Beyond Spell-checking: Word-checking

An attention based Transformer approach

Abdurrahman Shahid

A report presented for the Study and Research Work
(Travaux d'Etude et de Recherche - TER)

Université de Lille

FACULTÉ DES SCIENCES ET TECHNOLOGIES
Département Mathématiques

2021

Abstract

Contents

CHAPTER 1

Introduction

This report is intended to present and discuss the work accomplished during the TER. In this chapter we outline the key ideas of this TER and introduce some of the concepts needed to understand the reminder of this report.

1.1 The general framework

Let's begin by presenting the issue we will try to solve throughout this report. As we grow up, we are able to make less and less mistakes when writing a sentence or a long text. However, some words are more difficult to spell correctly than others. Especially, words called *homophones* can be quite challenging. What are homophones ? **Homophones** are words that are pronounced the same, but they have different writings and different meanings. For instance, in the French language - the language used for all of the studies in this TER - the words "leur" and "leurs" are homophones. To spell properly these homophones, we must use information such as the context in which they occur. To make matters worse, when humans make mistakes, they are very confident that they have chosen the right word (see **??**). Then we can wonder, how can we avoid these pitfalls ?

We can certainly treat such a question by leveraging knowledge from many different fields of study. For instance, we can use knowledge and concepts from linguistics to help us address the issue of choosing the right homophone. We can further use brain imaging techniques to spot any differences between the condition where we choose the right form and the condition we don't. The latter methods could be, broadly speaking, categorized as the cognitive sciences approach. While cognitive sciences are arguably well suited for this task, we won't take this approach. Instead, we will take a more computer science oriented approach and tackle the issue of homophones by using **deep learning** (commonly abbreviated as **DL**) and the latest breakthrough in **natural language processing** (commonly abbreviated as **NLP**) known as the *Transformer* [5]. The Transformer has democratized and brought to a new level the use of *attention* mechanisms in NLP and sequential problems in general. More precisely, we wil try to answer the following question: **can we can implement an effective attention based Transformer model for homophone correction ?**

We discuss what we mean by the term *effective*, and the terms *homophone correction* in chapter 3. Throughout this chapter, we will have a glimpse of what *attention* and *Tranformer model* mean, but their in-depth explanation is left for the next chapter.

1.2 DL and NLP prerequisites

In this section we will review basic concepts of DL and NLP needed to understand the material present in chapter 2 and chapter 3. Throughout this report, we won't get into too much details when it comes to mathematical theory or more technical methods usually employed in DL. Indeed, this is not the objective here and is beyond the scope of this report, in addition to the fact that many great resources on these topics already exist (see [4], and [2]). However, we will always try to give an intuitive explanation, enough to understand what the different components are used for. Also, additional explanations are given in the appendices. Most of the pages are dedicated to explain the in-depth functioning of the Transformer model and its building blocks, along with the experiment that have been done.

1.2.1 Deep Learning prerequisites

In deep learning we use terms like loss function, gradient, optimizer, true label, parameters, layers, units, activation function, etc. For an exhaustive survey see [4], [3] and [1]. We will mainly focus on the components used to implement the Transformer.

