CS336 Lecture 3: Language Model Architecture and Hyperparameters

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: From the 2017 Transformer to Modern Frontier Waves

Building a large language model requires making dozens of coupled structural decisions: where to normalize hidden vectors, which nonlinearity to apply inside feed-forward layers, how to inject token order, how wide and deep to make the network, and how to keep optimization stable and inference fast at scale. If you start from the original 2017 Transformer decoder (Vaswani et al., 2017) and train it naively at modern parameter counts and sequence lengths, training destabilizes, memory bandwidth bottlenecks throughput, and long-context generation grinds to a halt. Every major architectural change over the past decade was forced into existence by one of those concrete failure modes.

To see what changed and why, begin by reconstructing the exact structure of the 2017 Transformer decoder stack. Imagine the network diagram flowing vertically from bottom to top:

1. **Input stage**: A discrete token is mapped to a continuous vector via a token embedding table, and a fixed sinusoidal positional encoding vector is added directly to it (`Token Emb + Sine PE`). For position pos and feature pair index i, the two coordinates are:

$$PE_{(pos,\,2i)} = \sin\!\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \qquad PE_{(pos,\,2i+1)} = \cos\!\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

   Here pos is the integer token index in the sequence, d_{model} is the model's hidden dimension (the width of the residual stream), and $2i$ and $2i+1$ index adjacent pairs of coordinates. The denominator $10000^{2i/d_{\text{model}}}$ produces a geometric progression of wavelengths across feature dimensions, so low indices oscillate rapidly with position while high indices vary slowly.
2. **Repeated block ($N\times$)**: The embedded sequence enters a stack of N identical layers. Each layer contains two sublayers:
   - **Masked Multi-Head Attention**: Standard dense causal attention where every token attends to all earlier tokens and itself across multiple parallel heads. A residual skip connection branches off before attention and rejoins immediately after attention at an **`Add & LayerNorm`** block.
   - **Feed-Forward Network (FFN)**: A two-layer multilayer perceptron with a ReLU activation and additive bias vectors:

$$\mathrm{FFN}(x) = \max(0,\, xW_1 + b_1)\, W_2 + b_2$$

     Here $x \in \mathbb{R}^{d_{\text{model}}}$ is a token's hidden row vector, $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ projects it up to a wider intermediate dimension d_{ff}, $b_1 \in \mathbb{R}^{d_{\text{ff}}}$ shifts the pre-activation, $\max(0, \cdot)$ zeroes negative entries element-wise, $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ projects back down, and $b_2 \in \mathbb{R}^{d_{\text{model}}}$ shifts the output. A second

residual skip connection branches off before the FFN and rejoins at a second `Add & LayerNorm` block.

3. **Output stage**: After N blocks, a linear projection and softmax produce next-token probabilities (`Softmax / Linear`).

Every single one of the four core choices in this 2017 blueprint—additive sinusoidal position encodings, ReLU FFNs with biases, post-norm LayerNorm placed directly on the residual stream, and dense multi-head attention—has been revised in modern large language models.

Now contrast that 2017 stack against the modern dense backbone, often called the **"LLaMA-like" architecture**. Reconstructing its block diagram from bottom to top reveals a fundamentally different dataflow:

1. **Input stage**: Only the **Token Embedding** is applied at the bottom; positional information is no longer added to the token embedding at all.
2. **Unobstructed residual highway**: Running straight up the left side of the diagram is a clean vertical spine containing *only* addition nodes (`+`), with no normalization blocks sitting in its path.
3. **Attention branch**: The residual vector forks off to the right, passes through **RMSNorm**, then through **Causal Attention with RoPE (no bias)**, and is added back into the residual highway at the `+` node.
4. **FFN branch**: Above that addition, the residual vector forks off to the right again, passes through **RMSNorm**, then through a **SwiGLU FFN (no bias)**, and is added back into the residual highway at the second `+` node.
5. **Output stage**: Because the residual spine itself was never normalized inside the repeated blocks, a single **Final RMSNorm** is applied at the very top right before the **LM Head** (the final linear projection to vocabulary logits).

DEFINITION

Residual Stream (Residual Highway): The main additive pathway carrying each token's representation $x_l \in \mathbb{R}^{d_{\text{model}}}$ from the input embedding through all N Transformer blocks to the final output head, where each attention or feed-forward sublayer reads from the stream and adds its computed update back onto it.

