Early LLM (transformer) layers pay attention to many tokens to gain context, later layers focus on fewer "important" tokens

Vuk Rosić
vukrosic

For the impatient experts, I will explain the conclusion of my experiment immediately (beginners, don't worry, I will explain everything later step by step).

I pre-trained a small 80 million parameter LLM on 8M text tokens and looked at the attention's **key** vectors:

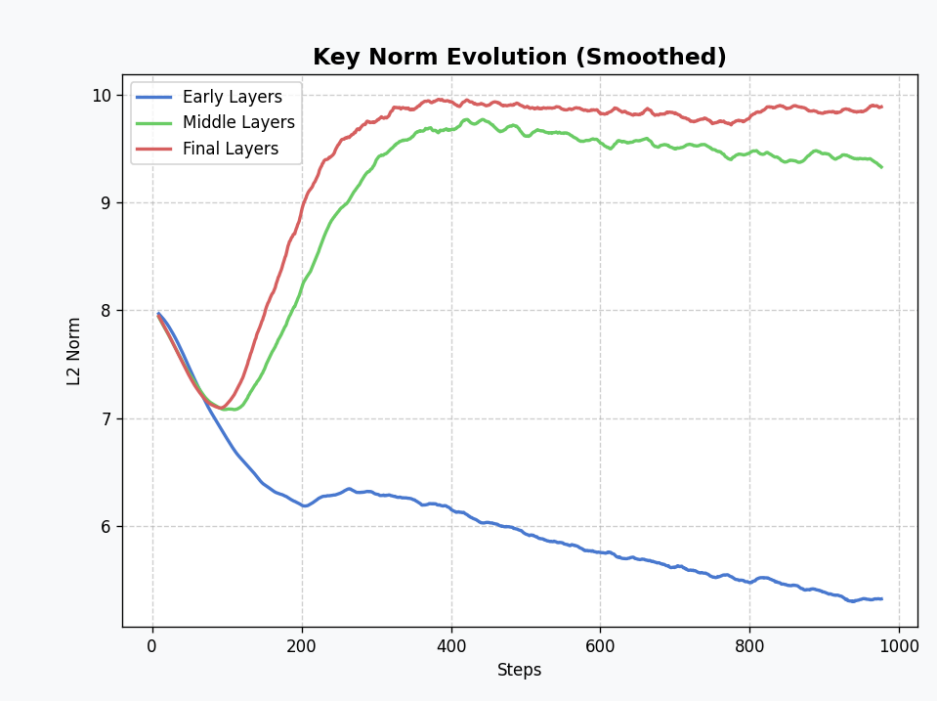


Figure 1: Evolution of the average Key Norm over ∼1000 training steps. "Early Layers" = layers 0-1, "Middle Layers" = layers 10-11, "Final Layers" = layers 20-21.

We can see that L2 Norm of attention keys in early layers is becoming lower as LLM trains, which means LLM is paying attention to more tokens and taking context from them.

L2 Norm has an effect of making softmax probabilities more uniform (similar), so LLM is

paying attention to more tokens equally.

In later layers L2 Norm is becoming higher, which means LLM is focusing on fewer "important" tokens.

Now let me explain everything step by step

Prerequisites

1. Dot Products Scale with Dimension

The dot product of two vectors measures how much they "agree" - how well they point in the same direction. You compute it by multiplying corresponding numbers and adding them up.

Example with 2D vectors:

Take $a = [2, 3]$ and $b = [4, 1]$:

$$a \cdot b = (2 \times 4) + (3 \times 1) = 8 + 3 = 11$$

The result is a single number. A **large positive** dot product means the vectors point in a similar direction; a value **near zero** means they are unrelated; a **large negative** value means they point in opposite directions.

Key insight: If you add more dimensions (more numbers to each vector), the dot product tends to grow in magnitude. This is because you are summing up more terms.

