The Math behind transformers

Eddie Conti

2023/2024

Contents

1 Introduction: Large Language Models (LLM)

Large Language Models is a type of language model that is able to 'understand' and generate language. These models are extremely expensive in terms of computational costs ranging from millions to billions of parameters. This aspect further emphasizes how actually demanding is to reproduce language that is correct and consistent. LLMs are artificial neural networks (mainly transformers) and are characterized by

- Context-depending mechanism;
- Attention mechanism;
- Multitask-training using huge amounts of text;
- Parallel processing.

The first aspect to deal with is the embedding mechanism. A Word embedding is a representation of a word, or a set of word, typically using a real-valued vector that encodes some features of it. The representing vector has tens or hundreds of dimensions, which means that we are embedding words in a "large" dimensional space. By means of that, we expect that words that are closer in the vector space are similar in meaning. In mathematical terms, if we have a vocabulary V of size $|V| = m$, the embedding is a function

$$E\colon V \to \mathcal{M}_{\mathbb{R}}(m, n)$$

where $\mathcal{M}_{\mathbb{R}}(m, n)$ is the space of real matrices of dimension $m \times n$. Thereby, $E(\omega) \in \mathbb{R}^n$, namely a word is represented by a row vector with n entries. There are several techniques that performs embedding: the simplest is to associated each word $v_i \in V$ with the unit vector $e_i = (0, \dots, 0, 1, 0, \dots, 0)$ where 1 is in the $i-$th position. This means that if we perform $v_i \cdot v_j = 0$, namely every word are represented by orthogonal and so independent vectors. However, when representing the text with vectors we aim at capturing the similarity of the word based on the context: in this regards we expect that words as 'tree', 'leaf' are similar while 'train' and 'rock' are dissimilar. This is not the only goal desirable: usually language is mostly auto-referencial in the sense that we have words that refers/replace other words even though their meanings are not related. To be more clear, let us consider the following example

Luca is studying in Spain. He is very happy.

In the previous sentence, the word 'He' is replacing 'Luca' and we have somehow to capture this aspect of language. Furthermore, to be able to create an user interactive tool it is essential to capture this auto-referencial aspect at a high level. Let us delve our intuition with the following example by considering this text

I was talking to Charlie about his old car to find out if it was still running reliably.

For a machine to understand this sentence, it has to figure out that 'it' refers to 'car'. A neural network should be able to answer the following question

What is the current state of Charlie's old car?

We are going to explore in the next pages how this is assured by the attention mechanism.

If the context understanding is an essential requirement to mimic the natural language, LLM are characterized also by the possibility to perform a multitude of tasks. This is a general and good practice when producing an architecture: in our case we aim at the possibility to produce language, to dialogue, to analyze text etc. In order to achieve both objectives, LLM process all the words at once: this is a crucial difference to recurrent neural networks (RNN). Let us now delve into the first steps of transformers.

1.1 Seq2seq

Commonly, LLMs are used for text generation and encoder-decoder is the standard modeling paradigm for sequence-to-sequence tasks. The encoder reads source sequence and produces its representation; while the decoder uses source representation from the encoder to generate the target sequence. To formalize it given an input sequence $x = (x_1, \ldots, x_n)$ we aim at finding $y^* = (y_1^*, \ldots, y_m^*)$ such that

$$y^* = \underset{y}{\operatorname{argmax}}\, p(y|x)$$

In the case of language models, the output depends on the previous generated outputs, the probability of a sequence can be decomposed as

$$p(y_1, \ldots, y_m) = p(y_1) \cdot p(y_2|y_1) \ldots p(y_m|y_1, \ldots, y_{m-1}) = \prod_{t=1}^{m} p(y_t|y_{<t})$$

and so taking into account the input and the parameters the aim is to determine

$$y^* = \underset{y}{\operatorname{argmax}} \prod_{t=1}^{m} p(y_t|y_{<t}, x).$$

Neural seq2seq models are trained to predict probability distributions of the next token given previous context (source and previous target tokens). At the timestep t a model predicts a probability distribution

$$p^{(t)} = p(*|y_1, \ldots, y_{t-1}, x_1, \ldots, x_n).$$

The target at this step is $p^* = \text{one-hot}(y_t)$, i.e., we want a model to assign probability 1 to the correct token, y_t, and zero to the rest. The standard loss function is the cross-entropy loss:

$$Loss(p^*, p) = -p^* \log(p) = -\sum_{i=1}^{|V|} p_i^* \log(p_i)$$

but since only one p_i^* (for the correct token y_t) is non-zero we get

$$Loss(p^*, p) = \log(p_{y_t}) = -\log(p(y_t|y_{<t}, x)).$$

Hence, at each step we maximize the probability a model assigns to the correct token. Now that we know how to the model is trained, we focus on how to generate the sentence, i.e,

how to find y^* that satisfies

$$y^* = \underset{y}{\operatorname{argmax}} \prod_{t=1}^{m} p(y_t|y_{<t}, x).$$

The total amount of feasible solution is $|V|^n$ which is too large, hence usually there two approaches can be used to find an approximate solution: Greedy Decoding and Beam Search. The former, consists of generating at each a token with the highest probability. Even though it is a good baseline, mathematically

$$\underset{y}{\operatorname{argmax}} \prod_{t=1}^{m} p(y_t|y_{<t}, x) \neq \prod_{t=1}^{m} \underset{y_t}{\operatorname{argmax}}\, p(y_t|y_{<t}, x).$$

As a consequence, Beam Search, tries to tackle this issues by keeping track of several most probably hypotheses. At each step we continue each of the current hypotheses and pick top-N of them.