WORKED EXAMPLE

Trace a single token vector $x_0 \in \mathbb{R}^{d_{\text{model}}}$ through one layer of the 2017 decoder versus one layer of the modern dense backbone to see the structural difference directly.

- **2017 decoder (Layer 1)**:
  1. $h_0 = x_0 + PE_0$ (position vector added directly into the representation).
  2. $u_1 = \text{LayerNorm}\big(h_0 + \text{MHA}(h_0)\big)$ (the sum itself is re-centered and rescaled).
  3. $x_1 = \text{LayerNorm}\big(u_1 + \max(0, u_1 W_1 + b_1) W_2 + b_2\big)$ (again normalized on the main path).
- **Modern dense backbone (Layer 1)**:

1. $u_1 = x_0 + \mathrm{Attn}_{\mathrm{RoPE}}\big(\mathrm{RMSNorm}(x_0)\big)$ (normalization sits on the branch; position is injected by rotating queries and keys inside attention).
2. $x_1 = u_1 + \mathrm{SwiGLU}\big(\mathrm{RMSNorm}(u_1)\big)$ (no biases; x_0 passes straight into x_1 with unit weight).

Interpretation: In the 2017 block, x_0 is altered by two nonlinear LayerNorm operations per layer. In the modern block, $x_1 = x_0 + \Delta_{\mathrm{attn}} + \Delta_{\mathrm{ffn}}$, preserving a direct additive path from the input to every deeper layer.

Architecture did not stop evolving once the field settled on that dense backbone. Looking from 2020 through late 2026, open language model design falls into three distinct waves:

Era	Representative Models	Defining Architectural Pattern
Historical baselines (2020–2024)	GPT-3 (2020), PaLM (2022), Llama 2 (2023), Llama 3 (July 2024), DeepSeek-V3 (Dec 2024)	Dense Transformer converges on Pre-RMSNorm + RoPE + SwiGLU + GQA; DeepSeek-V3 introduces 671B total / 37B active MoE + MLA + MTP
2025–early-2026 wave (Apr 2025–Feb 2026)	Llama 4 (Apr 2025), DeepSeek-V3.2 (Dec 2025), Qwen 3.5, Kimi K2 (July 2025), GLM-4.7, Nemotron 3, Gemma 4, OLMo 3	Pre-norm RMSNorm backbone; QK-Norm & QK-Clip; interleaved local/full or DeltaNet + full attention; Nemotron 3 LatentMoE; DeepSeek DSA & MTP
Late-2026 frontier (June–Oct 2026)	DeepSeek-V4 / V4.1-Flash (Sept 2026), Kimi K3 / K3.1 (July 2026), GLM-5.3 (2026), Qwen3.8-Flash-Next (Aug 2026), MiniMax-M3 (June 2026), Nemotron 3.5 (Oct 2026)	Ultra-sparse MoE (up to 896 experts); 1M-context sparse + linear hybrids (CSA2, DSA, QSA, DeltaNet/Mamba); mHC & Attention Residuals; Muon + gated stability

COMMON MISCONCEPTION

You might think that the late-2026 frontier models threw away the dense "LLaMA-like" Transformer block and started from an unrelated architecture. That breaks down when you inspect their layers: almost every late-2026 frontier model retains the exact dense block fundamentals—RMSNorm off the residual spine, gated `*GLU` feed-forward experts, and RoPE variants—while innovating on top of them in four targeted areas: FFN sparsity, KV-cache and attention compression, wider residual connections, and large-scale training stability.

KEY IDEA

Every architectural change from the 2017 Transformer to the modern dense backbone (Pre-RMSNorm, RoPE, SwiGLU, bias removal) was driven by optimization stability, quality per parameter, or data-movement efficiency; late-2026 frontier models keep those dense block fundamentals while layering sparsity, long-context attention compression, multi-stream residuals, and stronger stability controls on top.

The remainder of this lecture unpacks each of these design decisions from first principles, starting with the single most consequential change to the Transformer's skeleton: moving normalization off the residual path.

