Dissecting BERT Part 1: Understanding the Transformer

Miguel Romero Calvo Follow

Draft · 19 min read

This is Part 1/2 of Dissecting BERT written jointly by Francisco Ingham and Miguel Romero.

Many thanks to Yannet Interian for her revision and feedback.

In Part 1 we are going to examine the **Transformer** architecture in depth. This blog will explain the full architecture as described in Attention Is All You Need. In Part 2 we will dive into the novel modifications that make BERT particularly effective.

> ***Note:*** *If you are only interested in understanding how BERT works, you need only read and understand the encoder part of this article since that is all BERT uses*

Notation

Before we begin, let's define the notation we will use throughout the article:

> ***emb_dim****: Dimension of the embedding*

> ***input_length**: Length of the input sequence*

> ***target_length**: Length of the target sequence + 1. The +1 is a consequence of the shift.*

> ***hidden_dim**: Size of the hidden layer on the position-wise feed forward networks.*

> ***vocab_size**: Amount of words in your vocabulary (deviated from the corpus).*

> ***target input**: We will use this term interchangeably to describe the input string (set of sentences) or sequence in the decoder.*

Introduction

The **Transformer** is an attention-based architecture for Natural Language Processing (NLP) that was introduced in the paper **Attention Is All You Need** a year ago.

Transfer learning for NLP has quickly become a standard for state of the art results since the release of **ULMFiT** earlier this year. After that, remarkable advances have been achieved by combining the **Transformer** with transfer learning. Two iconic examples of this combination are **OpenAI GPT** and Google AI's **BERT.**

This series aims to:

1. Provide an intuitive understanding of the **Transformer** and **BERT**'s underlying architecture.
2. Explain fundamental principles of what makes **BERT** so successful in NLP.

This blog post will cover the whole **Transformer** architecture, as presented in the original paper.

To explain this architecture we will take the *general to specifics* approach. We will start by a basic understanding of the problem, then we will understand how the data flows through the model and finally we will deep dive into each of the layers to see that what they are doing and how.

Level 1: Problem

The problem that the **Transformer** addresses is translation. To translate a sentence to another language, we want our model to:

- Be able to capture the relations between the words in the input sentence.
- Combine the information contained in the input sentence and what has already been translated at each step.

Imagine that the goal is to translate a sentence from English to Spanish and we are given the following sequences of tokens:

> *X = ['Hello', ',', 'how', 'are', 'you', '?'] (Input sequence)*
> *Y = ['Hola', ',', 'como', 'estas', '?'] (Target sequence)*

First, we want to process the information in the input sequence *X* by combining the information in each of the words of the sequence. This is done inside the part of the **Transformer** called encoder.

Once we have this information mixed in the output of the encoder we want to combine this with the target sequence. This is done in the decoder.

Encoder and decoder are specific parts of the **Transformer** architecture as illustrated in Figure 1. We will investigate each of them in detail in *Level 3-b: Layers*.

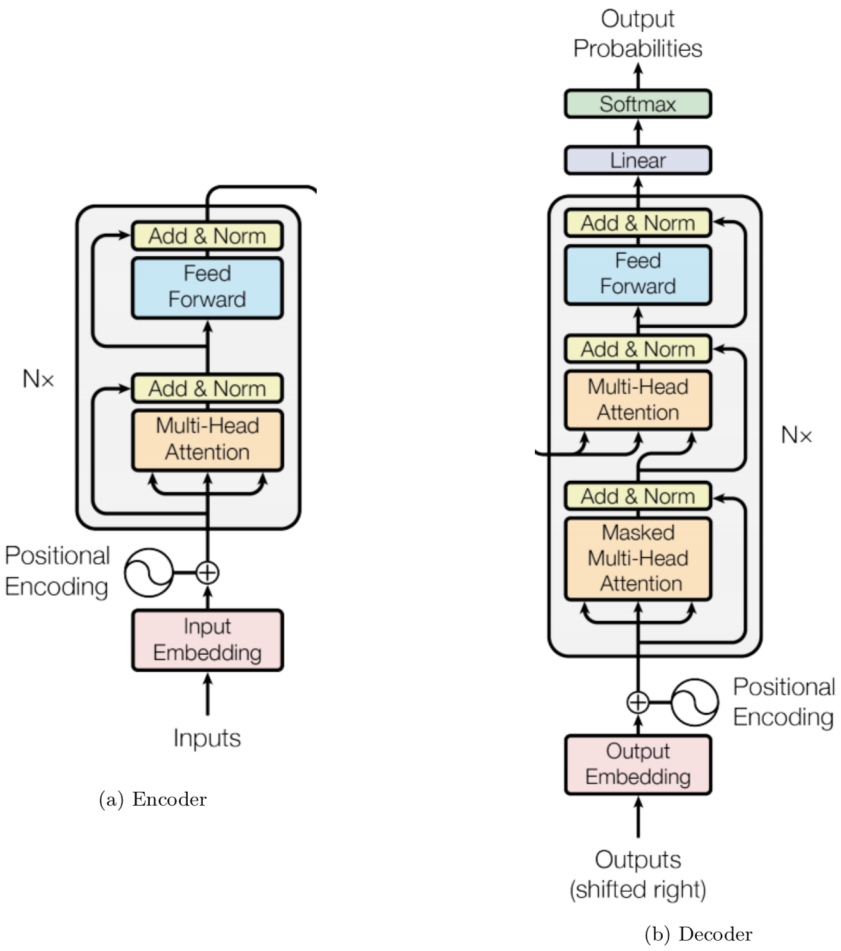


Figure 1: **Transformer** architecture divided into encoder and decoder

Level 2: Flow of Information

The flow of data through the architecture is as follows:

1. The model represents each token as a vector of dimension *d_embedding*. We then have a matrix of dimensions (*input_length*) x (*emb_dimb*) for a specific input sequence.

2. It then adds positional information (positional encoding). This step will return a matrix of dimensions (*input_length*) x (*emb_dim*), just as in the previous step.

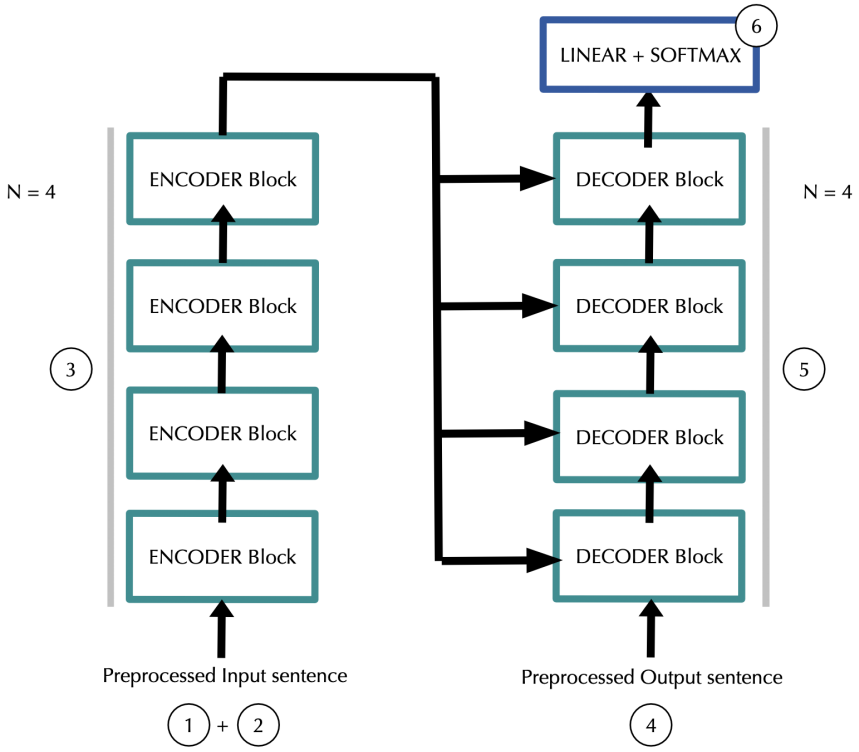
3. The data goes through N encoder blocks. After that, we obtain a matrix of dimensions *(input_length)* x *(emb_dim)*.

4. The target sequence is masked and sent through the decoder's equivalent of 1) and 2). The output has dimensions *(target_length)*

x *(emb_dim)*.

5. The result of 4) goes through N decoder blocks. In each of the iterations, the decoder is using the encoder's output 3). This is represented in Figure 2 by the arrows from the encoder to the decoder. The dimensions of the output are *(target_length)* x *(emb_dim)*.

6. Finally, it applies a fully connected layer and a row-wise softmax. The dimensions of the output are *(target_length)* x *(vocab_size)*.



