Understanding Transformers: The Math behind ChatGPT

Grant Yang
Period 4

Introduction

First proposed in the famous 2017 paper "Attention is All you Need" by Vaswani et al. [1], the transformer architecture is the basis of all modern large language models. Originally intended as an architecture to tackle machine translation, it has proven to be incredibly generalizable and has found applications in diverse areas ranging from computer vision (in Vision Transformers) to audio transcription (in OpenAI's Whisper). In this paper, I aim to give the reader a solid understanding of the mathematics and low-level mechanisms that power these cutting-edge technologies.

Overview

Fundamentally, transformers operate on a set of nodes (i.e. vectors), which are arranged in some graph. Computations are then performed in two steps: exchanging information using an *attention mechanism*, followed by individual processing. Thus, the connections in the graph would indicate how information can flow between nodes during the information exchange step. After the initial talking phase, there is a thinking phase, where each node is individually processed using a traditional feed-forward neural network. These two steps make up a single *block* of computation, which is repeated multiple times.

The end result is a transformed version of the original set of nodes. In the case of a large language model, each node would represent a vectorized word or *token*, so an LLM might transform ["A", "quick", "brown", "fox"] into ["quick", "brown", "fox", "jumped"], repeating this process until an end-of-stream token is reached. However, a complication is the fact that transformers have no sense of the order of the nodes, so a *positional encoding* must be added to the tokens before they are passed to the transformer. Training a transformer to predict the next token is called *pre-training*, and modern chatbots also undergo a *fine-tuning* step where human feedback is incorporated using the *proximal policy optimization* reinforcement learning algorithm.

The Attention Mechanism

As a first step for performing attention, nodes generate three vectors: the *query*, *key*, and *value* vectors. The query vector encodes the type of information that the node is looking for, the key vector encodes the type of information the node has, and the value vector encodes the actual content of the node's information. Each node that is looking to share information generates a key and a value vector, and each node that is looking to gather information generates a query vector. Note that nodes can both gather information and share information at the same time. The query and key vectors (but not the value vector) must have the same dimension, since we will need to take a dot product between them to see if a node is sharing the information that another node is looking for.

To formalize this and avoid any confusion between the node's value and the node's value vector, I will write the yth node's value as $\overrightarrow{n_y}$ and its corresponding value vector as $\vec{v_y}$. I will use this index y to index nodes that are sharing information and the index x for nodes that are gathering information. Thus, the

query and key vectors will be written as $\vec{q_x}$ and $\vec{k_y}$ respectively. All three of these vectors are generated by simple matrix multiplications, where the values of the nodes $\overrightarrow{n_x}$ or $\overrightarrow{n_y}$ are left-multiplied by a learnable weight matrix:

$$\vec{q_x} = W_Q \overrightarrow{n_x} \tag{Eq. 1}$$

$$\vec{k_y} = W_K \overrightarrow{n_y} \tag{Eq. 2}$$

$$\vec{v_y} = W_V \overrightarrow{n_y} \tag{Eq. 3}$$

To aggregate all the information that it gathers into a single vector $\vec{a_x}$, the node x takes the weighted average of the value vectors $\vec{v_y}$ of all the nodes y sharing information, weighted by a corresponding relevance score. This relevance score is computed by taking the dot product between the information-gathering node's query vector $\vec{q_x}$ and the information-sharing node's key vector $\vec{k_y}$. Summing y across all nodes that are sharing information to node x, we want to compute something similar to this sum:

$$\vec{a_x} \text{ is similar to } \sum_y \left(\vec{q_x} \cdot \vec{k_y}\right) \vec{v_y} \tag{Eq. 4}$$