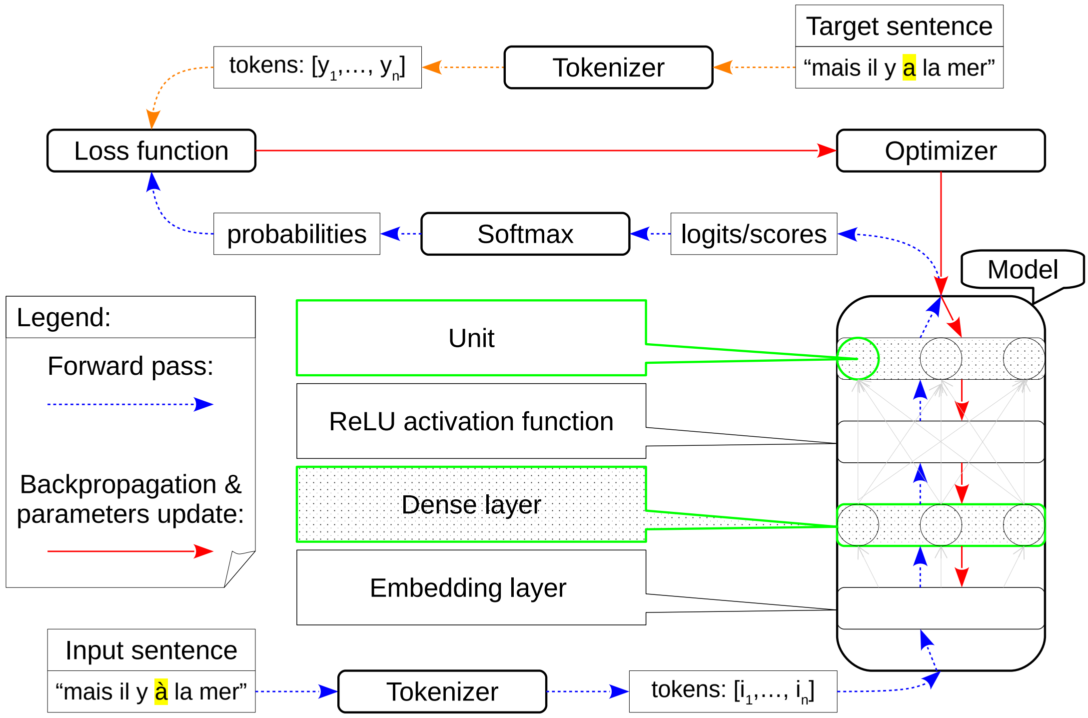

Figure 1.1: How different DL components can be related to each other during training. The model shown here is not the Transformer, and serves only as an illustration.

In the example in figure 1.1, we give the tokenized input (see 1.2.2 for more details) to the model, and then compare what the model outputs with the correct sentence i.e. target sentence. Target sentence is employed here to emphasize that, for each input sentence, we have the corresponding correct sentence (itself if it's already correct or the corresponding corrected sentence). Thus, the model is trained in a supervised setting. We want the model to correct the errors present - if any - in the input sentence, and output the corrected sentence. For instance in figure 1.1, the input contains an error whereas target sentence doesn't (see the highlighted terms "à" and "a"). That being said, let's review what each component in figure 1.1 does. First, the list of tokens (each token is a number) of length l goes through the **Embedding layer**, where each token is transformed into a vector of a given fixed dimension d called *embedding dimension*. Indeed, for every token in the vocabulary (see 1.2.2), the embedding layer has an associated vector of dimension d. For instance if $l = 11$ (there are 11 tokens) and $d = 64$, then the list of tokens is transformed into a matrix X having 11 rows and 64 columns ($X \in \mathbb{R}^{11\times 64}$). This newly created matrix is then fed to a **Dense layer**. Moreover, a dense layer consists of **units** and those *units* determine the shape of the weight matrix W representing the layer. What happens at the dense layer boils down to a matrix multiplication. Indeed, if we omit the bias term, and the number of units is equal to 16, then the following matrix multiplication occurs: $X \times W$ with $X \in \mathbb{R}^{11\times 64}$ and $W \in \mathbb{R}^{64\times 16}$ resulting in a matrix $X_{\text{new}} \in \mathbb{R}^{11\times 16}$. Oftentimes an activation function is applied to a layer's output. The activation function used in figure 1.1 - and also by the Transformer - is called **ReLU** activation function. For every real number x, $\text{ReLU}(x) = x$ if $x \geq 0$ and $\text{ReLU}(x) = 0$ if $x < 0$. ReLU is applied *element-wise* to the previous matrix $X_{\text{new}} \in \mathbb{R}^{11\times 16}$. Activation functions serve to inject non-linearity into our models. Finally, in figure 1.1, the output of the ReLU goes through a second Densely connected layer. If this second layer has $|V|$ units, then we get a new matrix $S \in \mathbb{R}^{11\times|V|}$ ($S = X_{\text{new}} \times W_2$ with $X_{\text{new}} \in \mathbb{R}^{11\times 16}$ and $W_2 \in \mathbb{R}^{16\times|V|}$). As we will see in section 1.2.2, $|V|$ denotes the size of the vocabulary, i.e. the total number of tokens that make up the vocabulary being used by the Model. Notice that this time we didn't use an activation function after the dense layer. Thus, elements of S can be any real number, we call these numbers *logits*. Next, we use the **Softmax** function to get *probabilities*. We apply softmax along the rows of S and obtain the probability matrix P:

$$P = softmax_j(S) \quad \text{i.e.} \quad P_{ij} = \frac{\exp(S_{ij})}{\sum_{j=1}^{|V|}\exp(S_{ij})} \tag{1.1}$$