✎ Check Your Understanding

1. **Fill in the blank:** In the modern dense backbone, because normalization is moved off the main residual highway inside each block, a single _____ must be added at the very top of the stack right before the LM head.
   Answer: Final RMSNorm (or final normalization layer). Because each layer adds unnormalized sublayer outputs directly onto the residual stream, one final normalization is required before projecting to vocabulary logits.
2. **Yes or No (with justification):** Do late-2026 frontier models such as DeepSeek-V4.1-Flash and Kimi K3 abandon the dense block primitives (RMSNorm, gated FFNs, and RoPE variants) established in earlier open models?
   Answer: No. They keep those core block primitives and innovate on top of them via ultra-sparse MoE, compressed and sparse/linear hybrid attention, multi-stream or attention residuals, and Muon/gated stability mechanisms.

Section 2: Residual Pathways and Normalization Placement

Recall that deep neural networks rely on residual connections to propagate gradients cleanly backward from the loss to early layers during backpropagation. In the original 2017 Transformer, normalization was placed directly on the residual spine right after each addition. That seemingly minor placement choice caused severe training instability as models grew deeper, forcing researchers to rely on fragile learning-rate warmup schedules just to prevent divergence on step one.

Compare the two placements side by side, matching the diagram adapted from Xiong et al. (2020). On the left side of the diagram sits **Post-LN (2017)**: a thick vertical spine enters from the bottom as x_l, branches right into `Attn`, rejoins at `Add` on the spine, and immediately passes through a `Norm` box sitting *on* the spine before continuing upward to the `FFN` sublayer, a second `Add`, and a second `Norm`

box on the spine. On the right side sits **Pre-LN (Modern)**: the vertical spine enters as x_l and travels straight up through only `Add` nodes; each branch forks right into `Norm` *first*, then into `Attn` (or `FFN`), and rejoins the spine at `Add`.

Written algebraically for a generic sublayer F (either attention or FFN):

$$\text{Post-LN:} \quad x_{l+1} = \mathrm{Norm}\big(x_l + F(x_l)\big)$$

$$\text{Pre-LN:} \quad x_{l+1} = x_l + F\big(\mathrm{Norm}(x_l)\big)$$

Break both equations apart term by term. In both formulas, $x_l \in \mathbb{R}^{d_{\text{model}}}$ is the residual state entering sublayer l, x_{l+1} is the state leaving it, $F(\cdot)$ is the sublayer transformation, and $\mathrm{Norm}(\cdot)$ rescales a vector to controlled variance (we analyze LayerNorm and RMSNorm specifically in Section 3). In Post-LN, the innermost operation is the addition $x_l + F(x_l)$, and the outer operation wraps that entire sum inside $\mathrm{Norm}(\cdot)$. In Pre-LN, the innermost operation normalizes only the branch input $\mathrm{Norm}(x_l)$, the sublayer transforms that normalized copy via $F(\cdot)$, and the outermost operation adds the result cleanly onto the untouched x_l.

> **DEFINITION**
> **Post-Norm (Post-LN)**: A Transformer block layout that applies normalization *after* the residual addition, $x_{l+1} = \mathrm{Norm}(x_l + F(x_l))$, placing the normalization Jacobian directly on the main residual path.

> **DEFINITION**
> **Pre-Norm (Pre-LN)**: A Transformer block layout that applies normalization *before* each sublayer on the residual branch, $x_{l+1} = x_l + F(\mathrm{Norm}(x_l))$, leaving an unobstructed identity path along the residual spine (paired with one final normalization before the output head).

Why does moving that single `Norm` box off the spine transform training dynamics? Examine the empirical and theoretical evidence from Xiong et al. (2020), depicted in the bar chart of **Expected Gradient Norm of FFN W^1 at Initialization** across a 6-layer Transformer (horizontal axis: Layer 1 = earliest through Layer 6 = deepest; vertical axis: Gradient Expectation from 0.0 to above 1.0):

- **Pre-LN bars (blue)**: Almost perfectly flat across all six layers, hovering uniformly near 0.20 from Layer 1 through Layer 6.
- **Post-LN bars (orange)**: Severely imbalanced across depth. Layer 1 starts tiny at roughly 0.08, Layer 2 rises to 0.12, Layer 3 to 0.27, Layer 4 to 0.37, Layer 5 jumps to 0.76, and Layer 6 explodes off the top of the chart to roughly 1.25—more than $15\times$ larger than Layer 1.

