Attention & Transformers: This short set of notes is all you need!

Dimitrios Tanoglidis

1 Introduction

This is a short, pedagogical, set of notes on the (self)-attention mechanism, and the deep learning architecture that utilizes this mechanism, transformers. Transformers have revolutionized Natural Language Processing (NLP) and, more recently, have also emerged as a powerful paradigm for computer vision applications (Vision Transformers - ViTs). Transformer-based architectures are behind most of the AI applications that have made it to the news recently, including chatGPT and DALL-E.

In this set of notes, our focus is on understanding the main concepts (and maths) behind the self-attention mechanism and the transformer block, which is the fundamental block that can be used to build more complicated transformer-based architectures. We are *not* going to describe any architecture or model (such as the Large Language Models as BERT, GPT etc) or how they are being trained. Furthermore, we start our exploration *in media res*, showing the modern self-attention mechanism and omitting the historical introduction about the attention mechanism as part of a Recurrent Neural Network model. We also do *not* cover variations of the attention mechanism that **do** exist. The interested reader can find some useful references at the end of this set of notes.

The power of transformers, over Recurrent Neural Networks (RNNs) in NLP, is that it can process a large number of sequence data in parallel, speeding up the process and avoiding problems such as vanishing gradients.

These notes are far from original. Despite the title, they are also limited, and you certainly **need** to study other resources if you want to better understand the attention mechanism, its historical evolution, alternatives, and different transformer-based architectures. I refer the interested reader to some excellent tutorials and papers in Sec. 4.

2 Self-attention

2.1 Basic form

The most important mechanism behind any modern transformer is the **self-attention** mechanism. We will start by presenting a very simple version of it, that clearly illustrates the basic idea behind it, and then we can generalize to present the form of self-attention as it is actually used in transformer architectures.

Let's suppose we have an input sequence of d-dimensional *vectors* $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. These can represent a (flattened) part of an image, be vector representations of words (word embeddings) etc. The point is that *inputs of similar context are represented by similar vectors.* The self-attention mechanism is nothing but an **operation** that **transforms** the input sequence to an output one, $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$, where each element of the output is a weighted sum of the input sequence:

$$\mathbf{y}_i = \sum_j W_{ij} \mathbf{x}_j. \tag{1}$$

The weight matrix is derived from the inputs; specifically, we want to give higher weight to those inputs that are more similar to $\mathbf{x}_i$, so first we calculate the dot product:

$$w_{ij} = \mathbf{x}_i^T \mathbf{x}_j. \tag{2}$$

However, the above dot product can take any possible value; we want to convert it to weights that are in the range [0,1] and their sum equals to 1, so we pass this result through a softmax function to generate the final weights W_{ij}:

$$W_{ij} = \frac{\exp w_{ij}}{\sum_j \exp w_{ij}} = \text{softmax}(w_{ij}) \tag{3}$$

That's it! This is the attention mechanism! The initial sequence of vectors has been **transformed** into a different one, a weighted combination of all the other vectors, with weights being larger for input vectors that are more similar (=attends more to them).

2.2 Adding learnable weights: query, key, and value parametrization

We haven't discussed why the above method of transforming a sequence can be beneficial in machine learning tasks, but this can be better (and more intuitively) understood once we add and (and discuss) one more layer of complexity. Let's do that!

The first thing to notice in the above formulation is that we have not introduced any learnable parameters so far; the weights W_{ij} are solely determined by the input vectors. A model based on that basic operation cannot be optimized for a specific task (e.g. classification) via the standard machine learning methods, but only by changing the input sequence embeddings.

In the above formulation, every vector $\mathbf{x}_i$ is used three times in three different roles (that's easier to see if we write the sum directly as $y_i = \sum_j \text{softmax}(x_i^T x_j) x_j$). This allows us to introduce three *learnable* weight matrices, $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_W$, that can be multiplied with the input vectors to give three new vectors, called Query, Key, and Value:

$$\text{Query: } \mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i \tag{4}$$

$$\text{Key: } \mathbf{k}_i = \mathbf{W}_K \mathbf{x}_i \tag{5}$$

$$\text{Value: } \mathbf{v}_i = \mathbf{W}_V \mathbf{x}_i \tag{6}$$

The names come from the three different functions. In the basic form of self-attention, the vector $\mathbf{x}_i$:

1. Compared with all other vectors (similarity) to create the weights for the output $\mathbf{y}_i$. (Query)
2. Compared with all the other vectors (similarity) to create the weights for the output $\mathbf{y}_j$ - notice the difference in the index. (Key)
3. Once the weights have been calculated, it is used in the sum to construct the output (Value).