P_{ij} gives for the i^{th} output token, the probability that this token is the j^{th} token of the vocabulary. In our example (figure 1.1), $P \in \mathbb{R}^{11\times|V|}$, hence the model outputs probabilities for 11 tokens (corresponding to 11 rows in P). Intuitively, we can consider that the i^{th} output of the model, is the j^{th} token of the vocabulary, such that P_{ij} is the maximum i.e. $j = argmax_j(P_{i,.})$, for those who are familiar with this notation.

Furthermore, as we have the correct sentence i.e. target sentence, we also have the correct tokens i.e. target tokens, we tokeize the target sentence to get them. The same tokenizer is used to tokenize both the input sentence and the target sentence. Consequently, we can compare how well the model is doing for the task of correcting incorrect sentences. We compare the model's output - the probability matrix P - with the target tokens from the target sentence. This is accomplished by the **Loss function**, it takes as input both the list of target tokens and the probability matrix P, and it calculates the *loss*, which is a measure of how different the model's output is from what is expected i.e. target sentence. We want this loss to be as low as possible. If we denote the loss function by L, and all the trainable parameters by θ, then parameters update is given by:

$$\theta_{\text{new}} = \theta_{\text{old}} - \epsilon \times \nabla_\theta L \quad \text{where} \quad \nabla_\theta L, \text{ is the gradient of } L \text{ w.r.t } \theta \tag{1.2}$$

We notice that, $\nabla_\theta L$ - obtained via the *backpropagation algorithm* - indicates the direction in which θ is moved, and ϵ specifies by how much it is moved. Additional update options can be configured and the **Optimizer** handles their implementation. This ends the quick review of the basic DL concepts needed to get the big picture of the problem. Next, we review some concepts related to NLP and the loss function in detail.

1.2.2 Natural Language Processing prerequisites

In order to train and evaluate a model, we have to first define a vocabulary. This vocabulary is linked to the way we *tokenize* textual data. Indeed, when we encode textual data using a **Tokeinzer**, the resulting token representation depends on the tokens the tokenizer is able to handle. For instance, a character-level tokenizer encodes each character separately, while a world-level tokenizer encodes each word separately. However, we will use for the Transformer a *subword-level* tokenizer. The subword tokenizer is first trained on a text corpus, then *the most frequent chunks* [see 2, for more details] of text are each associated with a number between 1 and $|V|$. Here, $|V|$ indicates the maximum number the tokenizer can emit. Moreover, at the beginning (respectively end) of a sentence, we add the <*SOS*> token (respectively the <*EOS*> token), see figure 1.2 for an example.

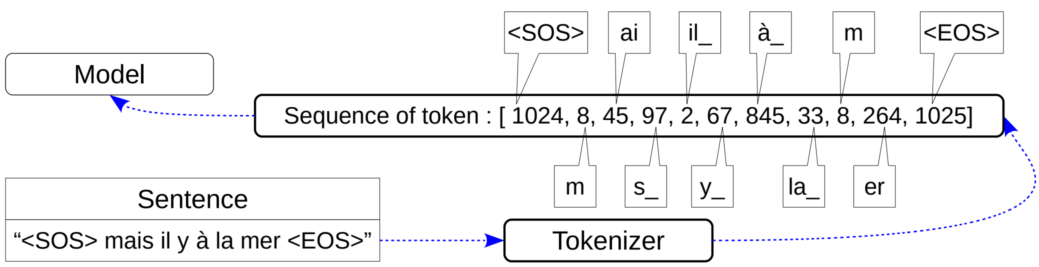

Figure 1.2: How Subword Tokenizer encodes sentences.

