The self-attention mechanism from the decoder
- The attention mechanism from the decoder

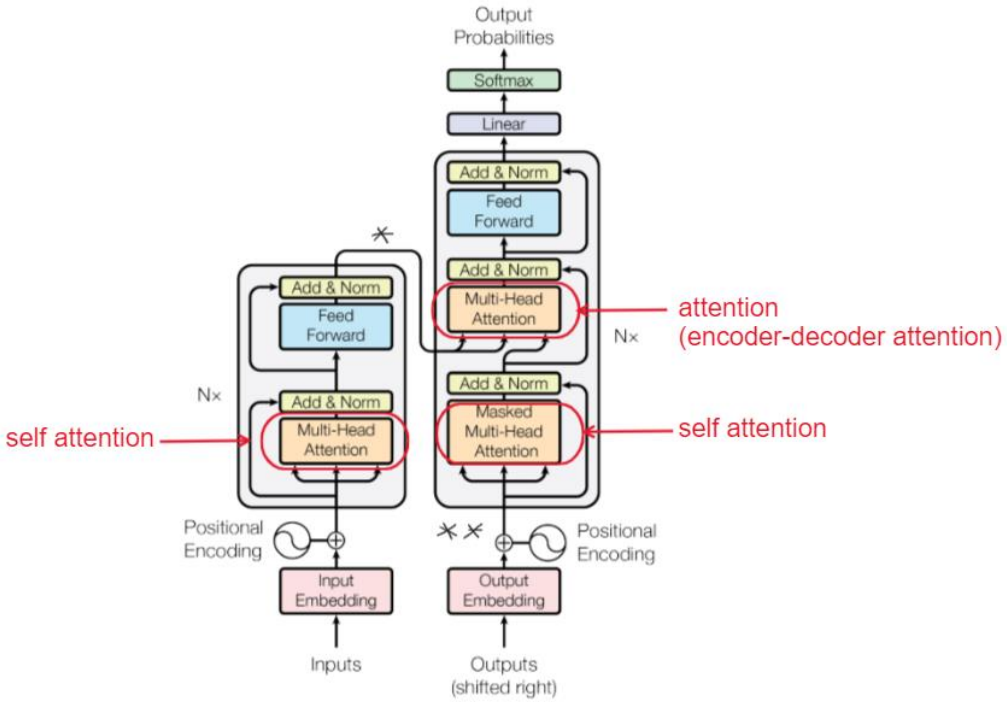

Figure 12. Types of attention mechanism used in the transformer's architecture

The self-attention mechanism from the encoder converts a set of embeddings $X1, X2, \dots, Xn$ (resulted from the P.E.E. processing of the tokens from the input sentence) into a set of contextualized embeddings $CX_1, CX_2, \dots, CX_n$. Note that according to notations from the previous section, $X_i = P_i + E_i$, $for\ i = \overline{1,n}$. The association between the original embeddings and the contextualized representations is 1:1. In case of the machine translation task, the resulted contextualized embeddings represent an encoding of the sentence in the source language, as these representations capture information regarding the syntactic and semantic dependencies between the tokens of the source sentence.

The (masked) self-attention mechanism from the decoder has the same purpose as the previous mechanism, but this time works on the inputs (**) given to the decoder, instead on the inputs given to the encoder. Therefore, this self-attention mechanism outputs a contextualized representation of the partial translation fed to the decoder, which captures the syntactic and semantic relations between the tokens from the partial translation. This type of attention is masked; we will discuss about masking at the end of this section, after we introduce some new notions.

Figure 13 summarized the general idea behind the two self-attention previously discussed mechanisms used by the transformer. Notice that in case of self-attention, the keys, the queries and the values are represented by the same embeddings $(X1, X2, \dots, Xn)$.

