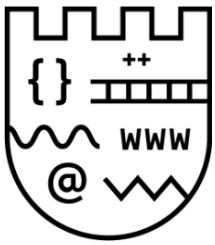
ARISTOTLE UNIVERSITY OF THESSALONIKI
FACULTY OF SCIENCES
SCHOOL OF INFORMATICS

Attention and Transformer architectures

Postgraduate Studies Programme
"Artificial Intelligence"

Subject: Advanced Computer Vision



Palaskos Achilleas

December 2022

This Page is intentionally left blank

Contents

This Page is intentionally left blank

Chapter 1

Word Embeddings

Chapter 1 contents [1], [2] [3], [4], [5], [6], [7], [8], [9], [10]

1.1 Introduction

Natural Language Processing (**NLP**) is the field of science which studies the ways to program a computer so as to be able to process and analyze natural language data. Our natural language is mainly comprised of words, which, in the context ot NLP, are often referred to as tokens. The main problem with words is that, although they may be comprehensible by humans, they are not interpretable by computers. In order for a word to be a valid input for a computer it has first to be transformed into a vector of numbers, because numbers are understandable by computers.

The first notable attempts to deal with this problem were the **One-Hot encoding** and the **TF-IDF** (**T**erm **F**requency–**I**nverse **D**ocument **F**requency) techniques. Both of these approaches belong to the general category of the **Bag-of-Words** representation, which name arises from the fact that words comprise a set (bag) of unrelated entities. In other words, in the Bag-of-Words approach, words are analyzed independently and as consequence grammar and order are disregarded. However, multiplicity is still taken into account.

To briefly understand how these initial techniques work, let's assume that we have a corpus that consists of $|S|$ sentences and the total number of unique words or terms that we accept as our input is $|V|$, where S is the set of all sentences in our

corpus and V is our Vocabulary, i.e. the set of all accepted words. On the one hand, in One-Hot encoding each word is represented by $|S|$-dimensional vectors, where its entry j is 1 if the word appears in sentence j, otherwise 0. As we can see, the One-Hot encoding representation of words does not take into account the number of times a word appears in a particular sentence, but only its existence. Therefore it's a very poor model.

On the other hand, the TF-IDF approach is a bit more complicated since it contains information about how often a word appears in a sentence. In fact, according to this approach each entry j of a word vector depends on the frequency that this particular word appears in sentence j. Therefore, this model captures much richer representations of words and as a consequence leads to better results.

However, both of these initial bag-of-words approaches were accompanied by three very significant problems:

1. both the TF-IDF and the One-Hot encoding matrices are very **sparse**, because the probability that a particular word appears in a sentence is very low, even for the most common words. Regarding these matrices, after numerous experiments, researchers have found, that neural networks are not good at handling these types of input data.

2. the dimensionality of both the TF-IDF and the One-Hot encoding matrices is very high, increasing the **computational cost** significantly. Indeed, if we assume that $|V| \sim 20,000 - 50,000$ and $|S| \sim 5,000$, then both matrices are of average size $|V| \times |S| = 35,000 \times 5,000$. This means that each word is a vector $\mathbf{w} \in \mathbb{R}^{\sim 5,000}$, which is not an ideal input for regular neural networks.

3. in both techniques, the **relationships** between the words are disregarded, because all words are treated as separate entities.

Researchers were trying to find better and richer representations for words. In fact the main efforts were aiming at creating more compact representations that would simultaneously be capable of capturing some major relationships between them. Here is were the idea of **Word Embeddings** becomes meaningful. This name might have come from the fact that these Word Embeddings manage to capture or "embed" the information of high dimensional vectors.

In fact, Word Embeddings is a learned way of representing words, such that those with similar meaning have a similar representation. In particular each word is mapped to a low-dimensional real-valued vector in such a way that those words which **generally** have similar meaning are placed in the same regions in this low dimensional space. The dimension of this new space is usually of the order of hundreds, in opposition to the initial word spaces, in the Bag-of-Word approaches, which were of the order of many thousands or even millions.

In conclusion, Word Embeddings are one of the key ideas in NLP that impressively increased the performance of the NLP models back to these days because on the one hand they brought the concept of context into play and on the other hand the dense representations of the words significantly decreased the computational cost. Because of their multidimensional advantages, they are still used as the initial layers in Deep Neural Networks.

1.2 Matrix Factorization for Recommender Systems

Let's assume that we have N users and D books and $\mathbf{R}$ is the $N \times D$ matrix, where all ratings are stored. In particular, the item r_{ij} indicates the rating that user i gave to book j. One key point to notice about $\mathbf{R}$ is that it is a sparse matrix. In fact, most of its values are missing, because most books have not been rated by any user. However, although those missing values may seem an inconvenience, in practice all recommender systems are based on this seemingly important drawback, because their purpose is to fill in these empty places.

1.2.1 Matrix Factorization

Matrix Factorization is an algorithm that belongs to a broader class of algorithms known as collaborative filtering. The main concept behind Matrix Factorization is that out of all users, there are going to be some, who will be similar to each other. This means that these users gave similar ratings to the same books. That is the basis for recommendations. For instance, if a book has been rated highly by a user i, who is similar to another user j that hasn't read the book yet, then that particular book would probably be a good recommendation for user j.

As we have already mentioned, one of the main problems of matrix $\mathbf{R}$ is its sparsity. Therefore, the first thing we would like to do, is to reduce its dimensions. One very common method for dimensionality reduction is called **SVD** (**S**ingular **V**alue **D**ecomposition). According to it, the ratings matrix can be split as follows:

$$\mathbf{R} = \mathbf{WSV}^T \tag{1.1}$$

where:

- $\mathbf{W} \in \mathbb{R}^{N \times K}$ is an orthogonal matrix.
- $\mathbf{S} \in \mathbb{R}^{K \times K}$ is diagonal matrix.
- $\mathbf{V} \in \mathbb{R}^{K \times D}$ is an orthogonal matrix.

If we set: $\mathbf{U}^T = \mathbf{SV}^T \in \mathbb{R}^{K \times D}$, then (1.1) becomes:

$$\mathbf{R} = \mathbf{WU}^T \tag{1.2}$$

where we usually choose: $K \ll D$ (figure (1.1))

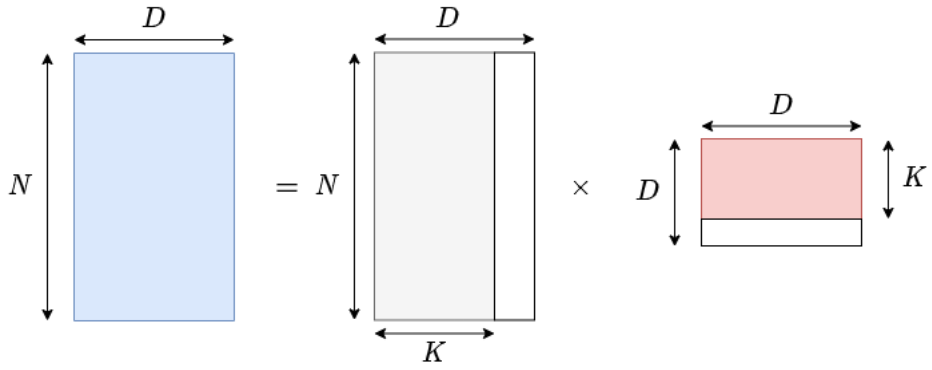


Figure 1.1: Matrix Factorization

In order to see more clearly why this method results in redundancy, let's use an example. Suppose that we have 5 million users and $500,000$ books. If we choose $K = 10$, then:

- the initial size of our ratings matrix is:

$$N \times D = 5 \times 10^6 \times 5 \times 10^5 = 25 \times 10^{11}$$

- whereas after factorization the total size of the two matrices is:

$$(N + D) \times K = 5.5 \times 10^6 \times 10 = 5.5 \times 10^7$$

Therefore we have a decrease of about:

$$\frac{25 \times 10^{11}}{5.5 \times 10^7} \cong 4.5 \times 10^4$$

which is quite large.

1.2.2 Problem Formulation

Now that we have our model described by equation (1.2), the question arises as to how to use it. The main problem comes from the fact that, when we multiply $\mathbf{W}$ by $\mathbf{U}^T$ we get a full $N \times D$ matrix $\hat{\mathbf{R}}$ with no missing values, which correspond to the ratings predictions. In fact, each entry $\hat{r}_{ij}$ is a prediction for the rating that user i would give to book j. But in reality $\mathbf{R}$ can not be decomposed, because most of its values are missing.

However, what with a closer look at the problem, we observe that we do not want to decompose $\mathbf{R}$. Rather our aim is to calculate only the missing values. Therefore, in order to get any single prediction we don't have to multiply the whole $\mathbf{W}$ matrix by the whole $\mathbf{U}$. Instead, we can just take the inner product of $\mathbf{w_i}$ and $\mathbf{v_j}$ for **only** those users i that haven't rated the books indicated by j, as follows:

$$\hat{r}_{ij} = \mathbf{w_i}^T \mathbf{u_j} \tag{1.3}$$

where $\mathbf{w_i}, \mathbf{u_j} \in \mathbb{R}^K$. Intuitively, we could say that the vector $\mathbf{w_i}$ corresponds to some user i and the vector $\mathbf{v_j}$ corresponds to some book j, while each of the K dimensions represents a latent feature, that may be arbitrary or could have some reasonable meaning. Therefore, (1.3) informs us about how similar a user vector is to a book vector.

For example let's suppose that $K = 3$ and that:

- for the user vector these three latent features indicate whether he likes novels, whether he cares about poetry and whether he likes romance respectively.

- for the book vector these three latent features indicate whether the book is a novel, whether it could be considered as poetry and whether it is romantic respectively.

Moreover, let's assume that we have the following vectors:

- $\mathbf{w_i} = [1\ 0\ -1]^T$, which represents a user that likes novels, doesn't care about poetry and dislikes romance.
- $\mathbf{u_j} = [1\ 0\ 0]^T$, which represents a book that is a novel, it is not poetry and has no romantic elements in it.

then: $\mathbf{w_i}^T\mathbf{u_j} = +1$, which means that this particular book i would be a good suggestion for user j. In addition, let's take another example, where:

- $\mathbf{w_i} = [-1\ 1\ -1]^T$, which represents a user that dislikes novels, likes care about poetry and dislikes romance.
- $\mathbf{u_j} = [1\ 0\ 1]^T$, which represents a book that is a novel, it is not poetry and has romantic elements in it.

then: $\mathbf{w_i}^T\mathbf{u_j} = -2$, which means that this particular book i would not be a good recommendation for user j. With this scheme, we can see that positive correlations yield positive numbers, whereas negative correlations yield negative numbers. On the other hand, zero seems to be neutral, indicating the number around which we should probably center our ratings, while implementing our recommendation system.

Now, in order to train our model, first we have to evaluate it. But to evaluate it we have to:

1. define our objective function
2. use only the ratings that are known

Based on the above, if:

$$\Omega = \{\text{set of all existing pairs } (i, j) \text{ where user } i \text{ rated book } j\}$$

then the total cost can be defined as:

$$J = \sum_{\substack{i \\ i,j \in \Omega}} \sum_{j} (r_{ij} - \hat{r}_{ij})^2 \tag{1.4}$$

Moreover, let's introduce the following two sets:

- $U_j = \{\text{set of users } i \text{ who rated book } j\}$
- $B_i = \{\text{set of books } j \text{ which user } i \text{ rated}\}$

Once we have our objective function, our goal is to minimize it with respect to its parameters. Since $\hat{r}_{ij} = \mathbf{w_i}^T \mathbf{u_j}$, then the unknowns of the problem are the vectors:

- $\mathbf{w_i}$, $i \in U_j$
- $\mathbf{u_j}$, $j \in B_i$

But our objective function is not yet complete. First let's introduce the **user bias** term b_i. This term has to do with the fact that different people rate things in a different way, because different numbers mean different things to different people. In other words users have biases. For instance, some people are optimistic and as a consequence they tend to give high ratings, whereas some others are pessimistic and therefore they tend to rate low.

Secondly, there is a **book bias** as well. To understand why this parameter makes sense, let's consider a popular book that many people like. Then our unbiased model might have latent features that capture some characteristics of the books. So, according to this, a popular book would have specific characteristics. But these attributes are not enough to predict how much someone would like this book, because there are also other books with the same characteristics, yet they are not good ones.

Moreover, it would be a good idea to center our dataset. To do this, we could also incorporate into our model a **global average** term μ. Finally, apart from all the previous additions, we could also include the corresponding **regularization** terms, so as to penalize for the large weights, preventing our model from overfitting. Therefore, our final objective function, that we want to optimize with respect to its parameters $\mathbf{W}, \mathbf{U}, \mathbf{b}, \mathbf{c}$ and μ can be written as follows:

$$J = \sum_{i \in U_j} \sum_{j \in B_i} (r_{ij} - \hat{r}_{ij})^2 + \lambda \left(||\mathbf{W}||_F^2 + ||\mathbf{U}||_F^2 + ||\mathbf{b}||_F^2 + ||\mathbf{c}||_F^2 \right) \tag{1.5}$$

where:

$$\hat{r}_{ij} = \mathbf{w_i}^T \mathbf{u_j} + b_i + c_j + \mu \tag{1.6}$$

1.2.3 Problem Solution

Before starting solving the optimization problem indicated by (1.5) we rewrite the objective function as follows:

$$J = \sum_{i \in U_j} \sum_{j \in B_i} \left(r_{ij} - \mathbf{w_i}^T \mathbf{u_j} - b_i - c_j - \mu \right)^2 + + \lambda \left(\mathbf{w_i}^T \mathbf{w_i} + \mathbf{u_j}^T \mathbf{u_i} + {b_i}^2 + {c_j}^2 \right) \tag{1.7}$$

Now, we take the derivatives with respect to its parameters:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{w_{i'}}} &= \sum_{j \in B_{i'}} 2 \left(r_{i'j} - \mathbf{w_{i'}}^T \mathbf{u_j} - b_{i'} - c_j - \mu \right) (-\mathbf{u_j}) + 2\lambda \mathbf{w_{i'}} = 0 \Leftrightarrow \\
&\Leftrightarrow \sum_{j \in B_{i'}} \left(\mathbf{w_{i'}}^T \mathbf{u_j} \right) \mathbf{u_j} + \lambda \mathbf{w_{i'}} = \sum_{j \in B_{i'}} \left(r_{i'j} + b_{i'} + c_j + \mu \right) \mathbf{u_j} \Leftrightarrow \\
&\Leftrightarrow \sum_{j \in B_{i'}} \left(\mathbf{u_j}^T \mathbf{w_{i'}} \right) \mathbf{u_j} + \lambda \mathbf{w_{i'}} = \sum_{j \in B_{i'}} \left(r_{i'j} + b_{i'} + c_j + \mu \right) \mathbf{u_j} \Leftrightarrow \\
&\Leftrightarrow \sum_{j \in B_{i'}} \mathbf{u_j} \left(\mathbf{u_j}^T \mathbf{w_{i'}} \right) + \lambda \mathbf{w_{i'}} = \sum_{j \in B_{i'}} \left(r_{i'j} + b_{i'} + c_j + \mu \right) \mathbf{u_j} \Leftrightarrow \\
&\Leftrightarrow \left(\sum_{j \in B_{i'}} \mathbf{u_j} {\mathbf{u_j}}^T + \lambda \mathbf{I} \right) \mathbf{w_{i'}} = \sum_{j \in B_{i'}} \left(r_{i'j} + b_{i'} + c_j + \mu \right) \mathbf{u_j} \Leftrightarrow \\
&\qquad \mathbf{w_{i'}} = \frac{\sum_{j \in B_{i'}} \left(r_{i'j} + b_{i'} + c_j + \mu \right) \mathbf{u_j}}{\left(\lambda \mathbf{I} + \sum_{j \in B_{i'}} \mathbf{u_j} {\mathbf{u_j}}^T \right)} \quad , i' \in U_j
\end{aligned} \tag{1.8}$$

Because equation (1.7) is symmetric with respect to $\mathbf{w_i}$ and $\mathbf{u_j}$, following exactly a similar procedure for $\mathbf{u_j}$ we get:

$$\mathbf{u_{j'}} = \frac{\sum_{i \in U_{j'}} \left(r_{ij'} + b_i + c_{j'} + \mu \right) \mathbf{w_i}}{\left(\lambda \mathbf{I} + \sum_{i \in U_{j'}} \mathbf{w_i} {\mathbf{w_i}}^T \right)} \quad , j' \in B_i \tag{1.9}$$

The next step is to take the derivative with respect to b_i:

$$\begin{aligned}
\frac{\partial J}{\partial b_{i'}} &= \sum_{j \in B_{i'}} 2 \left(r_{i'j} - \mathbf{w_{i'}}^T \mathbf{u_j} - b_{i'} - c_j - \mu \right) (-1) + 2\lambda b_{i'} = 0 \Leftrightarrow \\
&\Leftrightarrow \left(|B_{i'}| + \lambda \right) b_{i'} = \sum_{j \in B_{i'}} r_{i'j} - \mathbf{w_{i'}}^T \mathbf{u_j} - c_j - \mu \Leftrightarrow
\end{aligned}$$

$$\Leftrightarrow b_{i'} = \frac{\sum_{j \in B_{i'}} r_{i'j} - \mathbf{w_{i'}}^T \mathbf{u_j} - c_j - \mu}{|B_{i'}| + \lambda} \tag{1.10}$$

As previously, because (1.7) is symmetric with respect to b_i and c_j, following exactly a similar procedure for we get the following expression for c_j:

$$c_{j'} = \frac{\sum_{i \in U_{j'}} r_{ij'} - \mathbf{w_i}^T \mathbf{u_{j'}} - b_i - \mu}{|U_{j'}| + \lambda} \tag{1.11}$$

Finally, we get the expression for μ by taking the corresponding derivative and setting it equal to zero:

$$\begin{aligned} \frac{\partial J}{\partial \mu} &= \sum_{i \in U_j} \sum_{j \in B_i} 2 \left(r_{ij} - \mathbf{w_i}^T \mathbf{u_j} - b_i - c_j - \mu \right) (-1) = 0 \Leftrightarrow \\ &\Leftrightarrow \mu = \frac{\sum_{i \in U_j} \sum_{j \in B_i} \left(r_{ij} - \mathbf{w_i}^T \mathbf{u_j} - b_i - c_j \right)}{|U_j| + |B_i|} \end{aligned} \tag{1.12}$$

The problem with expressions (1.9)-(1.12) is that the variables we want to find depend on other variables that we also want to find. However, it can be proven that if we randomly initialize our variables and follow the calculations indicated by equations (1.9)-(1.12) then this procedures converges. This algorithm is known as **alternating least squares**.

1.3 GloVe (Global Vectors for word representation)

The main idea behind GloVe is to build a matrix $\mathbf{X}$, like the matrix $\mathbf{R}$ in recommender systems, which will also take context into account. In particular, $\mathbf{X}$ should be a term-by-term matrix, where each of its elements x_{ij} should have a higher value, if word i appears in the context of word j often and a lower value if ord i appears in the context of word j less often.

In **GloVe** what's important is how we quantify the **context distance**. This is done as follows: if a word is right beside another word then we add 1 to x_{ij}. If a word is within the context of another word and another word between them then we add $1/2$ to x_{ij}. If there are two words between the words i and j, then we add $1/3$ to x_{ij} and so on.

But in real world language some words appear quite more often than others, which means that if we create the words' histogram, it is going to be long tailed. As far as our matrix $\mathbf{X}$ is concerned, that means that most of its entries are zero. In other words, $\mathbf{X}$ is a sparse matrix, not in the same sense as with the recommender systems' matrix $\mathbf{R}$, where most values where missing, but in the sense that they are just zero.