Finally, we describe the loss function L used in figure 1.1, and also used by the Transformer. More precisely, L is the *Sparse Categorical Cross-Entropy Loss*. We saw in section 1.2.1, that this loss function takes as input the list of target tokens, and the probability matrix P from the model. Let's denote by y the list of target tokens, and $y_i \in \{1, ..., |V|\}$ the i^{th} token in the list. Moreover, l is both the length of y and the number of rows in P (in figure 1.1, $l = 11$). L is given by formula 1.3, and measures the loss for a pair input sentence/target sentence:

$$L(y, P) = -\sum_{i=1}^{l} \log(P_{i,y_i}) \tag{1.3}$$

We notice that, both the length of y and the number of rows in P must be equal (it's the same number l), we will see in the next section how it is ensured for the Transformer model. Additionally, if P_{i,y_i} tends to 1, then $\log(P_{i,y_i})$ tends to 0, and if P_{i,y_i} tends to 0, then $\log(P_{i,y_i})$ tends to $-\infty$ (then L tends to $+\infty$). Consequently, minimizing L forces P_{i,y_i} to be close to 1, i.e. the model gives a high probability for the i^{th} output token to be the i^{th} token in the target token list. Thus, the model *learns* to output what we expect it to output (the target tokens) given a certain input.

1.3 High-level picture of the problem setup

In this section, we look at the elements surrounding the training of the Tranfsormer. To begin with, the set-up used to train the Transformer is similar to the setting in figure 1.1, with changes only occurring at the model level. Indeed, as shown in figure 1.3, the Transformer is composed of two main blocks - an **Encoder** block and a **Decoder** block - along with a dense i.e. linear layer. Within these blocks, is implemented the *attention* mechanism. Moreover, *the Transformer takes as input two sequences*, one entering through the encoder and the other through the decoder. The encoder takes as input the list of tokens from the input sentence. However, the decoder takes as input the list of tokens from a truncated version of the target sentence. Indeed, from the entire target sentence, we extract two truncated sentences, one without the "<EOS>" token, the other without the "<SOS>" token. Both have the same length after tokenization, it's mandatory to use the loss function 1.3. For instance, in figure 1.3, from the target sentence "<SOS> mais il y a la mer <EOS>", we extract "mais il y a la mer <EOS>" and "<SOS> mais il y a la mer". The one without the "<EOS>" token is given as input to the decoder. The other one, without the "<SOS>" token is considered as the target sentence, and the resulting list of tokens (after tokenization) is given to the loss function.