1.1.1 Capturing the context: Attention mechanism

So far we introduced the framework of LLMs and in particular seq2seq models. The simplest architecture consists of two RNNs (LSTMs): encoder RNN reads the source sentence, and the final state is used as the initial state of the decoder RNN. The hope is that the final encoder state "encodes" all information about the source, and the decoder can generate the target sentence based on this vector. However, the constraint of condensing all information into a single vector fails to capture the complexity of language and often proves insufficient for synthesizing the input. This motivates the introduction of the concept of attention (see paper [2]).
The attention mechanism is the foundation of the transformer architecture that will be discussed afterwards. The basic idea is to allow the model to weigh the importance of each word within a sequence according to context. In this way, the model is able to capture long-range word relationships without relying on rigid sequential structures. In simple terms, the attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. To solve the problem of the bottleneck in LSTMs, attention mechanism gives representations for all source tokens: instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder.

In more details, figure 1 shows what the architecture performs: at every step given all the hidden states from the encoder, we produce a context vector that is concatenated to internal state and then we use a fully connected NN to produce the output word. In mathematical terms, denoted $s_1, \ldots, s_n$ all the encoder states and h_t the decoder state, we compute attention weights as

$$a_{t,k} = \frac{exp(\text{score}(h_t, s_k))}{\sum_{i=1}^{n} exp(\text{score}(h_t, s_i)} \quad k = 1, \ldots, n$$

where score is a scoring function that encodes how relevant is a source token for that target

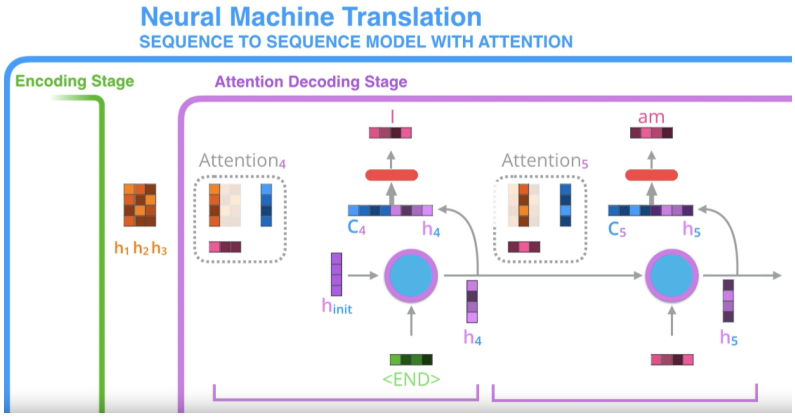


Figure 1: Details behind the mechanism

step. The most popular ways to compute attention scores are:

$$\begin{aligned}
&\text{Luong dot} \quad \text{score}(h_t, \hat{h}_s) = h_t^T \hat{h}_s \\
&\text{Luong multiplicative} \quad \text{score}(h_t, \hat{h}_s) = h_t^T W_a \hat{h}_s \\
&\text{Bahdanau} \quad \text{score}(h_t, \hat{h}_s) = v_a^T \tanh W_a[h_t; \hat{h}_s]
\end{aligned}$$

Therefore, the context vector is

$$c_t = a_{t,1} s_1 + \ldots + a_{t,m} s_m = \sum_{i=1}^{m} a_{t,i} s_i$$

here we concatenate c_t and h_t and compute

$$W[c_t; h_t]$$

where W is a matrix trained with all the other component that produces the output, i.e., the expected word.

1.2 The transformer architecture

A transformer is an architecture presented for the first time in the 2017 paper *Attention Is All You Need* for transforming one sequence into another one with the help of two parts: Encoder and Decoder.
Let us now explore the architecture in Figure 2. As we can see it is an Encoder-Decoder architecture: the former consists of encoding layers that process the input tokens iteratively one layer after another, while the latter consists of decoding layers that iteratively process the encoder's output as well as the decoder output's tokens so far.
The input of the layer is a sequence of tokens x_i which are presented to the network as one-hot encodings. As next step we reduce the dimensionality of the one-hot encoded vectors by multiplying them with a matrix denoted by W^E. In the original paper the size of the embedding in $d = 512$.

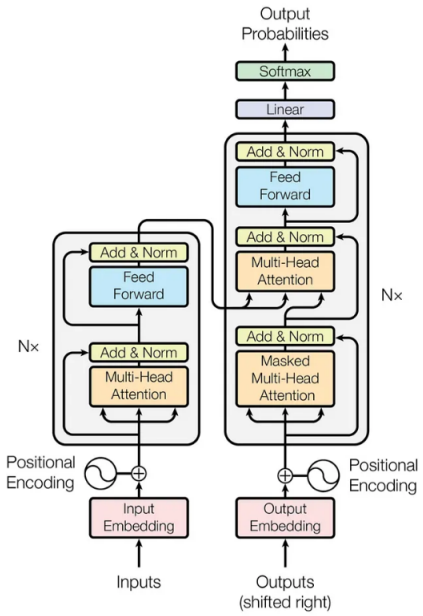


Figure 2: The transformer architecture. Image from [1].

1.2.1 Positional Encoding

As we already pointed out, all the tokens are presented to the Transformer simultaneously. However, this results in the loss of the original order of tokens in the input sequence. The purpose of positional encoding is to encapsulate the relative positions of tokens within a target sequence. This vector is then added to the embeddings. Before presenting the original function let us delve our intuition on why positional encoding is necessary. The general idea is to allow the model to differentiate between words with similar meanings but different positions in the sentence. For example if we consider the following

> Luca did not won the football tournament and he was happy;
>
> Luca won the football tournament and he was not happy.