Moreover, the non-zero values of matrix $\mathbf{X}$ are going to be quite large, depending on the number of documents we have in our corpus. A common technique usually implemented when values grow exponentially large is to apply the $\log$ function. But because $\log 0$ is not defined, what the researchers from Standford university did, is to add 1 before applying the $\log$ function. One more thing that the same researchers found to work well is to weight each entry of matrix $\mathbf{X}$ with the following function:

$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{x_{max}}\right)^{\alpha} & x_{ij} < x_{max} \\ 1 & x_{ij} \geq x_{max} \end{cases} \tag{1.13}$$

where $a = 0.75$ and $x_{max} = 100$. In this case the cost function is the following:

$$J = \sum_i \sum_j f(x_{ij}) \big(\log x_{ij} - \mathbf{w_i}^T \mathbf{u_j} - b_i - c_j - \mu\big)^2 \tag{1.14}$$

In the original GloVe paper the global average paramaeter μ was not included and the optimization problem was solved using gradient descent. However, the same problem can be solved with the use of the alternating least squares method. To apply this method the following equations are used:

$$\mathbf{w_i} = \left(\sum_j f(x_{ij}) \mathbf{u_j} \mathbf{u_j}^T\right)^{-1} \sum_j f(x_{ij})(\log x_{ij} - b_i - c_j - \mu)\mathbf{u_j}$$

$$\mathbf{u_j} = \left(\sum_i f(x_{ij}) \mathbf{w_i} \mathbf{w_i}^T\right)^{-1} \sum_i f(x_{ij})(\log x_{ij} - b_i - c_j - \mu)\mathbf{w_i}$$

$$\mathbf{b_i} = \left(\sum_j f(x_{ij})\right)^{-1} \sum_j f(x_{ij})(\log x_{ij} - \mathbf{w_i}^T \mathbf{u_j} - c_j - \mu)$$

$$\mathbf{c_j} = \left(\sum_i f(x_{ij})\right)^{-1} \sum_i f(x_{ij})(\log x_{ij} - \mathbf{w_i}^T \mathbf{u_j} - b_i - \mu)$$

Chapter 2

Advanced Deep Learning models in NLP

Chapter 2 contents [11], [12], [13], [14], [15], [16],

2.1 Introduction

In the previous chapter we examined a crucial stepping stone in the evolution of NLP, which are Word Embeddings. As we've seen they offer more compact representations of words and manage to reduce the computational cost tremendously. Although, Word Embeddings come with many advantages, they were not created to work on their own. In fact, their initial goal was to reduce the dimensionality of the input data so that it would be feasible then to feed it into a more complicated model.

The main reason for which Word Embeddings can not work isolatedly is that they find only **general** relationships between the words. But we know that these relationships change from sentence to sentence. Moreover the relationships between the words depends on their distance in the same sentence. Therefore it would be a good idea to create a model that would be able to "remember" the words that preceded in the sentence. The first notable Neural Networks that were used in combination Word Embeddings and aimed at remembering things from the past were the **Recurrent Neural Networks** (**RNNs**).

However, although RNNs were capable of partially recalling past things, in prac-

tice it was found that they were unable to capture long-term dependencies. Therefore, this important disadvantage of RNNs, led researchers to invent more complicated models such as the **gated RNNs**. The two major representatives of this category of Neural Networks are the **Gated Recurrent Unit** (**GRU**) and the **Long Short-Term Memory** (**LSTM**). But still, even these quite advanced and complicated deep learning models come with their own shortcomings. For that reason researchers tried to find even more efficient models and the outcome of their search was the **Bidirectional RNNs**.

2.2 Bidirectional RNNs

In order to better understand the motivation behind Bidirectional RNNs we will see two examples. Suppose we have the following incomplete sentences:

- "I am _____"
- "I am _____ tired"
- "I am _____ tired, because I didn't sleep last night"

In the first sentence "thirsty" seems to be a likely candidate, whereas the words "not" and "very" seem plausible in the second sentence. But "not" seems incompatible with the third sentence. Therefore, if we had a model that only looks at the previous words, then probably it would not give us a correct answer when predicting the missing word in the third sentence.

Also, let's consider another example. For instance let's assume that we have a Named Entity Recognition (NER) task for the word "general" in the following three sentences:

- "**General** Commander is a 2019 direct-to-video American action film produced by Philippe Martinez."
- "**General** Company is a European based international Company with activities all over the world."
- "**General** relativity is a theory of gravitation developed by Albert Einstein in the early of 1900's."

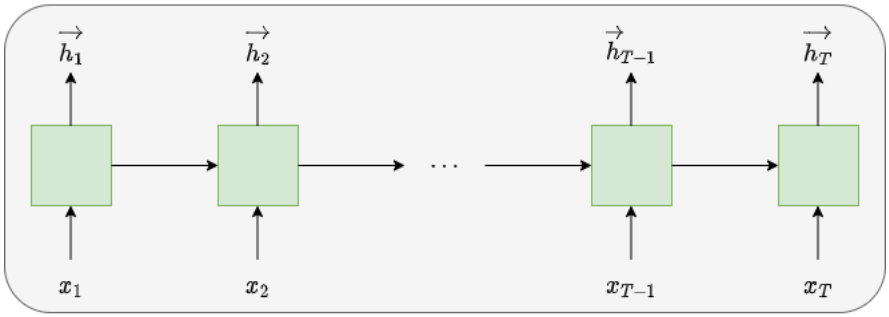


Figure 2.1: Bidirectional RNN (**forward** calculation)

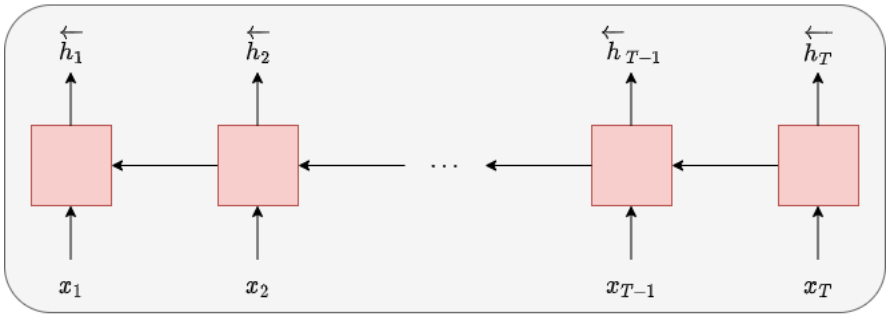


Figure 2.2: Bidirectional RNN (**backward** calculation)

In the first of the above sentences the "general" is an adjective and therefore it can not be labeled as anything special, such as a person, a company, a date etc. But in the second and third sentences the same word can be labeled as "company" and "movie" accordingly. In other words, in all of the cases the word "general" would be tagged differently. But the problem is that the word of our interest is at the beginning of the sentence. Therefore, an LSTM or GRU processing the sentence wouldn't see any words before the word "general", hence it should make an inference based on that word alone.

However, we as humans, understand that the word "general" has a different role in each sentence, because we can examine the whole sentence simultaneously. The solution to the problems, illustrated by the two previous examples, is to add one more model that will process the same sequence of input data in the opposite way.

That idea gave birth to the **Bidirectional RNNs**. The way these models work is the following: assuming that our input sequence has length T, first we do the usual RNN calculation to get the final hidden state $\overrightarrow{h_T}$ as we can see in figure (2.1).

Then, we use another RNN, but this time we read the sequence in reverse as we can see in figure (2.2). In other words, a bidirectional RNN is comprised of two RNNs, one going forward and the other backward. To get the final hidden state vector at time t, we concatenate the hidden state vectors at time t of these two RNNs and thus we get

$$\mathbf{h_t} = \left[\overrightarrow{h_t}, \overleftarrow{h_t}\right]$$

where if $\overrightarrow{h_t} \in \mathbf{R}^{T\times M}$ and $\overleftarrow{h_t} \in \mathbf{R}^{T\times M}$, then $\mathbf{h_t} \in \mathbf{R}^{T\times 2M}$. Generally, there are several ways to determine the final output of our model. However, the most common is to take the final hidden state of the forward RNN, $\overrightarrow{h_T}$, and the final hidden state of the backward RNN, $\overleftarrow{h_1}$, and concatenate them together as follows:

$$\mathbf{h_{out}} = \left[\overrightarrow{h_T}, \overleftarrow{h_1}\right] \tag{2.1}$$

Intuitively, the reason this makes sense is because for the vectors $\overrightarrow{h_T}$ and $\overleftarrow{h_1}$ to be created, the forward and backward RNNs have both processed the **whole** sequence but in opposite ways. After the $\mathbf{h_{out}}$ is created, we can pass it a dense layer in order to make our model more complex, so as to be able to find richer representations of our input. However, although bidirectional RNNs are powerful models, **they must not be used when our task is to predict the future**.

2.3 Sequence-to-Sequence models (Seq2Seq)

The bidirectional RNNs, although powerful models, have their own limitations. The most important limitation is that are confined to fixed sequence lengths. In other words, if we build a bidirectional RNN which accepts as its input sequences of length T, then it won't be able of handling sequences with length other than T. As a consequence, that crucial limitation renders them incapable of dealing with some NLP tasks.

The most well known NLP problem that bidirectional RNNs can not deal with is the language translation. The reason for this is that the number of characters

changes from language to language. For example the translation of "What time is it?" is "Τι ώρα είναι;". Here we see clearly that while the former sequence (english) consists of four words, the latter (greek) consists of three. Another type of task that bidirectional RNNs are incapable of solving is question answering. In that kind of task our model is asked a question, which consists of T words, but in most cases the answer will be of different length T'.

Let's take for instance the same question as before "What time is it?". In this case we do not want our model to translate it into another language, but to give us the proper answer, for example: "It is quarter to five". Here again we observe that the input and output sentences differ in length rendering bidirectional RNNs useless for this particular task.

The solution to this problem came by some researchers at Google who invented the **Sequence-to-Sequence models**, widely known as **Seq2Seq**. Actually, their architecture is a dual RNN system, as we can see from the following figure:

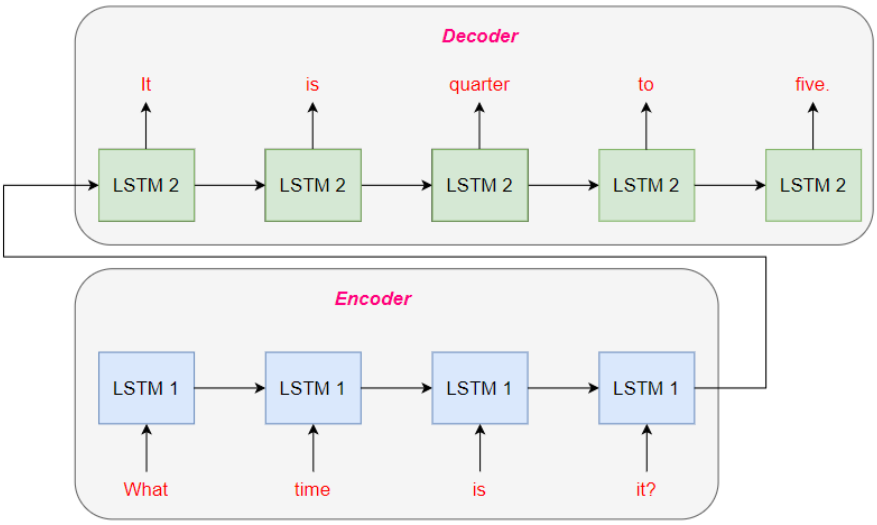


Figure 2.3: Seq2Seq model (example of question answering)

The first RNN which takes in the input is called the **encoder**, whereas the second RNN, which produces the output (translation or answer) is the **decoder**. What this architecture actually does is to take an input sequence of words and first create a compact representation of it. Then, in the second stage the decoder unfolds the

compressed vector representation creating the output. Another way to state this is that the encoder's goal is to fold up the input sequence into a small but useful and informative vector, whereas the decoder's job is to unravel that vector into a new sequence, which is the output.

In more detail, the encoder works like a standard RNN (in our case it is an LSTM). We pass in the input sequence and keep only the hidden state at the last time-step T, $\mathbf{h_T}$. In our example, this is $\mathbf{h_4}$ as we can see from figure (2.4):

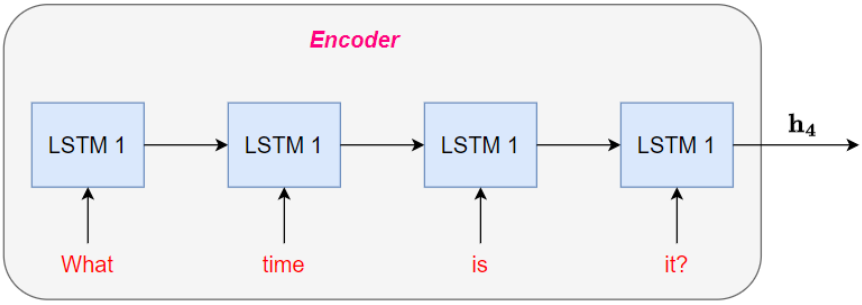


Figure 2.4: Seq2Seq model (example: encoder)

This vector of size M, is a vector that somehow represents the input data. That is the reason why the this first stage is called encoding. It's because $\mathbf{h_T}$ is a low-dimensional representation of the original data.

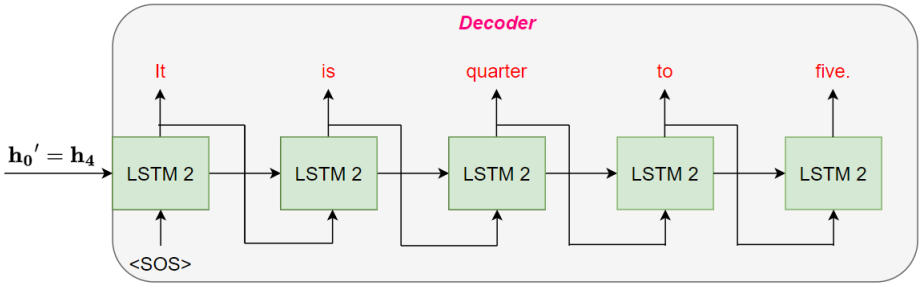


Figure 2.5: Seq2Seq model (example: decoder)

As far as the decoder is concerned, this is the novel part of the Seq2Seq architecture. At this part the RNN units are different, but they have the same size as the encoder RNNs. The reason we need to do this is because instead of passing in any

old hidden state into the decoder, we instead **pass in the last hidden state of the encoder** ($\mathbf{h_0}' = \mathbf{h_4}$). In general $\mathbf{h_0^{dec}} = \mathbf{h_T^{enc}}$.

During training, for the first input we pass in a special token to denote the **S**tart-**O**f-**S**entence ($\mathbf{x_1}$ = <SOS>). From $\mathbf{h_0}'$ and $\mathbf{x_1}$, $\mathbf{h_1}$ and $\hat{\mathbf{y}}_\mathbf{1}$ are calculated. But, $\hat{\mathbf{y}}_\mathbf{1}$ is a vector of probabilities, so from there we can take the argmax to pick the most likely word from our vocabulary.

After the first output is calculated, in order to produce the second word, the previous hidden state and some kind of input $\mathbf{x_2}$ are needed. One way to overcome this issue is just to pass as input in the second step the output $\hat{\mathbf{y}}_\mathbf{1}$ from the previous step. So generally $\hat{\mathbf{y}}_{\mathbf{t-1}}$ becomes $\mathbf{x_t}$. Another way to deal with this problem and which researchers have found to work even better is the so called **teacher forcing**. What we do in this case is instead of feeding the previously generated output into the encoder's input, we instead **feed in the true previous word**. In fact, what is actually needed is to offset the output sentence by 1. So even if we wrongly predicted the previous word, this method corrects it and therefore the model can predict the next word based on the real translation.

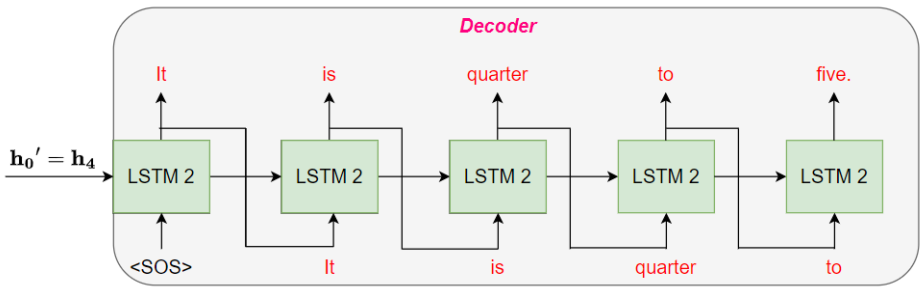


Figure 2.6: Seq2Seq model (example: decoder) - The true output is used as input in the next RNN unit

But the teacher forcing method can be implemented only during the training. When we want to make predictions during test time we don't have the true output and therefore this method can not be applied. For testing we go back into the original architecture, where we pass in the previous output into the next input. The problem with this approach is that during inference time, the input sequence is always going to be one. In other words we have to feed our model with only one

word at a time. We can not pass in the whole sequence, because the output at $t-1$ is a prerequisite for the output at time t to be generated.

2.4 Attention based models

Attention based models are a specific type of of Sequence-to-Sequence models. As the previous model that we have examined, their architecture consists of an encoder and a decoder, whose input and output accordingly, are generally sequences of different sizes. But their main difference lies in the use of a mechanism called **attention**.

In particular, let's assume that we have an input sequence of length T_x. In most cases of attention based models, the encoder consists of a bidirectional LSTM, as we can see in figure (2.7)

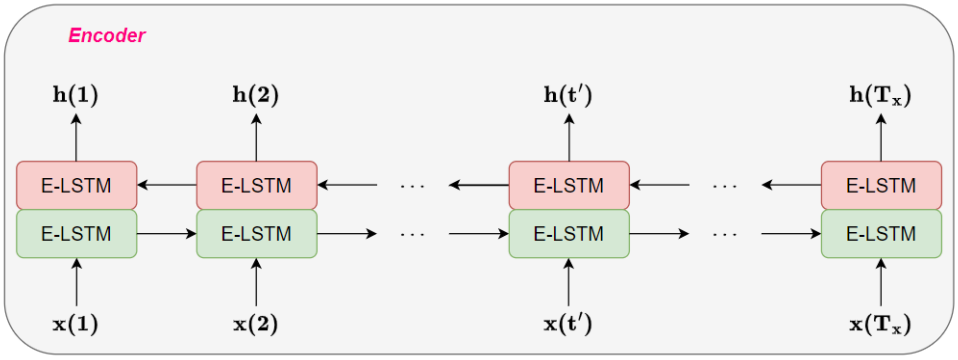


Figure 2.7: Attention based model (Encoder) - Note that in this figure the abbreviation E-LSTM stands for Encoder-LSTM

Since we have a bidirectional LSTM each of the T_x hidden states is of size $2M$, assuming that the hidden states of the E-LSTM are M-dimensional vectors. Hence, the output of the encoder is $T_x \times 2M$. The main difference between the Attention based models and the seq2seq models without attention is that whereas the latter care only about the last hidden state vector $\mathbf{h}(T_x)$, the former make use of all the hidden state vectors $\mathbf{h}(1), \mathbf{h}(2), \ldots, \mathbf{h}(T_x)$.

Subsequently, these vectors pass through the attention mechanism and the **context** vector is created, which is fed into the decoder. Generally, this context vector

informs us which hidden states we care most about. In other words, since it incorporates some learnable parameters, we could say that it adaptively quantifies the amount of attention that our model should pay to each of the hidden state vectors.