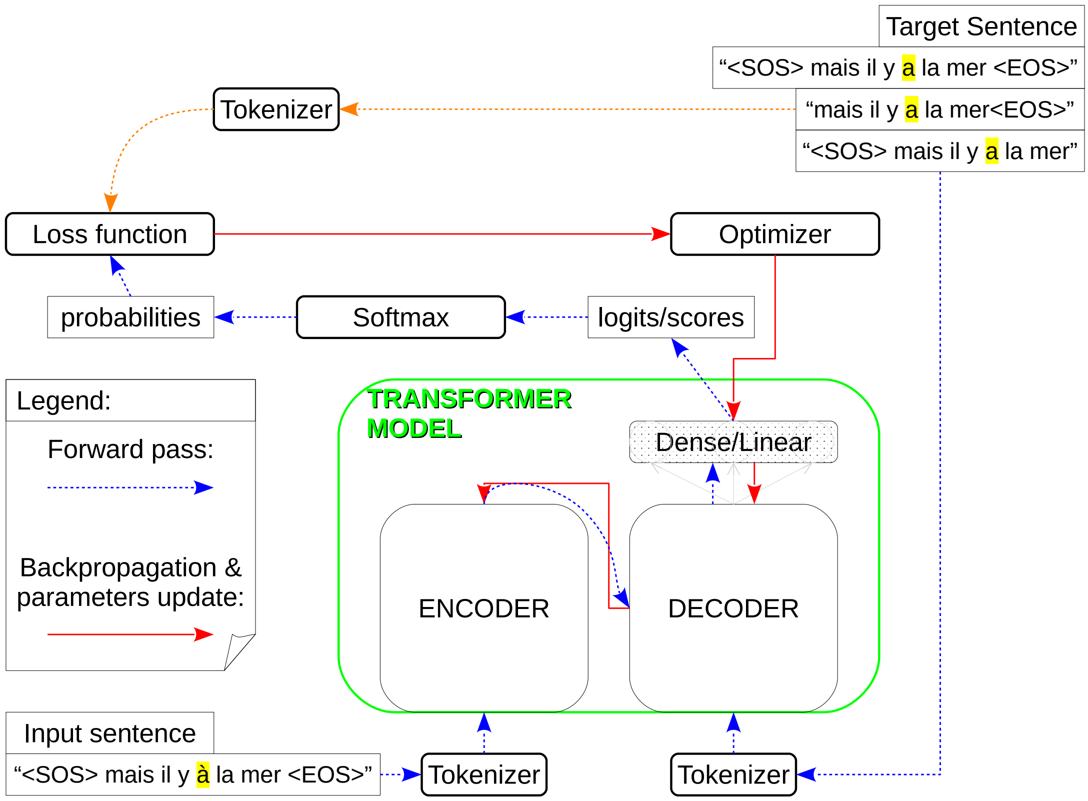

Figure 1.3: How the Transformer model fits into the big picture, with its Encoder and Decoder blocks along with a final linear layer. This figure illustrates the setting during the training phase.

We can also observe that the sequence given as input to the decoder is the sequence used as the target, but shifted by one token to the right. Since, the 1st token/word in the target sequence (for instace ”mais” in figure 1.3) is the 2nd token/word in the sequence given as input to the decoder, and this observation applies to all following tokens/words in the target sequence. This is indicated in figure 2.1 in chapter 2 by ”*Outputs (shifted right)*”.

In the next chapter, we unpack the encoder and the decoder to see the building blocks that make them up. We also present how the Transformer is used during the evaluation/test phase.

CHAPTER 2

The Transformer Explained

In this chapter, we will see how the inputs are transformed within the Transformer model, and how the *attention* mechanism works. As we have seen in figure 1.3, there isn't one input but two. Indeed, the model takes as input the input sentence (at the encoder level), for instance "<SOS> mais il y à la mer <EOS>", and what we call *the decoder input*, for instance "<SOS> mais il y a la mer". At training time, from these two inputs, the model is trained to output "mais il y a la mer <EOS>", which is the target i.e. true label. We will illustrate the inner working of the Transformer by following what happens to our sentence from the previous chapter, from the moment it enters the model at the encoder level and the moment the model outputs probabilities for each output token. See section 1.3 for a refresher and the explanation of the meaning of "*Outputs (shifted right)*" at the decoder level in figure 2.1. Throughout this chapter we will describe in detail each component shown in figure 2.1, therefore you should refer to it as many times as necessary.

2.1 The Encoder Input

We first tokenize the sentence "<SOS> mais il y à la mer <EOS>", which gives us a list (or a vector) of tokens. Moreover, suppose that after tokenization the sentence is $l = 11$ tokens long as in figure 1.2. Recall that tokens are just numbers, each one representing a specific subword (see section 1.2.2 for more details). This list of tokens is represented by

"**Inputs**" and this is what goes into the encoder.

