Mathematics Details of Transformers In Depth

Evan Dramko

March 2025

Contents

1 Introduction

1.1 Expected Background and Audience

This article is intended for someone who is already familiar with machine learning and neural networks, and is learning about Transformers and Attention. I created it because I found many excellent resources describing Transformers at a high-level, but very few which handled the computations. I couldn't find any convenient resource which described all the mathematics from scratch. So, this article is meant to serve as a supporting work for a reader after they understand the overall idea of the Transformer architecture. For convenience, I provide the following links to good high-level articles motivating Transformers and covering their main idea. This article describes them in regards to their motivating application, text processing. This article gives an even more high-level description of the state of the field regarding where Transformers are used and what they can do.

Throughout this article, we will reference Self-Attention without going into it in detail; a detailed description of Self-Attention mathematics can be found in another article on my website.

Since we are pursuing the calculations in detail, a reader will need an understanding of basic matrix operations and linear algebra, as well as familiarity with denoting terms as being an element of a space (ex: $A \in R^{i \times j}$).

1.2 Motivation

Since their introduction in 2017, Transformers have come to dominate the field of deep learning. They, or similar architectures, are already near-ubiquitous in natural language processing (NLP), computer vision (CV), and computational biology (compBio). They are becoming increasingly used in robotics, graph analysis and assorted physical science domains. Here, we will break down, in detail, the computations involved in Transformers.

2 High-Level Architecture

Transformers can take a variety of different forms. Broadly however, Transformers are made of Encoder and Decoder blocks. We will consider the mathematics of each in sections 4 and 5 respectively.

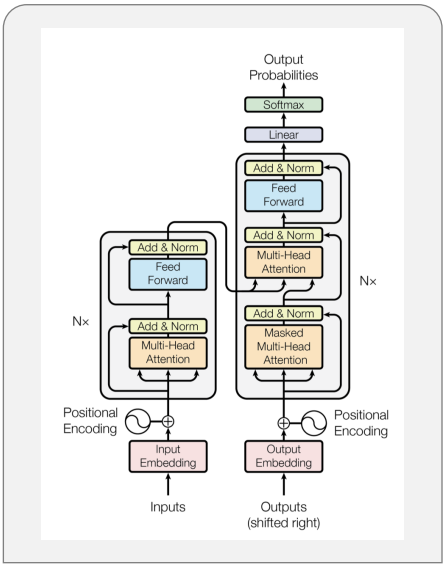
While the original proposed architecture was an Encoder followed by a Decoder, we often see models created as just an Encoder by itself, or just a Decoder by itself. A diagram of the architecture is included in 1a. A high level overview of the steps can also be seen by looking at the headings in the Table of Contents Section 4, 5.

3 Data Embedding and Positional Encoding

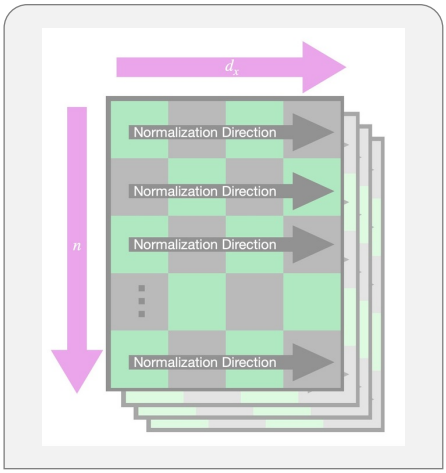
The Transformer architecture is used to handle a wide variety of situations and data types. However, the calculations require input to be a series of continuous-valued vectors. We must convert our data from its native form into a sequence of "tokens" (fundamental discrete component of the input) and represent each token with a vector of fixed length. The process of converting our data into this form is called **"Embedding"** the data. (Somewhat confusingly) we also call the resulting vectors **"Embeddings"**. Tokens refer to the embedding vector for a specific element of the input sequence.

We also often add a **Positional Encoding** to the data. This is a small value added to the embeddings so the model is able to identify the relative ordering of the tokens.

The process of creating embeddings and positional encodings has become quite advanced, and has spawned a whole field of research. We recommend further reading if you are trying to create your own embeddings for your data.



(a) Transformer diagram (from "Attention Is All You Need")



(b) Normalization axis in layerNorm (for both Encoder and Decoder)

4 Encoder Block

4.1 Input

Assume we have input embeddings: $X \in \mathbb{R}^{B,n,d_v}$ where we have B examples in our batch, a sequence length of n, and an embedding dimension of d_v. Following

standard practice, we will continue to refer to the data as X as it passes through the network. **Following standard notation, note that X no longer refers to a fixed input sequence, but is rather the label for the "data" as it is processed by the network.**

4.2 Self-Attention and Residual

We then perform Self-Attention on the embedding. The computations and intuition of Scaled Dot-Product Attention is quite involved, so we will not cover the computations of Self-Attention here. However, a very detailed analysis of it is available on my website. The Attention module outputs $attnScore_X \in \mathbb{R}^{B,n,d_v}$.

After Self-Attention is computed, we combine it with a residual connection, thus we update $X \leftarrow X + attnScore_X$.