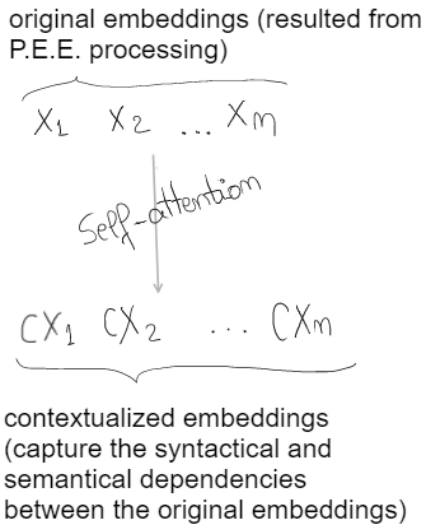

Figure 13. Self-attention in the transformer architecture – general idea

The attention mechanism from the decoder, also named encoder-decoder attention uses:

- as keys and also as values the contextualized representations (that encode the sentence in the source language) outputted by the encoder. Note that the same embeddings are used for keys, as well as for the values.
- as queries the contextualized representations (that describe a partial translation in the target language) outputted by the decoder.

Because it uses two different embedding sequences (one outputted by the encoder and one outputted by the decoder) this type of attention is also called cross-attention. This attention mechanism is also illustrated in the Figure 14 below (notations: K for keys, V for values and Q for queries):

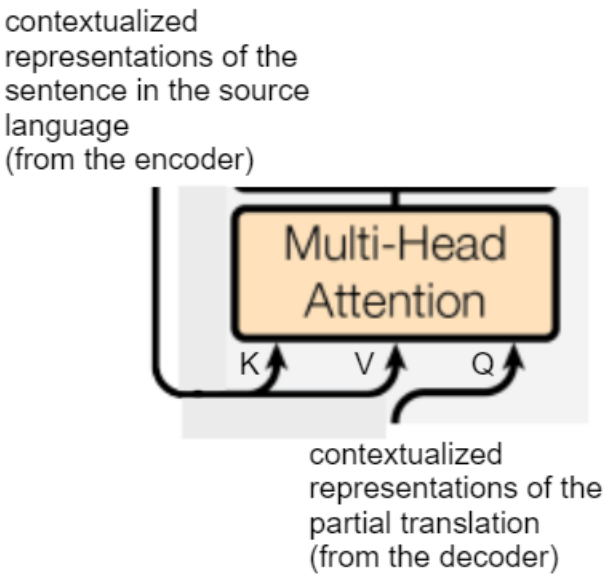

Figure 14. Encoder-decoder attention– general idea

The transformer uses the attention mechanism differently that what we have seen earlier in section 2.3.1.

The first difference is that it separates the three roles that an embedding can have: key, query and value, by learning the weights of three different matrices: W^k, W^Q *and* W^v. Each matrix corresponds to a different key/query/value layer.

The second difference is that it determines all contextualized embeddings for all queries in parallel, by using matrix multiplication operations.

Before describing in detail the (self)-attention mechanism used by the transformer, we need to introduce some notations.

All embeddings representing the keys are grouped in a matrix E_K, all embeddings representing the queries are grouped in a matrix E_Q and all embeddings representing the values are grouped in a matrix E_V. In each matrix E_A, where $A \in \{K, Q, V\}$, embeddings are placed on the rows, more exactly 1_{st} row corresponds to the first embedding, 2_{nd} row to the second, etc.

Matrices E_Q and E_K and E_V have the same number of columns (more exactly, the size of each embedding is the same), let it be d_{model}. Let the number of rows for E_Q be q and the number of rows for E_K and E_V be k. In case encoder-decoder attention is used, matrices E_K and E_V are identical. In case self-attention is used, all three matrices: E_K, $E_{V,}E_Q$ are identical.