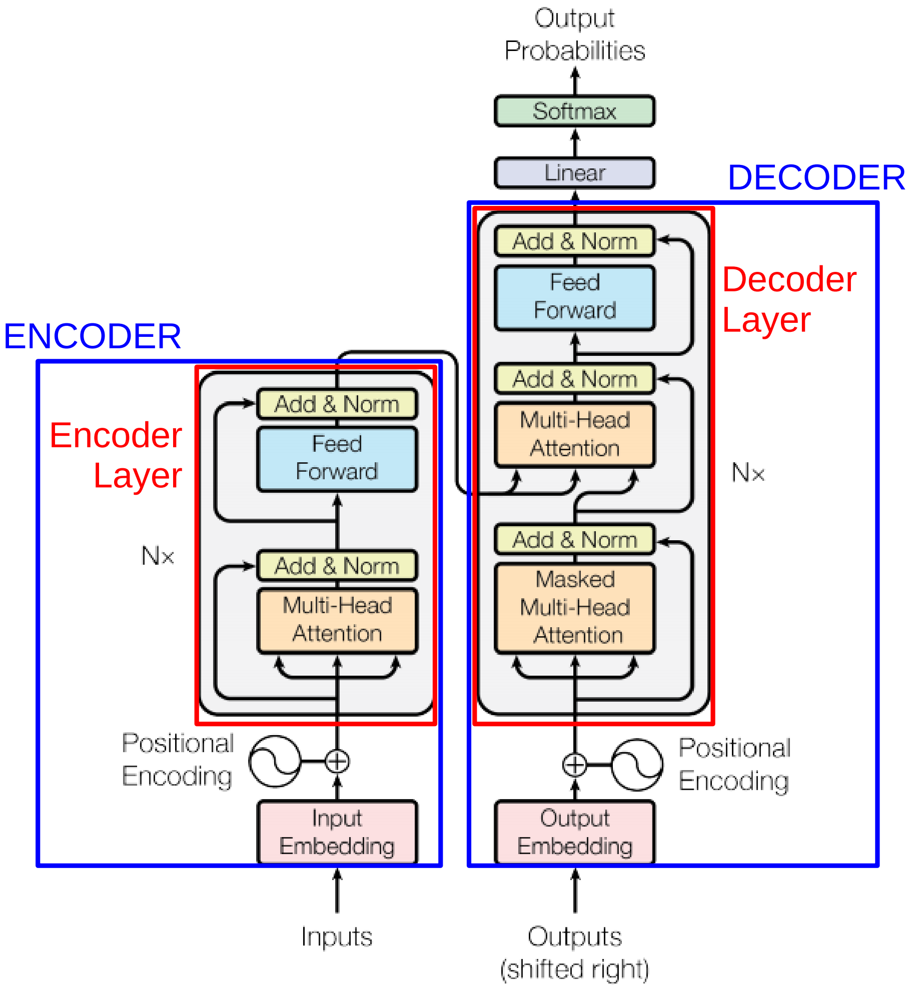

Figure 2.1: The Transformer - model architecture adapted from the original paper [5].

2.1.1 Input Embedding and Positional Encoding

The list of tokens goes through the *Embedding layer* (see 1.2.1, for the details) and we get a matrix $X \in \mathbb{R}^{l \times d_{model}}$. Here, d_{model} represents the dimension of the vectors representing each input token. If we set $d_{model} = 64$, then we get an "**Input Embedding**" matrix $X \in \mathbb{R}^{11 \times 64}$. Though we have encoded the information about words identity until now, we didn't encode the information about words position in the sentence. To inject the information about the position, we add X and a *sinusoidal* matrix R, with R having the same dimension as X. If $R \in \mathbb{R}^{l \times d_{model}}$, then each element of R is given by:

$$\forall i \in \{1, ..., l\}, \forall j \in \{1, ..., d_{model}\}, R_{ij} = \begin{cases} \sin(i/10000^{2j/d_{model}}) & \text{if } j \text{ is an even integer} \\ \cos(i/10000^{2j/d_{model}}) & \text{if } j \text{ is an odd integer} \end{cases}$$

R represents the "**Positional Encoding**". Let's call the resulting matrix $X_E \in \mathbb{R}^{l \times d_{model}}$, $X_E \in \mathbb{R}^{11 \times 64}$ in our example, encoding both the identity of the words and their position within the sentence. Then X_E enters the *Encoder Layer*. All the computations occurring within the latter can be repeated N times as indicated in figure 2.1.

2.1.2 Multi-Head Attention

First of all, **Multi-Head Attention** is made of multiple *Single-Head Attention*. Lets denote by *Attention* the calculations performed in a Single-Head Attention module and by h the number of Single-Head Attention composing the Multi-Head Attention module. The Attention module (and hence the Single-Head Attention) takes as input three matrices denoted by $Q \in \mathbb{R}^{l_q \times d_q}$, $K \in \mathbb{R}^{l_k \times d_k}$ and $V \in \mathbb{R}^{l_v \times d_v}$. We usually refer to these matrices by "Query" matrix, "Key" matrix and "Value" matrix respectively. In the encoder we obtain these matrices as follows:

