The Transformer Architecture

HY-673, 2023-2024, Spring

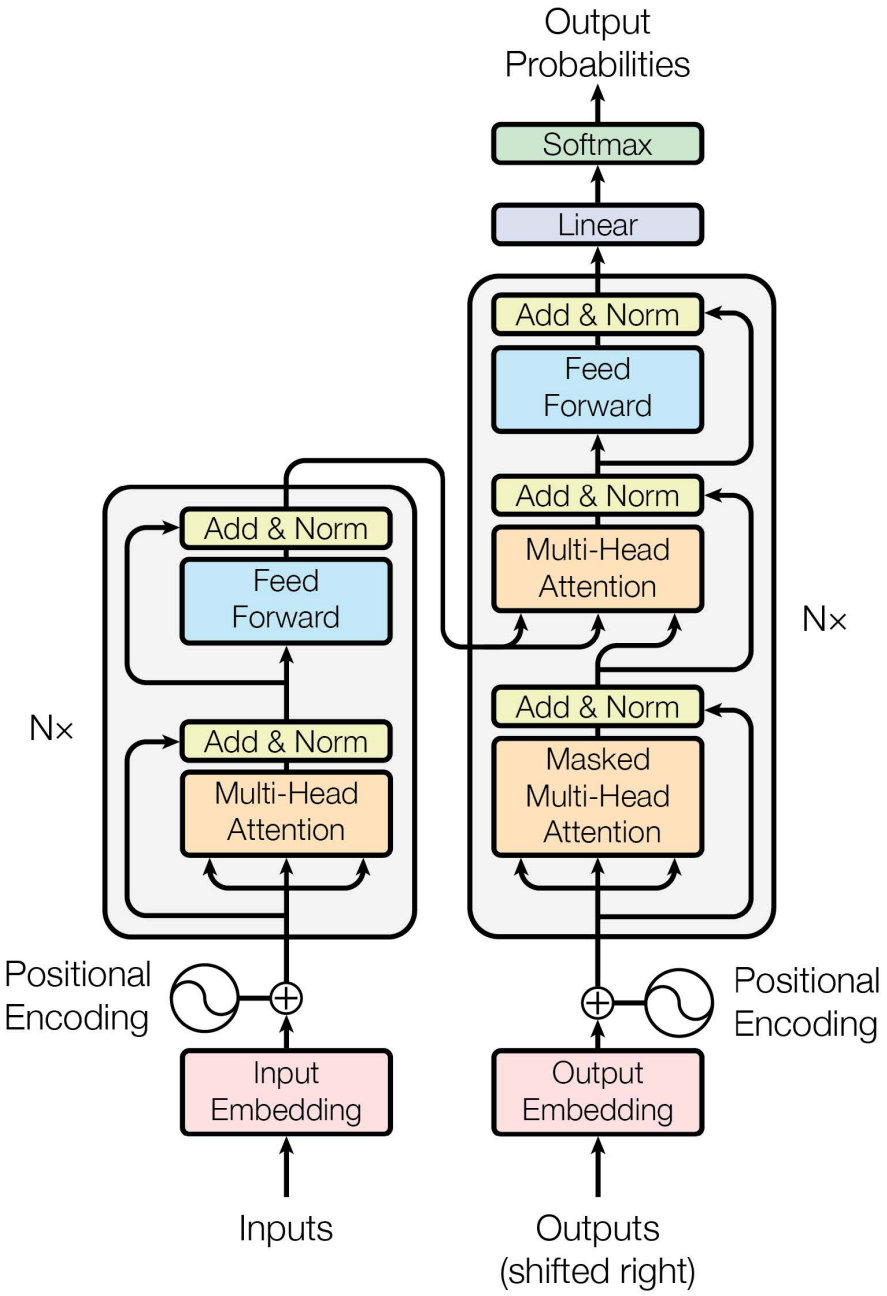

Figure 1: The original transformer architecture diagram.