In fact, in most cases, the attention mechanism is just the implementation of a weighted average on the hidden states. If we are at the time step t, which indicates the output we are currently trying to calculate attention for, then this can be formulated mathematically as follows:

$$\mathbf{c}(t) = \sum_{t'=1}^{T_x} \alpha(t, t')\mathbf{h}(t') \tag{2.2}$$

and is illustrated in figure (2.8):

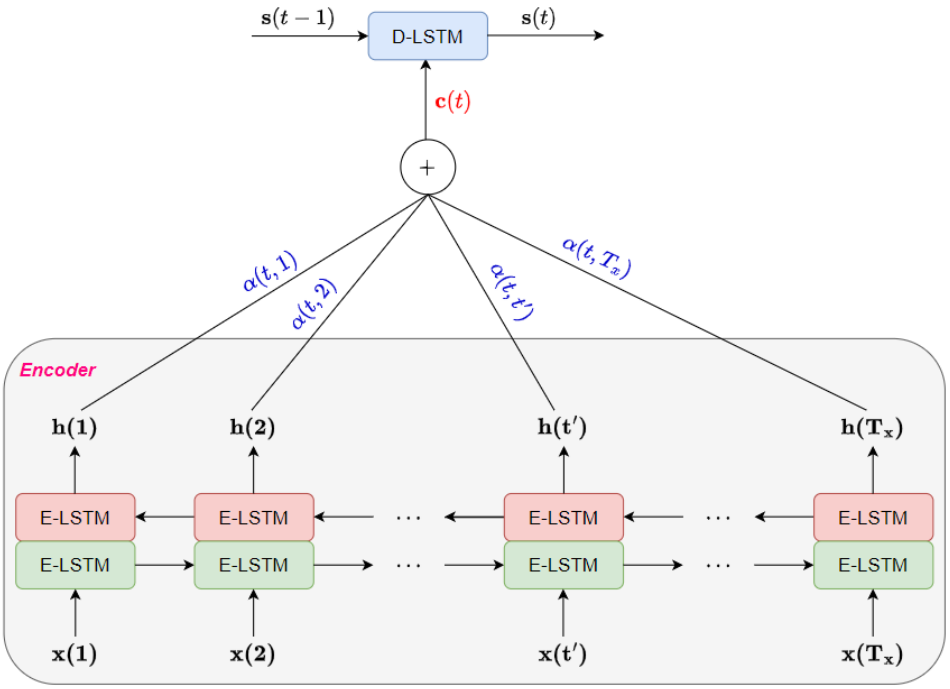


Figure 2.8: Attention based model (Attention Mechanism) - Note that in this figure the abbreviation D-LSTM stands for Decoder-LSTM

As we can see from (3.12), for each step of the output sequence we take into account all the steps of the input sequence. Furthermore, from the same equation

it is clear that the parameters α tell us how much attention we should pay at the input hidden state t' right now. But here the question about how our model learns these parameters becomes relevant.

The simplest way is to leave these weights as they are and they will be learned through the backpropagation algorithm. However, according to this approach the α's do not carry with them information either from the encoders' hidden states $\mathbf{h}(t')$ or from the decoders' hidden states $\mathbf{s}(t)$. Another, more sophisticated approach, would be to enforce the parameter α at time t to be dependent on both $\mathbf{s}(t-1)$ and $\mathbf{h}(t')$. The reason we would like to do that is, because it is logical that the attention weights depend both on where we are in the input sentence and where we are in the output sentence.

For example, suppose we want to translate "What time is it?" from english to greek "Τι ώρα είναι;"

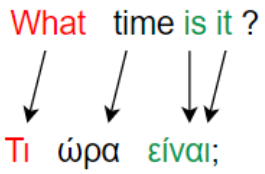


Figure 2.9: Example - Why it is a good idea for the parameters α to depend both the input and the output hidden state vectors

So we can see from figure (2.9) that when generating the first word of the translation, our attention is currently on the english word "What". When generating the second english word in the translation our attention is no longer in the english word "What", but rather has moved over to the word "time" and so on. Therefore, it is clear that which part of the input sentence we pay attention to must change as we generate new words in the output.

One way to implement this idea, is to concatenate the input and output hidden states $\mathbf{s}(t-1)$ and $\mathbf{h}(t')$ and then let them pass through a neural network. Finally, the attention weights are normalized by applying the softmax function over the input time step t, as follows:

$$\alpha(t,t') = softmax\Big(ANN\Big([\mathbf{s}(t-1), \mathbf{h}(t')]\Big)\Big), \quad t' = 1, \ldots, T_x \tag{2.3}$$

Furthermore, if we assume that the feed-forward network consists of only one hidden layer, then (2.3) can be written equivalently in the following form:

$$\alpha(t,t') = \frac{e^{\mathbf{W}^{[2]}(t,t')\left[f_{act}\left(\mathbf{W}^{[1]}(t,t')\begin{bmatrix}\mathbf{s}(t-1)\\ \mathbf{h}(t')\end{bmatrix}+\mathbf{b}^{[1]}(t,t')\right)\right]+\mathbf{b}^{[2]}(t,t')}}{\sum_{\tau=1}^{T_x} e^{\mathbf{W}^{[2]}(t,\tau)\left[f_{act}\left(\mathbf{W}^{[1]}(t,\tau)\begin{bmatrix}\mathbf{s}(t-1)\\ \mathbf{h}(\tau)\end{bmatrix}+\mathbf{b}^{[1]}(t,\tau)\right)\right]+\mathbf{b}^{[2]}(t,\tau)}} \tag{2.4}$$

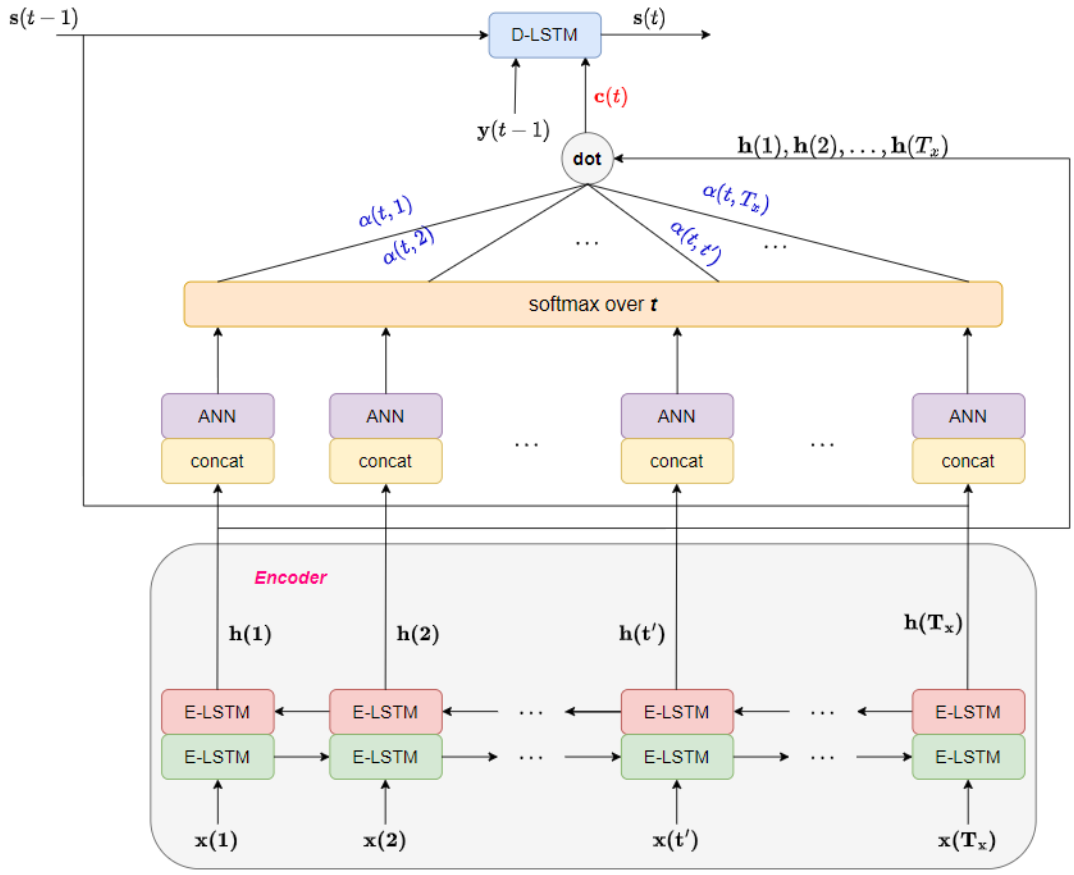


Figure 2.10: Attention based model - Architecture at time t

As far as the decoder is concerned, it is a regular multi-layered uni-directional LSTM. As it is illustrated in figure (2.10), the hidden state at time t depends on

the previous hidden state $\mathbf{s}(t-1)$, on the previous output $\mathbf{y}(t-1)$ and on the context vector $\mathbf{c}(t)$:

$$\mathbf{s}(t) = f(\mathbf{s}(t-1), \mathbf{y}(t-1), \mathbf{c}(t)) \tag{2.5}$$

Since in the original architecture of the LSTM unit it was designed to take only two vector inputs, in the Attention based model it was slightly modified so as compensate for the fact that know three input vectors are needed. In fact, for the context vector to be incorporated, the D-LSTM expressions were formulated as follows:

$$\mathbf{s}(t) = (1 - \mathbf{z}(t)) \circ \mathbf{s}(t-1) + \mathbf{z}(t) \circ \tilde{\mathbf{s}}(t) \tag{2.6}$$

where:

$$\begin{aligned}
\tilde{\mathbf{s}}(t) &= tanh(\mathbf{W}\mathbf{E}\mathbf{y}(t-1) + \mathbf{U}[\mathbf{r}(t) \circ \mathbf{s}(t-1)] + \mathbf{C}\mathbf{c}(t)) \\
\mathbf{z}(t) &= \sigma(\mathbf{W_z}\mathbf{E}\mathbf{y}(t-1) + \mathbf{U_z}\mathbf{s}(t-1) + \mathbf{C_z}\mathbf{c}(t)) \\
\mathbf{r}(t) &= \sigma(\mathbf{W_r}\mathbf{E}\mathbf{y}(t-1) + \mathbf{U_r}\mathbf{s}(t-1) + \mathbf{C_r}\mathbf{c}(t))
\end{aligned} \tag{2.7}$$

Finally, putting all together we get:

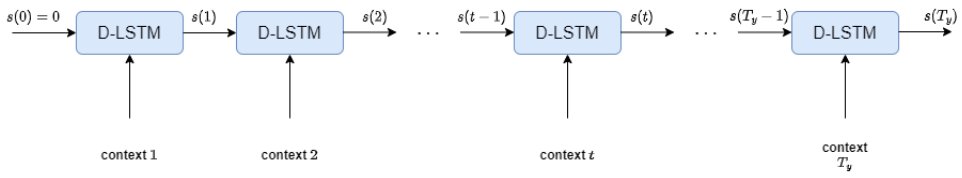


Figure 2.11: Attention based model - Stacking multiple decoder-LSTM-blocks together.

Figure (2.11) shows how the decoding process unfolds. In particular, the initial state is set to 0. The first decoder LSTM takes as its input the initial state and the first context vector and outputs the state $s(1)$. The second decoder LSTM takes as its input the previous state $s(1)$ and the second context vector and outputs the state $s(2)$. Generally, the decoder LSTM at time t takes as its input the previous state $s(t-1)$ and the context vector at time t and outputs the state $s(t)$. Since, we have assumed that the output sequence has length T_y, the last decoder LSTM takes as its input the state $s(T_y - 1)$ and the context vector at time T_y and outputs the last state $s(T_y)$.

Chapter 3

Transformers

Chapter 3 contents [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30],

3.1 Introduction

Transformers were first introduced in 2017, by a team at Google Brain, in the paper "**Attention Is All You Need**". A Transformer is a deep learning model that makes use of the mechanism of Self-Attention, in order to weight the input data, which most often are words or images, based on the their context. Although, nowadays, they are used both in the fields of Natural Language Processing (NLP) and Computer Vision (CV), due to their amazing efficiency, their initial purpose was to replace the already existing Sequence-to-Sequence (Seq2Seq) models in tasks like text summarization and translation. In particular, before Transformers the most commonly used models for NLP were the gated Recursive Neural Networks (gated RNNs) such as LSTMs and GRUs with some additional attention mechanisms. The main difference between Transformers and the previous state-of-the-art models is that the former do not have a recurrent structure

From the very beginning, Transformers gained immediate popularity, because of their astonishing improvements in the aforementioned problems, in comparison to other deep learning models that aim at processing sequential data, such as RNNs, GRUs or LSTMs.The main benefits from the use of Transformers are three:

- in opposition to the already existing sequence models, they are able to capture more complicated and longer term dependencies in the input data,

which makes them more efficient in solving real-world problems.

- they overcome the problem of vanishing gradients, which is a common issue in modern deep learning models, such as LSTMs, because they make use of the "skip connections". More specifically, gated RNNs process the input tokens sequentially, while simultaneously keep a hidden state vector for each time step, which represents the data prior to the input token. To extract information from a new input token at time t , the RNN combines it with the previous $(t-1)$ hidden state and creates the new hidden state. Although, theoretically, we could follow the same procedure up to the last token (for example the end of our sentence) and gain information from the whole sentence, this process has a serious drawback: the more far down the sequence we go, the less extractable and precise the information is carried through the hidden state vector. This is because, in order to train the model we have to implement the backpropagation algorithm, from where the problem of vanishing gradients arises.

- while gated RNNs process the input tokens one-at-a-time, Trasformers can simultaneously process the entire input data during training, which makes them highly parallelizable. This all-at-once processing of the input data makes the training procedure much faster and in quite short time gave the opportunity to researchers to the develop pre-trained systems such as BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), which were trained with large language datasets. Therefore, Transformers can be fine-tuned for specific tasks, which renders them a very powerful tool.

In figure (3.1) we can see the model architecture of the Transformer. Although from a first glance it seems quite complex, in the coming sections we will examine all of its particular constituents one by one. First, we will delve into the details of the left part of the diagram, which is the **Encoder** and then we will go through the right part of the diagram, which is the **Decoder**. At this point, it's important to note that like the seq2seq models, both the encoder and the decoder can work independently to each other in order to solve some particular task. However, figure (3.1) illustrates the original Transformer model, as is was presented in the paper "Attention Is All You Need".

Now, let's try to decipher the various parts of the Transformer. Suppose the input to our model is a sentence s_0 which consists of the words/tokens $\mathbf{w_0}, \mathbf{w_1}, \ldots, \mathbf{w_{T-1}}$,

where:

$$\mathbf{w_j} \in \mathbb{R}^{d_{vocab\_size}}, \quad j = 0, \ldots, T-1 \tag{3.1}$$

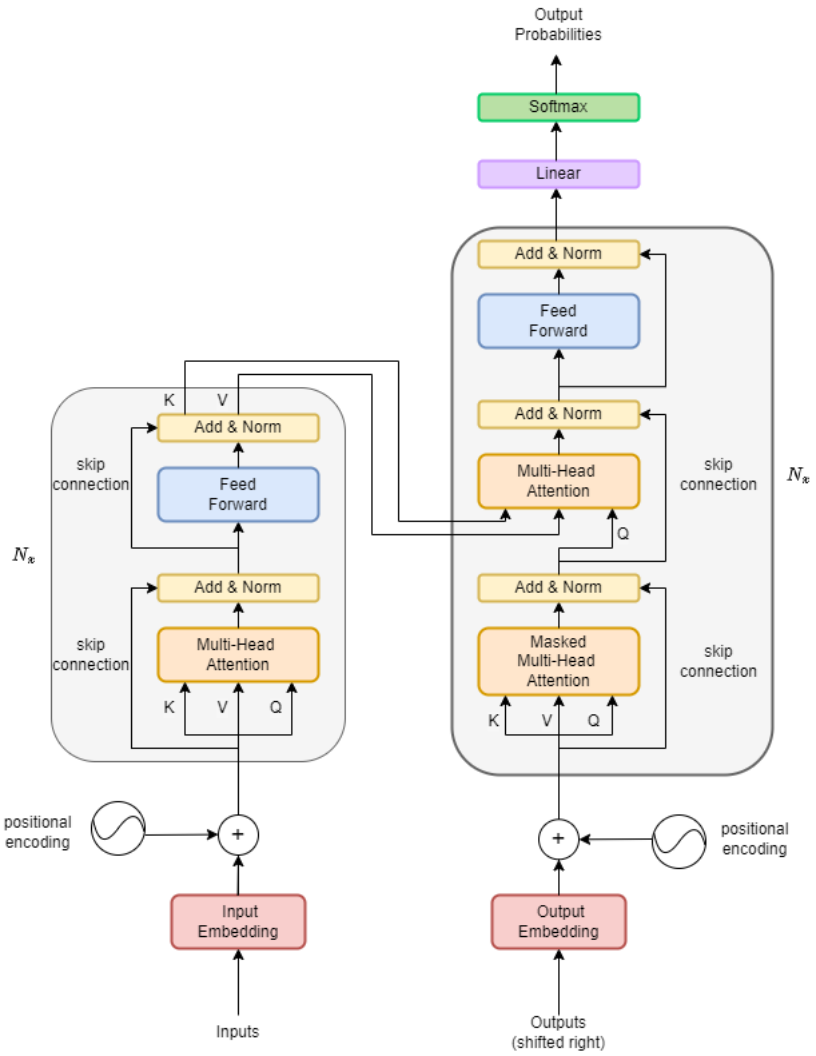


Figure 3.1: The Transformer: Encoder-Decoder Architecture

and d_{vocab_size} is the **vocabulary size**, for which usual values are $20,000 - 50,000$.

The first step according to the figure (3.1) is that the input tokens pass through an **embedding layer**. We have already mentioned what an embedding layer does and what its usefulness is and therefore no further explanation is needed, except that each word from (3.1) is transformed into an embedded word vector:

$$\mathbf{x_j} \in \mathbb{R}^{d_{model}}, \quad j = 0, \ldots, T-1 \tag{3.2}$$

where $d_{model} \ll d_{vocab\_size}$.

3.2 Positional Encoding

The next step is to study the positional encoding layer, how it works and why it is needed. Although the embedding layer is very useful in transforming our initial word vectors into lower dimensional vectors, it doesn't provide us with any information about the **position** of each particular word in the sentence. Consider what would happen if we randomly rearranged all the words in our input sentence. Intuitively this should probably confuse our model, just as it would definitely confuse us, because our **understanding** depends on the **order** of the incoming words.

Therefore, the whole purpose behind the techinque called **positional encoding** is to resolve this issue, that is to add this useful piece of information into the embedded words. In other words we would like the machine we are trying to build to be **permutation variant** or **position specific**, which means to be aware not only of the input word but also of its position in the sentence.

The order of words plays a very important role in any language. RNNs inherently take order into account, because they work in a sequential manner. But, because as we will see later, Transformers are capable of simultaneously processing the input tokens, we need to manually incorporate the appropriate positional information into our model.

The first and most simple idea would be to linearly assign a number to each token, that is: "0" for the first token, "1" for the second token and so on, creating a positional vector with the same size as the numbers of words that constitute the sentence. One issue that comes along with this approach is that the numbers representing each token may become quite large for big sentences. This could easily be solved by normalizing the positional vector.

But there is another more serious problem: how should we handle sentences of different size than the ones we used for training? If the test sentences were smaller in size than those in the training set, we could overcome this obstacle by padding them with zeros. But in the case where test sentences were larger, we would have to cut them in the middle, which is not an effective solution.

Therefore the researchers should come up with a solution or mechanism, which would ideally satisfy the following criteria:

- for each word it should output a unique encoding for each of its possible positions in the sentence.
- the encoding values should be bounded.
- it must be deterministic.
- the distance between the positions should be consistent regardless of the length of the sentence.

The encoding that the authors proposed is not a complex one, but very clever. The idea might have come from the fact that, since our embedded words are of size d_{model}, instead of representing the position of a word in the sentence by a single number, we could represent it by a vector the same size.