> ***Note**: In the paper's experiments, N was choosen to be equal to 6.*

> ***Note**: Different encoder or decoder blocks don't share weights.*

Level 3-a: Inputs

Remember that the described algorithm is processing both the input sentence and the target sentence to train the network. In this section, we discuss how given an input sentence (e.g. "Hello, how are you ?") and a target sentence (e.g. "Hola, como estás ?") we obtain a matrix

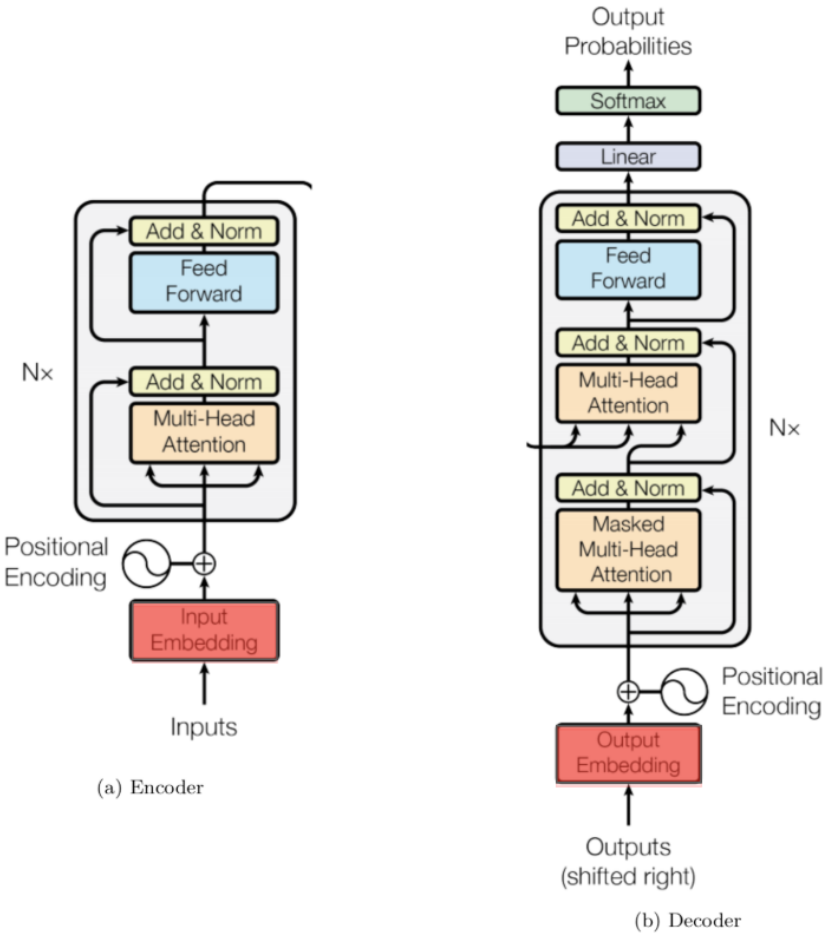
representing the input sentence for the encoder blocks and another matrix representing the target sentence for the decoder blocks.

The process is composed of two general steps:

1. Token embeddings
2. Encoding of the positions.

Moreover, this process is the same for both the input sentence X and the shifted target sentence Y.

From words to vectors



(a) Encoder

(b) Decoder

In the case of the encoder inputs, the mapping from the sentence to the matrix that will enter the encoder block is the same for training and test time (i.e a sentence to translate).

Tokenization, numericalization and embeddings do not differ from the traditional way it is done in NLP using RNNs. Given a sentence in a corpus:

> *" Hello, how are you?"*

The first step will be to tokenize it:

> *" Hello, how are you?" → ["Hello", ", ", "how", "are", "you", "?"]*

This will be followed by numeralization. That is, mapping each token to a unique integer on the corpus' vocabulary.

> *["Hello", ", ", "how", "are", "you", "?"] → [34, 90, 15, 684, 55, 193]*

> ***Note**: In the case of target input one more step is needed. This is:*

> *["Hola", ", ", "como", "estás", "?"]→["<SS>", "Hola", ", ", "como", "estás", "?"]*

Next, we will get the embedding for each word in the sequence. Each word of the sequence is mapped to a k-dimensional vector that the model will learn during training. You can think about it as a look-up of a vector for each token. The elements of those vectors are going to be treated as model parameters and will be optimised using back-propagation just like any other weights.

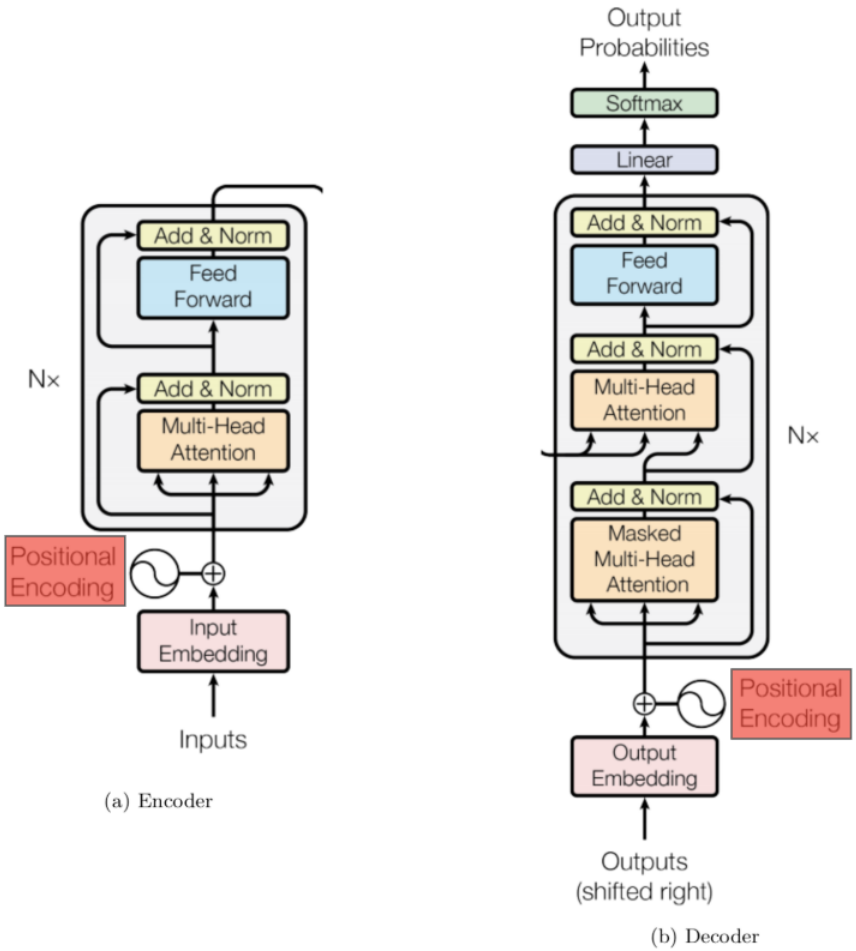
Therefore, for each token, we will look up its corresponding vector:

$$
\begin{aligned}
34 &\rightarrow E[34] = [123.4, 0.32, \cdots, 94, 32] \\
90 &\rightarrow E[90] = [83, 34, \cdots, 77, 19] \\
15 &\rightarrow E[15] = [0.2, 50, \cdots, 33, 30] \\
684 &\rightarrow E[684] = [289, 432.98, \cdots, 150, 92] \\
55 &\rightarrow E[55] = [80, 46, \cdots, 23, 32] \\
193 &\rightarrow E[193] = [41, 21, \cdots, 74, 33]
\end{aligned}
$$

Stacking each of the vectors together we obtain a matrix Z of dimensions length (*input_length*) x (*emb_dim*) :

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{ccccc} < & - & d_{model} & - & > \end{array} \\ \begin{pmatrix} 123.4 & 0.32 & \cdots & 94 & 32 \\ 83 & 34 & \cdots & 77 & 19 \\ 0.2 & 50 & \cdots & 33 & 30 \\ 289 & 432.98 & \cdots & 150 & 92 \\ 80 & 46 & \cdots & 23 & 32 \\ 41 & 21 & \cdots & 74 & 33 \end{pmatrix} \end{array}$$

Positional Encoding



(a) Encoder

(b) Decoder

At this point, we have a matrix representation of our sequence. However, these representations are not encoding the difference between the same word appearing in different positions of a sequence.