1 Introduction

In the seminal paper "Attention Is All You Need"[3], the authors introduce a novel architecture that addresses many of the shortcomings of RNN-based models in the context of machine translation. Despite advances in RNN encoder-decoder models, their sequential processing nature presents challenges for parallelization. The Transformer architecture replaces the reliance on RNNs and their hidden states entirely with attention-based operations, enhancing efficiency across various problem domains. This innovation significantly reduces bottlenecks in training state-of-the-art (SOTA) models, facilitating the implementation of large-scale models on extensive datasets.

2 High-Level Overview

The Transformer, depicted in fig. 1, embodies an encoder-decoder structure. Both the encoder (left side) and the decoder (right side) are comprised of N total blocks. The ensuing sections delve into the Transformer's components in greater detail, starting with a broad overview.

2.1 Embeddings

Consider a sequence, denoted F, as we briefly overview the embeddings and positional encodings:

1. **Tokenization**: F is tokenized into individual elements (e.g., words), represented as $F = (f_1, \ldots, f_m)$.
2. **Embedding**: Each token f_i is mapped to a high-dimensional vector $\mathbf{u_i}$ using the *embedding layer*.[1]
3. **Positional Encoding**: Positional encodings are added to the embedded vectors $\mathbf{u_i}$, imparting information about each token's position within the sequence. This addition is crucial because the (self) attention mechanism inherently lacks a sense of token order or position, in contrast to layers such as LSTMs or GRUs, which naturally encode sequential information due to their recurrent structure.

2.2 Encoder

Following the embedding process (section 2.1) for an input sequence during training, the resultant vectors are concurrently fed into the first encoder block. The output from one block is passed to the next, continuing through all N encoder blocks. Only the output of the final encoder block is directly used by the decoder, but it is used throughout all the decoder blocks.

Each encoder block comprises two primary components: (i) a Multi-Head Self-Attention layer and (ii) a Feed-Forward (FFN) block, with an intervening "Add & Norm" step. The "Add" denotes a residual connection adding the input to the output of each layer, while "Norm" refers to Layer Normalization[2] introduced in [4]. More details regarding Multi-Head attention will be discussed in 4.

The FFN block is a fully-connected neural network that applies two linear transformation with a ReLU activation in between. This enables the model to learn more complex relationships, and is applied to each position separately and identically. This means that the same FFN is used for all positions in the sequence, but it operates on them independently, ensuring that the model can maintain its ability to process input sequences in parallel. Mathematically, the FFN operation given an input x can be described as:s

$$\mathrm{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2, \tag{1}$$

where W_1 and W_2 are trainable weight matrices, b_1 and b_2 are the bias vectors, and $\mathrm{ReLU}(z) \triangleq \max(0, z)$.

[1] The embedding can be initialized randomly and learned during training, or pre-trained from other models.

[2] TL;DR: It subtracts the mean and divides by the standard deviation.

2.3 Decoder

The embedding process described earlier (section 2.1) also applies to the target (ground truth) sequences $E = (e_1, \ldots, e_\ell)$, with "shifted right" indicating the addition of a start-of-sequence token at the beginning and subsequent shifting of each token one position to the right. This shifting ensures that the model predicts the next word based on the preceding words in the target sequence during decoding.

Next, each decoder block comprises three main components: (i) a Masked Multi-Head Self-Attention, (ii) a Multi-Head Attention layer that connects the encoded source representation to the decoder, and (iii) a Feed-Forward (FFN) block. Similar to the encoder, each layer within the decoder is followed by an "Add & Norm" step.

