Self-Attention

This post specifically goes into the math behind **Attention** and *not* Transformers. Written for intuitive understanding of what attention really is, and how I can teach it in the Workshop.

At its core, attention is simple:

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

where:

- $Q \in \mathbb{R}^{n \times d}$ = Query Vector (the input basically)
- $K \in \mathbb{R}^{n \times d}$ = Key Vector (also the input, but transposed during the multiplication, (matmul rules!))
- $V \in \mathbb{R}^{n \times d}$ = Value Vector (also the input!)

However, in practice, all three are linear projections of the input, not the raw input itself. But conceptually, they come from the same matrix: the input embeddings.

Fundamentally, attention answers one question: **how do we make the tokens talk to each other?**

We'll break this down part by part:

1. the QK^T - measuring relevance
2. the $\sqrt{d}$ scaling - keeping variance stable
3. $\text{softmax} \times V$ - the magic of weighted aggregation

---

Why QK^T - Measuring Relevance

The query matrix Q is the matrix we receive as input, with shape `[# of tokens (sequence length), embedding dimension]`

The key matrix K is the same input matrix but transposed to `[embedding dimension, # of tokens]` so we can matrix multiply it with Q.

Why matmul is needed

Recall from linear algebra: the dot product tells you how *similar* one vector is to another. Matmul is simply the extension of this—a batched dot product of tokens.

When we compute QK^T, we're asking: *"how relevant is token i to token j?"* That's the entire job of QK^T.

The $\sqrt{d}$ Scaling - Keeping Variance Stable

Softmax turns scores into a probability distribution where relative scale determines importance. The value of each element determines how much *space* it occupies relative to others. **Softmax is incredibly sensitive to scale.** More precisely: the relative scale of elements determines how peaked the distribution is.

When we compute QK^T, the dot products can get *too* big. Why? The dot product sums d terms, and with 4096 dimensions (a fairly standard size), that's 4096 additions. The variance of the sum grows with dimension:

$$\mathrm{Var}\left(\sum_{i=1}^{d} x_i\right) = \mathrm{Var}(x_1) + \mathrm{Var}(x_2) + \cdots + \mathrm{Var}(x_d) \implies \mathrm{Var}\left(\sum_i x_i\right) \propto d$$

During backprop, this means only the big neurons learn while starving others of gradient flow. Large values saturate the softmax, suffocating smaller values and killing gradients. Without fixing this, the wider the model, the more the token distribution drifts from the original.

So we need to preserve the variance of the input distribution. LLMs are probability distribution machines—they learn distributions, not raw numbers. What defines a distribution? Mean and variance.

The input has mean 0 and some variance σ^2 the model decides. Since softmax is shift-invariant ($\mathrm{softmax}(x) = \mathrm{softmax}(x + c)$), we only care about variance. What's the variance of QK^T?

- $q_i \in Q$ has mean 0, variance σ^2
- $k_i \in K$ has mean 0, variance σ^2
- Product: $X_i = q_i \cdot k_i \implies \mathrm{Var}(X_i) = \sigma^4$ and $\mathrm{std}(X_i) = \sigma^2$ since, $\mathrm{Var}(q_i) = \sigma^2$ and $\mathrm{Var}(k_i) = \sigma^2$.

Summing over d dimensions:

$$S = \sum_{i=1}^{d} X_i \quad \Rightarrow \quad \mathrm{Var}(S) = d\,\sigma^4$$

$$\mathrm{Std}(S) = \sqrt{\mathrm{Var}(S)} = \sqrt{d} \times \sigma^2$$

The σ^2 is just some constant the model decides, but $\sqrt{d}$ scaling the variance is the problem. So we divide QK^T by $\sqrt{d}$ to renormalize and cancel out the scaling from the dot product additions.

Why divide by the term in standard deviation and not variance?

From probability theory, we know that If you scale a random variable X by a constant c:

$$\mathrm{Var}(cX) = c^2\mathrm{Var}(X)$$

from derivation, dot product score S has:

$$\mathrm{Var}(S) = d\,\sigma^4$$

You want the resulting variance to be independent of d.
So you're looking for a scaling factor c such that:

$$\mathrm{Var}(\mathrm{cS}) \propto \sigma^4$$

Plugging in the rule,

$$\mathrm{Var}(\mathrm{cS}) = \mathrm{c}^2 \cdot \mathrm{d}\,\sigma^2$$

To cancel the d, the only value for c remains:

$$c^2 = \frac{1}{d} \Rightarrow c = \frac{1}{\sqrt{d}}$$

- Variance measures *spread*.
- Standard deviation measures *scale*.
- Softmax feels **scale**, not variance directly.

$\mathrm{softmax} \times V$ - The Magic of Weighted Aggregation

After calculating token relevance via QK^T, we run softmax across tokens:

$$A = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) \in \mathbb{R}^{n\times n}$$

Softmax turns arbitrary scores into probabilities by exponentiating (making them positive) then normalizing (summing to 1):

$$\mathrm{softmax}(X) = \frac{e^{X_i}}{\sum_{j=1}^{|X|} e^{X_j}}$$

The dot product gives us relevance scores, but softmax converts these into *importance weights*: *"out of all these tokens, which pairs **matter** most?"* (Later, for LLMs, we'll mask the upper triangle to prevent future tokens from influencing past ones.) These are coefficients

of *importance* for tokens. the end result A answers the question, *how much information from token i should flow to token j?*

You can think of these scores as weights. We use them to *scale* our initial tokens via the value vector V. This matmul is a weighted sum of attention scores and actual token content:

$$\text{output}_i = \sum_{j=1}^{n} A_{ij} v_j$$

where:

- $A_{ij} =$ how much token i cares about token j
- $v_j =$ what token j has to offer

therefore, **Attention = where to look (QK^T) + what to take (V)**

```
>>> import torch
>>> import torch.nn.functional as F
>>> seq_len, embd = 4, 3
>>> x = torch.randn((seq_len, embd))
>>> x
tensor([[ 0.5245,  1.0470, -1.6467],
        [ 2.1202,  0.7773, -0.7941],
        [ 0.0405,  1.2870,  1.7759],
        [-0.5583,  1.5262, -1.6624]])
>>> q, k, v = x, x, x # learned linear projections of input in practice.
>>> inner = (q @ k.T) / (embd ** 0.5)
>>> inner
tensor([[ 2.3573,  1.8669, -0.8983,  2.3340],
        [ 1.8669,  3.3082, -0.1871,  0.7637],
        [-0.8983, -0.1871,  2.7782, -0.5836],
        [ 2.3340,  0.7637, -0.5836,  3.1204]])
>>> # softmax across columns
>>> # i.e. for each token, weights over all tokens sum to 1.
>>> sfm = F.softmax(inner, dim=1)
>>> sfm
tensor([[0.3805, 0.2330, 0.0147, 0.3718],  # token 0
        [0.1759, 0.7432, 0.0226, 0.0584],  # token 1
        [0.0228, 0.0464, 0.8997, 0.0312],  # token 2
        [0.2892, 0.0602, 0.0156, 0.6350]]) # token 3
>>> # attn to each token -> t0: [t0, t1, t2, t3] ...
>>> #                       t1: [t0, t1, t2, t3] ...
>>> #                       ...
>>> sfm.sum(dim=1) # probability distribution.
tensor([1., 1., 1., 1.])
>>> sfm @ v
```

```
tensor([[ 0.4867,  1.1658, -1.4037],
        [ 1.6363,  0.8799, -0.9368],
        [ 0.1293,  1.2653,  1.4716],
        [-0.0746,  1.3388, -1.5519]])
```