The approach chosen in the paper to add this information is adding to Z a matrix P with positional encodings.

$$Z + P$$

The authors chose to use a combination of sinusoidal functions. Mathematically:

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{model}}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{model}}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

In our example P would look like this:

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \overset{<}{} \\ \begin{pmatrix} \sin\left(\frac{0}{10000^{\frac{0}{d_{model}}}}\right) \\ \sin\left(\frac{1}{10000^{\frac{0}{d_{model}}}}\right) \\ \sin\left(\frac{2}{10000^{\frac{0}{d_{model}}}}\right) \\ \sin\left(\frac{3}{10000^{\frac{0}{d_{model}}}}\right) \\ \sin\left(\frac{4}{10000^{\frac{0}{d_{model}}}}\right) \\ \sin\left(\frac{5}{10000^{\frac{0}{d_{model}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{-}{} \\ \begin{pmatrix} \cos\left(\frac{0}{10000^{\frac{0}{d_{model}}}}\right) \\ \cos\left(\frac{1}{10000^{\frac{0}{d_{model}}}}\right) \\ \cos\left(\frac{2}{10000^{\frac{0}{d_{model}}}}\right) \\ \cos\left(\frac{3}{10000^{\frac{0}{d_{model}}}}\right) \\ \cos\left(\frac{4}{10000^{\frac{0}{d_{model}}}}\right) \\ \cos\left(\frac{5}{10000^{\frac{0}{d_{model}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{d_{model}}{} \\ \begin{pmatrix} \sin\left(\frac{0}{10000^{\frac{2}{d_{model}}}}\right) \\ \sin\left(\frac{1}{10000^{\frac{2}{d_{model}}}}\right) \\ \sin\left(\frac{2}{10000^{\frac{2}{d_{model}}}}\right) \\ \sin\left(\frac{3}{10000^{\frac{2}{d_{model}}}}\right) \\ \sin\left(\frac{4}{10000^{\frac{2}{d_{model}}}}\right) \\ \sin\left(\frac{5}{10000^{\frac{2}{d_{model}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{-}{} \\ \begin{pmatrix} \cos\left(\frac{0}{10000^{\frac{2}{d_{model}}}}\right) \\ \cos\left(\frac{1}{10000^{\frac{2}{d_{model}}}}\right) \\ \cos\left(\frac{2}{10000^{\frac{2}{d_{model}}}}\right) \\ \cos\left(\frac{3}{10000^{\frac{2}{d_{model}}}}\right) \\ \cos\left(\frac{4}{10000^{\frac{2}{d_{model}}}}\right) \\ \cos\left(\frac{5}{10000^{\frac{2}{d_{model}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{>}{} \\ \begin{pmatrix} \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \end{pmatrix} \end{array}$$

Observe that the resulting matrix

$$X = Z + P$$

that is the input of the first encoder or decoder block will have dimensions (*input_length*) x (*emb_dim*) or (t*arget_length*) x (*emb_dim*) respectively.

> ***Note**: The results with this approach allow us to obtain similar results as if we did it with learning weights with the advantage of having fewer parameters in the model.*

Particularities in the target input

The embedding and positional encoder parts are exactly like the ones for the input sequence. Just for the sake of completeness, observe that the token <SS> will have a learned embedded vector just as any other regular token. Following the previous notation, the matrix X entering in the decoder block will be:

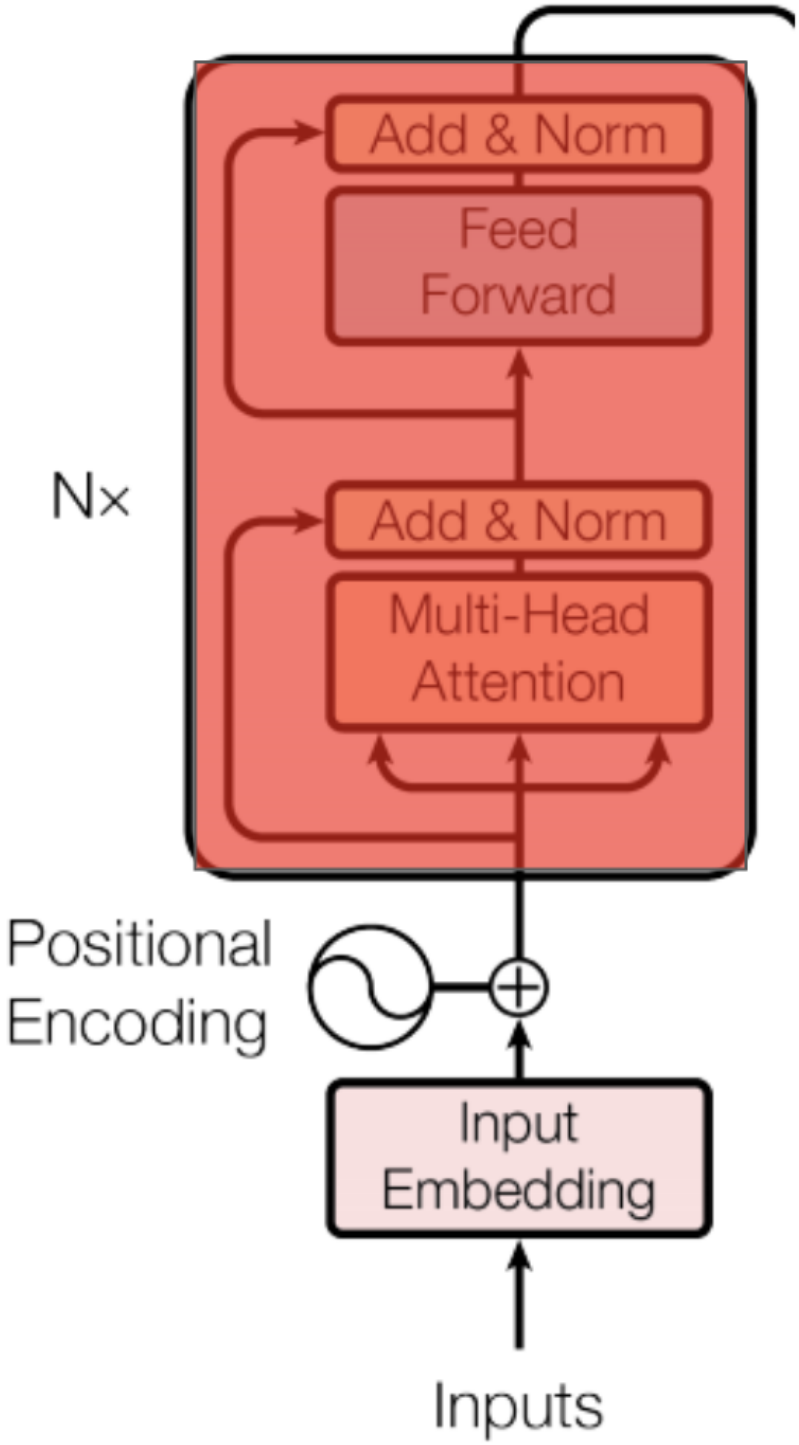
$$X = Z + P$$

where X has dimensions (target_length) x (embedding dimension).