$$Q = X_E W_Q \ , \quad K = X_E W_K \ , \quad V = X_E W_V \ , \quad \text{with } X_E \in \mathbb{R}^{l \times d_{model}} \tag{2.1}$$

with $W_Q \in \mathbb{R}^{d_{model} \times d_q}$, $W_K \in \mathbb{R}^{d_{model} \times d_k}$ and $W_V \in \mathbb{R}^{d_{model} \times d_v}$. Moreover, we have $depth = d_q = d_k = d_v = d_{model}/h$. If there is $h = 4$ heads (as in our implementation in chapter 3) then we have $depth = d_q = d_k = d_v = 64/4 = 16$. Single-Head Attention is then given by:

$$\text{Attention}(Q, K, V) = softmax_j(\frac{QK^T}{\sqrt{d_k}})V = AV, \quad \text{with } A = softmax_j(\frac{QK^T}{\sqrt{d_k}}) \tag{2.2}$$

More precisely, matrix $A \in \mathbb{R}^{l_q \times l_k}$ contains the **attention** weights. By using the softmax function along the columns we get, for each row of A, weights between 0 and 1. Afterwards, the matrix multiplication AV calculates a linear combination of the rows of $V \in \mathbb{R}^{l_v \times d_v}$ weighted by the attention weights found in A. Thus, if we denote by AV_i the i^{th} row in AV and V_i the i^{th} row in V then we have:

$$AV_i = \sum_{j=1}^{l_q} A_{ij} V_i \tag{2.3}$$

In other words, as each row in V represents a token in the token list, AV contains a new representation for each token that takes into account the information about the other tokens in the sentence. Therefore, the representation of a token or word is not fixed, it changes depending on the context (surrounding words) where it occurs. Attention weights dictate how much each of the surrounding words/tokens should contribute for the new representation of a given word/token. Notice also that, $l_q = l_k = l_v = l = 11$ here. To summarize each Single-Head outputs a matrix that contains a new representation for each token and the matrices W_Q, W_K, and W_V are learned by the model during training.

As we have h heads, we also have h sets of matrices $\{W_Q^i, W_K^i, W_V^i\}$ for $i \in \{1, ..h\}$ that are learned by the model during training for a Multi-Head Attention layer. Each head outputs a matrix, we then concatenate all these matrices along the columns to get a single matrix that we finally multiply by another matrix $W_O \in \mathbb{R}^{hd_v \times d_{model}}$. At the end of the Multi-Head Attention layer we have a matrix $X_{MHA} \in \mathbb{R}^{l_q \times d_{model}}$. With our numeric example we have $X_{MHA} \in \mathbb{R}^{11 \times 64}$.

2.1.3 Add & Norm

Each **Add & Norm** layer, takes as input, both the input and the output of the previous layer. It adds up both and then normalizes the result. The normalization used is *Layer Normalization*, which standardizes the output along the columns for each row of the matrix obtained after the aforementioned addition.

2.1.4 Feed Forward

Feed Forward layer consists of two Dense layers with a ReLU activation in between. See section 1.2.1 for a refresher on Dense layers and ReLU activation function. The first Dense layer outputs a matrix $X_{FF1} \in \mathbb{R}^{l \times dff}$ (we use $dff = 256$ in chapter 3) and the second Dense layer outputs a matrix $X_{FF2} \in \mathbb{R}^{l \times d_{model}}$. The weight matrices associated with both of the Dense layer are trainable weights learned during training. After the Add & Norm layer following the Feed Forward layer in the Encoder we can either repeat once again the Encoder layer (the output will be fed anew as X_E) or we take the output as the Encoder's output. For convenience we will denote the Encoder's output by $H \in \mathbb{R}^{l \times d_{model}}$. We usually refer to it as the Encoder's *Hidden states*. H is then employed by the Decoder's Multi-Head Attention layer (see figure 2.1). More precisely H is used to calculate the key matrix K and value matrix V as we will see in section 2.2.3. Now lets see how the Decoder processes its input.

