Deep Dive Into Attention: Intuition, Dimensions, and Calculations

Evan Dramko

March 2025

Contents

1 Overview **1**
- 1.1 Expected Background and Audience . . . 1
- 1.2 Motivation for the Article . . . 1
- 1.3 The Concept of Attention . . . 2

2 The Calculations **2**
- 2.1 Full Self-Attention . . . 2
- 2.2 Multiheaded Attention . . . 4
- 2.3 Extension to Cross Attention and Other Conditionings . . . 5
- 2.4 Masking in Attention . . . 5

3 Modern Variations **6**
- 3.1 Are There Other Types of Attention? . . . 6
- 3.2 Efficiency Concerns . . . 6
- 3.3 Low Rank V . . . 6

1 Overview

1.1 Expected Background and Audience

This article is intended for someone who is familiar with machine learning and neural networks, and is learning about Transformers and Attention. The main goal is to explain the details of the mathematics of Attention.

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 for the Article

Since its release in the now famous paper: "Attention Is All You Need", the Transformer architecture has become the basis of most modern neural networks

(NNs). While initially proposed for natural language processing tasks, it has become widely adopted in other areas including: computer vision, computational biology, graph processing, and many others. As the name suggests, the power of the Transformer comes from its Scaled Dot-Product Attention mechanism. Despite its widespread use, many practitioners are often "fuzzy" on exactly how it works. Herein, we will discuss both the intuition behind Attention as well as cover its computations in detail. One of the best ways to understand the mechanics of Attention is to focus on matrix dimensions; the dimensions at each intermediate step in the computation provide insight into the abstract idea is being represented.

1.3 The Concept of Attention

At a high level, Attention is computing an every-to-every interaction between elements of the input sequences. Given input sequences of length n, m, Attention computes all $n \cdot m$ interactions. This is done in two steps: 1) computing a "strength of signal" between each component, and 2) learning the "meaning" of each interaction.

For clarity, we will first cover full Self-Attention, then show how it is modified to create Multi-Head Attention and Cross-Attention. With that in mind, lets dive in to the calculations!

2 The Calculations

2.1 Full Self-Attention

Input to the Attention function is a sequence of n elements, where each element is represented by a continuous vector of values. In the case of discrete elements, they are often embedded into a fixed size continuous vector prior to being passed into attention. We denote the dimension of this embedded vector as d_X. These embedding vectors are then stacked into a matrix, where the first index denotes the sequence element, and the second index denotes the component of the embedded vector.

Now, take one such sequence $X \in \mathbb{R}^{n \times d_X}$. First, we generate the three primary components of the Attention computation: Q, K, V. We have trainable matrices $W_Q, W_K \in \mathbb{R}^{d_X \times d_k}$, and $W_V \in \mathbb{R}^{d_X \times d_v}$. Often, we choose $d_k = d_v$, though it is not required. There is no special meaning behind d_k, d_v; they are simply the amount of "information" that your model is allowed to work with when computing its Attention score.

So, we compute:

$$Q = XW_Q \in \mathbb{R}^{n \times d_k}$$
$$K = XW_K \in \mathbb{R}^{n \times d_k}$$
$$V = XW_V \in \mathbb{R}^{n \times d_v}$$

The *query matrix*, Q, can be thought of as asking a question about the data. The *key matrix*, K, is often thought of as the answer to the question, and the *value matrix*, V, tells us what that question and answer pair mean to the network.

Next, we compute the Attention Map (Map) as:

$$Map = \frac{QK^\intercal}{\sqrt{d_k}} \in \mathbb{R}^{n \times n}$$

The Attention Map can be thought of as assigning a score to how much the question and answer match for every possible pair of the n elements (hence its existence in an (n, n) space). The division by $\sqrt{d_k}$ is a rescaling term, whose purpose cannot be properly motivated until after the next step in the calculation.

As usual in machine learning, we want these values to be normalized between 0 and 1 to improve numeric stability in the calculations. Furthermore, we want all the values to sum to 1, meaning the data is analogous to a probability distribution. This helps interpretability, and prevents issues when we (later on) applying masking. To normalize all values between 0 and 1, we apply a row-wise (along the "key" dimension) softmax. Because we want each "question" to act independently of each other, we normalize along the rows/queries independently rather than normalizing across the entire matrix. Without any better notation for this, we typically just use the word *"softmax"* to represent the function.

$$\begin{aligned} &standardScore = softmax\left(\frac{QK^\intercal}{\sqrt{d_k}}\right) \\ &\text{enforces: } \sum_j standardScore_{i,j} = 1 \\ &\text{and } 0 < standardScore_{i,j} < 1;\ \forall i, j \end{aligned}$$

The division by $\sqrt{d_K}$ is a normalization factor derived from the expected value of the entries of $QK^\intercal$. Given that $Q_{i,j}, K_{i,j}$ are independently sampled from a distribution with mean 0 and variance σ^2, the expected variance of $QK^\intercal$ is $d_k \cdot \sigma^2$. As $d_k \to \infty$, the variance grows to infinity. In modern NNs, where scale is king, this becomes a problem. When applying softmax to a vector with such high variance, it approaches a one-hot vector (all entries are 0 except for one which is 1). Using one-hot vectors as the Attention Map would be best described as: "Which other element is most important", rather than "How does each element impact the others". In essence, we would lose the every-to-every comparison and instead get a "most important - to - every" calculation.