Without any additional information about the positions of the words, the model might treat the sentences similarly although the meaning is different. A first approach could be to add the index of the of the token to its representation: $x_i \mapsto W^E \cdot x_i + i$. However as we will see in the next pages we are going to perform matrix operations and exponentiation and $W^E \cdot x_i + i$ grows as the length of the sequence grows, which is dangerous as it can lead to exploding gradients. Hence, one possibility would be to normalize index i so it ranges values between $[0, 1]$. Unfortunately, also this strategy will confuse the model if the input text length varies since the result would contain different positional embeddings for the same positions for different input sequences. The solution presented in paper [1] involves using a sine and cosine function to create a d dimensional vector for each position in the sentence. The positional encoding is defined as a function $f : \mathbb{N} \to \mathbb{R}^d$ where the $i-$th entry

is

$$f(t)^{(i)} = \begin{cases} \sin(\omega_k \cdot t) & \text{if } i = 2k, \\ \cos(\omega_k \cdot t) & \text{if } i = 2k+1 \end{cases} \tag{1}$$

with

$$\omega_k = \frac{1}{N^{2k/d}}.$$

Alternatively, it can be expressed as a complex valued function as

$$f(t)^{(i)} = e^{it/\omega_k} \quad k = 0, \ldots, d/2 - 1.$$

In paper [1] $N = 10000$, but in general is a free parameter that should be significantly larger than the biggest k that in our notation is $d/2 - 1$. The main idea behind this function is that it allows one to perform shifts as linear transformations and this is useful to express the relative position. For the sake of simplicity, let us fix $i = 2k$

$$\begin{aligned} f(t+s)^{(i)} &= \sin(\omega_k \cdot (t+s)) = \sin(\omega_k \cdot t + \omega_k \cdot s) \\ &= \sin(\omega_k \cdot t)\cos(\omega_k \cdot s) + \sin(\omega_k \cdot s)\cos(\omega_k \cdot t) \\ &= a \sin(\omega_k \cdot t) + b. \end{aligned}$$

Similarly, for $i = 2k + 1$

$$\begin{aligned} f(t+s)^{(i)} &= \cos(\omega_k \cdot (t+s)) = \cos(\omega_k \cdot t + \omega_k \cdot s) \\ &= \cos(\omega_k \cdot t)\cos(\omega_k \cdot s) - \sin(\omega_k \cdot t)\sin(\omega_k \cdot s) \\ &= a' \cos(\omega_k \cdot t) + b', \end{aligned}$$

hence for a fixed offset s we can express the $(t+s)-$th position a linear combination of $t-$th position.
Furthermore, there is another relevant reason for this specific function: the output for a fixed t, i.e. for a fixed position, is a d dimensional vector

$$(\sin(\omega_1 \cdot t), \ldots, \sin(\omega_d \cdot t))$$

where t influences the oscillation of the sine/cosine function. As a consequence when changing the parameter t we obtain a different vector that is representative of that position of the word. The motivation of choosing both sine and cosine meet the need, as we showed above, to express the $\sin(t+s)$ and $\cos(t+s)$ as a linear transformation of $sin(t)$ and $cos(t)$.

1.2.2 Multi-Head attention

The next crucial step of the transformer architecture is the Multi-Head attention. At this point, we multiply the position-aware word embeddings X by two matrices, W^K and W^Q, to obtain the "query vectors" and "key vectors" defined as

$$Q = XW^Q, \quad K = XW^K.$$

Attention weights are then computed as

$$Softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)$$

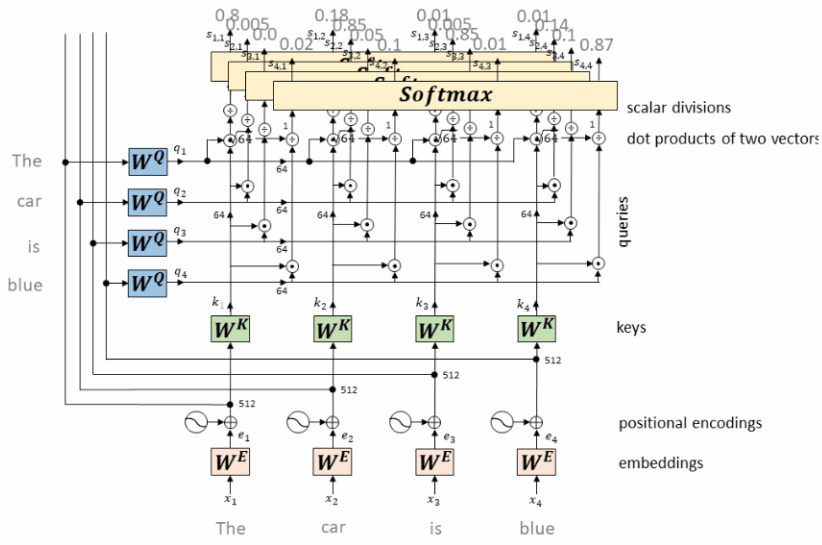


Figure 3: Embeddings, query and key vectors

where in the case of the paper we are dividing all weight factors by 8 (the square root of the dimension of the key vectors 64). Subsequently, we use the softmax function on the scaled factors as shown in picture 4.
The reason why are the dot products scaled with $\sqrt{64}$ before they arere fed into the softmax function is because they lead to more stable gradients. To understand this, we recall that the softmax function is a vector function

$$\begin{aligned} Softmax\colon \mathbb{R}^n &\to \mathbb{R}^n \\ z_i &\mapsto s_i := \frac{e^{z_i}}{\sum_j e^{z_j}}. \end{aligned} \tag{2}$$