The matrices $\mathbf{W}_Q \in \mathbb{R}^{d_q \times d}$, $\mathbf{W}_K \in \mathbb{R}^{d_k \times d}$, $\mathbf{W}_V \in \mathbb{R}^{d_v \times d}$ are *projection* matrices that project the vectors $\mathbf{x}_i$ into the lower-dimensional query, key, and value spaces. These can have, in principle, different dimensions, but in practice many times we choose $d_q = d_k = d_v$.

The attention weights can now be calculated via the **scaled dot-product** between the query and key vectors (the scaling is there for the stability of gradient calculations during training):

$$W_{ij} = \text{softmax}\left(\frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_k}}\right) \tag{7}$$

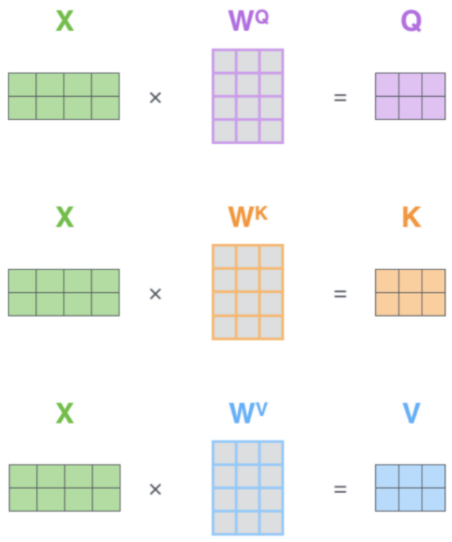


Figure 1: The matrix containing the input embeddings is multiplied with the three weight matrices, to produce the query, key, and value embeddings, respectively.

The output sequence is thus the weighted sum of values, where the attention weights are calculated as above:

$$\mathbf{y}_i = \sum_j W_{ij} \mathbf{v}_j. \tag{8}$$

In practice, we can define matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, whose $i-$th row being the vectors $\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i$, respectively. Fig. 1 shows how these matrices are created from the input and the weight matrices. Then, the **scaled dot-product** attention can be expressed parsimoniously in matrix form as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{1}{\sqrt{d_k}}\mathbf{QK}^T\right)\mathbf{V}, \tag{9}$$

where $\text{Attention}(Q, K, V) = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n]$. The graphical representation of this operation can be seen in Fig. 2. The product $\mathbf{QK}^T$ is usually referred to as the similarity matrix between the query and the key.

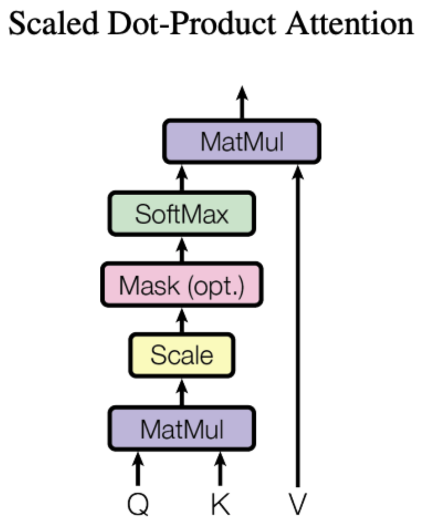
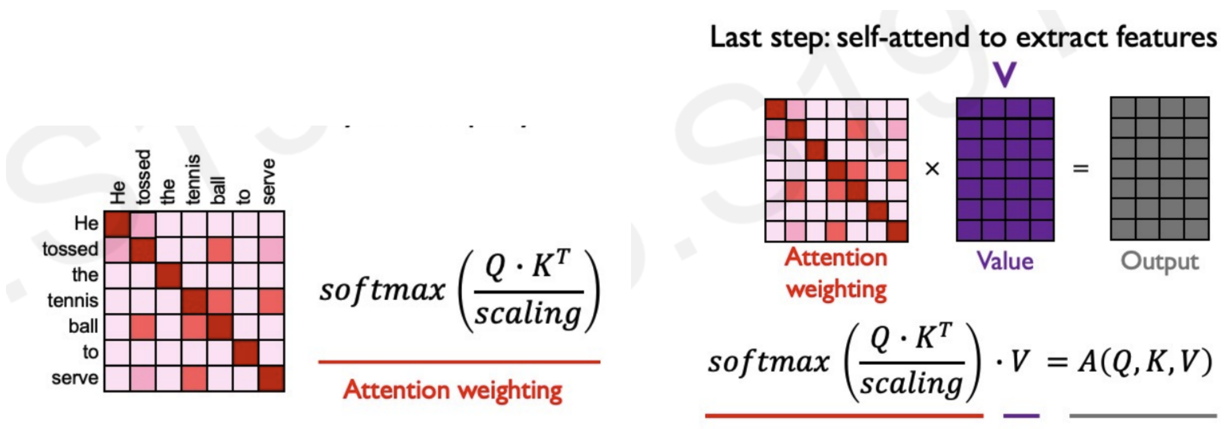