Let t_j denote the position of x_j in an input sentence and $f : \mathbb{R}^2 \rightarrow \mathbb{R}^{d_{model}}$ the function that produces the elements of the positional vector $\mathbf{p}(t_j, d_{model})$:

$$p_k^{(i)}(t_j, d_{model}) = f_k^{(i)}(t_j, d_{model}) \triangleq \begin{cases} \sin(\omega_k t_j) & \text{, if } \quad i = 2k \\ \cos(\omega_k t_j) & \text{, if } \quad i = 2k+1 \end{cases} \tag{3.3}$$

where:

$$\omega_k(d_{model}) = \frac{1}{10000^{\frac{2k}{d_{model}}}} \tag{3.4}$$

Note that the angular frequencies ω_k are a decreasing function of the vector dimension d_{model}. From 3.3 we can form the positional vector $\mathbf{p}(t_j, d_{model})$, which contains pairs of sines and cosines for different frequencies:

$$\mathbf{p}(t_j, d_{model}) = \begin{bmatrix} \sin(\omega_1 t_j) \\ \cos(\omega_1 t_j) \\ \\ \sin(\omega_2 t_j) \\ \cos(\omega_2 t_j) \\ \vdots \\ \sin(\omega_{d_{model}/2} t_j) \\ \cos(\omega_{d_{model}/2} t_j) \end{bmatrix} \tag{3.5}$$

where we choose d_{model} to be divisible by 2. Now, suppose we assume that d_{model} is fixed to a chosen value and we want to find how $\mathbf{p}(t_j, d_{model})$ changes with respect to t_j. Thus, let's take, for example, the position $t_j + \phi$, then:

$$\begin{aligned}
\mathbf{p}(t_j + \phi, d_{model}) &= \begin{bmatrix} \sin(\omega_1(t_j + \phi)) \\ \cos(\omega_1(t_j + \phi)) \\ \vdots \\ \sin(\omega_{d_{model}/2}(t_j + \phi)) \\ \cos(\omega_{d_{model}/2}(t_j + \phi)) \end{bmatrix} \\
&= \begin{bmatrix} \sin(\omega_1 t_j)\cos(\omega_1\phi) + \cos(\omega_1 t_j)\sin(\omega_1\phi) \\ \cos(\omega_1 t_j)\cos(\omega_1\phi) - \sin(\omega_1 t_j)\sin(\omega_1\phi) \\ \vdots \\ \sin(\omega_{d_{model}/2} t_j)\cos(\omega_{d_{model}/2}\phi) + \cos(\omega_{d_{model}/2} t_j)\sin(\omega_{d_{model}/2}\phi) \\ \cos(\omega_{d_{model}/2} t_j)\cos(\omega_{d_{model}/2}\phi) - \sin(\omega_{d_{model}/2} t_j)\sin(\omega_{d_{model}/2}\phi) \end{bmatrix} \\
&= \begin{bmatrix} \cos(\omega_1\phi) & \sin(\omega_1\phi) & \mathbf{0} & \cdots & \mathbf{0} \\ -\sin(\omega_1\phi) & \cos(\omega_1\phi) & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \cdots & \mathbf{0} & \cos(\omega_{d_{model}/2}\phi) & \sin(\omega_{d_{model}/2}\phi) \\ \mathbf{0} & \cdots & \mathbf{0} & -\sin(\omega_{d_{model}/2}\phi) & \cos(\omega_{d_{model}/2}\phi) \end{bmatrix} \begin{bmatrix} \sin(\omega_1 t_j) \\ \cos(\omega_1 t_j) \\ \vdots \\ \sin(\omega_{d_{model}/2} t_j) \\ \cos(\omega_{d_{model}/2} t_j) \end{bmatrix} \\
&= \mathbf{T}(\phi)\mathbf{p}(t_j, d_{model})
\end{aligned}$$

where:

$$\mathbf{T}(\phi) \triangleq \underbrace{\begin{bmatrix} \mathbf{R_1}^T(\phi) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{R_2}^T(\phi) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{R_{d_{model}/2}}^T(\phi) \end{bmatrix}}_{(d_{model}/2)\times(d_{model}/2)} \tag{3.6}$$

is a transformation matrix and $\mathbf{0} \in \mathbb{R}^{2\times 2}$. Also, note that in (3.6):

$$\mathbf{R_m}(\phi) \triangleq \begin{bmatrix} \cos(\omega_m\phi) & -\sin(\omega_m\phi) \\ \sin(\omega_m\phi) & \cos(\omega_m\phi) \end{bmatrix}, \quad m = 1, \ldots, d_{model}/2 \tag{3.7}$$

is the rotation matrix. Also, we can observe that $\mathbf{R_m}^T(\phi) = \mathbf{R_m}(-\phi) \quad \forall m = 1, \ldots, d_{model}/2$, therefore (3.6) can be written as follows:

$$\mathbf{T}(\phi) = \begin{bmatrix} \mathbf{R_1}(-\phi) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{R_2}(-\phi) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{R_{d_{model}/2}}(-\phi) \end{bmatrix} \tag{3.8}$$

In order to understand what these equations mean let's first take the simple case of 2-dimensions. Then from (3.2) and (3.8) we get:

$$\mathbf{p}(t_j + \phi, d_{model}) = \mathbf{R_1}(-\phi)\mathbf{p}(t_j, d_{model}) \tag{3.9}$$

Since $\mathbf{R_1}(-\phi)$ is a rotation matrix, what it actually does is to rotate the vector $\mathbf{p}(t_j, d_{model})$ by angle $-\phi$. Moreover, let's assume that the embedding vector of the word x_j is $\mathbf{e_j}$ and the corresponding position is t_j, Suppose, that we do the same thing as the researchers did, which is to add the positional encoding vector to the embedding vector and we create the **positional embedding vector** $\mathbf{p_e}(t_j, d_{model})$, then we get:

$$\mathbf{p_e}(t_j, d_{model}) = \mathbf{e} + \mathbf{p}(t_j, d_{model})$$

Now, let's assume that we change the position of the word $\mathbf{x_j}$ from t to $t + \phi$, then the positional embedding vector changes, because although the embedding vector is constant for a particular embedding word, the positional vector depends on the position shift ϕ. Therefore, we get:

$$\mathbf{p_e}(t_j + \phi, d_{model}) = \mathbf{e} + \mathbf{p}(t_j + \phi, d_{model}) \overset{(3.9)}{=} \mathbf{e} + \mathbf{R_1}(-\phi)\mathbf{p}(t_j, d_{model}) \tag{3.10}$$

where, since we are in the 2D-case, from (3.5) we get:

$$\mathbf{p}(t_j, d_{model}) = \begin{bmatrix} \sin(\omega_1 t_j) \\ \cos(\omega_1 t_j) \end{bmatrix} \tag{3.11}$$

which is a unit vector, since $|\mathbf{p}(t_j, d_{model})| = 1$.
From the last two equations it is clear that that the positional embeddings fall along a circle in the xy-plane, which is the red circle in the figure (3.2).

However, in reality the positional vector $\mathbf{p}(t_j, d_{model})$ is a d_{model}-dimensional vector. But our previous analysis is useful, because as we are going to prove that we can split this high dimensional vector into **multiple 2D-vectors**. We can re-write the expression for the shifted positional vector (3.2) using (3.8) as follows:

$$\begin{bmatrix} \mathbf{p}(t_1 + \phi, d_{model}) \\ \mathbf{p}(t_2 + \phi, d_{model}) \\ \vdots \\ \mathbf{p}(t_{d_{model}/2} + \phi, d_{model}) \end{bmatrix} =$$

$$\begin{bmatrix} \mathbf{R_1}(-\phi) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{R_2}(-\phi) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{R_{d_{model}/2}}(-\phi) \end{bmatrix} \begin{bmatrix} \mathbf{p}(t_1, d_{model}) \\ \mathbf{p}(t_2, d_{model}) \\ \vdots \\ \mathbf{p}(t_{d_{model}/2}, d_{model}) \end{bmatrix}$$

Hence:

$$\begin{cases} \mathbf{p}(t_1 + \phi, d_{model}) & = \mathbf{R_1}(-\phi)\mathbf{p}(t_1, d_{model}) \\ \mathbf{p}(t_2 + \phi, d_{model}) & = \mathbf{R_2}(-\phi)\mathbf{p}(t_2, d_{model}) \\ & \vdots \\ \mathbf{p}(t_{d_{model}/2} + \phi, d_{model}) & = \mathbf{R_{d_{model}/2}}(-\phi)\mathbf{p}(t_{d_{model}/2}, d_{model}) \end{cases}$$

where $\mathbf{p}(t_j + \phi, d_{model}), \mathbf{p}(t_j, d_{model}) \in \mathbb{R}^2 \quad \forall j = 1, \ldots, d_{model}/2$. Therefore the corresponding positional embedding vectors are:

$$\begin{cases} \mathbf{p_e}(t_1 + \phi, d_{model}) & = \mathbf{e} + \mathbf{R_1}(-\phi)\mathbf{p}(t_1, d_{model}) \\ \mathbf{p_e}(t_2 + \phi, d_{model}) & = \mathbf{e} + \mathbf{R_2}(-\phi)\mathbf{p}(t_2, d_{model}) \\ & \vdots \\ \mathbf{p_e}(t_{d_{model}/2} + \phi, d_{model}) & = \mathbf{e} + \mathbf{R_{d_{model}/2}}(-\phi)\mathbf{p}(t_{d_{model}/2}, d_{model}) \end{cases}$$

Notice that for the last equation to hold the positional vectors $\mathbf{p}(t_j, d_{model}) \quad \forall j = 1, \ldots, d_{model}/2$ have to be padded with $(d_{model} - 2)$ zeros, because as we have already mentioned $\mathbf{e}$ is a d_{model}-dimensional vector.

Therefore what we end up having is this: **the embedding vector $\mathbf{e}$ determines the center of a circle in the d_{model}-dimensional space. Then, the position of a word is determined by $d_{model}/2$ 2D-vectors, each one rotating clockwise with different frequency and independently to the others, on a hyperplane, which is parallel to the xy-plane.**

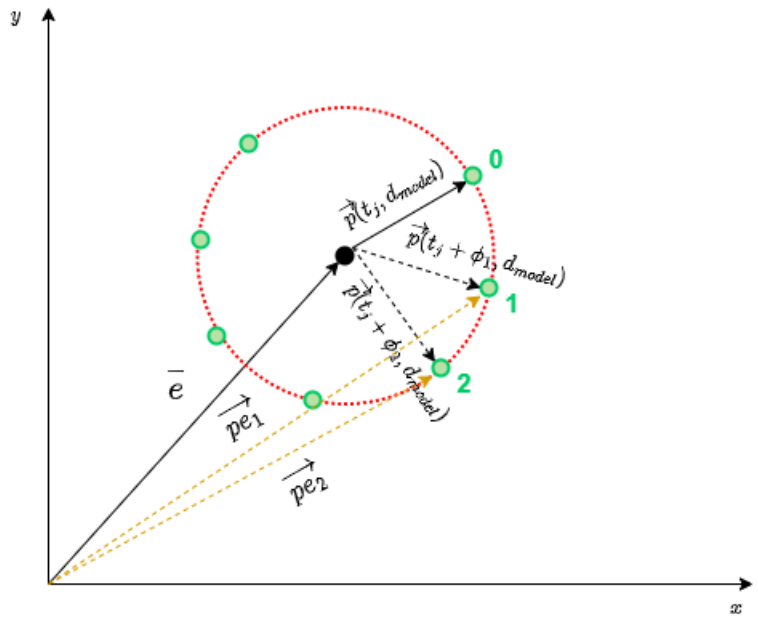


Figure 3.2: Positional Embeddings - 2D: this figure illustrates how the positional embedding vectors change as the position of the word x_j changes inside the sentence. Notice that all positional embeddings fall along the red circle.

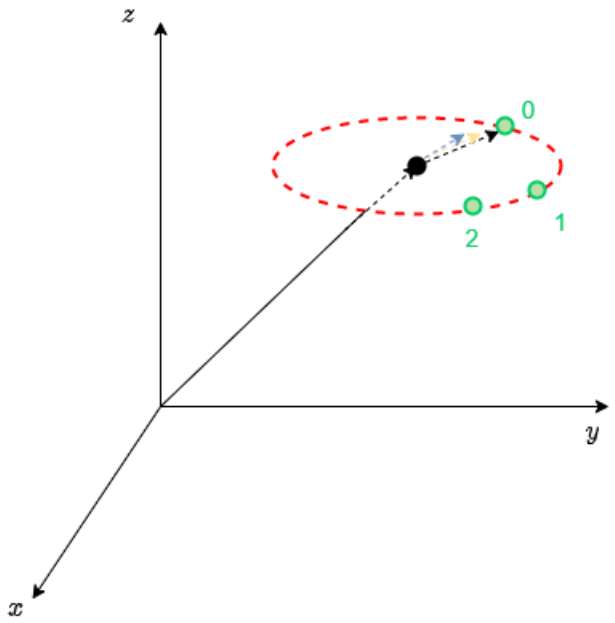


Figure 3.3: Positional Embeddings - 3D: this figure illustrates how the positional embedding vectors change as the position of the word x_j changes inside the sentence. Notice that all positional embeddings fall along the red circle, which is parallel to the xy-plane. Furthermore, at the position 0 there are 3 vectors with different colors, indicating that they are rotating with different frequencies, like the pointers of the common clocks. Although drawn with different sizes, in reality the 3 vectors have the same length, which is equal to 1.

3.3 The Transformer Block

Now that we have embedded our words and have found a way to uniquely and efficiently determine their position in a particular sentence it's time to investigate more advanced concepts behind the architecture of the Transformer. Self-attention is one such concept and more specifically it's an integral part of the the whole structure of the Transformer. The reason for that can be understood if we first define what it is. In this particular context, Self-Attention is the ability of a sentence to pay attention to itself. In other words, Self-Attention is a "measure of awareness" of each word about its context. Or we could say that it is a mechanism that quantifies how much dependency there is between the words of the same sentence.

3.3.1 Why is Self-Attention needed?

The main problem of words is that without context they are ambiguous, which means that without knowing what precedes or what follows them in many cases it is difficult to figure out what their proper meaning is. Suppose that we have the following sentence:

"Transformers is a very cool machine-learning model, because it very efficient"

Referring to this sentence, we can easily find examples of word ambiguity. For example, the word "cool" with no context is ambiguous, because it could also used to describe the weather. The same happens with the word "machine", which here goes with the word "learning". Therefore, the reason we would like the input sentence to pay attention to itself is **word disambiguation**.

From this we can derive that a machine which pays attention to its input, can use this information to answer questions about the input, where these answers appear as feature vectors. As an example, let's consider the following sentence:

"George drove to the bank"

- suppose the word "bank" pays attention to the word "drove". This might answer the question: "How did George arrive to the bank?".
- perhaps the word "bank" pays attention to the word "George". This might answer the question: "Who was at the bank?".
- maybe "George" pays attention to the word "bank", which may answer the question "Where did George go?".

It becomes clear that using Self-Attention helps our machine to answer a lot of different questions. This is very similar with Convolutional Neural Networks (CNNs), where we are looking for certain features in an image, like "Is there an eye in this image?" or "Is there a nose?". These questions, if properly answered, can lead to important conclusions, such as: "This is a face" or "This is not a face".

3.3.2 How Self-Attention works

First, we are going to start with a basic form of Self-Attention. In order to do that, let's review how the attention works in Seq2Seq RNNs. The main thing we are

trying to compute in RNNs with attention is the **context vector** $\mathbf{c}(t)$, which is the input to the decoder. Looking at the context vector more closely, we see that it is simply a weighted sum of the hidden states computed on the input sequence:

$$\mathbf{c}(t) = \sum_{t'=1}^{T_x} w(t, t')\mathbf{h}(t') \tag{3.12}$$

For each output timestep t we want to know which input timestep t' to pay attention to. This is indicated by the **attention weights** $w(t, t')$, which are computed as follows:

$$w(t, t') = \frac{e^{\tilde{w}(t,t')}}{\sum_{\tau=1}^{T_x} e^{\tilde{w}(t,\tau)}} \tag{3.13}$$

where:

$$\tilde{w}(t, t') = ANN(\mathbf{s}(t-1), \mathbf{h}(t')) \tag{3.14}$$

So, with this background in place, we can now begin to understand the mechanism of Self-Attention. Suppose that **after positional encoding** we have a sequence of feature vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_T}$ which correspond to the T input tokens. We would like to compute a new context vector, which is the weighted sum of these vectors. In practice, we are going to compute a context or **attention vector** for each of the input tokens, so we will have $\alpha_\mathbf{1}, \alpha_\mathbf{2}, \ldots, \alpha_\mathbf{T}$. Since the computation for each input token is the same, we can consider just one of these attention vectors $\alpha_\mathbf{i}$:

$$\alpha_\mathbf{i} \triangleq \sum_{j=1}^{T} w(i, j)\mathbf{x_j} \quad i = 1, \ldots, T \tag{3.15}$$

where:

$$\mathbf{x_j} \in \mathbb{R}^{d_{model}} \quad \forall j = 1, \ldots, T \tag{3.16}$$

the positional encoding vectors. Note that we changed the indexes from (t, t') to (i, j), because this tends to be the convention when describing Self-Attention. Also it's important to remember that, that in (3.15) the positional encoding vectors $\mathbf{x_j}$ are not "context aware", but only "position aware". For instance, if we have the following two sentences:

"The Transformer model is cool"
"The weather is cool"

the word "cool" would be mapped to the exactly the same positional encoding vector, because in both cases it lies in the fourth position in the sentence. However, we know that its meaning is different in both cases.

On the other hand, α's are context aware embeddings, because they take into account all the input tokens. As we can see from (3.15) the way we compute α's is similar to how we compute the context vector in attention for Seq2Seq RNNs. The same holds true for $\mathbf{w}$'s, which are computed as follows:

$$w(i,j) \triangleq softmax\left(\langle \mathbf{x_i}, \mathbf{x_j} \rangle\right) = \frac{e^{\mathbf{x_i}^T \mathbf{x_j}}}{\sum_{j'=1}^{T} e^{\mathbf{x_i}^T \mathbf{x_{j'}}}} \quad i,j = 1,\ldots,T \tag{3.17}$$

From the definition of the attention weights (3.17) we observe that they exclusively depend on the **inner products** between the positional embedding vectors. Because the inner product expresses the **similarity** between two vectors, we can therefore come to conclusion that w's contain the information about the dependencies between the input tokens. Furthermore, we apply the $softmax$ function to these inner products in order to normalize them.

However, there is still a problem with (3.17). In this expression the a's depend only on the input positional embeddings themselves. This means that for fixed input these attention weights will remain unchanged. In other words, there are **no learnable parameters**. So, our aim in the next section is to further parameterize the expression for the attention weights, which will render the Self-Attention mechanism more flexible and tunable for various tasks.

3.3.3 Scaled Dot-Product Attention

In order to incorporate some learnable parameters into equation (3.17), let's introduce the square matrix $\mathbf{P} \in \mathbb{R}^{d_{model} \times d_{model}}$, so as:

$$w(i,j) \triangleq \frac{e^{\mathbf{x_i}^T \mathbf{P} \mathbf{x_j}}}{\sum_{j'=1}^{T} e^{\mathbf{x_i}^T \mathbf{P} \mathbf{x_{j'}}}} \quad i,j = 1,\ldots,T \tag{3.18}$$

But from linear algebra we know that any real and square matrix, such as $\mathbf{P}$, can be **uniquely** decomposed into two matrices: an orthogonal one and lower triangular one, as follows:

$$\mathbf{P} = \mathbf{LO} \tag{3.19}$$

where:

- $\mathbf{L} \in \mathbb{R}^{d_{model} \times d_{model}}$ is a **lower triangular** matrix
- $\mathbf{O} \in \mathbb{R}^{d_{model} \times d_{model}}$ is an **orthogonal** matrix

Therefore, we can write:

$$\mathbf{x_i}^T \mathbf{P} \mathbf{x_j} \overset{(3.19)}{=} \mathbf{x_i}^T \mathbf{L} \mathbf{O} \mathbf{x_j} = \left(\mathbf{L}^T \mathbf{x_i}\right)^T \mathbf{O} \mathbf{x_j} = \left(\mathbf{U} \mathbf{x_i}\right)^T \mathbf{O} \mathbf{x_j} = \mathbf{q_i}^T \mathbf{k_j} \quad i, j = 1, \ldots, T \tag{3.20}$$

where:

$$\mathbf{U} \in \mathbb{R}^{d_{model} \times d_{model}}$$

is a **upper triangular** matrix.

$$\mathbf{q_i} \triangleq \mathbf{U} \mathbf{x_i}, \quad i = 1, \ldots, T \tag{3.21}$$

is called the **query** vector.

$$\mathbf{k_j} \triangleq \mathbf{O} \mathbf{x_j}, \quad j = 1, \ldots, T \tag{3.22}$$

is called the **key** vector.