There are notable differences between the encoder and decoder to highlight. Firstly, the inputs to the Masked Multi-Head Self-Attention layer in the decoder blocks are *masked*, which prevents each position from attending to subsequent positions in the sequence. This masking is crucial for ensuring that the prediction for a given position can only depend on known outputs, i.e., positions before it. The rationale behind this is straightforward: during inference, the model generates its predicted translation $\hat{E}$ one word at a time, based on the source sentence F. When predicting a particular word $\hat{e}_i$, the model can access only the words that have been generated so far, i.e., $\hat{e}_{<i}$. It cannot "see" future words $\hat{e}_{>i}$ because they depend on $\hat{e}_i$. Thus, masking is employed during training to mimic the sequential generation conditions the model encounters during inference. Importantly, during training, this masking enables the model to **simultaneously** predict the output for each position in a **single** forward pass.

Another distinction from the encoder is the presence of the second Multi-Head Attention layer, also known as the Encoder-Decoder Attention layer. Unlike the self-attention mechanisms at the start of each encoder and decoder block, its primary function is to integrate the encoded representation of the source sentence F with the target sequence in E, enabling the model to focus on relevant parts of the source sentence when generating each word of the translation. This Encoder-Decoder Attention mechanism is pivotal for aligning source and target language content, thereby enabling effective translation.

The final stages of the Transformer decoder involve a linear transformation followed by a softmax function, which produces output probabilities for each token in the target sequence. During training, the model uses the ground truth target sequence for the entire sentence and computes the loss for every word simultaneously, leveraging the parallel processing capabilities of the architecture. This contrasts with traditional sequence generation models like RNNs, where each word's prediction is made sequentially. However, during inference, the Transformer generates the output sequence one word at a time, feeding the selected word back into the decoder to generate the next one, until an end-of-sequence token is produced or a maximum sequence length is reached. This selection can be the word with the highest probability (greedy decoding) or use a more sophisticated method like beam search, which considers multiple possible sequences to find a more probable sequence of words.

3 Scaled Dot-Product Attention

Within the transformer model, an attention layer has three primary matrices: the query matrix $Q \in \mathbb{R}^{m \times d_k}$, the key matrix $K \in \mathbb{R}^{n \times d_k}$, and the value matrix $V \in \mathbb{R}^{n \times d_v}$. The query matrix is multiplied by the transpose of the key matrix. Then, this product is normalized with respect to the square root of the dimensionality of the keys d_k, and each row is passed through a softmax function to produce a probability vector. The resulting matrix, denoted $S \in \mathbb{R}^{m \times n}$ and referred to as the attention matrix (or attention weights), is then multiplied by the value matrix V to generate the output $O \in \mathbb{R}^{m \times d_v}$ of the attention layer:

$$\text{Attention}\,(Q, K, V) \triangleq O \triangleq S \cdot V \triangleq \text{softmax}\left(\frac{Q \cdot K^\intercal}{\sqrt{d_k}}\right) \cdot V. \tag{2}$$

Note that for the algebra to work out, the number of keys and values n must be equal, but the number of queries m can vary. In the case of **self-attention**, an input matrix $X \in \mathbb{R}^{m \times d_{\text{model}}}$ is projected into Q, K and V using separate trainable weights $W_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, that is:

$$Q = X \cdot W_Q,\ K = X \cdot W_K,\ V = X \cdot W_V. \tag{3}$$

We will start by writing $Q,\ K$ in terms of rows $\mathbf{q_i},\ \mathbf{k_j}$, and then express the product $QK^\intercal$ in terms of them:

$$Q = \begin{bmatrix} — & \mathbf{q_1} & — \\ — & \mathbf{q_2} & — \\ & \vdots & \\ — & \mathbf{q_m} & — \end{bmatrix},\ K^\intercal = \begin{bmatrix} | & | & & | \\ \mathbf{k_1} & \mathbf{k_2} & \dots & \mathbf{k_n} \\ | & | & & | \end{bmatrix} \implies Q \cdot K^\intercal = \begin{bmatrix} \mathbf{q_1} \cdot \mathbf{k_1} & \mathbf{q_1} \cdot \mathbf{k_2} & \dots & \mathbf{q_1} \cdot \mathbf{k_n} \\ \mathbf{q_2} \cdot \mathbf{k_1} & \mathbf{q_2} \cdot \mathbf{k_2} & \dots & \mathbf{q_2} \cdot \mathbf{k_n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{q_m} \cdot \mathbf{k_1} & \mathbf{q_m} \cdot \mathbf{k_2} & \dots & \mathbf{q_m} \cdot \mathbf{k_n} \end{bmatrix}, \tag{4}$$