2.2 The Decoder Input

The Decoder takes as input the target sentence without the <EOS> token. In our example (figure 1.3) it takes as input "<SOS> mais il y a la mer". Suppose that after tokenization, the length of the input tokens list is $m = 10$ (recall that l indicates the length of the Encoder input and $l = 11$ in our example). This list of tokens is represented by "**Outputs (shifted right)**" in figure 2.1.

2.2.1 Output Embedding and Positional Encoding

As with the Encoder input, the vector of tokens is first transformed into a matrix. We then add to it the positional encoding matrix $R \in \mathbb{R}^{m \times d_{model}}$. We denote the resulting matrix by $X_D \in \mathbb{R}^{m \times d_{model}}$. X_D then enters the *Decoder Layer*. As with the Encoder Layer, all the computation occurring inside the Decoder Layer can be repeated N times.

2.2.2 Masked Multi-Head Attention

Everything is the same as the Multi-Head Attention in the Encoder except that we use X_D instead of X_E in equation 2.1 and $l_q = l_k = l_v = m = 10$ here. What is new is the "**Masked**" part. As we give to the Decoder what it has to output but shifted by one token, it can just learn to output what it takes as input but shifted by one token.Thus, it won't learn anything interesting but to shift a sequence. To prevent the model from "cheating", we apply a **mask** to the matrix A containing the attention weights in equation 2.2. This mask zeroes out all the coefficients above the main diagonal i.e. $A_{ij} = 0$, if $j > i$. See**??** for an illustration. Therefore, in order to output the i^{th} token, the model can't look ahead and can only attend to tokens that that comes before it in the Decoder input. For our example, we get a final matrix $X_{MMHA} \in \mathbb{R}^{10 \times 64}$.

2.2.3 Add & Norm, Multi-Head Attention and Feed Forward

The Add & Norm layer is the same as the one in the Encoder. We continue to denote by X_{MMHA} the output of this layer. What is new in the next Multi-Head Attention layer, is that $H \in \mathbb{R}^{l \times d_{model}}$ from the Encoder get involved in the computations along with $X_{MMHA} \in \mathbb{R}^{m \times d_{model}}$. Indeed, H is used to calculate K and V and formula 2.1 becomes:

$$Q = X_{MMHA} W_Q \ , \quad K = H W_K \ , \quad V = H W_V \tag{2.4}$$

with $Q \in \mathbb{R}^{m \times d_q}$, $K \in \mathbb{R}^{l \times d_k}$ and $V \in \mathbb{R}^{l \times d_v}$. As in the Encoder we have $depth = d_q = d_k = d_v = d_{model}/h = 64/4 = 16$ for our example. Everything else (equation 2.2 and 2.3) remain the same. Finally the output of the Multi-Head Attention goes through an Add & Norm layer and a Feed Forward layer. The latter outputs a matrix $D \in \mathbb{R}^{m \times d_{model}}$, see figure 2.1. We can either repeat the Decoder Layer one more time (D will be given as input to the Masked Multi-Head Attention layer) or we can pass D to a final Linear layer.

2.3 From the Decoder Output to Probabilities

Matrix D goes through a final Linear layer which has $|V|$ units. Thus, it outputs a matrix $S \in \mathbb{R}^{m \times |V|}$ containing the *scores*, see figure 2.1. Then S is softmaxed along the rows and we get a matrix $P \in \mathbb{R}^{m \times |V|}$ that contains probabilities, see equation 1.1. For the bigger picture of the problem setup see figure 1.3.

2.4 The Transformer at evaluation time

CHAPTER 3

Homophone correction using The Transformer

APPENDIX A

Mathematics Appendix

APPENDIX B

Machine Translation and Sequence to Sequence Learning

APPENDIX C

Word checker: Materiel selection and Data preprocessing

APPENDIX D

Detailed code implementation of The Transformer

APPENDIX E

Transformer and other Applications

Bibliography

[1] François Chollet. *Deep Learning with Python*. Manning Publications, 2017. ISBN: 9781617294433.

[2] James H. Martin Dan Jurafsky. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Third Edition*. URL: https://web.stanford.edu/~jurafsky/slp3/.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, 2019. ISBN: 9781492032649.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. ISBN: 9780262035613.

[5] Ashish Vaswani et al. "Attention is all you need". In: *arXiv preprint arXiv:1706.03762* (2017).