Figure 2: Graphical representation of the scaled dot-product attention, Eq.(9). From the original "Attention is all you need" paper [1].

This similarity matrix is used to create the attention weights that capture the relevance of the combination of inputs ($\sim$ correlation) or the ML task at hand (e.g. classification, machine translation, etc.). On the left-hand side of Fig. 3 we can see the attention matrix for an NLP task where the inputs are words and the attention weights correspond to the most informative word combinations. Then these attention weight matrices can be used to multiply the matrix of value vectors to *extract* important features that can be subsequently passed to the next stage of the network (right-hand side of Fig. 3).

2.3 Multi-head attention

In the above we introduced the set of learned weight matrices $(\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V)$, combined known as an *attention head.* Those matrices learn useful relations between the input vectors (e.g. between words in a sentence or parts of an image - where a given value vector should *attend* more for a task such as classification or regression).



He
tossed
the
tennis
ball
to
serve
$softmax\left(\frac{Q\cdot K^T}{scaling}\right)$
Attention weighting
Last step: self-attend to extract features
V
×
=
Attention weighting
Value
Output
$softmax\left(\frac{Q\cdot K^T}{scaling}\right)\cdot V = A(Q,K,V)$

Figure 3: The attention weights capture the similarity/relevance between the input vectors (left). Then these weights can be used to extract features that can be subsequently passed to a deep network (right). Figures are taken from [5].

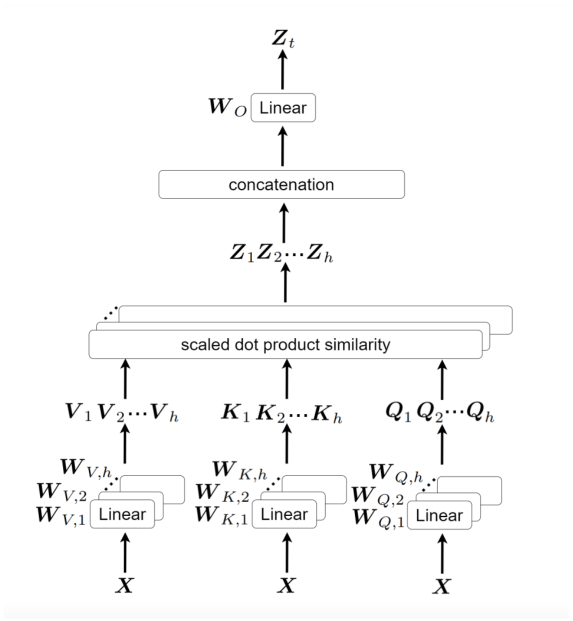


Figure 4: Multi-head attention with h attention heads. From [4].

Similarly to the case of CNNs, where one introduces multiple filters (learned weight kernels) to learn different types of features from an image, we can introduce multiple (learnable) attention heads that can capture different relationships between the input tokens.

The operations described in the previous section can be repeated h times, *in parallel*, producing matrices $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i,\ i = 1, \ldots, h$, as shown in Fig. 4. The multi-head attention output is then produced as:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O, \tag{10}$$

where:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \tag{11}$$

and $\mathbf{W}^O \in \mathbb{R}^{d \times hd_v}$ is a linear projection matrix that is used in order to project the concatenated vector to the output model space (gives the output array of vectors $\mathbf{y}_i$ that can be subsequently pass through feed-forward layers etc).