Example - same vectors, but with extra dimensions added:

$a = [2, 3, 1, 2]$, $b = [4, 1, 3, 2]$:

$$a \cdot b = (2 \times 4) + (3 \times 1) + (1 \times 3) + (2 \times 2) = 8 + 3 + 3 + 4 = 18$$

We went from 2 dimensions (dot product = 11) to 4 dimensions (dot product = 18). More dimensions $\rightarrow$ more terms $\rightarrow$ larger magnitude. For random vectors, the expected magnitude of the dot product grows proportionally to $\sqrt{d}$, where d is the number of dimensions.

This is the foundation of the entire analysis - and it is exactly why Transformers divide by $\sqrt{d_k}$ in attention: to cancel out this dimension-dependent growth.

2. What is Softmax?

Softmax converts a list of raw numbers (called **"logits"**) into a **probability distribution** - a list of values between 0 and 1 that all add up to 1. You can think of the outputs as **percentages**: they tell you what fraction of "attention" goes to each item.

Example 1 - Simple logits:

Logits: $[1.0, 2.0, 3.0] \rightarrow$ Softmax output (probabilities): $[0.09, 0.24, 0.67]$

Read this as: 9% goes to the first item, 24% to the second, and 67% to the third. They add up to 1.0 (100%).

Example 2 - Similar logits:

Logits: $[5.0, 5.1, 4.9] \rightarrow$ Softmax output: $[0.33, 0.34, 0.33]$

When the logits are almost the same, the probabilities are nearly equal - the model "can't decide" and spreads attention evenly.

Example 3 - Very different logits:

Logits: $[10.0, 1.0, 1.0] \rightarrow$ Softmax output: $[0.9998, 0.0001, 0.0001]$

When one logit is much bigger, softmax gives almost 100% of the probability to that item. The model is "laser-focused."

The formula looks scary, but don't worry about it - just know that softmax converts a list of numbers into another list that adds up to 1 and can be used as probabilities:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Critical property: Softmax is sensitive to the *spread* (range) of its input values.

- If the inputs are close together (e.g., $[2.1, 2.2, 1.9]$), the softmax output is nearly uniform: $[\sim 0.34, \sim 0.38, \sim 0.28]$.
- If the inputs are far apart (e.g., $[30, -10, 5]$), the softmax output concentrates on the largest: $[\sim 1.0, \sim 0.0, \sim 0.0]$.

This will be important later - **the attention's "key norm" in our experiment controls how spread-out those logits are**, which in turn controls how "sharp" or "diffuse" the attention becomes (if layer is distributing attention across more tokens, or focusing it on a few important tokens).

3. What is the L2 Norm?

The L2 Norm measures the total "length" or "magnitude" of a vector:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_d^2}$$

Example: A vector $[3, 4]$ has L2 norm $\sqrt{9 + 16} = 5$.

Scaling a vector changes its norm proportionally. If you multiply every component by the same factor, the norm multiplies by that same factor:

- Original vector: $[3, 4] \rightarrow$ Norm $= \sqrt{9 + 16} = 5$
- Doubled: $[6, 8] \rightarrow$ Norm $= \sqrt{36 + 64} = 10$ (exactly 2×5)
- Halved: $[1.5, 2] \rightarrow$ Norm $= \sqrt{2.25 + 4} = 2.5$ (exactly 0.5×5)

During our LLM pretraining experiment we will see that L2 Norm is growing in some layers and shrinking in others.

L2 Norm Shows How Focused Attention Is

This model applies `RMSNorm` to the Key vectors before attention. RMSNorm is a normalization technique that works in two steps:

1. **Normalize:** Divide the vector by its root-mean-square, forcing the average squared component to be 1:

$$\hat{x} = \frac{x}{\text{RMS}(x)}, \quad \text{where} \quad \text{RMS}(x) = \sqrt{\frac{1}{d}\sum x_i^2}$$