The inspiration for these names comes from the databases, because when we have a database we often want to query it to obtain the desired data. This means that we search for (query the database) specific data (values) in the database based on the appropriate keys. The database will return the data for which the query and the key match.

Before inspecting more closely why these vectors might be called this way, let's first start from what we have. If we suppose that our input sequence constitutes of T words, then the positional embedding matrix of this sequence is the following:

$$\mathbf{X} = \overbrace{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d_{model}} \\ x_{21} & x_{22} & \cdots & x_{2d_{model}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T1} & x_{T2} & \cdots & x_{Td_{model}} \end{bmatrix}}^{\text{latent feature dimension}} = \begin{bmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \vdots \\ \mathbf{x_T}^T \end{bmatrix}$$

As we can see, each row of $\mathbf{X}$ corresponds to a word and each column to a latent feature. Therefore the element x_{ij} quantifies the "amount" that the latent feature j

"participates" in word i. Although, what the latent features represent is probably something that only the model understands, for our convenience let's assume that some of them are comprehensible by humans. For example, suppose that three latent features are the following:

- what part-of-speech the word is.
- how important, generally, the word is.
- generally, how much it belongs to a particular topic.

At this point it's important to notice two things about $\mathbf{X}$:

1. a word is **uniquely** determined by its vector $\mathbf{x_i}$
2. since it is predetermined, its elements are fixed. Therefore, as it stands, our model is not allowed to change its values.

But in real language the values of $\mathbf{X}$ are not fixed. In fact, the **meaning** of the words changes from sentence to sentence. As a consequence, there is also a change in the amount that a certain word "belongs" to a particular latent feature. That's why in the latent feature examples above, the word "generally" was used to indicate that the values of $\mathbf{X}$ give us the general picture about the connection between a word and a latent feature and not a specific one, which depends on the current sentence. Therefore, it would be a good idea to modify $\mathbf{X}$, so that our model is capable of monitoring these changes in word meaning. Suppose, that we somehow transform $\mathbf{X}$ into a new matrix $\mathbf{V} \in \mathbb{R}^{T \times d_v}$. In fact if we define the function $f_t : \mathbb{R}^{T \times d_{model}} \to \mathbb{R}^{T \times d_v}$:

$$\mathbf{V}(s_t) = f_t(\mathbf{X}) = \overbrace{\begin{bmatrix} v_{11}(s_t) & v_{12}(s_t) & \cdots & v_{1d_v}(s_t) \\ v_{21}(s_t) & v_{22}(s_t) & \cdots & v_{2d_v}(s_t) \\ \vdots & \vdots & \ddots & \vdots \\ v_{T1}(s_t) & x_{T2}(s_t) & \cdots & v_{Td_v}(s_t) \end{bmatrix}}^{\textbf{new} \text{ latent feature dimension}} = \begin{bmatrix} \mathbf{v_1}^T(s_t) \\ \mathbf{v_2}^T(s_t) \\ \vdots \\ \mathbf{v_T}^T(s_t) \end{bmatrix}$$

Notice that all elements of $\mathbf{V}$ are a function of the current sentence s_t and that through this transformation our model also tries to find some new latent features, which may better represent our data. Hence, the new latent feature dimension is $d_v \neq d_{model}$. Moreover, since a word can have multiple nuances and meanings

according to the sentence it appears in, the corresponding vector $\mathbf{v_i}$ changes as well. Then what happens is that many different $\mathbf{v_i}$ vectors can correspond to the same word.

Now, let's try to give an explanation for why we might call these vectors query and key. Firstly, as far as the **key vector** is concerned, recall that the matrix $\mathbf{O}$ is orthogonal. This means that:

$$\mathbf{O}\mathbf{O}^T = \mathbf{I} \tag{3.23}$$

where:

- $\mathbf{I} \in \mathbb{R}^{d_{model} \times d_{model}}$ is the identity matrix.
- $\mathbf{O} = \begin{bmatrix} o_{11} & o_{12} & \cdots & o_{1d_{model}} \\ o_{21} & o_{22} & \cdots & o_{2d_{model}} \\ \vdots & \vdots & \ddots & \vdots \\ o_{T1} & o_{T2} & \cdots & o_{Td_{model}} \end{bmatrix} = \begin{bmatrix} \mathbf{o_1}^T \\ \mathbf{o_2}^T \\ \vdots \\ \mathbf{o_T}^T \end{bmatrix}$

In other words, $\mathbf{O}$'s **rows** are orthogonal unit vectors and form an **orthonormal basis** in $\mathbb{R}^{d_{model}}$. Taking that into account, from (3.30) we get:

$$\mathbf{k_j} = \mathbf{O}\mathbf{x_j} = \begin{bmatrix} \mathbf{o_1}^T \cdot \mathbf{x_j} \\ \mathbf{o_2}^T \cdot \mathbf{x_j} \\ \vdots \\ \mathbf{o_T}^T \cdot \mathbf{x_j} \end{bmatrix} = (\mathbf{o_1}^T \cdot \mathbf{x_j})\mathbf{o_1} + (\mathbf{o_2}^T \cdot \mathbf{x_j})\mathbf{o_2} + \cdots + (\mathbf{o_T}^T \cdot \mathbf{x_j})\mathbf{o_T} \tag{3.24}$$

Therefore, any vector, that belongs to this high dimensional space, can be written in a unique way as linear combination of the row vectors of $\mathbf{O}$. That means, $\mathbf{O}$'s rows function like the keys in database, because in the same way the keys are unique for a database, likewise the row vectors of $\mathbf{O}$ are in a way unique in the d_{model}-dimensional space.

Secondly, a vector, in order to be a **query vector**, must satisfy the following conditions:

1. it must be a function of the keys exclusively.
2. the function must be unique.
3. the function must be deterministic.

The criteria defined above, are derived from the fact that any query in a properly organized database can be answered by a unique and non-stochastic combination of its keys. From the definition of the key vector (3.30) we get:

$$\mathbf{k_j} = \mathbf{O}\mathbf{x_j} \Leftrightarrow \mathbf{O}^T\mathbf{k_j} = \mathbf{O}^T\mathbf{O}\mathbf{x_j} \overset{(3.23)}{\Longleftrightarrow} \mathbf{x_j} = \mathbf{O}^T\mathbf{k_j} \quad j = 1, \ldots, T \tag{3.25}$$

Hence, from (3.31) we get:

$$\mathbf{q_i} = \mathbf{U}\mathbf{x_i} \overset{(3.25)}{\Longleftrightarrow} \mathbf{q_i} = \mathbf{U}\mathbf{O}^T\mathbf{k_i}, \quad i = 1, \ldots, T \tag{3.26}$$

From (3.26) we observe that:

- Since for a given real square matrix $\mathbf{P}$ the matrices $\mathbf{U}$ and $\mathbf{O}$ are unique, (3.26) indicates that there is a unique linear transformation from keys to queries.
- if $\mathbf{P}$ is chosen to be deterministic, then both $\mathbf{U}$ and $\mathbf{O}$ are deterministic.
- Apart from the scaling-rotation matrix $\mathbf{U}\mathbf{O}^T$, the query vector depends **only** on the key vector.
- in the case of multiple queries we have:

$$\mathbf{q} = \sum_i c_i\mathbf{q_i} = \sum_i c_i\mathbf{U}\mathbf{O}^T\mathbf{k_i} = \mathbf{U}\mathbf{O}^T\sum_i c_i\mathbf{k_i} = \mathbf{U}\mathbf{O}^T\mathbf{k} \tag{3.27}$$

  where $c_i \neq 0, \forall i$. Therefore, from (3.27), we can see that multiple queries can be "answered" in a unique and deterministic way by multiple keys.

Therefore, we have proven that the vector defined by (3.31) satisfies the above three conditions and therefore we could call it a query vector.

Moreover, starting from the definition of the query vector, we could write:

$$\begin{aligned}\mathbf{q_i} = \mathbf{U}\mathbf{x_i} &= \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1d_{model}} \\ 0 & u_{22} & \cdots & u_{2d_{model}} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{d_{model}d_{model}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d_{model}} \end{bmatrix} \\ &= \begin{bmatrix} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1d_{model}}x_{d_{model}} \\ u_{22}x_2 + \cdots + u_{2d_{model}}x_{d_{model}} \\ \vdots \\ u_{d_{model}d_{model}}x_{d_{model}} \end{bmatrix}\end{aligned}$$

From the above expression we observe that, if we assume that $u_{ii} \neq \forall i = 1, \ldots, d_{model}$, each element of the vector is a linear combination of the values of different latent features. In particular, the first element contains information from all latent features, the second element includes all latent features except for the first one and so on. Generally, the n_{th} element of the query vector contains information from the features $x_n, x_{n+1}, \ldots, x_{d_{model}}$. Therefore, each time one new latent feature is taken into account and the model tries to capture a different kind of information.

Perhaps we could say that $\mathbf{q_i}$ tries to incorporate all the possible different linear transformations of the input vector $\mathbf{x_i}$, making use of the parameterized matrix $\mathbf{U}$. In other words, $\mathbf{q_i}$ attempts to "ask all possible questions" given a particular word in the d_{model}-dimensional space. In the language of databases, this translates to trying to make all the possible queries, on a particular characteristic/feature.

However, the previous analysis is for intuition only. In reality, since $\mathbf{P}$ is a matrix to be learned by the model itself, it is not necessary that it should be split according to (3.19) into an orthogonal and a lower triangular matrix. Also notice that (3.19) gives one possible factorization of matrix $\mathbf{P}$ and NOT the ONLY one.

Now, let's assume the real case, where the parameterization matrix $\mathbf{P}$ is split into two weight matrices $\mathbf{W^{(Q)}}$ and $\mathbf{W^{(K)}}$ as follows:

$$\mathbf{P} = \mathbf{W^{(Q)}}\mathbf{W^{(K)}}^T \tag{3.28}$$

where $\mathbf{W^{(Q)}}, \mathbf{W^{(K)}} \in \mathbb{R}^{d_{model} \times d_k}$. Then:

$$\mathbf{x_i}^T\mathbf{P}\mathbf{x_j} \overset{(3.28)}{=} \mathbf{x_i}^T\mathbf{W^{(Q)}}\mathbf{W^{(K)}}^T\mathbf{x_j} = \left(\mathbf{W^{(Q)}}^T\mathbf{x_i}\right)^T \mathbf{W^{(K)}}^T\mathbf{x_j} = \mathbf{q_i}^T\mathbf{k_j} \quad i,j = 1, \ldots, T \tag{3.29}$$

where:

$$\mathbf{q_i} \triangleq \mathbf{W^{(Q)}}^T\mathbf{x_i} \in \mathbb{R}^{d_k} \quad i = 1, \ldots, d_{model} \tag{3.30}$$

the query vector and:

$$\mathbf{k_j} \triangleq \mathbf{W^{(K)}}^T\mathbf{x_j} \in \mathbb{R}^{d_k} \quad j = 1, \ldots, d_{model} \tag{3.31}$$

the key vector. Therefore, because of (3.29) the attention weights in (3.18) can be written as:

$$w(i,j) = \frac{e^{\mathbf{q_i}^T \mathbf{k_j}}}{\sum_{j'=1}^{T} e^{\mathbf{q_i}^T \mathbf{k_{j'}}}} = softmax\left(\langle \mathbf{q_i}, \mathbf{k_j} \rangle\right) \quad i,j = 1,\ldots,T \tag{3.32}$$

But there is a problem with equation (3.32), that has to be considered, which is that the dot product may get quite large values. To illustrate this point, let's assume that the components of the vectors $\mathbf{q_i}$ and $\mathbf{k_j}$ are independent random variable with 0 mean and variance 1. In other words, if $\mathbb{E}[\cdot]$ is the expected value of a random variable, then:

- $\mathbb{E}\left[q_i^{(\xi)}\right] = \mathbb{E}\left[k_j^{(\xi)}\right] = 0$
- $Var\left[q_i^{(\xi)}\right] = \mathbb{E}\left[\left(q_i^{(\xi)}\right)^2 - \mathbb{E}\left[q_i^{(\xi)}\right]^2\right] = \mathbb{E}\left[\left(q_i^{(\xi)}\right)^2\right] = 1$
- $Var\left[k_j^{(\xi)}\right] = \mathbb{E}\left[\left(k_j^{(\xi)}\right)^2 - \mathbb{E}\left[k_j^{(\xi)}\right]^2\right] = \mathbb{E}\left[\left(k_j^{(\xi)}\right)^2\right] = 1$
- $\mathbb{E}\left[q_i^{(\xi)} k_j^{(\xi)}\right] = \mathbb{E}\left[q_i^{(\xi)}\right]\mathbb{E}\left[k_j^{(\xi)}\right] = 0$, since we have assumed independency between the components of the two vectors

But if we assume that the components of each vector are independent, then:

- $\mathbb{E}\left[q_i^{(\xi)} q_i^{(\xi')}\right] = \mathbb{E}\left[q_i^{(\xi)}\right]\mathbb{E}\left[q_i^{(\xi')}\right] = 0$, when $\xi \neq \xi'$
- $\mathbb{E}\left[k_j^{(\xi)} k_j^{(\xi')}\right] = \mathbb{E}\left[k_j^{(\xi)}\right]\mathbb{E}\left[k_j^{(\xi')}\right] = 0$, when $\xi \neq \xi'$

Then the inner product between the query and the key vectors can also be treated as a random variable and is given from the following expression:

$$\langle \mathbf{q_i}, \mathbf{k_j} \rangle = \sum_{\xi} q_i^{(\xi)} k_j^{(\xi)} \tag{3.33}$$

Then the mean of the dot product is:

$$\mathbb{E}\left[\langle \mathbf{q_i}, \mathbf{k_j} \rangle\right] = \mathbb{E}\left[\sum_{\xi} q_i^{(\xi)} k_j^{(\xi)}\right] = \sum_{\xi} \mathbb{E}\left[q_i^{(\xi)} k_j^{(\xi)}\right] = \sum_{\xi} \mathbb{E}\left[q_i^{(\xi)}\right]\mathbb{E}\left[k_j^{(\xi)}\right] = 0$$

and its variance:

$$
\begin{aligned}
Var\left[\langle \mathbf{q_i}, \mathbf{k_j} \rangle\right] &= \mathbb{E}\left[\left(\sum_{\xi} q_i^{(\xi)} k_j^{(\xi)}\right)^2\right] = \\
&= \mathbb{E}\left[\sum_{\xi}\left(q_i^{(\xi)}\right)^2\left(k_j^{(\xi)}\right)^2 + \sum_{\xi}\sum_{\xi' \neq \xi} q_i^{(\xi)} q_i^{(\xi')} k_j^{(\xi)} k_j^{(\xi')}\right] = \\
&= \sum_{\xi}\mathbb{E}\left[\left(q_i^{(\xi)}\right)^2\right]\mathbb{E}\left[\left(k_j^{(\xi)}\right)^2\right] + \\
&\quad + \sum_{\xi}\sum_{\xi' \neq \xi}\mathbb{E}\left[q_i^{(\xi)}\right]\mathbb{E}\left[q_i^{(\xi')}\right]\mathbb{E}\left[k_j^{(\xi)}\right]\mathbb{E}\left[k_j^{(\xi')}\right] = \\
&= \sum_{\xi=0}^{d_k - 1} 1 = d_k \Leftrightarrow \sigma\left[\langle \mathbf{q_i}, \mathbf{k_j} \rangle\right] = \sqrt{d_k}
\end{aligned}
$$

What we have shown through the above analysis is that the mean of the dot product between the query and the key vector is 0 while its variance, under some assumptions is equal to their size. What researchers observed is that for large values of d_k, the inner products grows large in magnitude and as a consequence pushes the softmax function into its asymptotic regions, where it has extremely small gradients. Since gradients are the essential part of the learning process, via the backpropagation algorithm, the result of this important observation is that it makes the training process much slower. In order to mitigate the effects of this problem the authors of the "Attention Is All You Need Paper" decided to divide the inner product by its standard deviation $\sqrt{d_k}$ **before** applying the softmax function, which the reason why this is called **scaled dot product attention**. Thus, (3.32) becomes:

$$
w(i,j) = softmax\left(\frac{\langle \mathbf{q_i}, \mathbf{k_j} \rangle}{\sqrt{d_k}}\right) \quad i,j = 1, \ldots, T \tag{3.34}
$$

Since the intuitive concept behind introducing learnable parameters in Self-Attention is to transform all the input vectors, that is all $\mathbf{x}$'s into their respective role as queries and keys, let's also define their corresponding **value** vector:

$$
\mathbf{v_i} \triangleq \mathbf{W^{(V)}}^T \mathbf{x_i} \in \mathbb{R}^{d_v} \quad i = 1, \ldots, d_{model} \tag{3.35}
$$

where $\mathbf{W^{(V)}} \in \mathbb{R}^{d_{model} \times d_v}$. Therefore the attention vectors given in (3.15) can be modified as follows:

$$\mathbf{W}^{(\mathbf{V})^T}\alpha_{\mathbf{i}} = \sum_{j=1}^{T} w(i,j)\mathbf{W}^{(\mathbf{V})^T}\mathbf{x_j} \overset{(3.19)}{\Longleftrightarrow}$$

$$\alpha_{\mathbf{i}}{}' = \sum_{j=1}^{T} w(i,j)\mathbf{v_j} \quad j = 1, \ldots, d_{model} \tag{3.36}$$

3.3.4 Example on Queries, Keys and Values

In order to make what we have achieved, through the previous analysis, more concrete, let's consider the following input sentence:

"Transformers is a very cool machine-learning model, because it is very efficient"

This is a useful example, because the word "cool" can have multiple meanings. So "cool" will correspond to our query vector $\mathbf{q_i}$. Our model wants to know in what sense do we mean the word "cool". In order to do that we might want to pay attention to other words in the same sentence, such as "Transformers" and "model". In order to compute these attention weights, we are going to calculate all the inner products between this query vector and all key vectors: $\mathbf{k_j}, j = 1, \ldots, T$:

$$\tilde{w}(i,j) = \langle \mathbf{q_i}, \mathbf{k_j} \rangle \quad i, j = 1, \ldots, T$$

For $\mathbf{q_i}$ fixed, the above expression will give us T attention scores. Then we find the attention weights by normalizing these attention scores, using the softmax function (3.32). Then these weights are used to form a weighted sum of the values $\mathbf{v_j}$, and thus we find the attention vector that corresponds to the word that interests us using (3.36). Overall, we have to calculate T attention vectors, one for each input word.

Now, in order to make the above procedure less abstract let's approach it from its geometrical point of view, instead of its equations. First of all, we can imagine a vector space where all the key vectors (blue) lie. Note that there will be a key vector for every word in the input sentence, but only some of them are shown in figure (3.4). The red vector represents our query, which is the word "cool". Importantly, note that the same word can be both query and key, since it's possible for a word to pay attention to itself. That's why "cool' can show up twice, once as

a key and once as a query.