2.4 Positional Encoding

The self-attention mechanism, as we have described it so far, is oblivious to the position of the inputs ($\mathbf{x}$). In other words, unlike CNNs and RNNs, the Transformer architecture is invariant under permutations. However, for text sequences (sentences) or images, the order (or position) of the input token is important. A sentence can have a very different meaning (if it has any) depending on the order of words that compose it.

In order to include positional information in the transformer architecture, one can add a vector $\mathbf{p}_i \in \mathbb{R}^d$, where i denotes the position in the sequence, to the embedding vector $\mathbf{x}_i$ (remember that d denotes the embedding dimension):

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{p}_i. \tag{12}$$

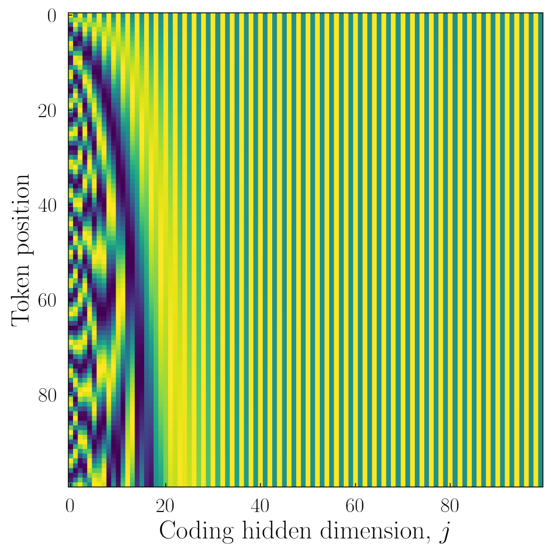


Figure 5: Plot of the values of the elements of the positional encoding vectors for tokens (inputs) at different positions. We use $d = 100$, the encoding dimension, and we consider $n = 100$ inputs, too.

The elements of the position vector $\mathbf{p}_i$ can be learned (in that case it is called a positional *embedding*) or set to have a specific form that *encodes* the position. In the original attention paper, a sinusoidal *positional encoding* vector was proposed, of the form:

$$\begin{cases} \mathbf{p}_i(2j+1) = \cos\left(\frac{i}{10000^{2j/d}}\right), \\ \\ \mathbf{p}_i(2j) = \sin\left(\frac{i}{10000^{2j/d}}\right), \end{cases} \tag{13}$$

For $j \in \{0, 1, \ldots, \lfloor d/2 \rfloor\}$ denoting the elements of the vector, where $\lfloor x \rfloor$ is the floor function, denoting the greatest integer that is not greater than x. In Fig. 5 we plot the elements of the position vector (coding hidden dimension $d = 100$) for an input of 100 tokens.

More recent transformer architectures (such as the GPT family) use learned positional embeddings, meaning that the network itself is let free to find the optimal vector elements that encode positional information instead of imposing a specific functional form.

3 The transformer block

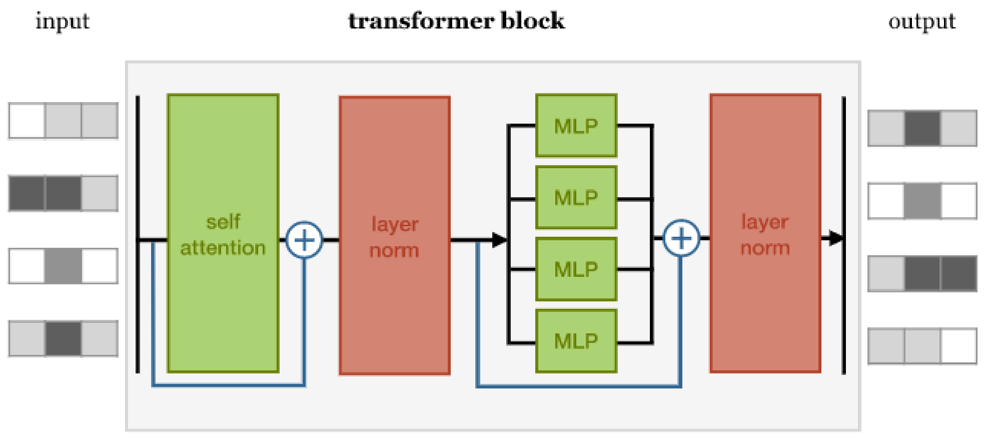


Figure 6: Graphical representation showing the main elements of a transformer block. Adapted from [2].