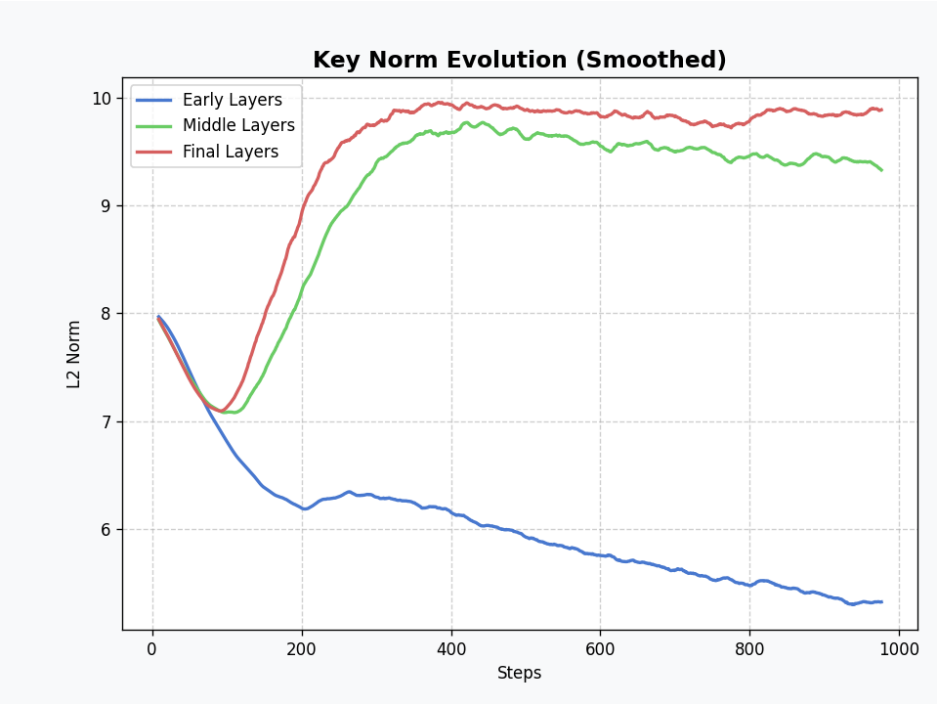
2. **Scale:** Multiply by a **learnable gain** vector γ: output $= \gamma \odot \hat{x}$.

Think of it this way: step 1 "resets" the vector to a standard size. Step 2 lets the model learn *how big* it actually wants each component to be.

At initialization, $\gamma = [1, 1, \ldots, 1]$ (all ones), so step 2 does nothing. After step 1 alone, the mean squared component is forced to be 1, so the expected L2 Norm of the output is:

$$\| \hat{x} \|_2 = \sqrt{\sum_{i=1}^{d} \hat{x}_i^2} = \sqrt{d \cdot \frac{1}{d}\sum \hat{x}_i^2} = \sqrt{d \cdot 1} = \sqrt{d}$$

With $d = d_{\text{head}} = 64$ (i.e., `d_model=512` divided by `n_heads=8`), the initial L2 Norm is $\sqrt{64} = 8.0$.



During training, gradient descent updates the gain γ. If the model learns gains greater than 1, the output norm rises above 8.0. If it learns gains less than 1, the norm drops below 8.0. **This is the mechanism by which the model controls the Key norm - it adjusts RMSNorm's learnable gain parameters.**

Mechanism: Sharpness Control

The attention mechanism computes **scaled dot-product attention**:

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

The $\frac{1}{\sqrt{d_k}}$ factor normalizes the dot products so they don't grow with dimension (recall from the Prerequisites - dot products grow with $\sqrt{d}$, so dividing by $\sqrt{d_k}$ cancels that out).

But on top of this standard scaling, the *magnitude* of K still acts as an additional multiplier. If the Key norm is α times larger, all dot products in $Q \cdot K^T$ are also α times larger, which changes the softmax behavior (recall - softmax is sensitive to the spread of its inputs).

- **Larger Key norm** $\rightarrow$ larger logits $\rightarrow$ sharper attention $\rightarrow$ lower effective temperature.

- **Smaller Key norm** → smaller logits → more diffuse attention → higher effective temperature.