Now, if we compute the partial derivatives for $k \neq i$

$$\begin{aligned} \frac{\partial s_i}{\partial z_i} &= \frac{\partial}{\partial z_i}\frac{e^{z_i}}{\sum_j e^{z_j}} = \frac{e^{z_i}}{\sum_j e^{z_j}} + e^{z_i}\frac{\partial}{\partial z_i}\frac{1}{\sum_j e^{z_j}} = \frac{e^{z_i}}{\sum_j e^{z_j}} - \left(\frac{e^{z_i}}{\sum_j e^{z_j}}\right)^2 = s_i \cdot (1 - s_i), \\ \frac{\partial s_i}{\partial z_k} &= \frac{\partial}{\partial z_k}\frac{e^{z_i}}{\sum_j e^{z_j}} = e^{z_i}\partial z_k \frac{1}{\sum_j e^{z_j}} = \frac{-e^{z_i}e^{z_k}}{\left(\sum_j e^{z_j}\right)^2} = -s_i \cdot s_k. \end{aligned}$$

Therefore, the Jacobian of (2) can be expressed as

$$J_s = \begin{pmatrix} s_1 \cdot (1 - s_1) & -s_1 \cdot s_2 & \cdots & -s_1 \cdot s_n \\ -s_2 \cdot s_1 & -s_2 \cdot (1 - s_2) & \cdots & -s_2 \cdot s_n \\ \vdots & \vdots & \ddots & \vdots \\ -s_n \cdot s_1 & -s_n \cdot s_2 & \cdots & -s_n \cdot (1 - s_n). \end{pmatrix}$$

It is immediate to observe that the Jacobian becomes zero if occurs one among

$$s_1 = (1, 0, \ldots, 0),\ s_2 = (0, 1, 0, \ldots, 0), \ldots,\ s_n = (0, \ldots, 0, 1). \tag{3}$$

However, the softmax function is not scale invariant, which means that the higher we scale the inputs, the more the largest input dominates the output. Therefore, for large inputs the softmax function is generating outputs which closely resemble the values in (3). To tackle the problem of the vanishing gradient is necessary to rescale the dot products as the larger the dimension d of the key vectors and query vectors, the larger the dot products will tend to be.

Once we have performed the softmax, we define a new matrix W^V leading to "values vectors"

$$V = XW^V$$

and finally the attention values are defined as

$$Attention(Q, K, V) = Softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V. \tag{4}$$

We can describe equation (4) in mathematical terms. Matrices W^K and W^Q are linear transformation that deforms the space. In multiplying QK^T we are performing

$$XW^Q(W^K)^T X^T$$

and since the dot product captures the similarity of two vectors, here we are computing how similar are the vectors in X embedded in two appropriate linear spaces that are able to separate dissimilar words and cluster similar words. The softmax function is applied because when expressing input x_i as

$$x_i = a_1 x_1 + \dots a_n x_n$$

we want all the coefficients $a_i > 0$ and for stability reasons that

$$\sum_{i=1}^{n} a_i = 1.$$

Finally, the embedding that captures the similarity is transformed according to V. The motivation for that is, since our aim is to produce the next word in a sentence, the embedding defined by V suits this task since "knows" when two words appear in the same context because it is moving the space where words are either clustered or separated.
However, this is just part of architecture: the attention step is done many times leading to multi-head attention, which is described as it follows

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= Concat(head_1, \dots, head_h)W^O \\ \text{where } head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{5}$$

We concatenate all our attention embeddings and then apply a linear transformation W^O that not only reduces the dimensionality but takes the best from each embedding.

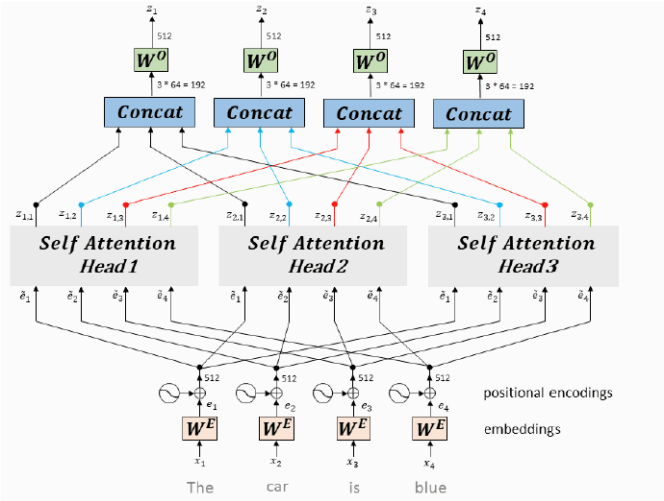


Figure 4: Multi-Head Attention

1.2.3 Stack of encoders

The output of the multi-head attention is an embedding for each input. At this point, a layer-normalization just subtracts the mean of each vector and divides by its standard deviation. This operation is done to obtain a more stability in the gradient. The outputs of the self-attention layer are fed to a fully connected feed-forward network: this consists of two linear transformations with a ReLU activation in between. The dimensionality of input and output is 512, and the inner-layer has dimensionality 2048 described by:

$$FFN(z) = \max(0; zW_1 + b_1)W_2 + b_2.$$

Finally, we again employ a residual connection around the fully connected feed-forward layer, followed by layer normalization. All the operations so far presented consists of an encoder in the architecture. However, the entire encoding component is a stack of six encoders and it is important that they do not share weights.

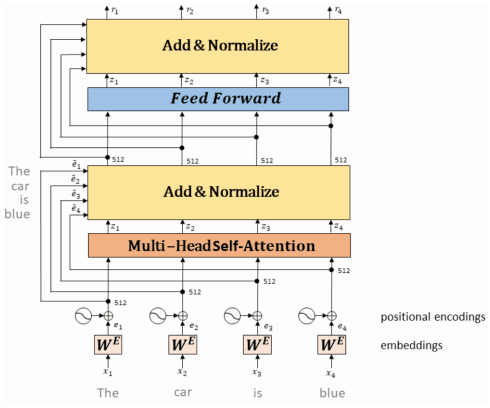


Figure 5: The encoder is composed by 6 sub-encoders

The general motivation for such a depth architecture is that in order to be able to capture the relationship between word, which is a crucial aspect as we said for translating and producing language, it is essential to look at the input text in many different ways. Similarly to what happens to images with convolutional NN, here each encoder is capturing a depth-dependent connection among the input. Language is extremely complex and heterogeneous and therefore we need a complex structure that is able to apprehend all facets of it.