4.3 LayerNorm

Next, we perform layer normalization (LN). This performs normalization across the *feature channels* [1] of the data. In the layer after LayerNorm, each token will be independently processed. So, when we normalize along the feature channels, we normalize each token individually.

Recall that our data is $X \in \mathbb{R}^{B,n,d_v}$, so we are independently normalizing each of $B \times n$ vectors of length d_v. Figure 1b shows that we normalize the d_v values in each feature channel for each of the n tokens. Thus, each token gets its own normalization value. The formula for this is:

$$LayerNorm(X_{i,j}) = \gamma_k \cdot \frac{X_{i,j,k} - \mu_{i,j}}{\sigma_{i,j} + \epsilon} + \beta_k$$

where each of the $B \cdot n$ different $X_{i,j} \Leftrightarrow X[i,j,:]$ denotes the individual tokens of the data, and $\mu, \sigma \in \mathbb{R}^1$ are the mean and standard deviation of the values of X along that token. $\gamma, \beta \in \mathbb{R}^{d_v}$ are learnable parameters. The ϵ is a small constant used to ensure numerical stability in the calculations, and is sometimes actually used inside the calculation of σ by computing: $\sigma = \sqrt{\sigma^2 + \epsilon}$.

Notice that if we define $a = \frac{X-\mu}{\sigma}$, then $LayerNorm(X) = \gamma \cdot a + \beta$, which is the same form as the formula for a linear layer.

Now, $X \leftarrow layerNorm(X) \in \mathbb{R}^{B \times n \times d_v}$.

4.4 FFN/MLP and residual

Following layer normalization, feed-forward network (FFN) (a multi-layer perceptron (MLP)) is applied to each token. Typically, we just have a single hidden layer (two weight matrices). While we could have any number of layers, empirical results have found that the best tradeoff between time, accuracy, generalization, and storage space comes from the single hidden layer FFN.

[1] "Feature channels" refers to processing along each token.

We denote the dimension of the hidden layer as d_{ff}, so we have $W_1 \in \mathbb{R}^{d_v \times d_{ff}}$, and $W_2 \in \mathbb{R}^{d_{ff} \times d_v}$. Thus, it takes an input vector $\mathbf{a} \in \mathbb{R}^{d_v}$, and performs: $\mathbf{a} \leftarrow W_2(\sigma(W_1^T \mathbf{a}))$ where here σ indicates the chosen activation function (ReLU, tanh, etc).

Then, we apply the **same** W_1, W_2 and FFN to each token in our data. So, we have:

$$
\begin{aligned}
X &\in \mathbb{R}^{B \times n \times d_v} \\
\mathbf{x}_{i,j} &= X[i, j, :] \in \mathbb{R}^{1 \times 1 \times d_v} \\
\mathbf{x}_{i,j} &\leftarrow W_2(\sigma(W_1^T \mathbf{x}_{i,j})); \forall i, j
\end{aligned}
$$

We also add a residual again at this stage. So, $X \leftarrow X + FFN(X)$, and we still have: $X \in \mathbb{R}^{B \times n \times d_v}$.

4.5 Recap

This finishes one encoder block. Often, several of these blocks will be "stacked" on top of each other to create a full encoder. Some models only use encoder blocks; the most common example of an encoder only model would be BERT.

5 Decoder

As one might expect, when there is an encoder there is also a decoder. Unlike the encoder, the decoder part of a model is slightly different than just stacking decoder blocks on top of eachother (although that is certainly a part of it!) This section covers the computations of the decoder block in detail, as well as how we use decoders in practical deployment.

5.1 Decoder Block

5.1.1 Input Data

We assume that our data is already a series of embedded representations of tokens (refer to 3, 4.1) and is of shape: $X \in \mathbb{R}^{B,n,d_v}$ where we have B examples in our batch, a sequence length of n, and an embedding dimension of d_v.

5.1.2 Self-Attention

This component is the same as the Self-Attention component in the encoder block: We perform Self-Attention on the embedding. We do not cover the computations of Self-Attention here, but a very detailed analysis of it is available on my website. The Attention module outputs $attnScore_X \in \mathbb{R}^{B,n,d_v}$.

After Self-Attention is computed, a residual connection is used, so we update $X \leftarrow X + attnScore_X \in \mathbb{R}^{B,n,d_v}$.

5.1.3 Cross-Attention + Residual

In an encoder-decoder model, we will compute Cross-Attention. This is not applicable to a decoder only model, as it pulls values from the encoder. In the case of a decoder only architecture (like GPT), simply skip this section and continue on to the FFN section.

Here, we compute an Attention score between across the values in the encoder and the decoder (hence "Cross" Attention). For efficiency, we will pull the Q values from the previous self-attention layer, and we only recompute K, V based on the current data X. The intuition here is that we are looking for the same thing as we did in Self-Attention, we are just now checking for "information" from the encoder side of the model.

Following the computation of Cross-Attention, we will add a residual connection. Thus, we find that $X \leftarrow X + crossAttnScore_X \in \mathbb{R}^{B,n,d_v}$.