where each $\mathbf{q_i} \cdot \mathbf{k_j}$ is a dot (or inner) product, hence the name "Scaled Dot-Product Attention". As a reminder, for vectors, say, $\mathbf{a} = (a_1, a_2, \dots, a_n),\ \mathbf{b} = (b_1, b_2, \dots b_n)$, the inner product between them is calculated as:

$$\mathbf{a} \cdot \mathbf{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots a_n \cdot b_n. \tag{5}$$

Next, we obtain our attention matrix (or attention weights) S by scaling each element by $1/\sqrt{d_k}$ and applying the softmax function row-wise. The scaling happens in order to prevent the dot products from becoming too large in magnitude, which in turn aids in maintaining more stable gradients. This scaling has become a standard component of attention mechanisms in many subsequent models and architectures that utilize attention. Visit [1] to read more on the matter of scaling. In any case, S will be:

$$S = \text{softmax}\left(\frac{Q \cdot K^\intercal}{\sqrt{d_k}}\right) = \begin{bmatrix} \text{softmax}\left(\frac{1}{\sqrt{d_K}}\left\langle \mathbf{q_1} \cdot \mathbf{k_1}, \mathbf{q_1} \cdot \mathbf{k_2}, \dots, \mathbf{q_1} \cdot \mathbf{k_n} \right\rangle\right) \\ \text{softmax}\left(\frac{1}{\sqrt{d_K}}\left\langle \mathbf{q_2} \cdot \mathbf{k_1}, \mathbf{q_2} \cdot \mathbf{k_2}, \dots, \mathbf{q_2} \cdot \mathbf{k_n} \right\rangle\right) \\ \vdots \\ \text{softmax}\left(\frac{1}{\sqrt{d_K}}\left\langle \mathbf{q_m} \cdot \mathbf{k_1}, \mathbf{q_m} \cdot \mathbf{k_2}, \dots, \mathbf{q_m} \cdot \mathbf{k_n} \right\rangle\right) \end{bmatrix} \tag{6}$$

$$= \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{bmatrix}. \tag{7}$$

Each element $s_{i,j}$ essentially represents the weight (or importance) that the i-th query places on the j-th key. Regarding the softmax function, which is also known as softargmax or normalized exponential function, it converts a vector of N real numbers into a probability distribution of N possible outcomes. In words, softmax applies the standard exponential function to each element z_i of the input vector $\mathbf{z}$, and normalizes these values by dividing by the sum of all these exponentials. Hence, for a vector $\mathbf{z}$ of N real numbers, the standard (unit) softmax: $\mathbb{R}^N \mapsto (0,1)^N,\ N \geq 1$, is defined by the formula:

$$\text{softmax}(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}},\ i = 1, \dots, N,\ \mathbf{z} = (z_1, \dots, z_N) \in \mathbb{R}^N. \tag{8}$$

As a result of the softmax operation, each row i of S contains non-negative elements and $\sum_{j=1}^{n} s_{i,j} = 1$. So, each row of the attention matrix S can be viewed as a discrete probability distribution, or probability mass function (PMF), over the keys for the given query. Finally, we multiply this matrix by V to get O:

$$\text{Attention}\,(Q, K, V) = O = S \cdot V = \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{bmatrix} \cdot \begin{bmatrix} — & \mathbf{v_1} & — \\ — & \mathbf{v_2} & — \\ & \vdots & \\ — & \mathbf{v_n} & — \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} s_{1,i} \cdot \mathbf{v_i} \\ \sum_{i=1}^{n} s_{2,i} \cdot \mathbf{v_i} \\ \vdots \\ \sum_{i=1}^{n} s_{m,i} \cdot \mathbf{v_i} \end{bmatrix} \tag{9}$$