1.2.4 The decoder

The decoder is responsible for sequentially generating output based on the information processed by the encoder. In its generality the structure is similar to the encoder's one, even though particular attention must be given to some aspects. First of all, the decoder takes positional information and embeddings of the output sequence as its input. Here, during self-attention, the decoder employs a mask to ensure that each position can only be influenced by preceding positions in the sequence. This mask is crucial to ensure that the model generates sequential output correctly without "looking ahead" during generation. The masked- multi head attention is pretty similar to what we described before, however we apply a method to prevent computing attention scores for future words.

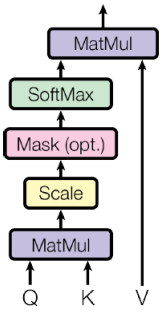


Figure 6: Details on how attention coefficients are computed

As Figure 6 points out the mask is applied after the scaling operation. Let S be the matrix of scaled values and M the masking matrix defined as

$$\begin{pmatrix} 0 & -\infty & -\infty & \cdots & -\infty \\ 0 & 0 & -\infty & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & -\infty \\ 0 & 0 & \cdots & \cdots & 0 \end{pmatrix}.$$

If we add matrix M to matrix S we obtain a matrix of masked scores whose elements are s_{ij} if $i \geq j$ and $-\infty$ otherwise. At this point if we apply the softmax to each row vector (the values for each token) if we have $-\infty$ then the output of the softmax will be 0. As a

consequence, we end up with a matrix of the form

$$\begin{pmatrix} s'_{1,1} & 0 & 0 & 0 & 0 \\ s'_{2,1} & s'_{2,2} & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s'_{n-1,1} & s'_{n-1,2} & \cdots & s'_{n-1,n-1} & 0 \\ s'_{n,1} & s'_{n,2} & \cdots & \cdots & s'_{n,n} \end{pmatrix}.$$

The attention coefficients are computed as

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(M + \frac{QK^{\text{T}}}{\sqrt{d_k}}\right)V. \tag{6}$$

It is worthy to note that even if in (6) we end up with a full matrix, the $j-$th row, i.e, the attention coefficients for input x_j, are computed as

$$(s'_{j,1}, \ldots, s'_{j,j}, 0, \ldots, 0) \cdot V.$$

In other words, they are linear combinations of $s'_{j,1}, \ldots, s'_{j,j}$ and this encodes exactly that the decoder is only influenced by preceding positions in the sequence. To sum up, the output of the first multi-headed attention is a masked output vector with information on how the model should attend on the decoder's input.
In the second multi-headed attention layer the decoder interacts with the output of the encoder through cross-attention. In this way, the decoder interacts both with the ongoing output (at the beginning we have a <start> character) and focus on different parts of the encoder's input during output generation. Here, the architecture proceeds similarly to the encoder's one: it processes the inputs with feed forward layers and normalization. Residual connections are applied to move smoothly through the architecture for the back propagation.
Finally, the output of the final pointwise feed forward layer goes through a final linear layer, that acts as a classifier. The classifier is as big as the number of classes you have, i.e. words. Here a softmax will produce probability scores between 0 and 1. We take the index of the highest probability score, and that equals to our predicted word. The decoder then takes the output, add's it to the list of decoder inputs, and continues decoding again until a <end> token is predicted. Therefore, the decoding steps produces a word that is the most probable according to the input and its context and the previous word generated: in this way we gained the ability to produce meaningful sentences that are related to the input sentence.

2 The amount of parameters in LLM

Transformers are the main architecture employed for LLM. Despite their notable capabilities in understanding and generating language such as GPT, a crucial aspect of these models lies in their substantial number of parameters, particularly when applied to the English language. The willing to capture the richness of linguistic context has led to architectures of considerable size, comprised of millions or even billions of parameters. As it is clear from Chapter 1, in order to comprehend language it is essential to use a huge variety of matrices (i.e. word representations) and long distance word relationship. The largest models can have a context window sized up to 32000. For example, GPT-4; while GPT-3.5 has a context window sized from 4000 to 16000, and legacy GPT-3 has had 2000 sized context window. Also, the significant economic costs in terms of computational resources and infrastructure must taken into account.
Within this context, the challenge emerges of balancing the expressive power of advanced language models with the need to optimize computational efficiency.

Name	Release Date	Number of parameters
GPT-1	June 2018	117 million
BERT	October 2018	340 million
GPT-2	February 2019	1.5 billion
GPT-3	May 2020	175 billion
LaMDA	January 2022	137 billion
Minerva	June 2022	540 billion

Table 1: List of some LLMs with the amount of parameters

As it is clear from Table (1) training transformer-based architectures can be expensive, especially for long inputs.

2.1 Transformers for long sequences: changing the attention

Transformer architecture suffers some limitations when it comes to long sequences as the computational complexity increases quadratically with the sequence length n. Indeed, most of the complexity of the architecture lies in the self-attention, where we produce the attention coefficients. Each token x_i is related to every other token x_j in order to capture the relationship between words. This means that the resulting complexity is $O(n^2)$. Most transformer models are fixed in their sequence length as in the case of BERT model [6] which is limited to 512 tokens. However, being able to cover a variety of tasks is a desirable aspect and when it comes to document summarization, DNA processing or any task that requires long sequences, the training becomes practically infeasible, due to the enormous computational cost. In the next pages we are going to cover different techniques that reduce this quadratically dependency on sequence length. These approaches try to switch from the quadratic dependency to a linear dependency.