Level 3-b: Layers

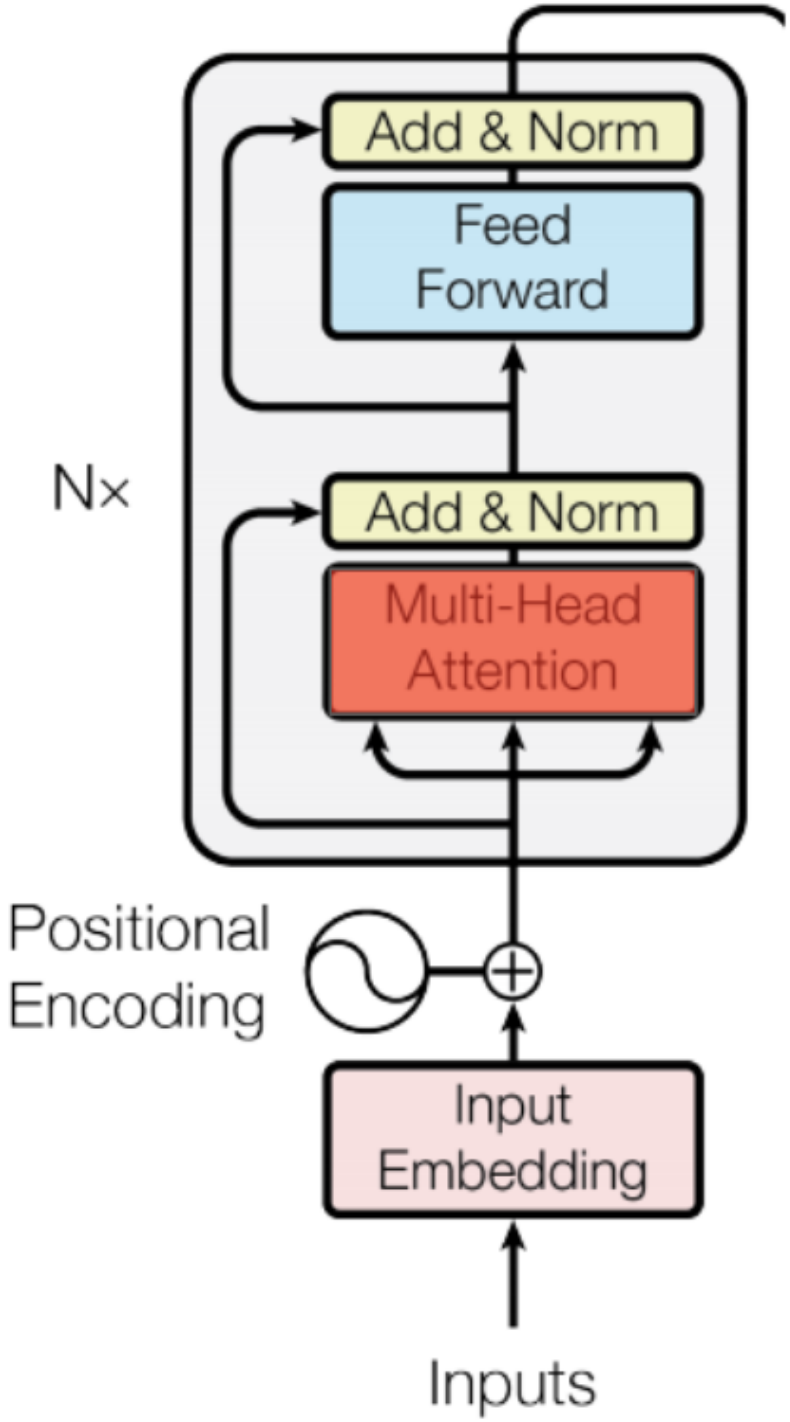
Encoder block

A total of N encoder blocks are chained together to generate the encoder's output. A specific block is in charge of building relationships between the input's vectorial representations and encode them in its output.



Intuitively, doing this iterative process through the blocks will help the neural network capture more complex relationships between words in the input sequence. You can think about this process as iteratively building the meaning of the input sequence as a whole.

Multi-Head Attention



The **Transformer** uses **Multi-Head Attention**, which means it computes attention *h* different times with different weight matrices and then concatenates the results together.

The result of each of those parallel computations of Scaled Dot-Product Attention is called a head. We are going to denote a specific head and the associated weighted matrices with the subscript *i*.

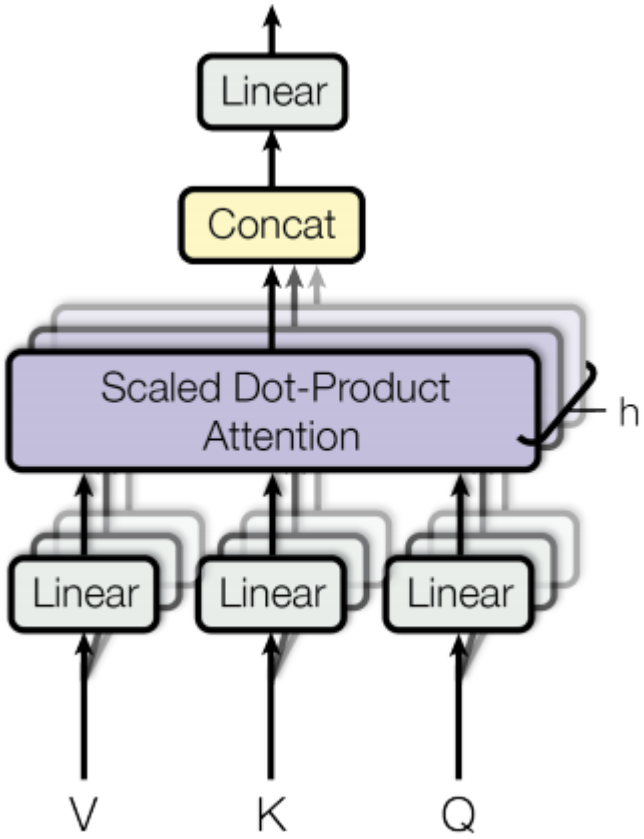


Figure 7: Illustration of the parallel heads computations and their concatenation.

As shown in Figure 7, once all the heads have been computed they will be concatenated. This will result in a matrix of dimensions (*input_length*) x (*h*d_v*). Afterwards, a linear layer with weight matrix W^0 of dimensions (*h*d_v*) x (*emb_dim*) will be applied leading to a final result of dimensions (*input_length*) x (*emb_dim*). Mathematically:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Each head is going to be characterised by three different projections given by matrices:

$$W_i^K \text{ with dimensons } d_{model}\text{x}d_k$$
$$W_i^Q \text{ with dimensons } d_{model}\text{x}d_k$$
$$W_i^V \text{ with dimensons } d_{model}\text{x}d_v$$

To compute a head we will take the input matrix *X* and separately project it with the above weight matrices.

$$XW_i^K = K_i \text{ with dimensons } input\_length \text{ x } d_k$$
$$XW_i^Q = Q_i \text{ with dimensons } input\_length \text{ x } d_k$$
$$XW_i^V = V_i \text{ with dimensons } input\_length \text{ x } d_v$$

> ***Note**: In the paper d_k and d_v are set such that d_k = d_v = emb_dim/h*

Once we have *K_i*, *Q_i* and *V_i* we use them to compute the Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Graphically:

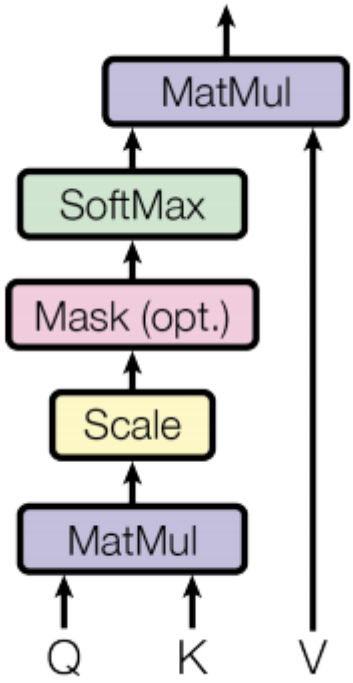


Figure 8: Illustration of the Dot-Product Attention.

> ***Note**: In the encoder block the computation of Attention do not uses mask. We will see in upcoming sections that the decoder usess masking.*

Let's start by understanding what is attention doing by looking at the matrix product between *Q_i* and *K_i* transposed.

$$Q_i K_i^T$$

Remember that *Q_i* and *K_i* were different projections of the tokens into a *d_k* dimensional space. Therefore we can think about the dot product of those projections as a measure of similarity between tokens projections. For every vector projected through *Q_i* the dot product with the projections through *K_i* measure the similarity between those vectors. If we call *v_i* and *u_j* the projections of the *i-th* token and the *j-th* token through *Q_i* and *K_i* respectively, their dot product can be seen as:

$$v_i u_j = \cos(v_i, u_j)||v_i||_2||u_j||_2$$

Thus, it can be seen as a measure of how similar is the direction between u_i, v_j and how large are their lengths. The more similar the direction the bigger and for a fixed u_i the further the v_j from the origin the bigger the dot product (for a fixed direction).

You may like to think about this matrix product as the one encoding the relationships between each of the tokens in the input sequence.

> ***Note**: The result of the matrix multiplication will have dimensions (input_length)* x *(input_length)*.

After that, the matrix is divided element-wise by the square root of *d_k*.

The next step is a **Softmax applied row-wise**. One independent Softmax for the elements of each row.

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

In our example, this could be:

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 78.49 & 43.29 & 1.2 & 41.74 & 91.43 & 74.47 \\ 95.84 & 28.78 & 57.13 & 68.20 & -60.94 & 26.85 \\ -95.69 & -52.16 & 17.00 & 45.71 & 48.49 & 64.35 \\ -69.92 & 85.16 & 94.94 & 91.04 & -92.83 & 77.49 \\ 65.85 & 55.85 & 62.54 & -97.46 & 76.38 & 13.20 \\ -30.05 & -4.52 & 76.02 & 42.35 & 15.29 & 63.61 \end{pmatrix} \end{array} \Longrightarrow$$

Before Softmax

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{ccccccc} Hello & , & how & are & you & ? & \end{array} \\ \begin{pmatrix} 72.40 * 10^{-06} & 1.23 * 10^{-21} & 6.51 * 10^{-40} & 2.62 * 10^{-22} & 9.99 * 10^{-01} & 4.30 * 10^{-08} \\ 1.00 * 10^{+00} & 7.51 * 10^{-30} & 1.54 * 10^{-17} & 9.91 * 10^{-13} & 8.15 * 10^{-69} & 1.09 * 10^{-30} \\ 3.12 * 10^{-70} & 2.51 * 10^{-51} & 2.72 * 10^{-21} & 8.03 * 10^{-09} & 1.29 * 10^{-07} & 9.99 * 10^{-01} \\ 2.47 * 10^{-72} & 5.54 * 10^{-05} & 9.80 * 10^{-01} & 1.98 * 10^{-02} & 2.77 * 10^{-82} & 2.58 * 10^{-08} \\ 2.67 * 10^{-05} & 1.21 * 10^{-09} & 9.75 * 10^{-07} & 3.17 * 10^{-76} & 9.99 * 10^{-01} & 3.64 * 10^{-28} \\ 8.59 * 10^{-47} & 1.05 * 10^{-35} & 9.99 * 10^{-01} & 2.38 * 10^{-15} & 4.21 * 10^{-27} & 4.07 * 10^{-06} \end{pmatrix} \end{array} \begin{array}{c} \\ = 1 \\ = 1 \\ = 1 \\ = 1 \\ = 1 \\ = 1 \end{array}$$

After Softmax

Resulting in rows with numbers between zero and one that sums to one. The next step will be the product of this with *V_i*:

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

Example 1

For the sake of understanding let suppose a dummy example:

Suppose that the resulting first row of

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

is [0,0,0,0,1,0]. Hence, because the 1 is in the 5th position of the vector, the result will then be:

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \end{array} \begin{array}{c} d_v \\ \begin{pmatrix} v_{Hello} \\ v_{,} \\ v_{how} \\ v_{are} \\ v_{you} \\ v_{?} \end{pmatrix} \end{array} = \begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} d_v \\ \begin{pmatrix} v_{you} \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \end{pmatrix} \end{array}$$

Where *v_{token}* is the projection through *V_i* of the token representation. Observe that in that case the word *"hello"* ends up with a representation based on the 4th token *"you"* of the input for that head.