Think of each query in Q looking at the keys K, deciding how important each key is (via the attention scores in S), and then using those importance scores to pull information from the corresponding values in V. The more a query "cares" about a specific key, the more it will "pull" from the corresponding vector $\mathbf{v_j}$ when constructing its own aggregated information vector. The takeaway here is that the Attention mechanism results in a series of weighted averages of the rows of V, where the weighting depends on the input queries and keys. Each of the m queries of Q results in a specific weighted sum of the value vectors. Now, let's zoom in on an individual row i of S:

$$\text{softmax}\left(\frac{1}{\sqrt{d_K}}\left\langle \mathbf{q_i}\cdot\mathbf{k_1},\ldots,\mathbf{q_i}\cdot\mathbf{k_n}\right\rangle\right) = \frac{1}{L}\left\langle \exp\left(\frac{\mathbf{q_i}\cdot\mathbf{k_1}}{\sqrt{d_k}}\right),\exp\left(\frac{\mathbf{q_i}\cdot\mathbf{k_2}}{\sqrt{d_k}}\right),\cdots,\exp\left(\frac{\mathbf{q_i}\cdot\mathbf{k_n}}{\sqrt{d_k}}\right)\right\rangle, \tag{10}$$

where L is the normalization constant $L = \sum_{j=1}^{n}\exp\left(\frac{\mathbf{q_i}\cdot\mathbf{k_j}}{\sqrt{d_k}}\right)$. Looking at how S is constructed, the origin of the names "queries," "keys," and "values" is clearer. As in a hashtable, this operation picks out desired values via corresponding, one-to-one keys. The keys that we seek are indicated by the queries, and we can express the dot product between a given key and query in terms of the angle θ between them:

$$\mathbf{q_i}\cdot\mathbf{k_j} = |\mathbf{q_i}|\,|\mathbf{k_j}|\cos(\theta). \tag{11}$$

Ignoring magnitudes for a moment, in fig. 2 we see that exponentiation amplifies positive cosine values while diminishing negative ones (reducing their relative influence). Hence, the closer in angle (indicating stronger agreement) the key $\mathbf{k_j}$ and query $\mathbf{q_i}$ are, the more pronounced their presence in the attention vector becomes.

Revisiting machine translation, each row of the keys, queries, and values in this setting represents a vectorized depiction of elements within a sequence. A notable limitation of RNN-based models is their struggle to effectively utilize information from elements observed far in the past, i.e., they face challenges in connecting sequential information that is spaced widely apart. Techniques such as leveraging attention on hidden states and utilizing bidirectional models were devised to address this shortcoming. Scaled dot-product attention, specifically, facilitates the efficient and rapid association of each element in a sequence with all elements in another sequence, as well as with every other element within the same sequence.