The following steps are followed by the transformer to determine the contextualized representations:

- A queries matrix Q are determined by multiplying the embeddings that represent the queries E_Q with the matrix representing the query layer: W^Q. Same thing happens for the keys matrix $K$and values matrix V. Matrices W^Q and W^K have a number of columns equal to d_k, while matrix W^V has a number of columns equal to d_v. Because these matrices W^Q, W^K, W^V change the number of columns (the size of the embeddings) of the resulted matrices: $Q,\ K, V$, they are also called projection matrices/projection layers.

$$\begin{aligned} Q &= E_Q \cdot W^Q \\ K &= E_K \cdot W^K \\ V &= E_V \cdot W^V \end{aligned} \tag{8}$$

- The queries matrix Q (which contains one query per row) is multiplied with the transpose of the keys matrix K^T (which contains one key per column) to obtain a matrix $Q \cdot K^T$ of attention scores. The attention score $(Q \cdot K^T)_{ij}$ is obtained by multiplying query on row i in Q with key on column j in K^T. This

step resembles very much step i) from section 2.3.1., with the difference that now attention scores are calculated for all queries, in parallel by using matrix multiplications.

- The values in the matrix $Q \cdot K^T$ are scaled, by being all divided to value d_k (the size of an embedding corresponding to a key or a query). [7] explains why this scaling is needed; the main idea is that with the increase of the embeddings' sizes d_k, the values in the matrix $Q \cdot K^T$ become large, making the gradients small and thus, hindering the training process.

- The Softmax function is applied to each scaled element in the matrix $Q \cdot K^T$. The result is a matrix α, containing the attention distributions (one per row for each query). This process corresponds to step ii) from section 2.3.1.

$$\alpha = softmax(\frac{Q \cdot K^T}{\sqrt{d_k}}) \quad (9)$$

- The matrix Y of contextualized representations is determined by multiplying the matrix α of attention distributions with the values matrix V. This is analogous to step iii) from section 2.3.1. i_{th} row in Y contains the contextualized embeddings calculated for the i_{th} query.

$$Y = \alpha \cdot V \quad (10)$$

Figure 15 highlights the elements described above, which form what we will call an attention layer.

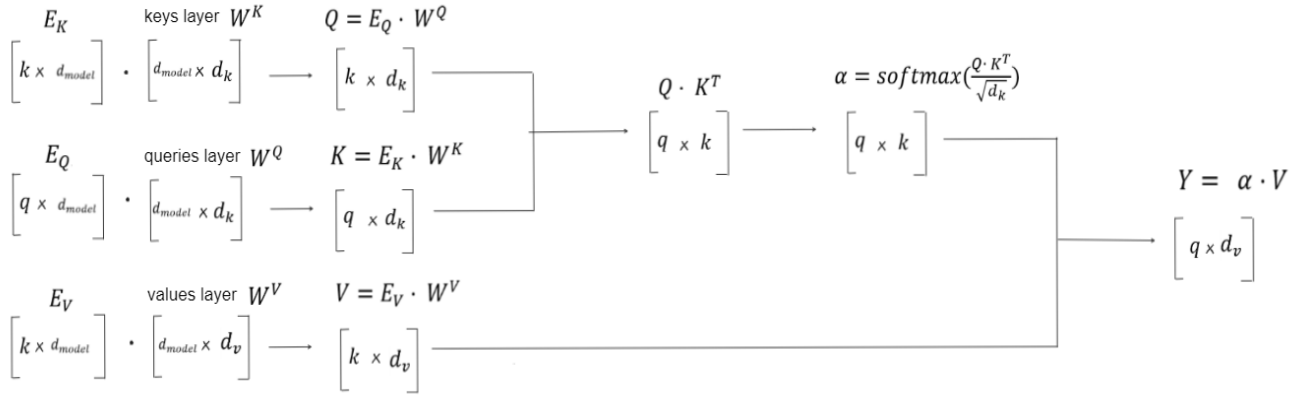

Figure 15. (Self-)attention layer in the transformer's architecture - detailed steps

We previously mentioned that the self-attention mechanism from the decoder is a masked type of attention. We will explain first why masking is needed and then how it is performed.