Supposing an equivalent example for the rest of the heads. The word *"Hello"* will be now represented by the concatenation of the different projections of other words. The network will learn over training time which relationships to learn and which tokens to relate to each other.

Example 2

Let us now complicate the example a little bit more. Suppose now our previous example in the more general scenario where there isn't just a single one per row but decimal positive numbers that sum to 1. Let's visualise that:

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 0.1 & 0 & 0.06 & 0.1 & 0.6 & 0.14 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \end{array}$$

If we do as in the previous example and multiply that to *V_i*:

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 0.1 & 0 & 0.06 & 0.1 & 0.6 & 0.14 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \end{array} \begin{array}{c} d_v \\ \begin{pmatrix} v_{Hello} \\ v_{,} \\ v_{how} \\ v_{are} \\ v_{you} \\ v_{?} \end{pmatrix} \end{array} =$$

Which result in a matrix where each row is a composition of the projection through *V_i* of the token's representations:

$$\begin{matrix} & < - - - - - - - - - - - - - d_v - - - - - - - - - - - - - > \\ Hello & 0.1v_{Hello} + 0v_, + 0.06v_{how} + 0.1v_{are} + 0.6v_{you} + 0.14v_? \\ , & \cdots\cdots\cdots\cdots\cdots\cdots \\ how & \cdots\cdots\cdots\cdots\cdots\cdots \\ are & \cdots\cdots\cdots\cdots\cdots\cdots \\ you & \cdots\cdots\cdots\cdots\cdots\cdots \\ ? & \cdots\cdots\cdots\cdots\cdots\cdots \end{matrix}$$

Observe that we can think about the resulting "representation" of "Hello" as a weighted centroid of the projected vectors through *V_i* of the input tokens.

Thus, a specific head captures a specific relationship between the input tokens. Now, if we do that *h* times (a total *of* h heads) the model is capturing *h* different relationships between input tokens.

Following up, assume that the example above referred to the first head. Then the first row would be:

$$V_{Hello,1} = 0.1v_{Hello} + 0v_, + 0.06v_{how} + 0.1v_{are} + 0.6v_{you} + 0.14v_?$$

Then the first row of the result of the **Multi-Head Attention** layer, i.e. the representation of "Hello" at this point, will be

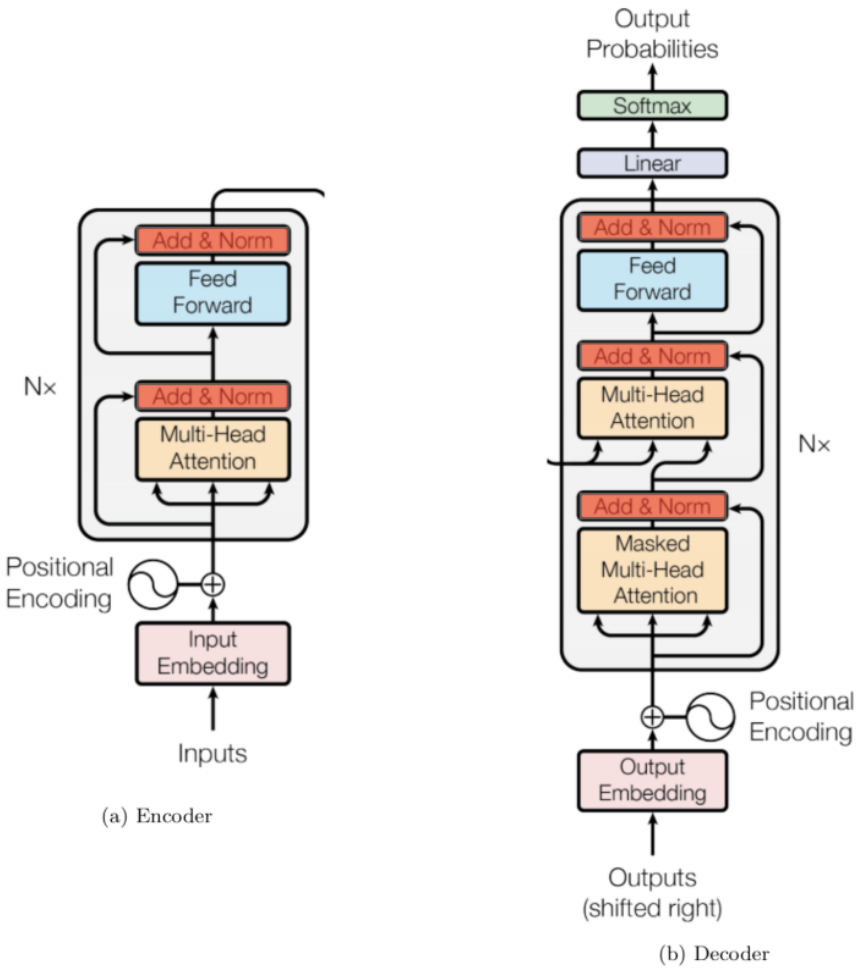
$$Concat(V_1, V_2, ..., V_h)W_0$$

Which is a vector of length *emb_dim* given that the matrix *W_0* has dimensions (*d_v*h*) x (*emb_dim*).

Applying the same logic in the rest of the rows/tokens representations we obtain a matrix of dimensions (*input_length*) x (*emb_dim*).

Thus, at this point, the representation of the token is the concatenation of *h* weighted centroids through the *h* different learned projections.

Dropout, Add & Norm



(a) Encoder

(b) Decoder

Before this layer there always is a layer for which inputs and outputs have the same dimensions (MultiHead Attention or Feed-Forward). We will call that layer *Sublayer()* and its input *x*.

After each layer (be it a **Feed Forward** or **MultiHead Attention**) dropout is applied with probability 0,1. Call this result Dropout(Sublayer(x)). This result is added to the Sublayer's input *x*, and we get:

> x + Dropout(Sublayer(x))

Observe that in the context of a **Multi-Head Attention** layer, this means adding the original representation of a token, x, representation based on the relationship with other tokens. It is like telling the model:

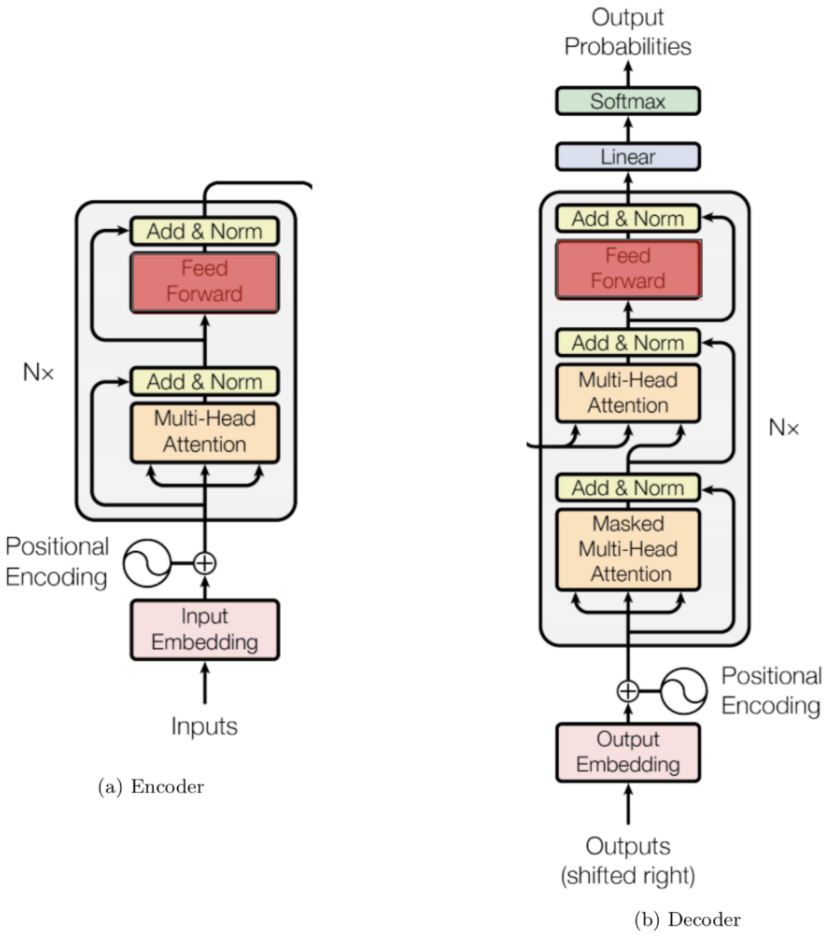
> *"Learn the relationship with the rest of the tokens, but, don't forget your own meaning or get the strongest relationship with yourself!"*

Finally, a token-wise/row-wise normalisation is computed with the mean and standard deviation of each row. This improves the stability of the network.