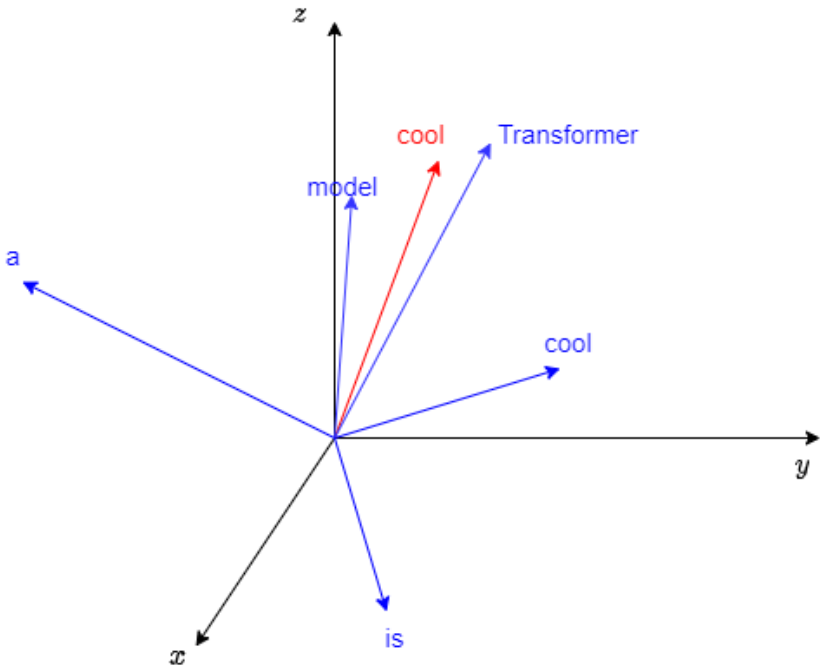


Figure 3.4: 3D-example of Keys and Queries

What we are actually trying to do is find which key matches best with our query, which in vector space could be translated as the key vector that yields the largest dot product with our query vector. In this case, that would be the word “Transformer”. Indeed, as we can see, “cool” and “Transformer” are very close together, hence they are similar and would be considered the best match, which makes sense, since in this particular sentence the word “cool” refers to the word “Transformer”. Furthermore, note that the query vector for the word “cool” is close to the key vector of the word “model”, which is also logical, since “Transformer” is a model.

So, now we can think of what the attention weights might be. Intuitively, the inner product between the query and the keys other than “Transformer” or “model” will either be small or negative, in either case insignificant. Whereas the dot product between the query and “Transformer” or “model” will probably be large

and positive. Moreover, when we apply the softmax to these numbers, it will give us a number close to 1 for the inputs that are large relative to the others. Therefore, since after finding the attention weights, the attention output vector is the weighted sum of the value vectors weighted by these weights, in our case we will most probably end up having something like:

$$\mathbf{a'}_{cool} = w(cool, Transformer)\mathbf{v}_{Transformer} + w(cool, model)\mathbf{v}_{model} + \\ + \sum_{other} w(cool, other)\mathbf{v}_{other}$$

where:

$$w(cool, Transformer), w(cool, model) \gg w(cool, other) \quad \forall other \in vocabulary$$

because the softmax function makes the big differences even bigger. Therefore the above expression for the attention output vector can be approximated as follows:

$$\mathbf{a'}_{cool} \cong w(cool, Transformer)\mathbf{v}_{Transformer} + w(cool, model)\mathbf{v}_{model}$$

To conclude this example, we saw the role that the query and key vectors play from a geometrical perspective and we examined the way the softmax distributes the attention weights over the input words. In particular, we came to the conclusion, that although it doesn't ignore any of the input words, it tends to choose much bigger weights for the most important ones.

3.3.5 The full-vectorized Scaled Dot-Product Attention

The problem with equation (3.34) is that we have to calculate the inner product for all possible combinations of query and key vectors, that is $O(T^2)$, since T is the length of input sequence. But, because the attention weights **depend only on the dot product operation** we could possibly write equation (3.34) using matrix multiplications, which are nothing else but a series of inner products. This way we can benefit from the faster algorithms that underlie these mathematical operations.

We can do this following the next procedure. First let's define the matrix:

$$\mathbf{X} \in \mathbb{R}^{T \times d_{model}} \tag{3.37}$$

which contains all the input positional embedding vectors. That is, each row of X corresponds to one positional embedding vector from the input sequence. Then we can write the queries, keys and values in vectorized form as follows:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathbf{Q})} \in \mathbb{R}^{T \times d_k} \tag{3.38}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathbf{K})} \in \mathbb{R}^{T \times d_k} \tag{3.39}$$

$$\mathbf{X} = \mathbf{X}\mathbf{W}^{(\mathbf{V})} \in \mathbb{R}^{T \times d_v} \tag{3.40}$$

because: $\mathbf{W}^{(\mathbf{Q})}, \mathbf{W}^{(\mathbf{K})} \in \mathbb{R}^{d_{model} \times d_k}$ and $\mathbf{W}^{(\mathbf{V})} \in \mathbb{R}^{d_{model} \times d_v}$. Based on (3.38), (3.39) and (3.40), (3.34) can be further vectorized in the following manner:

$$\mathbf{W} = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \tag{3.41}$$

where $\mathbf{W} \in \mathbb{R}^{T \times T}$. Mathematically, $\mathbf{Q}\mathbf{K}^T$ means that we multiply every row of $\mathbf{Q}$ by every row of $\mathbf{K}$. These rows are different query and key vectors for different input tokens, which makes sense, since for every query and key vector we have T attention weights, one for each input, making up $T \times T$ attention weights in total. Also, note that after we apply the softmax to this matrix, the dimensions do not change. Furthermore, using (3.40) and (3.41) we can further vectorize (3.36):

$$\mathbf{A} = \mathbf{W}\mathbf{V} = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{3.42}$$

What we have achieved thanks to equations (3.37)-(3.43) is that we calculate the queries, keys and values **for all input positional embedding vectors simultaneously** and this is true for the attention weights $\mathbf{a}$ and the attention matrix $\mathbf{A}$. Finally, note that in code we further vectorize the attention computation by handling multiple samples/sentences at once, So, instead of $T \times d_{model}$ input we have $N \times T \times d_{model}$.

To conclude, it's important to pinpoint the difference between Scaled Dot-Product Attention and Self-Attention. While the former is the attention operation which can act on **any** queries, keys and and values, the latter is where the queries, keys and and values are all computed on the same input sentence.

3.3.6 Self-Attention efficiency and comparison to RNNs

From the last section it is clear that every word in the input sentence will pay attention to some other word, possibly itself, in the same sentence. This means that

we must compute T^2 attention weights for each sentence, which grows fast as T increases. Eventually, the number of computations will get so large that, at some point, we will need to limit the size of the inputs.

However, attention is, basically, still superior to RNNs for the following two reasons:

1. Firstly, we should observe from (3.34) that to compute the attention weights for $\mathbf{q_i}$ we didn't have to consider any other query $\mathbf{q_j}, j \neq i$. This is a first hint that we could parallelize this computation. This is unlike RNNs, where for each hidden state $h(t)$, we must have finished computing the previous hidden state $h(t-1)$, since $h(t)$ depends on that value.

2. Another reason attention is superior to RNNs, is that they have trouble with long sequences. Indeed, due to the vanishing gradients, they have difficulty in remembering things too far in the past. Gated-RNNs, like GRUs or LSTMs, were designed to fix this issue, but there is still a limit, where these advanced units stop performing well. In fact, this was demonstrated in the original RNN attention paper. There it was shown that, as the number of input tokens increases, RNNs without attention dropped their performance, whereas RNNs with attention maintained their performance.

Another interesting fact to consider is that both RNNs and Self-Attention are able to handle variable length sequences. On the on hand, in RNNs, this is possible, since each $h(t)$ depends on $h(t-1)$ and its weights don't depend on sequence length . On the other hand, this is also true for Self-Attention, because the size of the weights $\mathbf{W^{(Q)}}$, $\mathbf{W^{(K)}}$ and $\mathbf{W^{(V)}}$ doesn't depend on T.

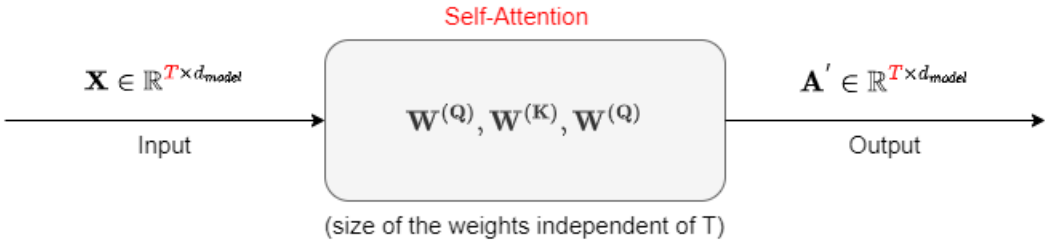


Figure 3.5: Self-Attention

Therefore, from figure (3.5), we can see that the Self-Attention box is capable of handling variable length sequences. Although, do note that, because we want to use matrix multiplications, every input in the same batch must be of the same size

3.3.7 Attention Mask

Masking is the final addition to the Self-Attention mechanism. There are two why we need the attention mask:

1. The first reason is because we need to pad our sequences, so that each sentence in a single batch has the same length. We want to do that, because then we can benefit from the fast matrix multiplication algorithms, since we will have full-vectorized forms. But when we introduce these pad tokens, we do not want our model to pay attention to them as they are irrelevant for understanding the sentence.

2. The second reason is mostly related to the decoder, which we are going to study in a subsequent section. However, the overall concept is the following: as we recall from the seq2seq model, the decoder is autoregressive, meaning that it only looks at past tokens in order to predict the next token. One way to accomplish this is to hide all the tokens beyond the last time step by using a mask. Essentially, what we want is to set the attention weights for irrelevant tokens, such as the padding ones, to be zero.

Here, it's important to pinpoint that the attention mask is applied **before the softmax** and not after, according to figure 3.6. The corresponding mathematical modification is shown in the following expression:

$$\mathbf{A} = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \circ \mathbf{M}\right)\mathbf{V} \tag{3.43}$$

where $\mathbf{M}$ is the attention mask matrix and $\circ$ denotes element-wise multiplication. In more detail, from what we have mentioned previously, in $\mathbf{M}$ we set:

- $m_{ij} = 1$, for locations, where we want to pay attention to.
- $m_{ij} = 0$, for locations, where we do not want to pay attention to.

The problem with the previous approach is that $softmax(0) = 1 \neq 0$ and thus it won't results in the corresponding attention being zero. The researchers resolved

this issue by setting $-\infty$ as the values of the attention mask matrix $\mathbf{M}$, where we do not want to pay attention to.

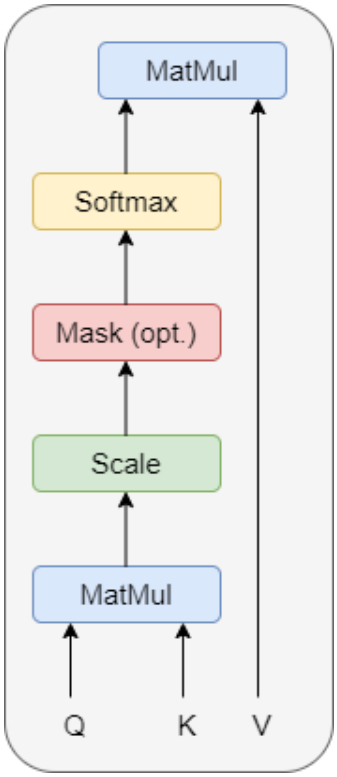


Figure 3.6: Scaled Dot-Product Attention

3.4 Multi-Head Attention

Now, that we have well established the concepts of Self-Attention and Scaled Dot-Product Attention, let's go one step further and introduce the idea behind Multi-Head Attention and why we need it. As it stands the Self-Attention attention mechanism only allows each token to pay attention to **some** of the rest tokens in the sequence, due to the softmax, which, as we have already pointed out, exaggerates the important relations and makes them **very important**.

Let's take for example the same sentence we had in the previous section:

"Transformers is a very cool machine-learning model, because it very efficient"

and let's assume that our model, through Self-Attention has learned that the most important words in this sentence that are related to the word "Transformers" are the words "machine" and "model" (see figure 3.7).

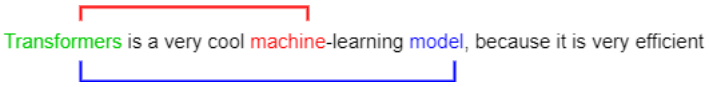


Figure 3.7: The word "Transformers" paying attention to the words "machine" and "model"

Although this is certainly a good and promising result about our model, it's still not quite enough, because we lose some of the information contained in this sentence. For instance our model has probably learned that "Transformers are a machine learning model", but it is not yet aware of its efficiency. We would like our model to be able to answer a question like: "Are Transformers an efficient machine learning model?". But in order to be able to correctly answer such a question it should learn to pay attention to the word "efficient" as it does for the words "machine" and "model"(see figure 3.8).

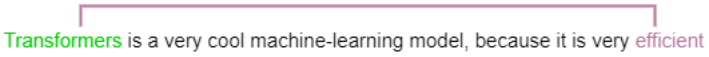


Figure 3.8: The word "Transformers" paying attention to the word "efficient"

In order to achieve the desired results, researchers thought about adding more attention layers, which will be able to **simultaneously** capture **multiple different dependencies** among the input tokens. For instance, one head would learn to connect the word "Transformers" with the word "model", another head would be responsible to connect it with the word"machine", while still another head would probably learn to connect "Transformers" with the word "efficient".

Specifically, suppose that we would like to compute the Self-Attention operation h-times. Each individual's Self-Attention output is called **head**. Therefore we would like to have h different sets of weights $\mathbf{W_i^{(Q)}}, \mathbf{W_i^{(K)}}, \mathbf{W_i^{(V)}}, i = 1, 2, \ldots, h$ as we can see in figure (3.9):

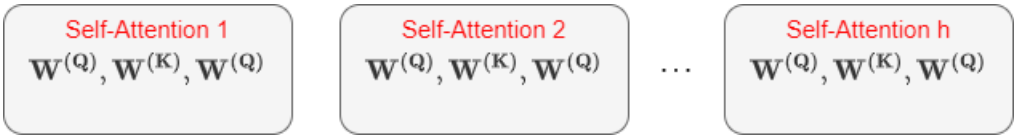


Figure 3.9: Multiple Self-Attention

After computing Self-Attention h-times, we still need a way to combine the results back into a single block of sequence data, that could be treated, somehow, like a single entity. The way researchers decided to approach this problem is to just **concatenate all the heads** along the feature dimension:

$$\mathbf{C} = Concat(\mathbf{A_1}, \mathbf{A_2}, \ldots, \mathbf{A_h}) \in \mathbb{R}^{T \times hd_v} \tag{3.44}$$

This is more clearly illustrated in figure (3.10).

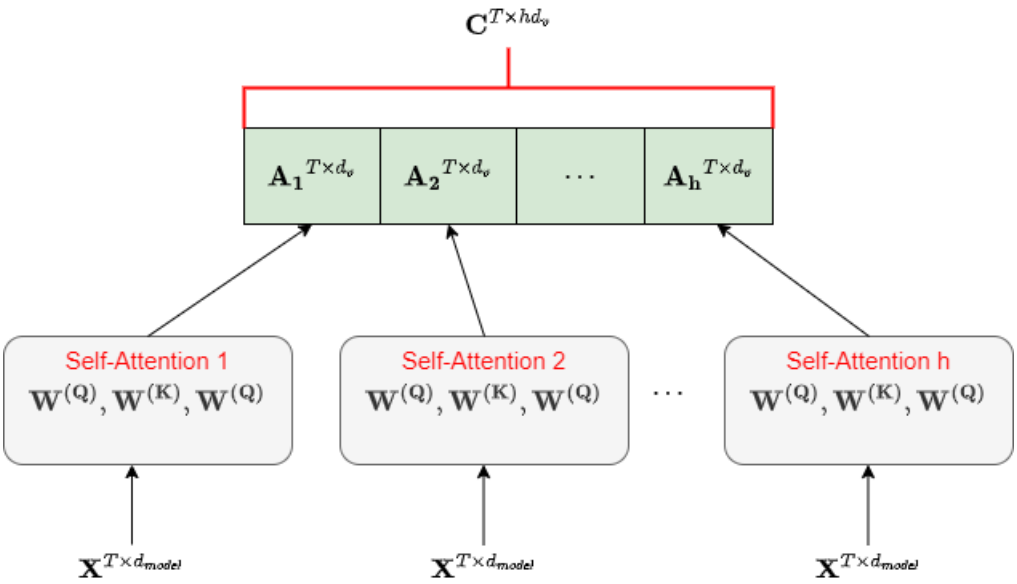


Figure 3.10: Concatenation of h Self-Attention Heads

One good analogy of the concatenation process in the Multi-Head Attention is with Convolutional Neural Networks (CNNs). In CNNs we use convolutional layers to produce features from images. For example, if a filter is shaped like an eye, then it will find eyes, whereas if it is shaped like an ear, it will find ears. But

in the case we would like to find both characteristics, we need to pass our image through multiple convolutional filters at once. Then, we concatenate the results, which become image representations of where these features were found.

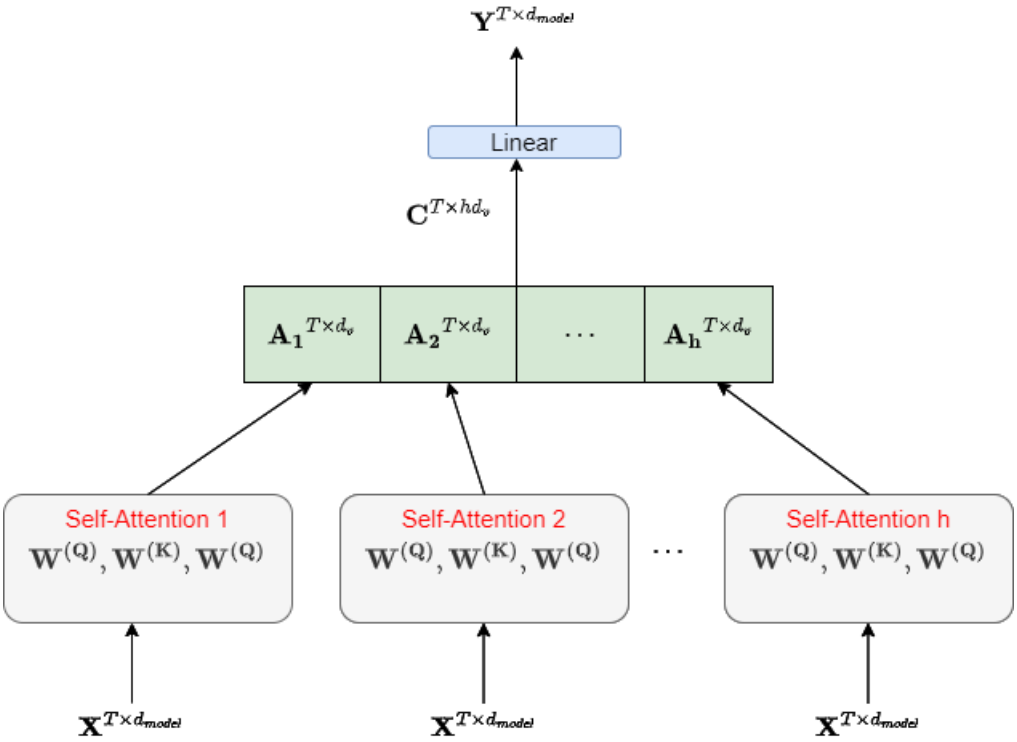


Figure 3.11: Multi-Head Attention

The last addition to the Multi-Head Attention is to **project** the concatenation matrix $\mathbf{C}$ back to the initial space of the input positional embedding matrix $\mathbf{X} \in \mathbb{R}^{T\times d_{model}}$. We do this by simply multiplying by an output weight matrix $\mathbf{W^o} \in \mathbb{R}^{hd_v\times d_{model}}$. Thus, we get:

$$\mathbf{Y} = \mathbf{CW^o} \in \mathbb{R}^{T\times d_{model}} \tag{3.45}$$

This way we have created a machine that produces **outputs of the same size as its inputs**. That's a crucial point in understanding the structure of Transformers. As we will see in a later sections, We can benefit from this important structural property by stacking many of these "blocks" together, without having to pay attention

each time to the input and output shapes. Therefore, we can easily create more complicated structures in order to achieve deeper representations of our input data and extract more valuable information.