2.1.1 Sparse attention

Sparse attention refers to an attention mechanism where the attention of each token is limited to a subset of other tokens. BigBird, a transformer model developed by Google Research (see [7]), implements a sparse attention mechanism that reduces the quadratic dependency to linear. As they proved, the proposed sparse attention can handle sequences of length up to $8x$ of what was previously possible using similar hardware. As a consequence, BigBird drastically improves performance on various NLP tasks such as question answering and summarization. To be more precise, BigBird is a combination of global attention, local attention and random attention. In particular, BigBird consists of three main parts:

- a set of g global tokens attending on all parts of the sequence;
- all tokens attending to a set of w local neighboring tokens;
- all tokens attending to a set of r random tokens.

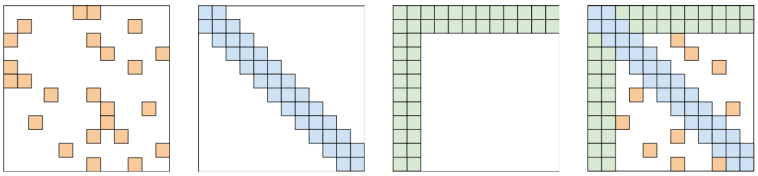


Figure 7: White color indicates absence of attention. (a) Random attention with $r = 2$, (b) sliding window attention with $w = 3$, (c) global attention with $g = 2$, (d) the combined BigBird model.

In mathematical terms, let $X = (x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$ an input sequence. The generalized attention mechanism is described by a directed graph D whose vertex set is $\{1, \ldots, n\}$. In this scenario, we denote by $N(i)$ the out-neighbors set of node i in D, or, in other terms, the set of inner products that the attention mechanism will consider. The generalized attention is defined as

$$\text{ATTN}_D(X)_i = x_i + \sum_{h=1}^{H} \sigma\Big(Q_h(x_i) K_h(X_{N(i)})^T\Big) \cdot V_h(X_{N(i)}), \tag{7}$$

where $Q_h, K_h : \mathbb{R}^d \to \mathbb{R}^m$ are query and key functions, $V_h : \mathbb{R}^d \to \mathbb{R}^d$ is a value function, σ is a scoring function such as softmax and H denotes the number of heads. Also note $X_{N(i)}$ corresponds to the matrix formed by only stacking $\{x_j : j \in N(i)\}$ and not all the inputs. It is clear that (7) reduces to (4) if $N(i)$ is full, in the sense that we are considering the relationship among all the input words. The general idea behind BigBird architecture is that most contexts within NLP and computational biology have data which displays a great deal of locality of reference. In this regards, fixed token x_i, tokens x_j for $i - w/2 \leq j \leq i + w/2$ are the most relevant for a context comprehension. Furthermore, adding random attention in the key function, allow to capturing less obvious or less frequent information that may be crucial for understanding the context. Furthermore, introducing randomness in attention

can make the architecture more robust to variations in input data. Lastly, in contexts where relevant information may be distant or not immediately related, random attention could facilitate capturing broader and non-local relationships. To conclude, the architecture involves the use of global attention, i.e, tokens that attend to all tokens in the sequence and to whom all tokens attend to. Concretely, we choose a subset $G \subseteq \{1, \ldots, n\}$ such that we explore the relationship between x_i and x_j for $i \in G$ and all j. This latter element is added in order to connect indirectly any pair (x_i, x_j). In fact, the article is based on the mathematical result that every complete graph can be approximated by random graphs. In this scenario, these three elements aims at reproducing the original attention graph. As proven in Theorem 3 in [7], the complexity using sparse attention reduces to $O(n)$.

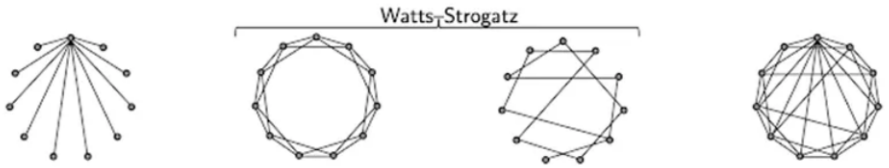


Figure 8: The effect of global, window and random attention in generating a sparse graph. Here, tokens represent nodes and the similarity scores calculated between tokens represent the edges. It is important to note that the number of edges is equal to the number of inner products that the attention mechanism will consider.

An alternative that shares similarities with BigBird is Longformer, presented in [8]. This architecture scales linearly with the input sequence, making it efficient for longer sequences. It is composed by three elements that combined form the sparse attention: sliding window, dilated sliding window and global attention.

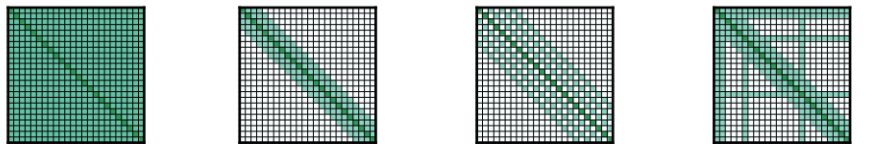


Figure 9: Visual explanation of the attention patterns in Longformer

First of all, given a fixed window size w, each token attends to $1/2w$ tokens on each side. Therefore, the computation complexity of this pattern is $O(n \times w)$, which scales linearly with input sequence length n. By using multiple stacked layers of such windowed attention, this results in a large receptive field that is able to capture information across the entire input. To further increase the receptive field without increasing computation, the sliding window can be "dilated" by gaps of size dilation d. In a transformer with l layers, assuming w, d fixed, the receptive field is $l \times d \times w$, which can reach tens of thousands of tokens even for small values of d. In paper [8] they underline that in multi-headed attention, each attention head computes a different attention score. Therefore settings with different dilation configurations per head improves performance by allowing some heads without

dilation to focus on local context, while others with dilation focus on longer context. In this way, we are "breaking down" the full attention in different experts that still aims at exploring the context at different scales.
Similarly to BigBird, we add global attention on few pre-selected input locations to allow enough flexibility to learn task-specific representations. Since the number of such tokens is small relative to and independent of n the complexity of the combined local and global attention is still $O(n)$. The attention coefficients are still computed as

