Dissecting BERT Appendix: The Decoder

Miguel Romero CalvoFollow

Nov 27, 2018 · 11 min read

Unleash the force of self-attention

This is the appendinx of Understanding BERT written by Miguel Romero and Francisco Ingham. Each article was written jointly by both authors. If you did not already, please refer to Part 1 to understand the Encoder architecture in depth since this post assumes its prior understanding.

Many thanks to Yannet Interian for her revision and feedback.

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.
>
> **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.

In this blog post, we are going to examine the Decoder in depth; the part of the Transformer architecture that are not used in BERT. We will reference the Encoder to explain the full **Transformer**'s architecture.

> **Note:** If you are only interested in understanding how BERT works, the parts of the **Transformer** described in this blog post are not relevant.

This article is organized as follows:

1. The Problem the **Transformer** aims to solve.
2. The Information Flow.
3. The Decoder.

Problem

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

- Be able to capture the relationships 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 Encoder.

Once we have this information in the output of the Encoder we want to combine it 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 the Decoder in detail in *Level 3-b: Layers*.

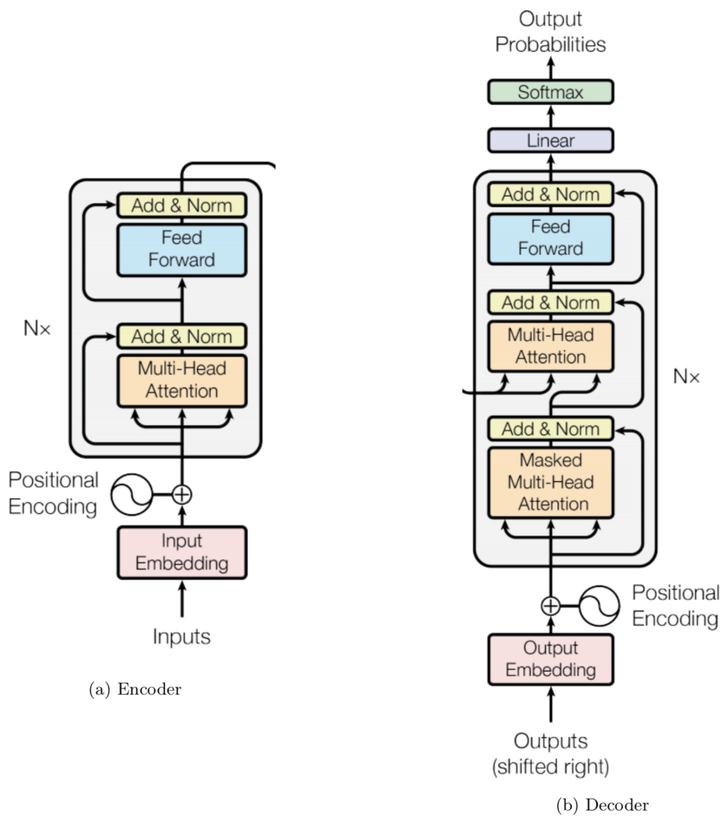
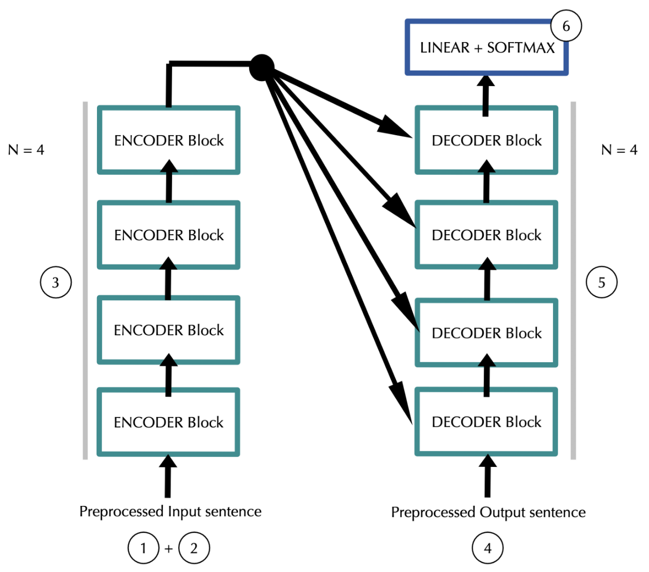


Figure 1: **Transformer** architecture divided into Encoder and Decoder

Information Flow

The data flow through the architecture is as follows:

1. The model represents each token as a vector of dimension *emb_dim*. 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).*



The dimensions of the input and the output of the decoder blocks are the same. Hence, it makes sense to use the output of one decoder block as the input of the next decoder block.