In the previous section, we mentioned that the decoder is autoregressive and that it generates the output word by word. Therefore, a word *w* generated at a certain time step cannot be used to predict the words generated before him, as *w* was created much later. In conclusion, we need to prevent the decoder form conditioning on future tokens.

To do this, we apply a look-ahead mask M to the matrix $Q \cdot K^T$ containing the attention scores, so that all future generated words are not influencing the predictions of each currently generated word. This mask is represented by a matrix with the same size as the matrix $Q \cdot K^T$, in which all values are equal to 0, except for the upper triangle (which doesn't contain the diagonal), in which all values are $-\infty$. The mask M is summed up to matrix $Q \cdot K^T$, resulting in the final matrix containing the masked attention scores. Figure 16 depicts this final matrix resulted from the masking process.

q1•k1	−∞	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞	−∞
q4•k1	q4•k2	q4•k3	q4•k4	−∞
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

Figure 16. Example of a matrix containing masked attention scores

2.3.3. Multi-*head attention*

All the three types of attention mechanism used by the transformer (which are highlighted in red in Figure 12), are of type multi-head attention. Next, we will see what it is multi-head attention and why it is used in the transformer's architecture.

The attention layer presented in the previous section takes as input three matrices of embeddings (E_Q for queries, E_K for keys and E_V for values) and outputs a matrix Y. This final matrix Y contains a set of q contextualized representations (one representation for each query), each one having a size d_v. Even though the attention layer can model some dependencies between the embeddings, it is hard for a single attention layer to capture all dependencies between the tokens. For this reason, multiple attention layers/heads are used. These layers function separately from one another, having similar components at the same depths. They learn different aspects of the dependencies between the tokens.

Each attention head has its own projection matrices, which are not shared with the other attention heads. Let W_i^Q, W_i^K, W_i^V be the projection matrices associated to the i_{th} attention layer.

Let h be the number of attention heads. Let Y_i be the matrix containing the contextualized representations which are outputted by the i_{th} attention layer. The attention heads are able to computed the matrices Y_i in parallel; at the end of this process, all matrices Y_i are concatenated one to another forming the matrix Y' of size $q * (h \cdot d_v)$. The i_{th} row of Y' contains a contextualized representation size $h \cdot d_v$, corresponding to the i_{th} query.

Y' is passed through a linear projection layer such that the final contextualized embeddings have a size equal to d_{model}. More exactly, Y' is multiplied to a projection matrix W^O of size $(h \cdot d_v) * d_{model}$, resulting a final matrix Y'' of size $q * d_{model}$.

Figure 17 from [1] and Figure 18 from [8] contain images that intuitively describe the multi head attention mechanism.

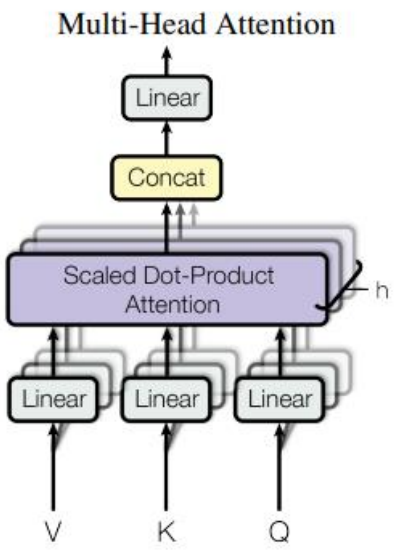

Figure 17. Multi-head attention mechanism – image from original paper "Attention is all you need"