3.5 The Feed-Forward layer

One important thing to notice up **until this point** is that **all transformations** applied to our input are **linear**. But only by linear transformations our machine can not be powerful enough to solve real world problems, because in most cases our available data require highly non-linear functions to extract the appropriate information. Therefore we **need to add non-linearity** to our existing structure. But we know that Feed-Forward Neural Networks (ANNs) are highly non-linear, which makes them capable of learning such rich representations of our data.

One way we can accomplish this is by passing the output of the Multi-Head Attention layer through an ANN. In the "Attention Is All You Need" paper an ANN was used with one hidden layer and a $ReLU$ activation in the middle, whereas no activation was used for the output. However, this is not the case for all Transformers.

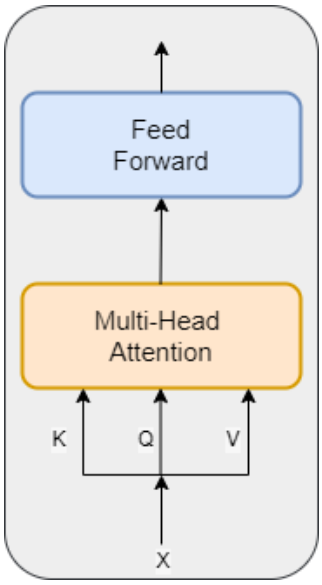


Figure 3.12: Adding an ANN to introduce non-linearity

3.6 Layer Normalization

The next block we are going to add is the normalization layer. As we can see in figure (3.13), this layer appears both after the Multi-Head Attention block and after the ANN block, so that eventually we have four sequential steps. The purpose of the of the Layer Normalization block is similar to the batch normalization. Actually the only but important difference is that while the latter implements normalization across the samples, the former normalizes across the features.

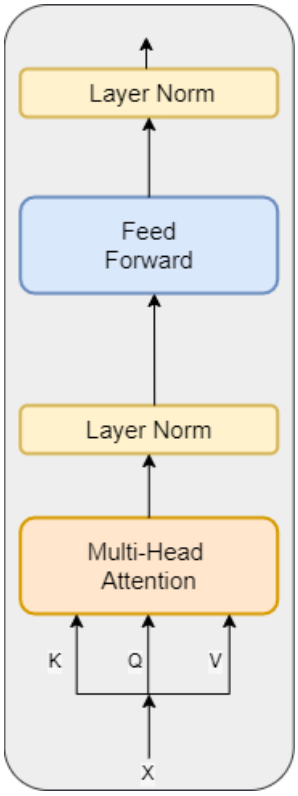


Figure 3.13: Adding Layer Normalization

In particular, what this layer does, can be described by the following equation:

$$\mathbf{x}_{norm} = g\frac{\mathbf{x} - \mu}{\sigma} + b \tag{3.46}$$

where:

- $\mathbf{x}$: the feature vector across which we want to normalize.

- μ: the mean of the feature vector $\mathbf{x}$
- σ: the standard deviation of the feature vector $\mathbf{x}$
- g: the adaptive gain
- b: the adaptive bias
- $\mathbf{x}_{norm}$: the normalized output vector

Note that in (3.46) g and b are learnable parameters. What we actually try to achieve through this equation is the following: first we transform our feature vector into a standardized one, that is a vector with zero mean and standard deviation of 1, by subtracting its mean and dividing by its standard deviation:

$$\mathbf{x}_{st} = \frac{\mathbf{x} - \mu}{\sigma} \tag{3.47}$$

Then we multiply by g and add b:

$$\mathbf{x}_{norm} = g\mathbf{x}_{\mathbf{st}} + b \tag{3.48}$$

This way, we make final feature vector to have:

- $\mu_{\mathbf{x}_{norm}} = b$
- $\sigma_{\mathbf{x}_{norm}} = g$

The reason we do these transformations is because we know that for neural networks it helps to normalize our data before passing it to them. Furthermore, since g and b are learnable parameters, we not only normalize the data. We "re-normalize" it as well, through g and b, making them part of our model. This way we "ask" our model to find the best "re-normalization" of the normalized data, that is the most suitable values for g and b.

Generally, if the values of the input feature vector $\mathbf{x}$ are too small or too large, the performance of ANNs degrades. Hence, following the above preocedure keeps the data within a standard range of values. Moreover, instead of just performing this operation at the input, it helps to turn it into a layer, so that it can repeatedly be applied even between intermediate layers of the network.

On the other hand, the practical aspect of this block is to help ANNs train faster, because we manage to the input data at ranges where the gradients of our activation functions take their largest values.

3.7 Skip Connections

Skip connections are called "skip", because they skip some layer of the Transformer architecture and feed some other layer. **Their output is the same as their input**. As we can see from figure (3.1) skip connections are used in two places in the Transformer architecture:

1. the first place they are found in, is to transfer the positional embedding matrix $\mathbf{X}$, without changing it, to the "Add and Norm" layer.

2. the second place they are found in, is to transfer the input of the "Feed-Forward" layer, without changing it, to the second "Add and Norm" layer.

The **main reason** for using skip connection is to **deal with the problem of vanishing gradients**. Recall that in neural networks we have an objective function J that we want to minimize. To do that we using gradient descent. The update rule of gradient descent without momentum and learning rate a is given by the following equation:

$$\xi' \leftarrow \xi - a\frac{\partial J}{\partial \xi}, \quad \xi', \xi \in \mathbb{R} \tag{3.49}$$

In the above equation the partial derivative of our objective function with respect to the weights is calculated via the backpropagation algorithm, which iteratively optimizes these parameters making use of chain rule from calculus. In particular, from multi-variate calculus we know that if:

$$J = f(\mathbf{w^{(i)}}) = f(w_1^{(i)}, \ldots, w_D^{(i)})$$

and

$$w_j^{(i)} = g_j(\xi), \quad j = 1, \ldots, D$$

then

$$\frac{\partial J}{\partial \xi} = \sum_j \frac{\partial J}{\partial w_j^{(i)}} \frac{\partial w_j^{(i)}}{\partial \xi} \tag{3.50}$$

From equation (3.50), we can see that the partial derivative of the objective function with respect to a parameter ξ of our model is expressed as a sum of product of some other partial derivatives. But it's worth noticing that, in deep learning models, the magnitude of all these partial derivatives are often less than 1, that is:

$$\left|\frac{\partial J}{\partial w_j^{(i)}}\right| < 1, \qquad \left|\frac{\partial w_j^{(i)}}{\partial \xi}\right| < 1$$

We usually make that decision, because, in general, multiplication with absolute values less than 1 offers some sense of training stability, although this is not accompanied by a strict mathematical theorem.

However this choice comes with an serious drawback. As we go back to the initial layers of our neural network more terms are added to the multiplication stage of (3.50), inside the summation, which are less than 1, independent of the sign. As a result, **the gradient for the initial parameters of the neural network becomes really small, in fact close to zero**. Therefore, the parameters of the early layers are actually not updated at all, as (3.49) informs us.

In order to resolve this issue, in 2015 the concept of **skip connection** was introduced. This idea was first proposed for deep Convolutional Neural Networks but later it became part of most deep architectures. Intuitively, the main benefit from these connections is that they **provide an alternative path for the gradient, during the backpropagation**.

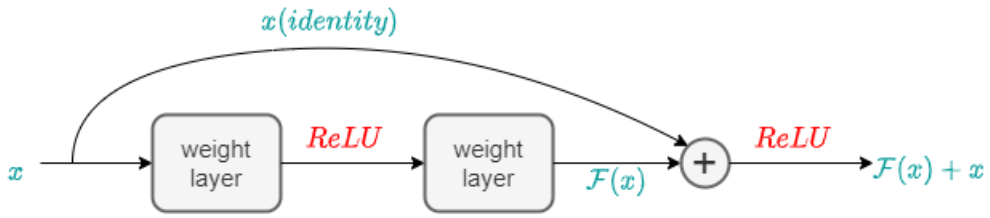


Figure 3.14: Skip/Residual Connenction

According to the figure above, we can calculate the partial derivative of our objective function like this:

$$\frac{\partial J(G(x))}{\partial x} = \frac{\partial J}{\partial G}\frac{\partial G}{\partial x} = \frac{\partial J}{\partial G}\left(\frac{\partial F(x)+x}{\partial x}\right) = \frac{\partial J}{\partial G}\left(\frac{\partial F(x)}{\partial x}+1\right) \tag{3.51}$$

Apart from the vanishing gradients there is another reason that skip connections are used. In particular, in many problems **we would like the information captured by the early layers to be passed as it is to the later layers, so that they can learn directly from it**. We do this, because, according to experiments, it was observed that the initial layers tend to learn feature that correspond to lower semantic information that is extracted from the input. If we had not used the skip connections, that information would have turned to abstract.

3.8 Encoder Architecture

The encoder architecture is presented in the following figure:

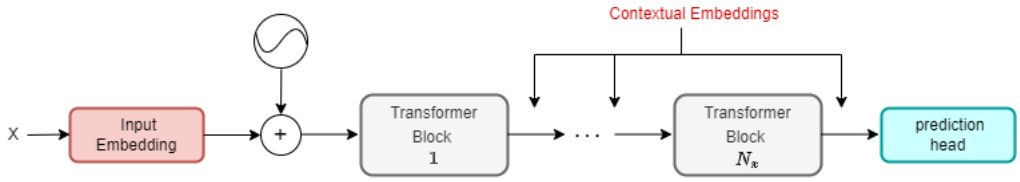


Figure 3.15: Encoder Architecture

First, let's suppose our input is a sequence of T word tokens ($T \times d_{vocab}$). Then we put this through an embedding layer. After the embedding layer ($T \times d_{model}$) we add the positional encodings ($T \times d_{model}$) and the output is of the same shape. Next, we pass the previous output through a whole series of Transformer blocks. But we know that these blocks do not change the shape of the data from input to output. So the input to every Transformer block is $T \times d_{model}$ and the output the same. Effectively, we end up with T vectors, each of size d_{model}.

Now, let's focus at the "states" between the Transformer blocks. When we are working with RNNs, we refer to these as hidden states $h(t)$. We could, of course, use the same terminology in the case of Transformers, but it's typically not used. Instead, we normally call these vectors "contextual embeddings".

First of all, they are called "embeddings", because they are just like the first layer of embeddings in terms of their shape. No matter where we are, at the beginning of the network or at the end, we still have T vectors, each of size d_{model}. The difference is that at the beginning of the model, those embeddings come from a predetermined table and do not take context into account. However, this changes when we get at the end of the network, which does take context into account thanks to attention. Therefore the "meaning" of the initial pure words changes as the corresponding embeddings move through the Transformer blocks. Or we could say that it gets transformed based on the surrounding context.

1. Encoder for text classification
   Suppose we want to to text classification, which involves assigning a single label to an entire text document. With RNNs we normally take the final

hidden state, push that through a final dense layer and then apply the softmax. However, with Transformers we actually take the embedding output and follow the same procedure. In this case we take the first output, because with Transformers order doesn't matter, since everything can pay attention to everything else. Of course, this is opposed to a typical RNN, which only pays attention to the past hidden state.

2. Encoder for token classification
   In this problem our aim is to classify every token in the document to which part of speech it belongs to. In this case, we can take every contextual embedding and pass each of them through a final dense layer followed by a softmax.

3.9 Decoder Architecture

As we can see in figure (3.16), the decoder architecture is almost the same as the encoder.

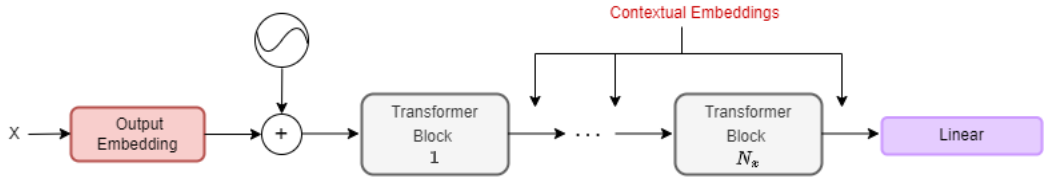


Figure 3.16: Decoder Architecture

The main task of the decoder is to predict the next token in a sentence. We could do this naively, but for Transformers there is a very specific method. To go into more detail about how the decoder works let's start with an example. For instance assume the following popular, in the domain of NLP, sentence:

"The quick brown fox jumps over the lazy dog"

Suppose that we pass on the input:

"The quick brown fox jumps over the lazy _____"

then we would train the Transformer to output "dog". We would do this for all documents in our dataset. In particular, we would turn each document into multiple effective samples:

$$\text{"The"} \longrightarrow \text{"quick"}$$
$$\text{"The quick"} \longrightarrow \text{"brown"}$$
$$\text{"The quick brown"} \longrightarrow \text{"fox"}$$

where the left-hand-side expressions are the inputs to our model, whereas right-hand-side expressions are our targets. In other words all the tokens in a document have the chance to be the target, given just the previous tokens and therefore we are building a model that estimates the probability:

$$p(\mathbf{x_t}|\mathbf{x_{t-1}}, \mathbf{x_{t-2}}, \ldots, \mathbf{x_1})$$

Now that we have stated the problem that the decoder is called to solve, let's describe its architecture. As input we have a sequence of tokens, which get converted into embedding vectors and combined with positional encodings they pass through a series of Transformer blocks. At the output we have a final linear layer, which is used for classification and has N_V output nodes, where N_V is our token vocabulary size (fig. 3.16).

However, the main difference of the decoder lies in the construction of the Transformer blocks. As we can see from figure (3.17), instead of using plain Multi-Head Attention, we use Multi-Head Attention **with masking**, which is also called **causal Multi-Head Attention**. It is causal, because it requires that the model only looks at the past. Furthermore, one crucial point to be understood, is that this isn't technically part of the model itself, but instead we induce this causality by inserting a specific mask ourselves.

Note that masking is not an absolute necessity in the decoder architecture, but a beautiful trick, so as to force attention weights to pay attention only to past input tokens. Recall that our attention weights are a $T \times T$ square matrix where $w(i, j)$ quantifies the amount of attention the output token i pays to the input token j. Consequently, we want $w(i, j) > 0$ only when $i \geqslant j$. We can accomplish this by setting the values of the mask according to (3.52):

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{3.52}$$

Then by multiplying our attention weights we get our **masked attention weights**:

$$\mathbf{W_M} = \mathbf{W} \circ \mathbf{M} \Leftrightarrow w_M(i,j) = \begin{cases} w(i,j) & ,i \geqslant j \\ 0 & ,i < j \end{cases} \tag{3.53}$$

Note that in the expression of the causal attention matrix we also allow $i = j$, which means that the diagonal elements of $w_M(i,j)$ are non-zero. In other words, the output token i can pay attention to itself, because the output is a shifted version of the input. Also notice that in (3.53) what we actually want is only the **lower triangular** portion of the attention matrix $\mathbf{W}$ to have non-zero values.

Now that we have examined what masking does, let's now talk about **how it helps**. The brute force method of predicting the next token in a sequence based on the past ones is the following· suppose we the same input sentence as previously:

"The quick brown fox jumps over the lazy dog"

Then in order to predict the token "quick" we should create the sample "The". In order to predict the token "brown" we need to create the sample "The quick" and so on. In other words, the brute-force method requires from us to **manually** create the input tokens, because, otherwise, in the train set, the decoder would pay attention to future tokens as well.

But, by using the trick of masking we can improve this cumbersome method, because it allows us to have T outputs and any input sequence need to be passed once. In other words, we don't have to create separate input samples. From this one pass our model can **simultaneously** learn all of the subsequences.

Finally note that, as we can see from figure (3.1), in the original paper "Attention Is All You Need", the decoder was constituted by two Multi-Head attention layers. In particular the first Multi-Head attention layer is a regular one, whereas the second is a causal Self-Attention using a mask. However, this need not always

be the case. Ultimately, decoder's structure depends on the particular task that Transformer is called to tackle.

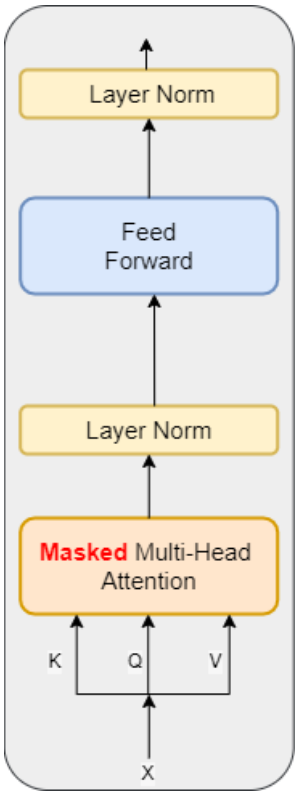


Figure 3.17: Decoder Transformer block

3.10 Encoder-Decoder connection

Another important fact about the Transformer architecture, which is not clear from the typical diagram shown in figure (3.1) is which encoder output is connected to which encoder input. First, notice that the N_x index beside each Transformer block indicates the number of times we stack each Transformer block. In the "Attention Is All You Need" paper N_x was set to 6.

And here is the point of ambiguity. In particular, it's not clear whether we connect each encoder output to the corresponding decoder input or **we take only the final**

encoder output and plug that into each decoder input. The correct answer is the second one and it is more clearly illustrated in figure (3.18)

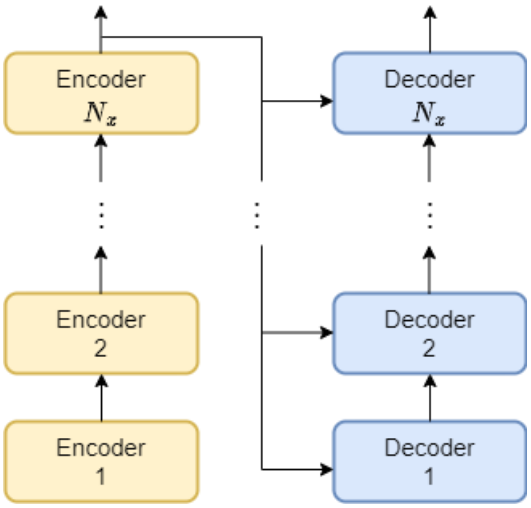


Figure 3.18: Encoder-Decoder connection

The above expanded diagram shows that the data passes through all the encoder layers first. Only the final encoder output gets passed to the decoder and it's the same output that is plug into each decoder input.

Also note that, from figure (3.1), the encoder output becomes V and K for the decoder, while the Q comes from the previous Multi-Head Attention layer. This makes sense, because the encoder creates the keys and the corresponding values, whereas the decoder makes the desired queries. Recalling our previous analogy with databases, we could say that we input to the database the keys and the corresponding values, while in order to output something we first have to create a query.

Now that we have the intuition of why the encoder and decoder are connected with each other this way, let's see how the equation for the Attention matrix (3.43) changes. Since, as we have already mentioned, the keys and values come from the encoder while the queries come from the decoder, we have:

$$\mathbf{A}(\mathbf{Q_{dec}}, \mathbf{K_{enc}}, \mathbf{V_{enc}}) = softmax\left(\frac{\mathbf{Q_{dec}}\mathbf{K_{enc}^T}}{\sqrt{d_k}}\right)\mathbf{V_{enc}} \tag{3.54}$$

where:

- $\mathbf{V_{enc}} \in \mathbb{R}^{T_{input} \times d_v}$, $\mathbf{K_{enc}} \in \mathbb{R}^{T_{input} \times d_k}$, $\mathbf{Q_{dec}} \in \mathbb{R}^{T_{output} \times d_k}$.
- the subscript "dec" denotes "decoder", whereas the subscript "enc" denotes "encoder".
- equation (3.54) is also known as **Cross-Attention**.
- the second dimension of $\mathbf{K_{enc}}$ and $\mathbf{Q_{dec}}$ must be the same for the matrix multiplication to be a valid operation.
- generally the sequence length of the input is **not equal** to the sequence length of the output.
- $\mathbf{W} = \mathbf{Q_{dec}}\mathbf{K_{enc}^T} \in \mathbb{R}^{T_{output} \times T_{input}}$ are the **cross-attention weights**.