5.1.4 FFN

This layer – (or sub-layer) more accurately – is computed in the same way as it was in the encoder. We reference section 4.4 for details.

After each token is passed through the FFN layer, it remains in shape $\mathbb{R}^{B,n,d_v}$.

5.1.5 Projection Into Vocabulary

Now, we have $X \in \mathbb{R}^{B,n,d_v}$. We need to get from a representation of our tokens as a continuous embedding of length d_v into something we can interpret directly. To do this, we multiply by a linear layer, W_V of size d_v, V where V is the size of our vocabulary. Thus, we see:

$$X \leftarrow XW_V \in \mathbb{R}^{B,n,V}$$

This gives us logits over the vocabulary size for each token. We apply a $softmax$ to turn the logits into a probability distribution.

5.2 Teacher Forcing

5.2.1 What Is Teacher Forcing?

When using a decoder for sequence-to-sequence tasks (like text generation) we often use a method called *Teacher Forcing* to train the model.

In this method, we feed in the entire sequence to the model, and mask off future tokens during the attention. This ensures that future words cannot influence their own prediction (otherwise this would be a lot easier!) [2]. We then calculate the loss from the prediction at each of the B, n steps to their actual values, and perform backpropagation. In this way, we can train on the entire sequence length in parallel.

[2] In this situation you already have the answer key for the next word prediction!

The benefit of this sort of training compared to one-at-a-time token generation is that bad predictions earlier in the sequence will not affect the accuracy of future predictions because the future token predictions are conditioned on the actual data. This ensures that the best possible learning for each token in the sequence occurs, and leads to faster convergence.

5.2.2 Issues and Solutions

An issue that arises with Teacher Forcing is exposure bias: during testing time which is inherently sequential one-at-a-time generation, small errors in prior token predictions can compound over long sequences and lead to highly inaccurate results.

Another similar issue comes up with evaluating the model: the generated sequence is always being "corrected" for each new time step which leads to overinflated results.

To handle these issues, we often start with always using Teacher Forcing for training, and then slowly transition to using one-at-a-time token generation.

5.3 Autoregressive Generation

During inference time, decoder models produce a single token at a time, a process called *autoregressive generation*. What we have formulated in the last section predicts n tokens where n is the input sequence length. The change in behavior when we use autoregressive generation is due to a modification of the way we compute Attention. If you aren't deeply familiar with Attention, this would be a great time to refer to my other post on Attention (available on my website).

Assume we have generated tokens 1 to $(t-1)$ in our sequence, and we are now generating token t. Then, within the Attention block, we must calculate: $Q_t = x_t W_Q$; $K_t = x_t W_K$; $V_t = x_t W_V$. For each head, we see shape: $K_i \in \mathbb{R}^{B \times h \times 1 \times d_k}$; $V_i \in \mathbb{R}^{B \times h \times 1 \times d_v}$.

Assume that we have cached (saved) the values for $K_{1:t-1}, V_{1:t-1}$ [3]. Then, we want to append the value for K_t, V_t to K_{t-1}, V_{t-1} along the t dimension. Since K_{t-1}, V_{t-1} have been constructed this way, they have shapes $\mathbb{R}^{B \times h \times (t-1) \times d_k}$, $\mathbb{R}^{B \times h \times (t-1) \times d_v}$ respectively. After we append K_t, V_t, we now get:

$$K_{1:t} \in \mathbb{R}^{B \times h \times t \times d_k}$$
$$K_{1:t}^T \in \mathbb{R}^{B \times h \times d_k \times t}$$
$$V_{1:t} \in \mathbb{R}^{B \times h \times t \times d_v}$$

Then, we calculate:

[3] As we will see: $K_{1:t-1} \in \mathbb{R}^{B \times h \times (t-1) \times d_k}$

$$\mathbf{a} = \frac{Q_t K_{1:t}^T}{\sqrt{d_k}} \in \mathbb{R}^{B \times h \times 1 \times t}$$

We then apply a softmax over the t dimension: $softmax(a, dim = -1)$. This normalizes the sequence, and gives a score for how much the tokens produced thus far are "contributing" to the next token.

Intuition Break: In "regular" Attention we apply the softmax over the "key" dimension, so that each query has its own normalized score. In autoregressive generation, normalizing along the t dimension does the same thing... it is normalizing the effect of each previous token which is the "question" affecting the value of the new token.

After the softmax, we then multiply:

$$Output = \mathbf{a} V_{1:t} \in \mathbb{R}^{B \times h \times 1 \times d_k}$$

This concludes the changes to the attention mechanism in autoregressive generation that allows the model to produce a single token at a time (notice the "n" dimension is 1). We proceed to do the concatenation and projection of the heads as usual, and then continue through the rest of the components/sub-layers of the decoder block as usual.

6 Conclusion

That is it! We have covered the standard Transformer architecture from start to finish. We describe both the encoder and decoder portions and all the calculations they use. Many variations of the architecture have emerged. Some target efficiency, others add in a convolution operation, and many other "goals" are considered. However, the base of all these architectures is the setup we covered.