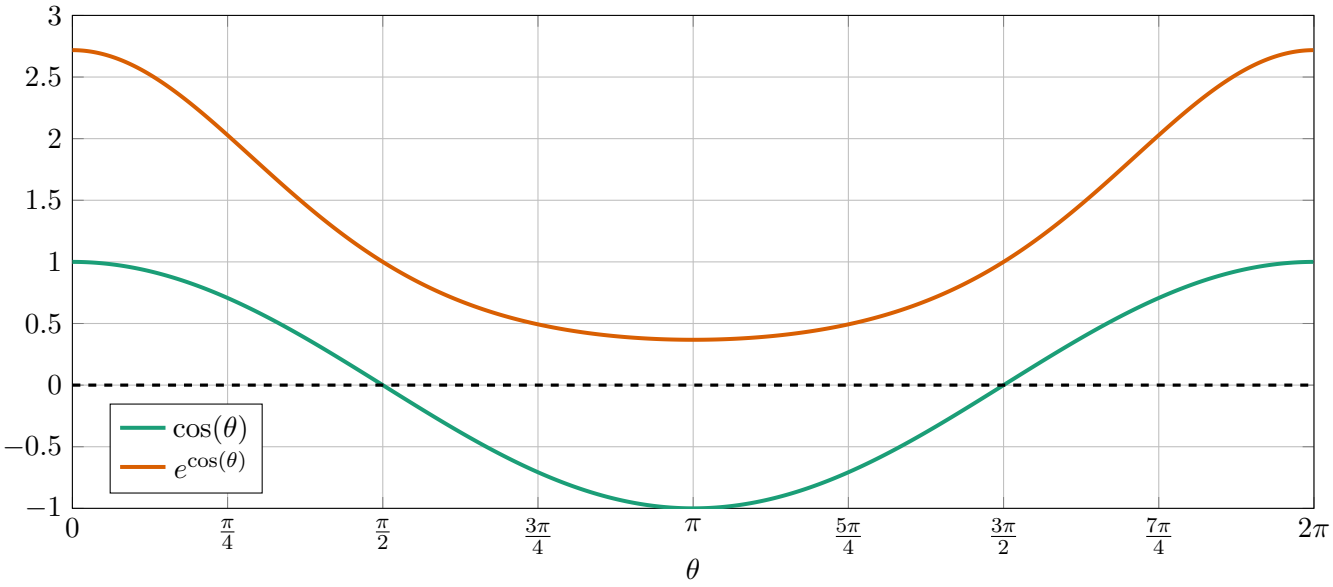

Figure 2: Exponentiated cosine and normal cosine plots from 0 to 2π.

4 Multi-Head Attention

Multi-Head Attention is an extension of Scaled Dot-Product Attention, in which different linear transformations are applied to the queries, keys, and values, producing different sets of inputs for the attention function. These operations are performed **in parallel** for all input sets, and their outcomes (heads) are concatenated side-by-side. The final result is achieved through a linear transformation of this concatenated matrix with the trainable output weight matrix W_O, resulting in a matrix with the desired dimensionality:

$$\text{MultiHead} = \text{Concatenate}\left(O^{(1)}, \ldots, O^{(h)}\right) \cdot W_O. \tag{12}$$

Each output $O^{(i)}$ is called a **head**, hence the name Multi-Head Attention. The i-th head is simply the result of running the Scaled Dot-Product Attention on the i-th set of transformed queries, keys, and values, i.e.:

$$O^{(i)} = \text{Attention}\left(Q^{(i)}, K^{(i)}, V^{(i)}\right). \tag{13}$$

Likewise, given an input matrix $X \in \mathbb{R}^{m \times d_{\text{model}}}$, then in the case of **self-attention**, those will be:

$$Q^{(i)} = X \cdot W_Q^{(i)},\ K^{(i)} = X \cdot W_K^{(i)},\ V^{(i)} = X \cdot W_V^{(i)}, \tag{14}$$

where given a hyperparameter h indicating the number of attention heads, we have that $W_Q^{(i)} \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_K^{(i)} \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_V^{(i)} \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W_O \in \mathbb{R}^{(h \cdot d_v) \times d_{\text{model}}}$.