The output of those layers is then:

$$LayerNorm(x + Dropout(Sublayer(x)))$$

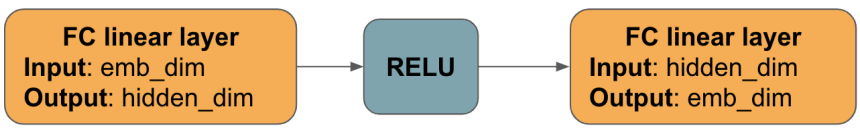
Position-wise Feed-Forward Network



(a) Encoder

(b) Decoder

This step is composed of the following layers:



Mathematically, for each row in the output of the previous layer:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where W_1 and W_2 is a (d_embedding) x (d_F) and (d_F) x (d_embedding) matrix respectively.

Observe that during this step, vector representations of tokens don't "interact" with each other. It is equivalent to run the calculations row-wise and stack the resulting rows in a matrix.

The output of this step will be of dimension (input_length) x (*emb_dim*).

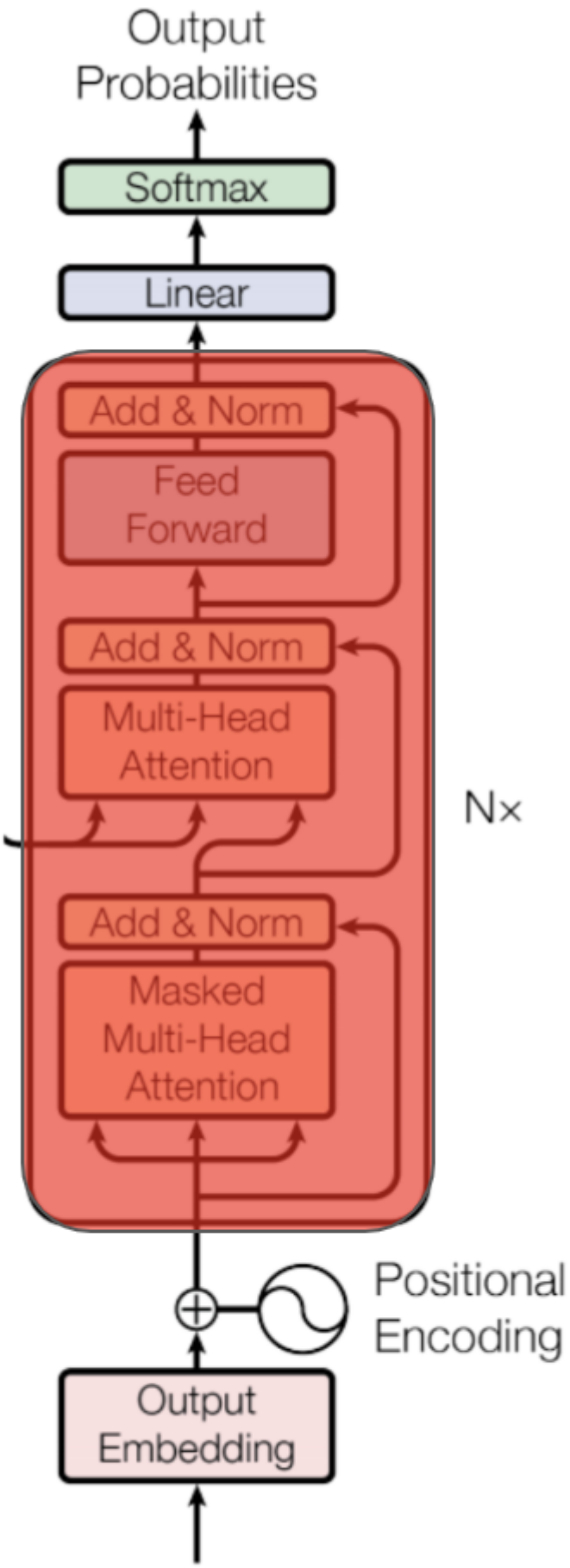
Observations about the encoder

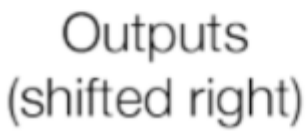
1. BERT uses only the encoder part of the **Transformer** architecture as described in Attention Is All You Need. It is not necessary to understand the whole architecture. If you are here for BERT you can now move directly to Part 2.
2. The dimensions of the input and the output of the encoder block are the same. Hence, it make sense to use the output of one encoder block as the input of the next encoder block.

Decoder

In this section we will cover those parts of the decoder that differ from those covered in the encoder.

Decoder block—Training vs Testing


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Positional Encoding
Output Embedding



During test time we don't have the ground truth. The steps, in this case, will be as follows:

1. Compute the embedding representation of the input sequence.
2. Use a starting sequence token, for example <SS> as the first target sequence: [<SS>]. The model gives as output the next token.
3. Add the last predicted token to the target sequence and use it to generate a new prediction [<SS>, Prediction_1,…,Prediction_n]
4. Do step 3 until the predicted token is the one representing the End of the Sequence, for example <EOS>.

During training we have the ground truth i.e. the tokens we would like the model to output for every iteration of the above process. Since we have the target in advance, we will give the model the whole shifted target sequence at once and ask it to predict the non-shifted target.

Following up with our previous examples we would input:

> *[‘<SS>’,’Hola’, ‘,’, ‘ como’, ‘estas’, ‘?’]*

and the expected prediction would be:

> *[’Hola’, ‘,’, ‘ como’, ‘estas’, ‘?’,’<EOS>’]*

However, there is a problem here. What if the model sees the expected token and uses it to predict itself? For example, it might see *‘estas’* at the right of *‘como’* and use it to predict *‘estas’*. That’s not what we want because the model will not be able to do that a testing time.

We need to modify some of the attention layers not to allow the model to see information on the right (or down in the matrix of vector representation) but allow it to use the already predicted words.

Let's illustrate this with an example. Given:

> *['<SS>','Hola', ',', 'como', 'estas', '?']*

we will transform it into a matrix as described above and add positional encoding. This would result in a matrix:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} \begin{array}{ccccc} < & - & d_{model} & - & > \end{array} \\ \begin{pmatrix} -3.521 & 23.625 & \cdots & -3.041 & 21.150 \\ 28.256 & -27.21 & \cdots & -30.80 & 44.232 \\ -25.34 & -39.38 & \cdots & 44.016 & 18.240 \\ 22.126 & 14.527 & \cdots & -18.22 & 48.169 \\ 49.093 & -48.61 & \cdots & -17.78 & 26.766 \\ 11.692 & -40.30 & \cdots & -4.356 & 32.497 \end{pmatrix} \end{array}$$

And just as in the encoder the output of the decoder block will be also a matrix of sizes (*target_length*) x (*emb_dim*). After a row-wise linear (a linear layer in the form of matrix product through the right) and a Softmax per row this will result in a matrix for which the maximum element per row indicates the next word.

That means that the row assigned to "<SS>" is in charge of predicting "Hola", the row assigned to "Hola" is in charge of predicting "," and so on. Hence, to predict "estas" we will allow that row to directly interact with the green region but not with the red region in the following Figure 13.