Note: In the Attention Is All You Need experiments, N was chosen to be 6.

> Note: The blocks do not share weights

Inputs

Remember that the described algorithm is processing both the input sentence and the target sentence to train the network. The input sentence will be encoded as described in The Encoder's architecture. In this section, we discuss how given a target sentence (e.g. "Hola, como estás ?") we obtain a matrix representing the target sentence for the decoder blocks.

The process is exactly the same. It is also composed of two general steps:

1. Token embeddings
2. Encoding of the positions.

The main difference is that the target sentence is shifted. That is, before padding, the target sequence will be as follows:

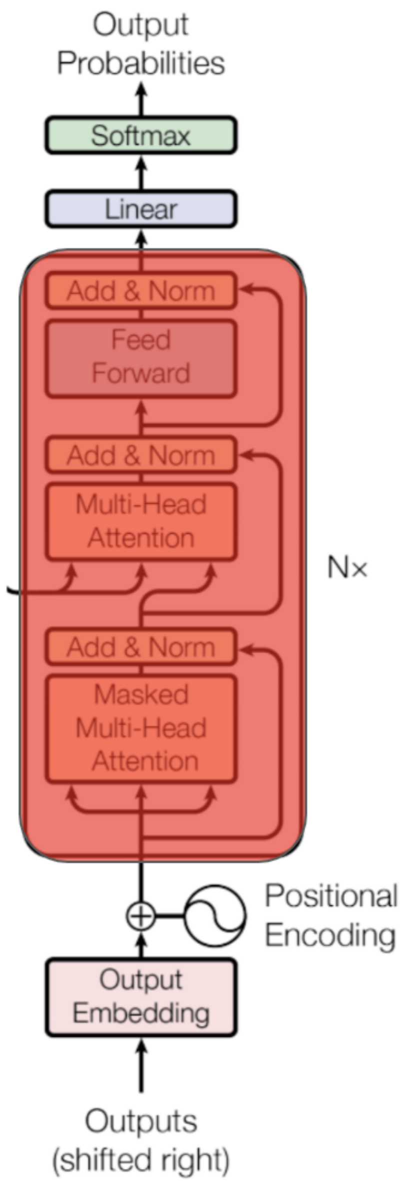
> ["Hola", ", ", "como", "estás", "?"]→["", "Hola", ", ", "como", "estás", "?"]

The rest of the process to vectorize the target sequence will be exactly as the one described for input sentences in The Encoder's architecture.

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



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 to prevent the model of seeing 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:

> [”,’Hola’, ‘,’, ‘ como’, ‘estas’, ‘?’]

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

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

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 on 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 Figure 13.

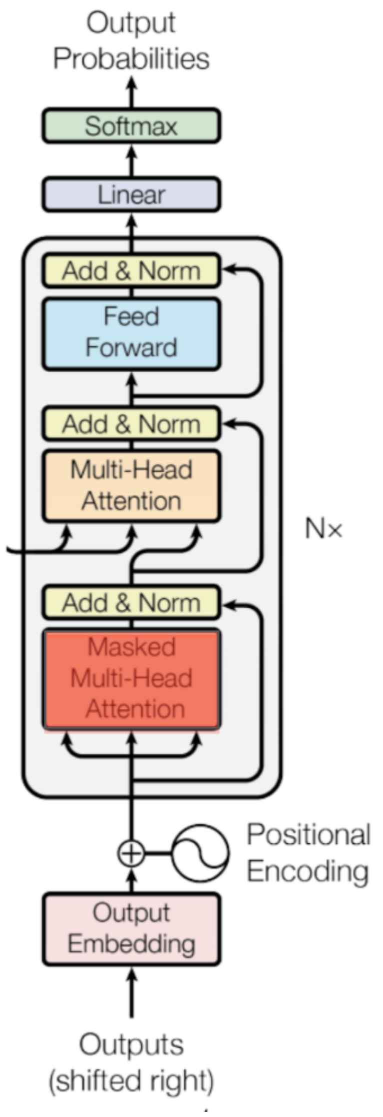
$$\begin{array}{c|ccccc}
 & < & - & d_{emb\_dim} & - & > \\
\hline
<SS> & -3.521 & 23.625 & \cdots & -3.041 & 21.150 \\
Hola & 28.256 & -27.21 & \cdots & -30.80 & 44.232 \\
, & -25.34 & -39.38 & \cdots & 44.016 & 18.240 \\
como & 22.126 & 14.527 & \cdots & -18.22 & 48.169 \\
estás & 49.093 & -48.61 &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

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.