Analyzing the image

- **Case 1: Low Norm**
  - **Norm:** ∼5.3 (below the 8.0 baseline)
  - **What happens:** All dot products $Q \cdot K^T$ are scaled down (by a factor of ∼5.3/8.0 ≈ 0.66 compared to baseline). The logits fed into softmax are closer together.
  - **Result:** A more diffuse attention distribution. The model attends to many tokens roughly equally. This is useful for **aggregating broad context** - for example, building up a representation of the overall sentence structure.
- **Case 2: High Norm**
  - **Norm:** ∼9.8 (above the 8.0 baseline)
  - **What happens:** All dot products are scaled up (by a factor of ∼9.8/8.0 ≈ 1.23 compared to baseline). The logits fed into softmax are pushed further apart.
  - **Result:** A more peaked attention distribution. The model concentrates attention on a small number of tokens (or even a single token). This is useful for **precise information retrieval** - for example, copying a specific fact or entity from earlier in the context.

Note on the examples: The exact logit values depend on both the Query and Key vectors - specifically, how well they align. The Key norm only controls the *scale* of those logits, not their specific values. Two situations with the same Key norm but different Q-K alignments will produce different logit patterns.

Experimental Analysis

- **Model:** 88M parameter Transformer (22 layers, `d_model=512`, 8 heads, `d_head=64`).
- **What's measured:** The L2 norm of Key vectors **after RMSNorm** (i.e., after the learnable gain is applied, but before Rotary Position Embeddings are added). This isolates the effect of the learned scaling.
- **Training:** 8M tokens (∼1000 gradient steps, `batch_size=4`, `seq_len=2048`).

Very few layers remain near the initialization value of 8.0 by the end of training. The model pushes most layers toward one of two operating regimes (dampened or amplified), suggesting

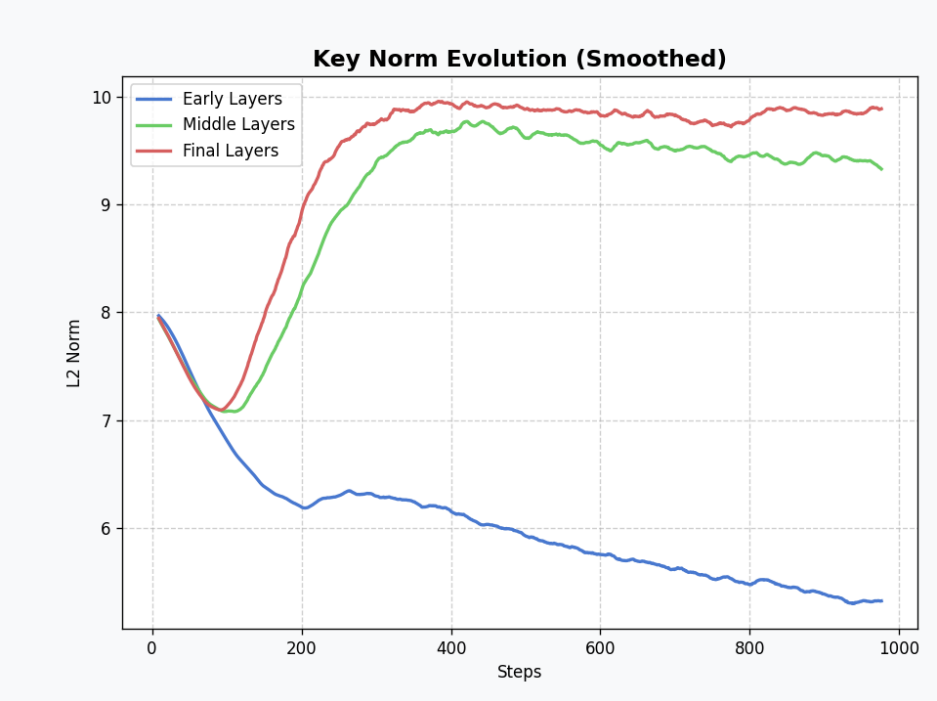


Figure 2: Figure 1: Evolution of the average Key Norm over ∼1000 training steps. ”Early Layers” = layers 0-1, ”Middle Layers” = layers 10-11, ”Final Layers” = layers 20-21.

that the ”neutral” baseline is suboptimal and the model benefits from **distinct functional specialization** across depths.

The final layers have slightly higher norms than the middle layers (∼9.8 vs ∼9.3), suggesting the last layers need the sharpest attention for the final prediction - consistent with their role as the ”decision-making” layers closest to the output.