Okay, lets recap so far: we used our data to condition question and answer matrices, and then computed a score for how much each element in our sequence is answering the question. Now, we want to apply meaning to the $(question, answer)$ pair. To do this, we use the *value matrix*, V.

$$Output = standardScore \times V \in \mathbb{R}^{n \times d_v}$$
$$= softmax\left(\frac{QK^\intercal}{\sqrt{d_k}}\right)V$$

Through this step, we use a learned matrix to apply the "meaning" (V) of the answer (K) to the question (Q). Notice that we end with a matrix in $\mathbb{R}^{n \times d_v}$; this corresponds to a value embedding $1 \times d_v$ for each of the n elements in the sequence.

Thus, we reach the final formulation of scaled dot-product attention, as can seen in "Attention Is All You Need".

2.2 Multiheaded Attention

Often, there are complicated relationships between elements of our sequence. A single Q matrix may not be sufficient to capture all the "questions" needed. While it is certainly possible to replicate the Attention module and run it many times in parallel, this is quite slow. To address this, Multiheaded Attention was introduced.

First we define an attention head as a set of independant attention calculations. This means that everything we have worked through up until now is a single Attention head. Multiheaded Attention divides the full size Attention head into smaller sub-heads (to match with literature, we will call them simply "heads". Each head/sub-head is an independent Attention head, just over a smaller section of the data).

By convention, we rename d_X to be d_{model}. Then, we define the number of heads as h. In most implementations (including the the pyTorch nn.MultiheadAttention), all heads are forced to be the same size. Thus, h must divide d_{model}. Now, we set $W_Q, W_K, W_V \in \mathbb{R}^{d_{model} \times d_{model}}$. Then, as we do in the single head case, we compute:

$$Q = XW_Q \in \mathbb{R}^{n \times d_{model}}$$
$$K = XW_K \in \mathbb{R}^{n \times d_{model}}$$
$$V = XW_V \in \mathbb{R}^{n \times d_{model}}$$

Now, we break from the single head setup. After computing Q, K, V, we split them into h heads along the d_{model} (embedding) dimension. This allows each head to operate on a different portion of the input, and specialize to the values there rather than having to handle the whole embedding. We redefine d_k to be the size of each head, thus $d_k = \frac{d_{model}}{h}$. Then, each of the h heads has the shape:

$$Q^i \in \mathbb{R}^{n \times d_k}$$
$$K^i \in \mathbb{R}^{n \times d_k}$$
$$V^i \in \mathbb{R}^{n \times d_k}$$

We perform attention as normal for each head, calculating: $Q^i K^{i\mathsf{T}} \in \mathbb{R}^{n \times n}$. We apply row-wise softmax and rescale just like we did before, then perform matrix multiplication by V^i. This creates:

$$Output = standardScore \times V \in \mathbb{R}^{n \times d_k}$$

After doing this in parallel for all h heads, we have h matrices of size (n, d_k). We concatenate them to create a single matrix of size $(n, h \times d_k) = (n, d_{model})$. Finally, multiplication by a projection matrix of size (d_{model}, d_{model}) yields the final output of size (n, d_{model}), which is the same size that we had in single headed Attention.

2.3 Extension to Cross Attention and Other Conditionings

Now that we have covered the details of Self-Attention, how can we derive Cross-Attention from it. Fortunately, the answer is very easy! Instead of conditioning K on X, we use a different matrix X'. In an encoder-decoder framework with cross-attention in the decoder, X' would be taken from the encoder.

We can also extend this to other cases. If we want a question matrix Q or values matrix V that doesn't depend on our input data X, we can instead remove the XW_Q or XW_V step, and learn Q or V directly. (If you are using the pyTorch nn.MultiheadAttention module, you can use any constant matrix (like that of all $1s$) as the argument to the respective parameter.)

2.4 Masking in Attention

In some cases, we do not want to allow every element in the sequence to influence every other one. While there are many different kinds of masking, we consider the illustrative example of "causal masking". In next token prediction during text generation you do not want to allow future words to impact previous ones (they haven't been spoken/written yet!). To prevent this, we zero out any values in the Attention map that we do not want to influence the calculation. This is referred to as "masking" the Attention mechanism. It should be noted that this does not "weaken" the effect of any word unfairly, because the row-wise softmax will standardize the total score for every element.

3 Modern Variations

People have improved upon this mechanism to create specialized variants. Some of the most common variants are included throughout this section.

3.1 Are There Other Types of Attention?

Yes! Scaled dot-product attention was not the first type of attention proposed. Most notably, Bahdanau Attention and Luong Attention appeared a few years prior. In fact, Dot-Product Attention itself appeared prior to "Attention Is All You Need"; it was identical to Scaled Dot-Product Attention but without the normalization by $\sqrt{d_k}$.

3.2 Efficiency Concerns

It is worth noting that the number of interactions in Self-Attention is $O(n^2)$. In very long context windows (when n is large), this can become prohibitively expensive. Additionally, there are alternatives that work better in distributed training of NNs over many GPUs.

Notable variants addressing these concerns are: Ring Attention, FlashAttention (all one word), Multi-Query Attention, Nystrom attention, and the aforementioned Bahdanau attention.

3.3 Low Rank V

Low rank factorizations of a matrix are the decomposition of a large matrix into the product of two smaller matrices. In many cases we will break V into a low rank factorization by:

$$V = UA$$

where $U \in \mathbb{R}^{n \times r}$ and $A \in \mathbb{R}^{r \times d_k}$, and $r << n$. The following table breaks down the memory and compute costs of making this change:

	Memory	Computations
V	$n \cdot d_k$	n^3
UA	$(n \cdot r) + (r \cdot d_k)$	$n^2 r + n r d_k$