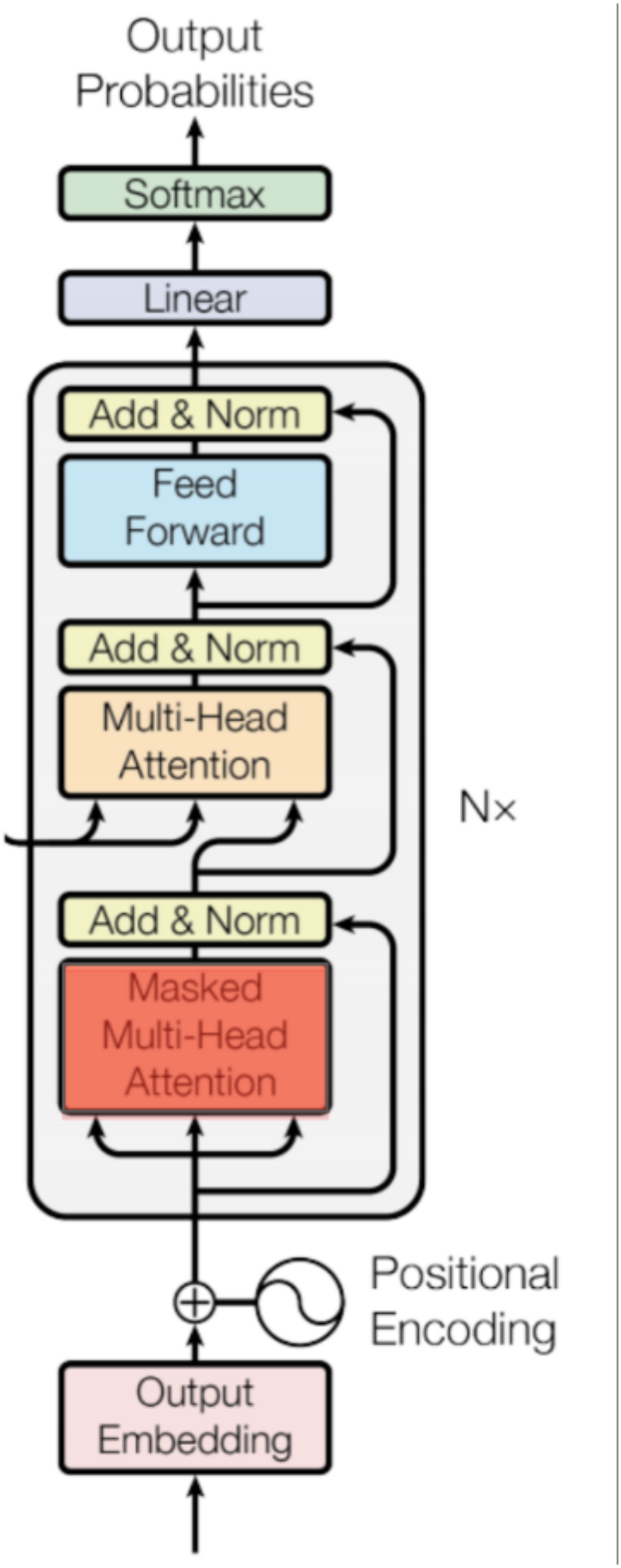
$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} \begin{array}{ccccc} < & - & d_{model} & - & > \end{array} \\ \begin{pmatrix} -3.521 & 23.625 & \cdots & -3.041 & 21.150 \\ 28.256 & -27.21 & \cdots & -30.80 & 44.232 \\ -25.34 & -39.38 & \cdots & 44.016 & 18.240 \\ 22.126 & 14.527 & \cdots & -18.22 & 48.169 \\ 49.093 & -48.61 & \cdots & -17.78 & 26.766 \\ 11.692 & -40.30 & \cdots & -4.356 & 32.497 \end{pmatrix} \end{array}$$

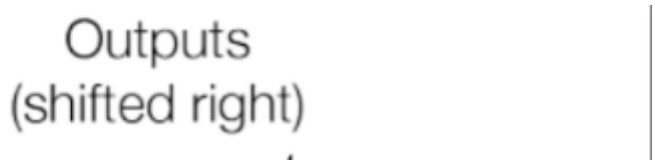
Figure 13: Diagram of what words is the row of "como" going to be able to see.

Observe that we don't have problems in the linear layers because they are defined to be token-wise/row-wise in the form of a matrix multiplication through the right.

The problem will be in **Multi-Head Attention** and the input will need to be masked. We will talk more about masking in the next section.

Masked Multi-Head Attention


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Positional Encoding
Output Embedding



This will work exactly as the **Multi-Head Attention** mechanism but adding masking to our input.

The only **Multi-Head Attention** block where masking is required is the first one of each decoder block. This is because the one in the middle is used to combine information between the encoded inputs and the outputs inherited from the previous layers. There is no problem in combining every target token's representation with any of the input token's representations (since we will have all of them at test time).

The modification will take place after computing:

$$\frac{Q_i K_i^T}{\sqrt{d_k}}$$

Observe that this is a matrix such as:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} <SS> & Hola & , & como & estás & ? \end{array} \\ \begin{pmatrix} -29.59 & -6.044 & -13.48 & 29.626 & 45.840 & -48.69 \\ -15.26 & 46.884 & -45.50 & 21.835 & 24.514 & -17.68 \\ 30.225 & -2.567 & 4.6751 & 10.244 & 4.2682 & -11.86 \\ -8.656 & -13.94 & -29.00 & 48.459 & 22.416 & -39.63 \\ -5.156 & 48.210 & 8.5994 & -20.29 & -33.36 & -17.24 \\ -46.01 & 10.281 & -39.73 & 28.344 & 25.826 & 20.824 \end{pmatrix} \end{array}$$

Now, the masking step is just going to set to minus infinity all the entries in the strictly upper triangular part of the matrix. In our example:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} <SS> & Hola & , & como & estás & ? \end{array} \\ \begin{pmatrix} -29.59 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -15.26 & 46.884 & -\infty & -\infty & -\infty & -\infty \\ 30.225 & -2.567 & 4.6751 & -\infty & -\infty & -\infty \\ -8.656 & -13.94 & -29.00 & 48.459 & -\infty & -\infty \\ -5.156 & 48.210 & 8.5994 & -20.29 & -33.36 & -\infty \\ -46.01 & 10.281 & -39.73 & 28.344 & 25.826 & 20.824 \end{pmatrix} \end{array}$$

That's it! Now we will continue as described in the **Multi-Head Attention** for the encoder.

Let's now dig in what does it means mathematically to set those elements to minus infinity. Observe that if those entries are relative attention measures per each row, the larger they are, the more attention we need to pay to that token. So setting those elements to minus infinity is mainly saying: " For the row assigned of predicting *"estás"* (the one with input *"como")*, ignore *"estás"* and *?*". Our Softmax output would look like this:

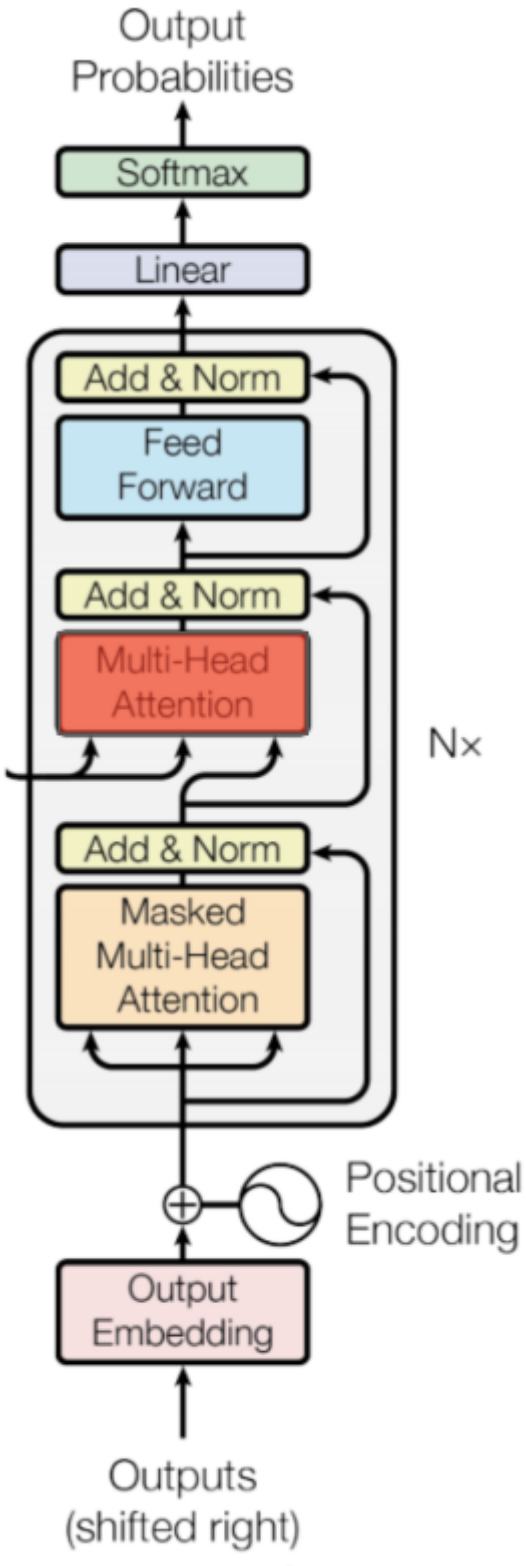
$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} <SS> & Hola & , & como & estás & ? \end{array} \\ \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 & 0 \\ 1.026 * 10{-}27 & 1.0 & 0 & 0 & 0 & 0 \\ 0.99 & 5.73 * 10{-}15 & 8.013 * 10{-}12 & 0 & 0 & 0 \\ 1.56 * 10{-}25 & 7.95 * 10{-}28 & 2.29 * 10{-}34 & 0 & 0 & \\ 6.65 * 10{-}24 & 1.0 & 6.27 * 10{-}18 & 1.78 * 10{-}30 & 3.75 * 10{-}36 & 0 \\ 4.72 * 10{-}33 & 1.32 * 10{-}08 & 2.52 * 10{-}30 & 0.92 & 0.07 & 5 * 10{-}4 \end{pmatrix} \end{array}$$

The relative attention of those tokens that we were trying to ignore has indeed gone to zero.