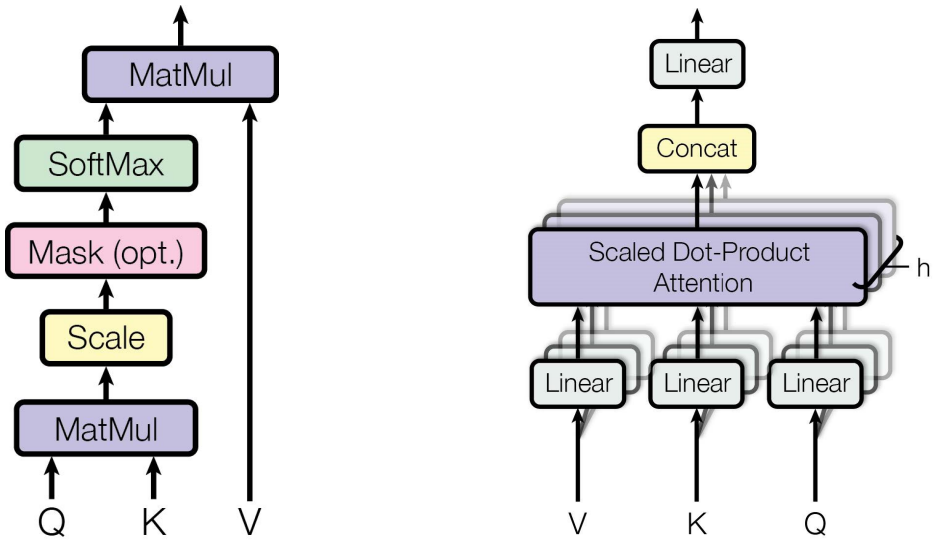

Figure 3: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Let's perform a quick sanity check that the matrix multiplication works out. Initially, we understand that each output from a head, $O^{(i)}$, will possess the same number of rows as $Q^{(i)}$ and an equivalent number of columns as $V^{(i)}$. Given that $Q^{(i)}$ is dimensioned as $\mathbb{R}^{m \times d_k}$ and $V^{(i)}$ as $\mathbb{R}^{n \times d_v}$, it follows that $O^{(i)}$ is dimensioned as $\mathbb{R}^{m \times d_v}$. Concatenating h such outputs yields a matrix of dimensions $\mathbb{R}^{m \times (h \cdot d_v)}$. The subsequent multiplication by W_O produces a matrix of dimensions $\mathbb{R}^{m \times d_{\text{model}}}$. This progression is logical: beginning with m queries in Q and culminating in m responses via the MultiHead operation reflects the intended transformation and dimensional consistency.

Observe that each head in the computation undergoes a distinct linear transformation for the key, query, and value matrices, with these transformations being learned during the training process. For those acquainted with convolutional neural networks (CNNs), the concept of heads and their corresponding weight matrices can be analogously understood as akin to the various channels and their learned kernels within CNN layers. Both mechanisms initiate from the same input and concurrently generate multiple representations of this input, each unveiling unique facets of the data.

5 Conclusion

In this document, we explored the fundamental components of the Transformer architecture as introduced in "Attention is All You Need". Grasping these elements from first principles is pivotal for delving into the myriad Transformer-based models that followed. For instance, BERT, as described in [5], utilizes a series of encoder blocks, trained with a novel objective function tailored to the encoder's inherent bidirectionality. Similarly, the GPT series[6], expands on this by incorporating increasingly extensive sequences of decoder blocks. The pace of research and development in Transformer-based models remains rapid and vibrant. Recent studies have unveiled Transformers' efficacy across diverse domains, including computer vision problems, within GANs, and in interdisciplinary fields linking natural language processing to computer vision, exemplified by OpenAI's DALL · E project [7]. Additionally, there has been significant research focused on rendering Transformers more compact and efficient. On the development front, the community continues to unveil large, pre-trained Transformer models. These models, accessible for immediate application or further fine-tuning, cater to a broad spectrum of specific use cases.

References

[1] J. S. Lee, "Transformers: a primer," 2021. [Online]. Available: https://www.columbia.edu/~jsl2239/transformers.html#attention_paper

[2] wikipedia, "Softmax function." [Online]. Available: https://en.wikipedia.org/wiki/Softmax_function

[3] 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.

[4] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] wikipedia, "Chatgpt." [Online]. Available: https://en.wikipedia.org/wiki/ChatGPT

[7] ——, "Dalle." [Online]. Available: https://openai.com/research/dall-e