We have casually mentioned transformers a few times so far, but what is actually a transformer? Although initially the term transformers was a reference to the architecture proposed in [1] for the task of machine translation, we can now call a **transformer any architecture that has a transformer block as a basic building element**. The main mechanism inside a transformer block is, in turn, the multi-head self-attention we previously described (note that other mechanisms beyond the scaled dot-product self-attention exist, but their description is beyond the scope of these notes).

We can see a graphical representation of a typical transformer block (although variations do exist) in Fig. 6. It is composed of a self-attention layer, followed by *layer normalization*, a series of standard multi-layer perceptrons (MLPs) forming a feed-forward layer, and one more layer normalization. There are also some skip connections (blue arrows). Let's discuss in a bit more detail those other elements of a transformer block.

- Skip connections: Similar to the skip connections that one can find in residual neural networks (ResNets), the skip connection adds the input $\mathbf{X}$ to the output of the multi-head attention. If we denote as $\mathbf{Z} = \text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$, we get that after the skip connection:

  $$\mathbf{Z}' = \mathbf{Z} + \mathbf{X}. \tag{14}$$

  where $\mathbf{X}$ is a matrix containing the input vectors (including the positional encodings). This help retain some information about the original sequence.

- Layer normalization: Layer normalization is a variation of the batch normalization that you may have seen in other discussions of deep learning architectures and performance optimization. Let's assume a hidden layer with H hidden units, h_i. Layer normalization standardizes those hidden units and makes them have zero mean and unit variance by first calculating the mean and variance over the H units:

  $$\mu = \frac{1}{H}\sum_{i=1}^{H} h_i, \tag{15}$$

  $$\sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(h_i - \mu)^2}. \tag{16}$$

  And then redefining the output of every hidden unit as:

  $$\hat{h}_i = \gamma \frac{h_i - \mu}{\sigma}, \tag{17}$$

  γ being a trainable parameter. Layer normalization improves both the training time and the generalization performance of the transformer.

- Multi-layer perceptron (MLP): A single MLP is applied independently to each output vector after the layer normalization. This MLP adds extra complexity to the model and allows transformations on each sequence element separately.

  That's it! We have covered all the main elements of the transformer block, and in principle, you should be able with minimal extra effort to understand the construction of more complicated transformer-based architectures and models, like the original transformer encoder-decoder model, the GPT family etc.

4 References and where to go from here

This set of notes gave a very basic description of the self-attention mechanism. Also, far from being original, I based my notes on a number of excellent tutorials that exist out there. I strongly recommend the following resources for a deeper understanding of transformers, their historical development, and alternatives to the scaled dot-product attention described here:

[1] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L.,

et al., 2017, arXiv:1706.03762.: *The original "Attention is all you need" paper that started the Transformer revolution.*
[2] `https://peterbloem.nl/blog/transformers`: *Excellent blog post; our treatment here was* ***heavily*** *based on it. Strongly recommended*
[3] `https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html?utm_source=substack&utm_medium=email`:
[4] `https://paperswithcode.com/paper/attention-mechanism-transformers-bert-and-gpt` *Nice, technical, survey paper. Also covers the historical development of the attention mechanism, as a mechanism to improve RNNs,*
[5] `http://introtodeeplearning.com/`: *Excellent introduction to deep learning, including sequence models and attention.*
[6] `https://lilianweng.github.io/posts/2018-06-24-attention/`: *A great blog post by Lilian Weng. Discusses the evolution and different flavors of the attention mechanism in detail. Great if you want to learn more about the concept of attention, beyond the simple scaled dot-product attention discussed here.*
[7] `https://lilianweng.github.io/posts/2020-04-07-the-transformer-family/`: *Very good discussion of different transformer types.*
[8] Dosovitskiy A., Beyer L., Kolesnikov A., Weissenborn D., Zhai X., Unterthiner T., Dehghani M., et al., 2020, arXiv, arXiv:2010.11929: *The paper that introduced Vision Transformers (ViT).*
[9] `https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html?utm_source=substack&utm_medium=email`: *Blog post with code that helps someone to understand the attention mechanism.*
[10] `https://nlp.seas.harvard.edu/2018/04/03/attention.html`: *Provides PyTorch code for each of the elements of the original transformer paper [1].*
[11] `https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html`