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

but we use Q_s, K_s, V_s to compute attention scores of sliding window attention, while Q_g, K_g, V_g to compute attention scores for the global attention, that encodes what we described above. In paper [8] the authors emphasise the good performance of the model for long sequences, similarly to BigBird.

2.1.2 Linear attention

In paper [10] the authors propose a method that is able to scale linearly with respect to the dimension of the output. They introduce the so called *linear transformer* that is a reformulation of self-attention by using a kernel-based formulation. To follow their notation, let us denote

$$V' = Softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V.$$

and the $i-$th row of V' can be expressed as

$$V_i' = \frac{\sum_{j=1}^n e^{Q_i^T K_j} V_j}{\sum_{j=1}^n e^{Q_i^T K_j}}. \tag{8}$$

Equation (8) can be generalized to any similarity function as

$$V_i' = \frac{\sum_{j=1}^n \mathrm{sim}(Q_i, K_j) V_j}{\sum_{j=1}^n \mathrm{sim}(Q_i, K_j)} \tag{9}$$

where $\mathrm{sim}(Q_i, K_j)$ is a non-negative function that expresses how similar are two vectors. Before continuing it is essential to introduce the following definitions and results.

Definition 1. An Hilbert space H is a reproducing kernel Hilbert space (RKHS) if the evaluation functionals $F_x \colon H \to \mathbb{R}$ such that $F_x(f) = f(x)$ are continuous, i.e.

$$|F_x(f)| = |f(x)| \leq M||f|| \quad \forall f \in H$$

for a given $M \in \mathbb{R}$.

Invoking Riesz's representation theorem, every RKHS has a special function associated to it, namely the reproducing kernel:

Definition 2. A reproducing kernel is a function $K \colon X \times X \to \mathbb{R}$ such that

1. $K_x(\cdot) \in H$, $\forall x \in X$ and
2. $(f, K_x) = f(x)$, $\forall f \in H$ and $x \in X$.

The intuition for a reproducing kernel is a function that is able to reconstruct the elements of an Hilbert space. Given a measure μ,

$$\begin{aligned} f(x) = (f, K_x) &= \int_X f(x') K_x(x')\, d\mu(x') \\ &= \int_X f(x') K(x, x')\, d\mu(x'), \end{aligned}$$

which shows that K is a function that is able to determine the punctual value of f if it accesses to all the information about f in terms of similarity to the interested point. From the uniqueness in Riesz's representation theorem it is immediate to conclude the following

Theorem 2.1. *Every reproducing kernel K induces a unique RKHS, and every RKHS has a unique reproducing kernel.*

The interesting property of reproducing kernel is that they are a measure of similarity in a different (usually big) dimensional space. Indeed, in view of Mercer-Hilbert-Schmit theorem, it holds that a reproducing kernel can be expressed as

$$K(x, y) = \phi(x)^T \phi(y) \tag{10}$$

where $\phi \colon X \to F$ is a feature map with F an Hilbert space. Viceversa, every feature map defines a unique reproducing kernel according to (10).
Losing generality, we can assume sim to be a reproducing kernel, so that $\text{sim}(a, b) = \phi(a)^T \phi(b)$. Therefore, substituting this in (9), we obtain

$$V_i' = \frac{\sum_{j=1}^n \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^n \phi(Q_i)^T \phi(K_j)}. \tag{11}$$

In this way we have an attention computation consisting only on dot products. Using associativity in (11) we get

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^n \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^n \phi(K_j)}.$$

Previous equation defines the *linear transformer* and it is crucial to underline that has a complexity of $O(n)$ because we can compute $\sum_{j=1}^n \phi(K_j) V_j^T$ and $\sum_{j=1}^n \phi(K_j)$ once and reuse them for every query and so n times. The same idea can be used for the masking such that the $i-$th position can only be influenced by a position j if and only if $j \leq i$, observing that we can express the attention as

$$V_i' = \frac{\sum_{j=1}^i \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^i \text{sim}(Q_i, K_j)}$$

and so

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^i \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^i \phi(K_j)}.$$

An illustrative example is based on first-order approximation of Taylor expansion of exponential function:

$$e^{Q_i^T K_j} \approx 1 + Q_i^T K_j.$$

To ensure $1 + Q_i^T K_j \geq 0$ we can normalize Q_i and K_j using the l_2 norm so that $-1 \leq Q_i^T K_j \leq 1$. Equation (8) becomes

$$V_i' = \frac{\sum_{j=1}^{n} \left(1 + \left(\frac{Q_i}{||Q_i||_2}\right)^T \sum_{j=1}^{n} \left(\frac{K_j}{||K_j||_2}\right)\right) V_j}{\sum_{j=1}^{n} \left(1 + \left(\frac{Q_i}{||Q_i||_2}\right)^T \left(\frac{K_j}{||K_j||_2}\right)\right)}$$

and simplified as:

$$V_i' = \frac{\sum_{j=1}^{n} V_j + \left(\frac{Q_i}{||Q_i||_2}\right)^T \left(\frac{K_j}{||K_j||_2}\right) V_j}{n + \left(\frac{Q_i}{||Q_i||_2}\right)^T \sum_{j=1}^{n} \left(\frac{K_j}{||K_j||_2}\right)}$$