To compute the true average $\vec{a_x}$ instead of this simplified sum, we use a function called *softmax*. The softmax function operates on vectors and maps all components into the range $[0, 1]$ while ensuring they sum to 1. The exponential function e^x is used to map negative values into the positive range. The jth component of a softmaxed vector can be expressed as the following, where k sums across the entire vector $\vec{u}$, and u_j and u_k are the jth and kth components:

$$\text{softmax}(\vec{u})_j = \frac{\exp(u_j)}{\sum_k \exp(u_k)} \tag{Eq. 5}$$

For the weighted average, we define a vector of all the relevance scores $\vec{q_x} \cdot \vec{k_y}$ and apply softmax to it. One can think of taking this vector and left-multiplying by a matrix of $\vec{v_y}$ vectors to compute the average:

$$\vec{r} \text{ is a vector where each component } r_i = \vec{q_x} \cdot \overrightarrow{k_{y_i}} \tag{Eq. 6}$$

$$\vec{a_x} = \left[\overrightarrow{v_{y_1}}, \overrightarrow{v_{y_2}}, ...\right] \text{ softmax}(\vec{r}) \tag{Eq. 7}$$

More sensibly, the weighted average $\vec{a_x}$ for the node x could be written as:

$$\vec{a_x} = \frac{\sum_y \exp\left(\vec{q_x} \cdot \vec{k_y}\right) \vec{v_y}}{\sum_y \exp\left(\vec{q_x} \cdot \vec{k_y}\right)} \tag{Eq. 8}$$

In practice, this $\vec{a_x}$ value is scaled by a factor of $1 / \sqrt{d_k}$, where d_k is the number of dimensions in the key vector. This keeps intermediate values in sensible ranges and makes training easier. The entire process of calculating the $\vec{a_x}$ value and scaling is called *scaled dot product attention*, and it makes up a single *attention head.*

Multiple attention heads can be used for *multiheaded attention.* Each head uses its own set of learnable W_Q, W_K, and W_V matrices to produce query, key, and value vectors and calculate the attention (i.e. the weighted averages $\vec{a_x}$). In theory, each attention head would be responsible for exchanging certain key

bits of information. The $\vec{a_x}$ vectors from all of the attention heads are concatenated together and left-multiplied by a learnable matrix W_O in a process called *output projection*. Finally, this value is added onto the node's value $\overrightarrow{n_x}$:

$$\overrightarrow{n_x} \to \overrightarrow{n_x} + W_O \text{ Concat}(\vec{a_x} \text{ from head}_1, \vec{a_x} \text{ from head}_2, ...) \quad \text{(Eq. 9)}$$

This concludes the attention mechanism, or talking step, as information from all of the $\overrightarrow{n_y}$ vectors has been added into $\overrightarrow{n_x}$.

Individual Processing

Next, we move onto the thinking step: the node's value $\overrightarrow{n_x}$ is fed into a small two-layer neural network, and the output is added onto the node's value again. With some activation function f (usually ReLU or some variant of it), we can write this step mathematically as:

$$f(x) = \text{ReLU}(x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases} \quad \text{(Eq. 10)}$$

$$\overrightarrow{n_x} \to \overrightarrow{n_x} + W_2 f\left(W_1 \overrightarrow{n_x} + \vec{b_1}\right) + \vec{b_2} \quad \text{(Eq. 11)}$$

The weight matrices W_1 and W_2 as well as the bias vectors $\vec{b_1}$ and $\vec{b_2}$ are learnable, but the bias vectors are sometimes omitted. The threshold function is applied element-wise on the vector, and we skip applying it the second time to avoid limiting the output range.

Only using addition to modify the nodes' values helps to address the *vanishing gradient* problem, where the partial derivative of the cost function with respect to parameters shrinks to zero as neural networks grow deeper [2]. Because the final output is effectively a sum of the outputs of all intermediate layers, the gradient does not shrink as much when the transformer grows deeper. Each parameter is more directly connected to the final output, requiring fewer chain rule multiplications.

Layer Normalization

Another technique, *layer normalization*, is used to further alleviate the vanishing gradient problem [3]. For each application of this technique, two new learnable scalar parameters are introduced, γ and β. The layer's output is scaled and biased so that it has a mean of β and a standard deviation of $|\gamma|$. To apply layer normalization, the following formula is applied to each component x of the layer:

$$y = \beta + \gamma * \frac{x - E[x]}{\sqrt{\text{Var}[x] + 10^{-5}}} \quad \text{(Eq. 12)}$$

$E[x]$ is the expected value (mean) and Var[x] is the variance (the square of standard deviation). Values for the mean and variance are estimated using past outputs produced by the layer. A small factor ε (in this case 10^{-5}) is added to avoid division by zero.