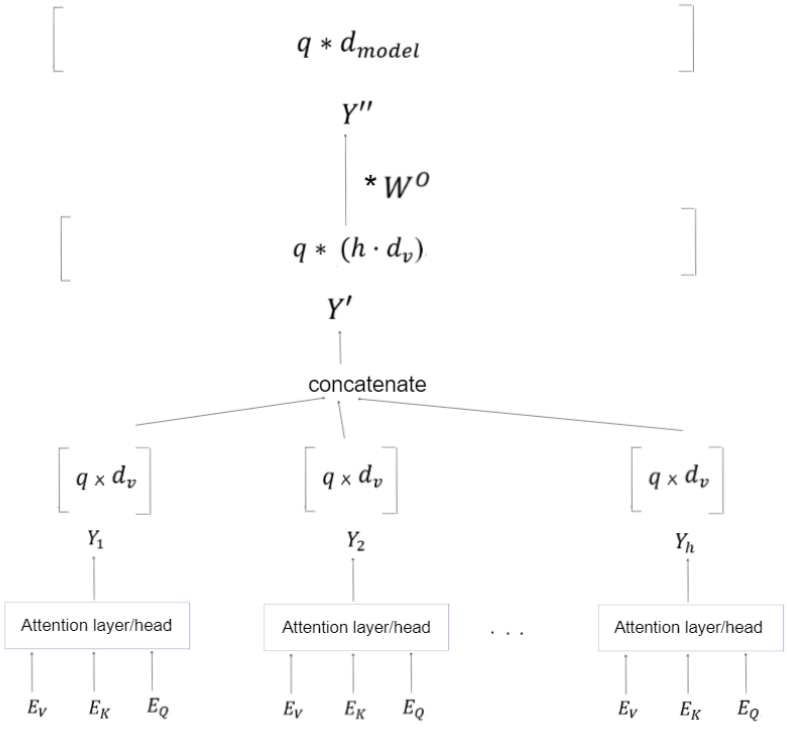

Figure 18. Multi-head attention mechanism – detailed process

2.4. Feed Forward Networks, layer normalizations and residual connections

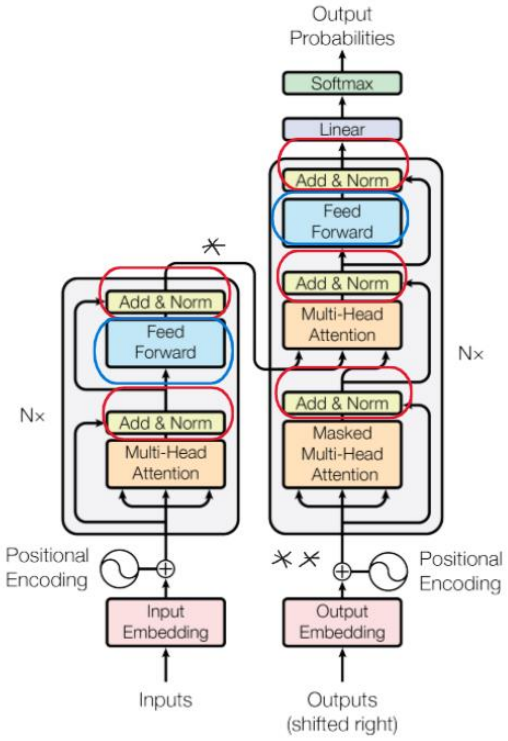

Figure 19. Highlights of the layer normalizations and residual connections (red) and of the feed forwards networks (blue) in the transformer's architecture

The last elements to talk about regarding the transformer's architecture are the layer normalizations, the residual connections and the feed forward networks integrated in the transformer's architecture.

Figure 19 emphasises in red the layer normalizations along with the residual connections ("Add & Norm") and in blue the feed forward networks. Figure 20 (taken from [8]) describes the detailed structure of an encoder block, in which details regarding the above mentioned components are visible.

Both the encoder and the decoder contain a fully connected feed forward network block, formed out of two linear layers with a ReLU transformation between them. These feed forward networks are used to further process the outputs from the attention mechanism, potentially adding more non linearity in the representations.

The transformer's encoder and decoder are composed of multiple complex layers, this indicating that the vanising gradient problem is likely to appear. Residual connections solve this problem, by connecting lower layers to higher layers such that the gradients are allowed to flow easier through the architecture. More exactly, each residual connection creates a shortcut that "skips" a certain block of the transformer (such as a feed forward network block or a multi-head attention block). The residual connections are implemented by adding the skipped block's input to its output and passing it forward.