At 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.

Masked Multi-Head Attention



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:

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

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:

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

That's it! The rest of the process is identical 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 to predicting *"estás"* (the one with input *"como")*, ignore *"estás"* and "*?*". Our Softmax output would look like this:

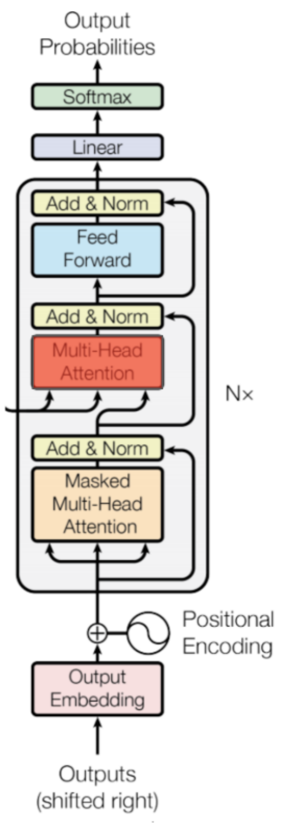
$$\begin{array}{c|cccccc} & <SS> & Hola & , & como & estás & ? \\ \hline <SS> & 1.0 & 0 & 0 & 0 & 0 & 0 \\ Hola & 1.026*10-27 & 1.0 & 0 & 0 & 0 & 0 \\ , & 0.99 & 5.73*10-15 & 8.013*10-12 & 0 & 0 & 0 \\ como & 1.56*10-25 & 7.95*10-28 & 2.29*10-34 & 1.12*10-39 & 0 & 0 \\ estás & 6.65*10-24 & 1.0 & 6.27*10-18 & 1.78*10-30 & 3.75*10-36 & 0 \\ ? & 4.72*10-33 & 1.32*10-08 & 2.52*10-30 & 0.92 & 0.07 & 5*10-4 \end{array}$$

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 clarify 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:

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

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).

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

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 like 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:

$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estas \\ ? \end{array} \begin{array}{cccccc} Hello & , & how & are & you & ? \\ \end{array} \hspace{-0pt} \begin{array}{c|cccccc} & Hello & , & how & are & you & ? \\ \hline <SS> & 0.9 & 0.0 & 0.1 & 0.0 & 0.0 & 0.0 \\ Hola & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ , & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ como & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ estas & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ ? & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{array}$$

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:

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

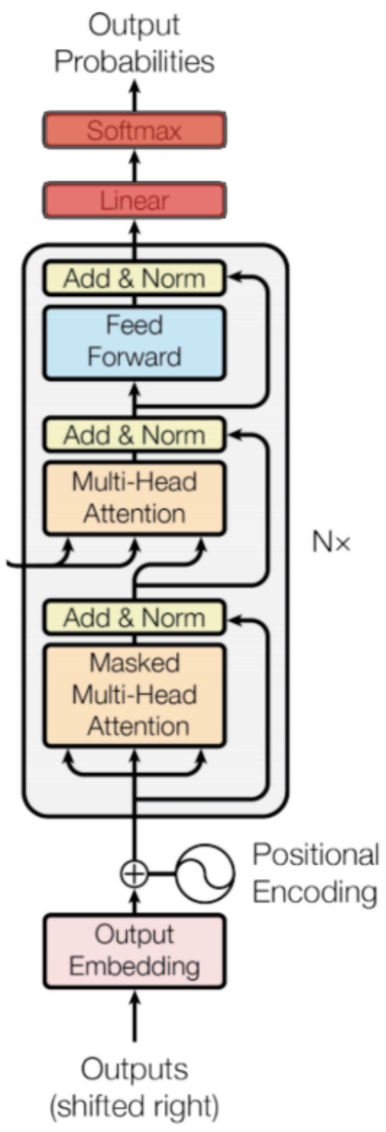
$$\begin{array}{c} \\ <SS> \\ Hola \\ , \\ como \\ estás \\ ? \end{array} \begin{array}{c} < - - - - - - - - - - - - - d_v - - - - - - - - - - - - - > \\ \begin{pmatrix} 0.9v_{Hello} + 0.0v_{,} + 0.1v_{how} + 0.0v_{are} + 0.0v_{you} + 0.0v_{?} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots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

As we can see, every token in the target sequence is represented in every head as a combination of encoded input tokens. Moreover, this 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.

Just like 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, for a specific row the result 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 matrix form this looks like:

$$XW_1$$

And applying a Softmax in each resulting row.

If you have any feedback please let us know in the comment section!

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., 2018.

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