In the original 2017 paper, layer normalization was applied to $\overrightarrow{n_x}$ node values twice per block: once after adding the $\vec{a_x}$ attention values and again after adding the feed-forward network [1]. However, many modern transformers use a different model proposed in a later paper, which showed that applying layer normalization before performing attention and before the feed-forward network provides better results and faster training times [4]. Only the values passed into the attention mechanism and neural network are normalized, not the stored values of $\overrightarrow{n_x}$.

Matrix Form of Attention

Attention can be efficiently calculated as a series of matrix multiplications. We can pack the query, key, and value vectors for all nodes together as rows in the matrices Q, K, and V. In other words, for x nodes gathering information and y nodes sharing, and with d_k-dimensional keys and d_v-dimensional values, the matrix Q is x by d_k, K is y by d_k, and V is y by d_v. Then, an attention head can be written as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{Eq. 13}$$

where softmax is applied along the rows of the x by y matrix (i.e. across y). Refer back to Equation 5 for the mathematical definition of softmax. This matrix is a table of relevance weightings between information-sharing and information-gathering nodes.

Although this $\text{Attention}(Q, K, V)$ form is mathematically equivalent to computing the $\vec{a_x}$ values as discussed previously, attention is often written in this matrix form [1].

Encoder vs. Decoder Architectures

Large language models are typically *decoder-only* transformers. In this case, the graph of connections has each node connected to all nodes that come before it but not any of the nodes after it. This pattern of attention is called *masked self-attention* as the nodes that come later are masked to the nodes before them. This way, each node can only look backwards in time, and the decoder can be used to generate text. A transformer can be trained to predict the next token given the last token and the ability to perform attention on the preceding tokens.

By contrast, in an *encoder* transformer, each node is connected to every other node. Encoders are used to perform tasks like sentiment analysis where no information needs to be hidden. The patterns of attention in both encoders and decoders are types of *causal self-attention*, as the sentence is performing attention within itself, and in theory words should attend to what logically modify them.

The original 2017 paper contained both an encoder and a decoder, so it had an additional component of *cross-attention*, which occurred in the decoder after masked self-attention and before the feed-forward network. The encoder processes the foreign-language text, and the decoder outputs the translated text. In cross-attention, the nodes outputted by the encoder network share information using key and value vectors, while the nodes in the decoder gather information with query vectors. Information is transferred across from the encoder to the decoder.

Conclusion

What I have presented in this paper has been an incomplete picture of modern large language models. There are still important training methods that I have not covered, like *dropout* (randomly zeroing values to avoid overfitting) and the *ADAM optimizer* (an improved algorithm for updating weights more sophisticated than simply adding partial derivatives). As mentioned in the overview, neither have I covered aspects like fine-tuning, tokenization, or positional encoding. Nevertheless, I hope that this paper has captured the core ideas of the transformer architecture and scaled dot product attention. For more details and code, OpenAI's Andrej Karpathy has a video lecture where he builds a simple transformer in PyTorch, which was an extremely helpful resource to me when writing this paper [5].

Bibliography

[1] A. Vaswani *et al.*, “Attention Is All You Need”. 2023.

[2] S. Basodi, C. Ji, H. Zhang, and Y. Pan, “Gradient amplification: An efficient way to train deep neural networks”, *Big Data Mining and Analytics*, no. 3, pp. 196–207, 2020, doi: 10.26599/BDMA.2020.9020004.

[3] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer Normalization”. 2016.

[4] R. Xiong *et al.*, “On Layer Normalization in the Transformer Architecture”, *CoRR*, 2020, Available: https://arxiv.org/abs/2002.04745

[5] A. Karpathy, “Let's build GPT: from scratch, in code, spelled out”. [Online]. Available: https://www.youtube.com/watch?v=kCc8FmEb1nY