Each residual connection is followed by a layer normalization procedure, which has the purpose to increase the transformer's training performance by keeping the values the model works with in a range suitable for gradient-based training. The layer normalization procedure is given as input a set of embeddings $X_1, X_2, \dots, X_n$. Each embeddings is normalized by substracting the "mean embedding" μ from it and dividing it by the "standard deviation embedding" σ. Formula for these two special embeddings are given below:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{10}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)^2} \tag{11}$$

Each embedding is finally multiplied by a parameter γ and summed up to a parameter β. Both γ and β parameters are learned during the trainng process. Equation (12) highlights the operations applied to an embedding X_i during the layer normalization process:

$$\text{final } X_i = \gamma \cdot \left(\frac{X_i - \mu}{\sigma}\right) + \beta \tag{12}$$

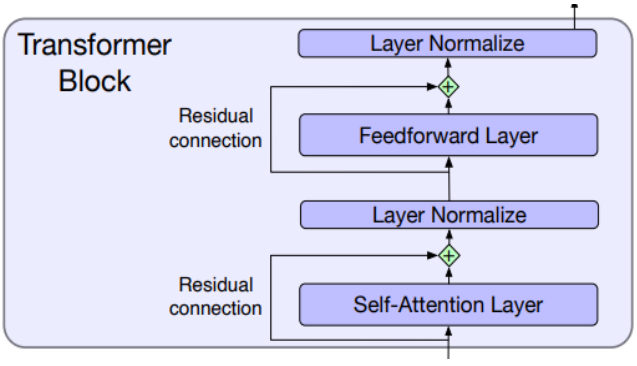

Figure 20. Detailed structure of an encoder block in the transformer's architecture

References

[1] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, 2017, vol. 30. Accessed: Feb. 04, 2023. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[2] Five reasons to embrace Transformer in Computer Vision. Accessed: Feb 04, 2023. [Online]. Available: https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/five-reasons-to-embrace-transformer-in-computer-vision/

[3] Hugging Face library. Accessed: Feb 04, 2023. [Online]. Available: https://huggingface.co/docs/transformers/

[4] Kazemnejad, Amirhossein, "Transformer Architecture: The Positional Encoding", Accessed: Feb 04, 2023. [Online]. Available: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

[5] Abigail See, "Natural Language Processing with Deep Learning CS224N/Ling284 Lecture 8: Machine Translation, Sequence-to-sequence and Attention". Accessed: Feb 04, 2023. [Online]. Available: https://web.stanford.edu/class/cs224n/slides/cs224n-2021-lecture07-nmt.pdf

[6] (https://stats.stackexchange.com/users/95569/dontloo), "What exactly are keys, queries, and values in attention mechanisms?". Accessed: Feb 04, 2023. [Online]. Available: https://stats.stackexchange.com/q/424127

[7] John Hewitt, "Natural Language Processing with Deep Learning CS224N/Ling284 Lecture 9: Self- Attention and Transformers". Accessed: Feb 04, 2023 [Online]. Available: https://web.stanford.edu/class/cs224n/slides/cs224n-2021-lecture09-transformers.pdf

[8] Daniel Jurafsky & James H. Martin, "Speech and Language Processing. Chapter 10. Transformers and Pretrained Language Models". Accessed: Feb 04, 2023. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/10.pdf

[9] Stanford CS224N: NLP with Deep Learning, Winter 2019, Lecture 14 – Transformers and Self-Attention. Accessed: Feb 04, 2023. [Online]. Available: https://www.youtube.com/watch?v=5vcj8kSwBCY&ab_channel=StanfordOnline

[10] Lukasz Kaiser, "Attention is all you need; Attentional Neural Network Models | Machine Leaning Masterclass" Accessed: Feb 04, 2023. [Online]. Available: https://www.youtube.com/watch?v=rBCqOTEfxvg