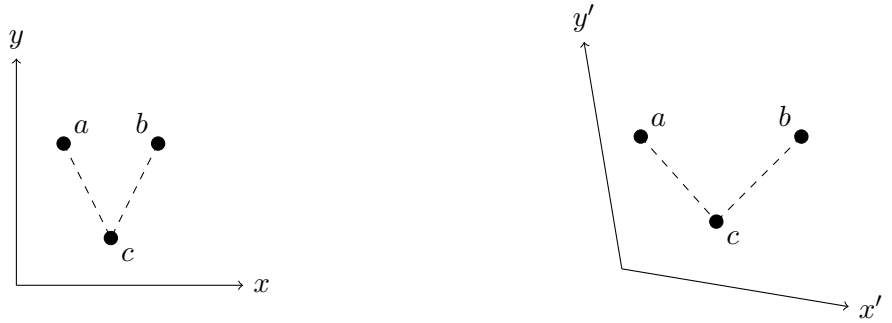
In paper [10] the authors show that their evaluation on image generation and automatic speech recognition demonstrates that linear transformer can reach the performance levels of transformer, while being up to three orders of magnitude faster during inference.

2.1.3 Matrix factorization

In matrix factorization we assume that the matrix corresponding to the self-attention values is low rank, i.e., not all items are independent of each other. Therefore, the matrix QK^T which is $n \times n$ is reduced to a matrix $n \times k$ with $k < n$ without significant loss in information. Since taking row-wise softmax of a square matrix of order n has complexity $o(n^2)$, reducing the matrix QK^T improves significantly the efficiency.

3 Chapter

In the previous chapter we showed three different approaches aiming at reducing the complexity of the self-attention mechanism. In this chapter we want to discuss an alternative method motivated by the mathematical understanding of the attention coefficients. To sum up, in the attention formula, matrix Q and K look at the similarity between words after projecting the input in two different spaces, while matrix V master the ability to predict the next word using the knowledge on the similarity of words. The key idea is to think of matrix as linear actions that deform the space, but why is this useful? Let us consider for the sake of simplicity two points $a, b \in \mathbb{R}^2$ and a third point c that is somehow in between a, b. We can think of c as a word that has multiple meaning and provided context it will shift towards either a or b.



When we provide context we are weighting each word according to all the other input words

$$s_t = a_{t,1}s_1 + \ldots + a_{t,m}s_m = \sum_{i=1}^{m} a_{t,i}s_i$$

As a consequence, in our simple example, c will shift towards either a or b. It is clear that the second space works better in moving c according to the context. However, if Q, K are responsible for moving the space, the way we measure the similarity is the dot product. Let us now denote $\mathcal{D} = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ and q a query. According to our discussion, we can generalize the definition of attention as

$$Attention(q, \mathcal{D}) = \sum_{i=1}^{n} \alpha(q, k_i)v_i, \tag{12}$$

where $\alpha(q, k_i) \in \mathbb{R}$ for $i = 1, \ldots, n$ are scalar attention weights. The function α is generally referred to as *attention pooling* and it can be thought as a similarity measure. In general we can ask α to have various properties, the most common, as done in the transformer architecture, is that weights $\alpha(q, k_i)$ form a convex combination, i.e.,

$$\sum_{i=1}^{n} \alpha(q, k_i) = 1, \qquad \alpha(q, k_i) \geq 0 \text{ for } i = 1, \ldots, n.$$

This is usually attained using the softmax

$$\alpha(q, k_i) = \frac{exp(\alpha(q, k_i))}{\sum_{j=1}^{n} exp(\alpha(q, k_j))}.$$

In paper [12] the authors delve into the multi-head attention mechanism and evaluate the contribution made by individual attention heads in the encoder to the overall performance of the model. They found that for a translation task, only a small number of heads are important for translation and these heads play interpretable "roles". They showed that most of the heads can be pruned without significant loss in quality, but after the training process. Inspired by their work, our aim is to explore the possibility lightening the architecture by finding an optimal attention pooling function for a given task.

3.1 Attention pooling functions

References

[1] A. Vaswani et al., *Attention Is All You Need*, NIPS, 2017

[2] D. Bahdanau, K. Cho, Y. Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, arXiv, 2014

[3] M.T. Luong, H. Pham, C. D. Manning, *Effective Approaches to Attention-based Neural Machine Translation*, arXiv, 2015

[4] R.S. Navid, *Introduction to Transformers*, Institute of Computational Perception CP Lectures, 2020

[5] A. Kak, C Bouman, *Transformers: Learning with Purely Attention Based Networks*, Lectures on Deep Learning Purdue University, 2023

[6] J. Devlin, M. W. Chang, K. Lee, K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint, 2018

[7] M. Zaheer et al., *Big Bird: Transformers for Longer Sequences*, NIPS, 2020

[8] I. Beltagy, M. E. Peters, A. Cohan, *Longformer: The Long-Document Transformer*, arXiv, 2020

[9] I. Chalkidi et al., *An Exploration of Hierarchical Attention Transformers for Efficient Long Document Classification*, arXiv, 2022

[10] A. Katharopoulos et al., *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*, arXiv, 2020

[11] R. Li et al., *Linear Attention Mechanism: An Efficient Attention for Semantic Segmentation*, arXiv, 2020

[12] E. Voita et al., *Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned*, Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019

[13] S. Wang et al., *Linformer: Self-Attention with Linear Complexity*, arXiv, 2020

[14] Y. Tay et al., *Long range arena: a benchmark for efficient transformers*, arXiv, 2020

[15] A. Zhang et al., Dive into Deep Learning, 2023