When multiplying this matrix with *V_i* the only elements that will be accounted for to predict the next word are the ones into its right, i.e. the ones that the model will have access to during test time.

Observe that this time the output of the modified **Multi-Head Attention** layer will be a matrix of dimensions (*target_length*) x (*emb_dim*) because the sequence from which it has been calculated has a sequence length of *target_length*.

Multi-Head Attention—Encoder output and target



Observe that in this case, we are using different inputs for that layer. More specifically, instead of deriving *Q_i*, *K_i* and *V_i* from *X* as we have been doing in previous **Multi-Head Attention** layers, this layer will use both the Encoder's final output *E* (final result of all encoder

blocks) and the decoder's previous layer output *D* (the masked **Multi-Head Attention** after going through the **Dropout, Add & Norm** layer).

Let's first set the ground of the shape of those inputs and what they represent.

1. *E*, the encoded input sequence, is a matrix of dimensions (*input_length*) x (*emb_dim*) which has encoded, by going through 6 encoder blocks, the relationships between the input tokens.
2. *D*, the output from the masked **Multi-Head Attention** after going through the Add & Norm, is a matrix of dimensions (*target_length*) x (*emb_dim*).

Let's now dive into what to do with those matrices. We will use weighted matrices with the same dimensions as before:

$$W_i^K \text{ with dimensons } d_{model} \text{x} d_k$$
$$W_i^Q \text{ with dimensons } d_{model} \text{x} d_k$$
$$W_i^V \text{ with dimensons } d_{model} \text{x} d_v$$

But this time the projection generating *Q_i* will be done using *D* (target information), while the ones generating *K* and *V* will be created using *E* (input information).

$$DW_i^Q = Q_i \text{ with dimensons } target\_length \text{ x } d_k$$
$$EW_i^K = K_i \text{ with dimensons } input\_length \text{ x } d_k$$
$$EW_i^V = V_i \text{ with dimensons } input\_length \text{ x } d_v$$

for every head i=1,…, h.

The matrix W_0 used after the concatenation of the heads will have dimensions (*d_v*h*) x (*emb_dim*) just as the one used in the encoder block.

Apart from that, the **Multi-Head Attention** block is exactly the same as the one in the encoder.

Observe that in this case, the matrix resulting from:

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

is describing relevant relationships between "encoded input tokens" and "encoded target tokens". Moreover notice that this matrix will have dimensions (*target_length*)x(*input_length*).

Following up on our example:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estas \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 0.9 & 0.0 & 0.1 & 0.0 & 0.0 & 0.0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \end{array}$$

As we can see, every projected row-token in the target attends to all the positions in the (encoded) input sequence. This matrix encodes the relationship between the input sequence and the target sequence.

Repeating the notation we used in the **Multi-Head Attention** of the Encoder the multiplication with *V_i* results in:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estas \\ ? \end{array} \begin{array}{c} \begin{array}{cccccc} Hello & , & how & are & you & ? \end{array} \\ \begin{pmatrix} 0.9 & 0.0 & 0.1 & 0.0 & 0.0 & 0.0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \end{array} \begin{array}{c} d_v \\ \begin{pmatrix} v_{Hello} \\ v_, \\ v_{how} \\ v_{are} \\ v_{you} \\ v_? \end{pmatrix} \end{array} =$$

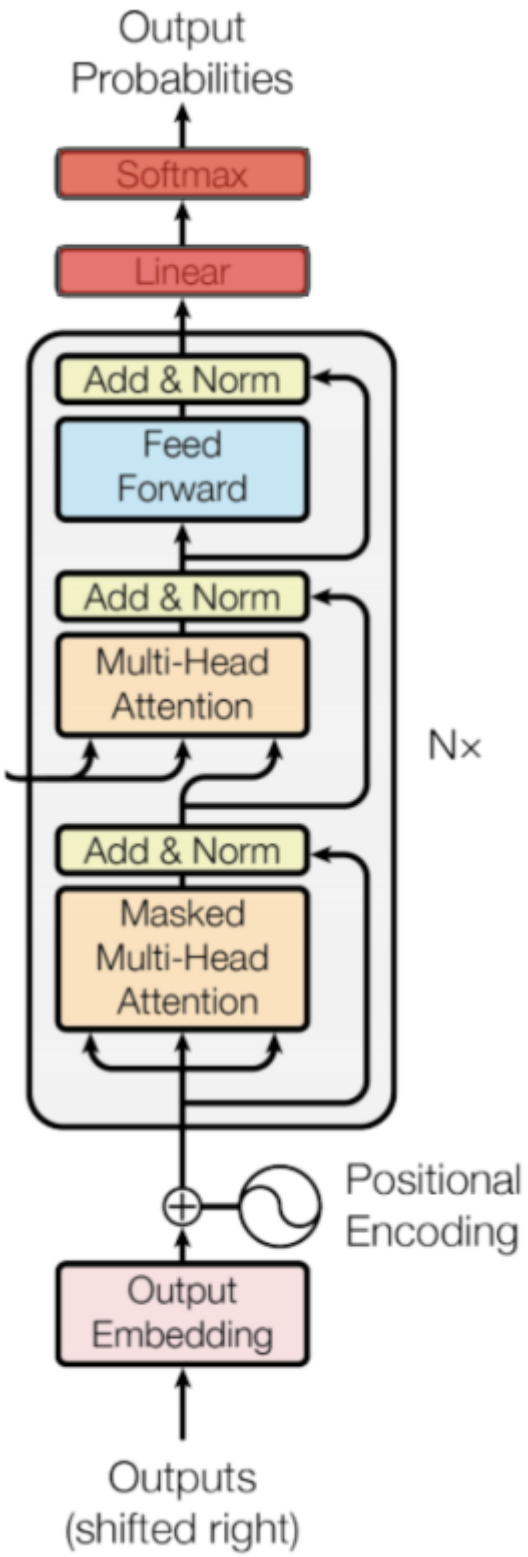
$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} <------------d_v------------> \\ \begin{pmatrix} 0.9v_{Hello} + 0.0v_, + 0.1v_{how} + 0.0v_{are} + 0.0v_{you} + 0.0v_? \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \end{pmatrix} \end{array}$$

As we can see, every token in the target sequence is represented in every head as a combination of encoded input tokens. Moreover, that will happen for multiple heads and just as before, that is going to allow each token of the target sequence to be represented by multiple relationships with the tokens in the input sequence.

As in the encoder block, once the concatenation has been carried out, we will take the product of that with W_0.

$$Concat(V_1, V_2, ..., V_h)W_0$$

Linear and softmax



This is the final step before being able to get the predicted token for every position in the target sequence. If you are familiar with Language Models, this is identical to their last layers.

The output from the last Add & Norm layer of the last Decoder block is a matrix X of dimensions (*target_length*)x(*emb_dim*).

The idea of the linear layer is for every row in *x* of *X* to compute:

$$xW_1$$

where *W_1* is a matrix of learned weights of dimensions (*emb_dim*) x (*vocab_size*). Therefore the result, for a specific row will be a vector of length *vocab_size*.

Finally, a softmax is applied to this vector resulting in a vector describing the probability of the next token. Therefore, taking the position corresponding to the maximum probability returns the most likely next word according to the model.

In a matrix form this looks like:

$$XW_1$$

And applying a Softmax in each resulting row.

> ***Note**: In training time the prediction of all rows matter. Given that at prediction time we are doing an iterative process we are just going to care about the prediction of the next word of the last token in the target/output sequence.*

> ***Note**: The architecture shares the weights between the two embeddings and this last linear layer.*

Observation

As we have seen, the positional encoding is not learned and is adaptable to different sequences lengths. Moreover, all the matrix operations are done such that the result of each layer has one of the dimensions the number of tokens in the input or output sequence. This

aspect makes it adaptable to data with different lengths which match the idea of having a model that finds the relations between the words independent of the amount of the length of the sentence.

References

Attention Is All You Need; Vaswani et al., 2017.

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding; Devlin et al., 2018.

The Annotated Transformer; Alexander Rush, Vincent Nguyen and Guillaume Klein.

Universal Language Model Fine-tuning for Text Classification; Howard et al.

Improving Language Understanding by Generative Pre-Training ; Radford et al., 2018