3.11 Training vs. Inference

Another issue that has to be cleared is how the processes of training and inference are implemented. The problem arises from the fact that during training we always pass in a shifted version of the true target into the decoder input. So if the sentence is our favourite:

"The quick brown fox jumps over the lazy dog"

then the input becomes:

"<SOS>, The, quick, brown, fox, jumps, over, the, lazy, dog"

where "<SOS>" stands for "Start Of Sentence", while the target is:

"The, quick, brown, fox, jumps, over, the, lazy, dog, <EOS>"

where "<EOS>" stands for "End Of Sentence". But this can't be the way our model makes predictions, because during inference time we don't know the true target. Instead, what we do is to simply **predict one token at a time** and then use our previously predicted tokens as input into the next time step.

In other words, during inference time we follow the next steps for the above example:

- at $t = 1$ the input is "<SOS>" and our model will output a word that may be the word "The" (output 1).
- at $t = 2$ the input is "<SOS>, output 1" and the model gives us another output, which may be the word "quick" (output 2).
- at $t = 3$ the input is "<SOS>, output 1, output 2" and the procedure continues this way.

In fact, at each time step we add to the input the last output of our model. However, we can change our input from:

"<SOS>, output 1, output 2, . . ., output $(t-1)$"

to:

"<SOS>, **real** output 1, **real** output 2, . . ., **real** output $(t-1)$"

In the second case, which is called **Teacher Forcing**, we essentially substitute the previous outputs of our model with the corresponding real values that were produced, since they are available during inference time. This way we feed our model with the values that should have produced and therefore it is supposed to learn by its own mistakes.

Chapter 4

Transformers in Computer Vision

Chapter 4 contents
[31], [32], [33], [34], [35], [36], [37], [38], [39], [40]

4.1 Introduction

Since the introduction of Transformers in 2017, they rapidly spread widely in the field of Natural Language Processing (NLP), becoming its dominant choice, due to their remarkable performance. Because of their need to be trained on large amount of data, the main idea that is applied is to pre-train the model on large text-datasets and then fine-tune them on smaller text corpus so as to solve specific tasks. For some years after the publication of the original paper "Attention Is All You Need" the use of the Transformer model in Computer Vision tasks was quite limited. In particular, in vision, the concept of attention initially was used combined with Convolutional Neural Networks (CNNs). Sometimes the attention mechanism was used to replace other components of the CNN architectures.

However the first remarkable attempt to completely replace CNNs for Computer Vision tasks came in 2021 with the paper "An image worth 16×16 words: Transformers for image recognition at scale". There, the Vision Transformer (ViT) model was proposed, which is nothing else but the encoder of the original Transformer model. The main idea was to apply the standard Transformer architecture for image classification tasks with the least possible changes and it was shown that the exclusive dependency on CNNs is not necessary. More specifically, the researchers of the aforementioned paper proved that a pure Transformer model ex-

emplified better performance and greater efficiency on image classification tasks.

4.2 Vision Transformer (ViT)

4.2.1 Pure Transformer

The main idea of ViTs is to apply the concept of Self-Attention on images, which is what the encoder part of the original Transformer model does. Since the basic unit of analysis for images is the pixel, one simplistic approach would be to apply this method on the pixel level, forcing each pixel to pay attention to every other pixel. But the problem with this technique is that, because the size of the Self-Attention matrix is a quadratic function of the number of pixels, this naive method can't be applied to real input images, since it is not scalable to their size. In other words, for a typical real image, measuring the similarity between every single pixel pair is prohibitive not only in terms of memory but of computation too.

Figure 4.1: Input image to ViT

In order to solve this problem of scalability, the idea that was applied in the original "An image worth 16×16 words: Transformers for image recognition at scale"

paper was to split the input image (4.1) into larger regions of neighboring pixels, called **patches**, as we can see in figure (4.2). In particular, suppose our initial image $\mathbf{x}$ has resolution H by W, where H, W are the number of pixels along the height and the width of the image and C is the number of channels. Under these assumptions $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$.

Therefore if the resolution of the $2D$-patches is P by P, then each patch can be represented by a $3D$-tensor $\mathbf{x}_p^{3D} \in \mathbb{R}^{P \times P \times C}$, where $N = HW/P^2$ is the resulting number of patches. Then our original image can be represented as $4D$-tensor $\mathbf{x}^{4D} \in \mathbb{R}^{N \times P \times P \times C}$.

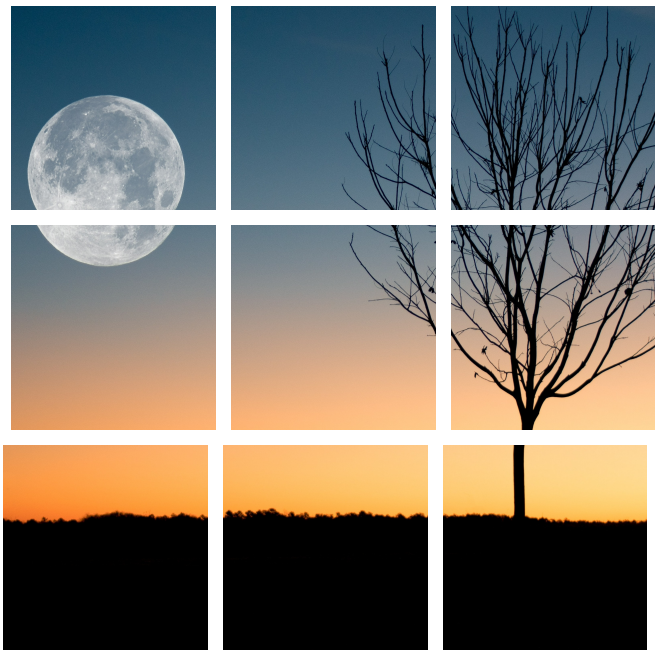


Figure 4.2: Image split into 9 patches before setting it as input to ViT.

Another problem that arises when trying to apply the Transformer architecture on images, is that it is unable to process grid-structured data. In other words $\mathbf{x}_p^{3D}$ is not a valid input into the Transformer model, since the Transformer is designed only to process sequential data. For that reason, what we do after we have created the patches and before feeding them into our Transformer is to **flatten** them. In other words we turn our $3D$-tensor into a vector of size D', where $D' = CP^2$ is the dimensionality of our flattened patches. As a result, for each one of the

patches, we end up having a vector $\mathbf{x}_p^{(i)} \in \mathbb{R}^{D'},\ \forall i = 1, \dots N$. Then, all these flattened patches, if put together in a single matrix, can form a $2D$-representation of our original image $\mathbf{x}^{2D} \in \mathbb{R}^{N \times D'}$.

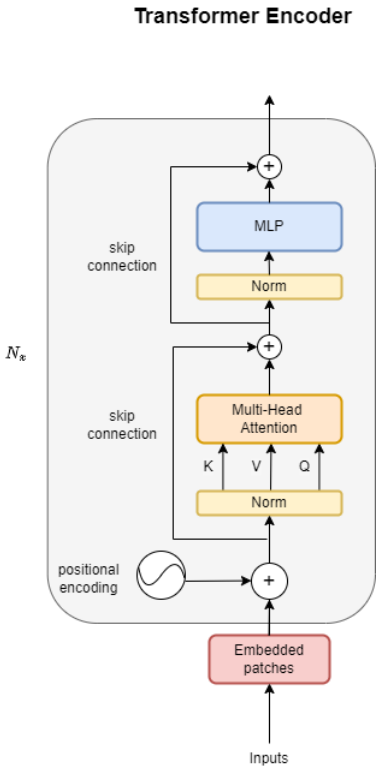


Figure 4.3: Transformer Encoder

But unfortunately, the vectors $\mathbf{x}_p^{(i)}$, which are the sequence tokens, like the words in NLP, are high dimensional. To see why this is true let's see an example. For instance, let's assume that the resolution of our input image is 1920×1080 pixels and we split it into 64 patches. Then the resolution of each patch will be 240×135 and if we flatten these patches we end up having vectors of size 32400, which are really high dimensional. Therefore we would like to decrease the dimensionality of each of these vectors. To achieve this, we pass the flattened patches through a linear projection layer and we create the **patch embeddings**, which are D-dimensional vectors with $D \ll D'$. Note that the weights of these linear transformations are trainable.

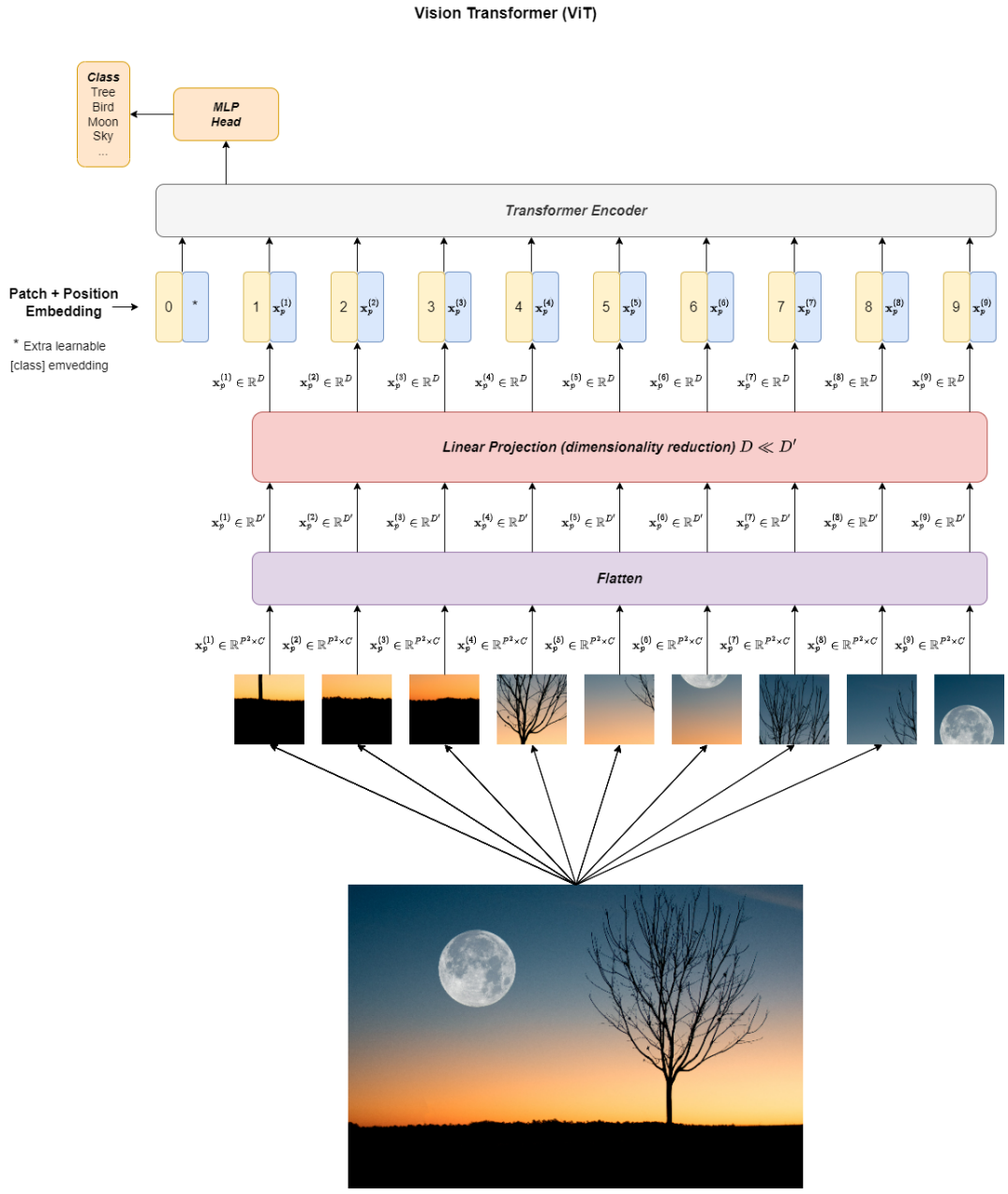


Figure 4.4: Vision Transformer (ViT)

After the low-dimensional patch embeddings are created, **position embeddings**

are added to them in order to incorporate positional information. The position embeddings that the researchers of the ViT paper used were 1-dimensional, since they didn't observe any significant increase in the overall performance by adopting more advanced position embeddings, like $2D$-embeddings.

4.2.2 Variants of ViT

Following the idea of the pure Vision Transformer, a whole gamut of variants of the ViT have been proposed to achieve better results in computer vision tasks. The main idea behind these new variants was to improve the efficiency of the Self-Attention mechanism, by making it capable of capturing short-range dependencies. Although the original ViT was good at extracting information about the relationship of distant regions in an image, it was observed that it was not so efficient in capturing local information.

The main models that were proposed so as to enhance locality were the following:

- TNT
- Twins
- CAT
- Swin Transformer
- Shuffle Transformer
- Region ViT
- DeepViT
- KVT
- XCiT

Since the architecture of a neural network plays an important role in the general field of Artificial Intelligence, new architecture designs have also been investigated. Apart from the original architecture of the ViT, which is a simple stack of transformer blocks with the same input-output shapes, the pyramid-like architecture was also adapted by many vision transformer models, such as HVT and PiT.

Moreover Neural Architecture Search (NAS) was implemented so as look for more efficient Transformer architecture, such as Scaling-ViT, ViTAS, AutoFormer and GLiT. Finally, it should be noted that there were also other directions been followed in order to improve the initial ViT model, such as positional encoding, removing attention, normalization strategy and shortcut connection.

4.2.3 Transformers with Convolution

Although Vision Transformers have presented unprecedented performance in many vision tasks, their inability to efficiently capture local information has led many researchers to the investigation of hybrid CNN-Transformer architectures. This because of the undoubted capability of the CNNs to capture locality.

There are plenty of works trying to incorporate convolutions into the Transformer model. Some of them are the CPVT, CvT, CeiT, LocalViT, CMT, LeViT, BoTNet and Visformer.

4.3 Applications of the Vision Transformer

According to (reference to the paper "A survey on Vision Transformer") the Transformer models were categorized by their application scenarios. In particular, in the aforementioned paper, they were divided into three categories: high/mid-level vision, low-level vision and video processing.

High/Mid-level vision

High/Mid-level vision deals with what we see in an image. In other words what objects and surfaces are experienced in it and how the information included in them is interpreted. More specifically, in this case Transformers perform the following tasks:

- Object detection
  - Transformer-based Set Prediction for Detection, such as the DETR and its improvements TSP-FCOS, TSP-RCNN, the Spatially Modulated Co-Attention (SMCA).
  - Transformer-based Backbone for Detection.

  - Pre-training for Transformer-based Object detection such Unsupervised Pre-training DETR (UP-DETR).

- Segmentation

  - Transformer for Panoptic Segmentation, such as Max-DeepLab and extensions of DETR.
  - Transformer for Instance Segmentation, such as VisTR and the Instance Transformer (ISTR).
  - Transformer for Semantic Segmentation, such the Semantic Transformer (SEMA) and others.
  - Transformer for Medical Image Segmentation, such as the Gated Axial-Attention (GAA) model and the Cell-DETR.

- Pose estimation

  - Transformer for Hand Pose Estimation, such as PointNet and Hand-Object Transformer Network (HOT-Net).
  - Transformer for Human Pose Estimation, such as METRO (Mesh Transformer)
  - Pedestrian Detection, such as the Pedestrian Detector (PED)
  - Lane Detection, such the LSTR.
  - Scene Graph, such as Graph R-CNN.
  - Tracking, such as TMT, TrTr, TransT and TransTrack.
  - Re-Identification, such as TransReID.
  - Point Cloud Learning

Low-level vision

Low-level vision deals with extracting information from images, such as features or descriptions, which usually are represented as images themselves. Although the applications of the low-level vision are quite limited, there are still a few published works, which include image super-resolution and image generation:

- Image Generation, such as the TransGAN, which is a GAN adopting the Transformer architecture, the ViTGAN, the Image Transformer and the Taming Transformer

- Image Processing, such the Texture Transformer Network for Image Super-Resolution (TTSR), the Image Processing Transformer (IPT), SceneFormer and iGPT.

Video processing

Because videos consist of highly sequential data, Transformers are perfectly suited for use on video tasks. In particular, in the context of videos, they perform the following tasks:

- Video Action Recognition, such as the Action Transformer or the I3D.
- Video Retrieval.
- Video Object Detection, such as the Memory Enhanced Global-local Aggregation (MEGA).
- Multi-task Leaning, such as the video multi-task transformer network.
- Frame-Video Synthesis, such as the ConvTransformer.
- Video Inpainting.

Bibliography

[1] Wikipedia, QR decomposition.

[2] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[3] Wikipedia, Orthogonal matrix.

[4] L. Programmer, [Udemy] Natural Language Processing with Deep Learning in Python.

[5] Wikipedia, Word embedding.

[6] J. Brownlee, Machine Learning Mastery: What Are Word Embeddings for Text?, 2019.

[7] Wikipedia, Natural language processing.

[8] M. Taboga, "QR decomposition", Lectures on matrix algebra., 2021.

[9] Wikipedia, GloVe.

[10] ——, Singular value decomposition.

[11] ——, Seq2seq.

[12] K. Taneja, Tutorial on Attention-based Models (Part 1), 2018.

[13] A. Zhang, C. Z. Lipton, M. Li, and J. A. Smola, Dive into Deep Learning: Bidirectional Recurrent Neural Networks.

[14] K. Taneja, Tutorial on Attention-based Models (Part 2), 2018.

[15] Wikipedia, Gated recurrent unit .

[16] L. Programmer, [Udemy] Deep Learning: Advanced Natural Language Processing and RNNs.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[18] B. Rohrer, Transformers from Scratch, 2021.

[19] L. Programmer, [Udemy] Data Science: Transformers for Natural Language Processing.

[20] J. Alammar, The Illustrated Transformer, 2018.

[21] J. R. Salazar, Transformers - Self-Attention to the rescue, 2022.

[22] J. Thickstun, The Transformer Model in Equations.

[23] A. K. Panda, Intuitive Maths and Code behind Self-Attention Mechanism of Transformers, 2021.

[24] Wikipedia, Transformer (machine learning model).

[25] IBM_Research and Harvard_NLP, exBERT lite (Explorable Transformers).

[26] N. Adaloglou, Intuitive Explanation of Skip Connections in Deep Learning.

[27] T. Denk, Linear Relationships in the Transformer's Positional Encoding.

[28] A. Kazemnejad, "Transformer architecture: The positional encoding," *kazemnejad.com*, 2019. [Online]. Available: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

[29] Wikipedia, Rotation matrix.

[30] N. Adaloglou, How Transformers work in deep learning and NLP: an intuitive introduction, 2020.

[31] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *ACM computing surveys (CSUR)*, vol. 54, no. 10s, pp. 1–41, 2022.

[32] Y. Liu, Y. Zhang, Y. Wang, F. Hou, J. Yuan, J. Tian, Y. Zhang, Z. Shi, J. Fan, and Z. He, "A survey of visual transformers," *arXiv preprint arXiv:2111.06091*, 2021.

[33] Wikipedia, Vision transformer .

[34] J. R. Siddiqui, What Are Vision Transformers And How Are They Important For General Purpose Learning?, 2022.

[35] B. Gaudenz, Vision Transformers (ViT) in Image Recognition – 2022 Guide, 2022.

[36] M. Bert, Transformers in Computer Vision, 2022.

[37] V. Perez, Transformers in Computer Vision: Farewell Convolutions!, 2020.

[38] C. Wolfe, Using Transformers for Computer Vision, 2020.

[39] N. Adaloglou, How the Vision Transformer (ViT) works in 10 minutes: an image is worth 16x16 words, 2021.

[40] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.