The reason comes directly from the chain rule. Under Pre-LN, unrolling L sublayers yields:

$$x_L = x_0 + \sum_{l=0}^{L-1} F_l\big(\mathrm{Norm}(x_l)\big)$$

Differentiating the loss $\mathcal{L}$ with respect to an early state x_k gives:

$$\frac{\partial \mathcal{L}}{\partial x_k} = \frac{\partial \mathcal{L}}{\partial x_L}\left(I + \sum_{l=k}^{L-1} \frac{\partial F_l(\mathrm{Norm}(x_l))}{\partial x_k}\right)$$

The identity matrix I guarantees that the output gradient $\frac{\partial \mathcal{L}}{\partial x_L}$ reaches every layer directly with unit gain, regardless of depth. Under Post-LN, there is no clean additive sum: every layer multiplies the backward gradient by the Jacobian of $\mathrm{Norm}(\cdot)$. Because adding independent sublayer outputs increases vector variance before normalization, dividing by the standard deviation $\sigma > 1$ repeatedly scales down gradients as they travel backward toward Layer 1, while leaving gradients near the output (Layer 6) huge.

WORKED EXAMPLE

Verify this scaling effect numerically with a single simplified residual update in $\mathbb{R}^2$, using root-mean-square scaling $\mathrm{Norm}(v) = v/\sqrt{\frac{1}{2}\|v\|_2^2}$.

Suppose the incoming residual state is $x_0 = [1,\ 1]^T$ (whose root-mean-square norm is $\sqrt{(1^2+1^2)/2} = 1$), and the sublayer produces $F(x_0) = [3,\ -1]^T$.

1. **Under Post-LN**:
   - First form the sum: $s = x_0 + F(x_0) = [4,\ 0]^T$.
   - Compute its RMS scale: $\mathrm{RMS}(s) = \sqrt{(4^2+0^2)/2} = \sqrt{8} \approx 2.828$.
   - Normalize the sum: $x_1 = s/\sqrt{8} = [\sqrt{2},\ 0]^T \approx [1.414,\ 0]^T$.
   - Notice what happened to the identity signal x_0: its contribution to x_1 was multiplied by $1/\sqrt{8} \approx 0.354$. If a 24-layer stack attenuates backward gradients by even a factor of 0.85 per layer along the spine, Layer 1 sees $0.85^{24} \approx 0.02$ of the gradient magnitude seen by Layer 24.
2. **Under Pre-LN**:
   - Since $\mathrm{RMS}(x_0) = 1$, $\mathrm{Norm}(x_0) = x_0$, so $x_1 = x_0 + F(\mathrm{Norm}(x_0)) = [1,\ 1]^T + [3,\ -1]^T = [4,\ 0]^T$.
   - Differentiating x_1 with respect to x_0 preserves the identity matrix I verbatim; no factor of $1/\sqrt{8}$ attenuates the residual pathway.

Interpretation: Post-LN couples the variance of $x_l + F(x_l)$ directly into the backward scaling of the residual highway, creating huge output-layer gradients and attenuated early-layer gradients at initialization. Pre-LN isolates the normalization Jacobian onto the branch so the residual highway remains an exact identity.

Empirical training runs confirm the theory: Salazar & Nguyen (2019) observed far smaller global gradient norm spikes under Pre-LN than under Post-LN. Originally, Pre-LN was motivated by the desire to train Transformers without a learning-rate warmup phase. Today, models still use warmup, and the

modern motivation for Pre-LN is **training stability and the ability to run larger learning rates in very deep networks**. Pre-norm is now the universal consensus across historical baselines, the 2025–early-2026 wave (Llama 4, Nemotron 3, DeepSeek-V3.2), and late-2026 frontier models. Only early historical architectures—the original 2017 Transformer, BERT, and OPT-350M—used Post-LN.

Non-Residual Post-Norm ("Double Norm")

Pre-LN solves gradient imbalance across depth, but it introduces a subtle secondary issue: because each layer adds an unnormalized vector $F(\mathrm{Norm}(x_l))$ onto the residual highway, an unusually large sublayer output can inject an activation spike straight into the residual stream, and the norm of x_l grows steadily with depth.

To bound each block's contribution without re-breaking the identity highway, several architectures adopt **non-residual post-norm** (also called **Double Norm**). In the block diagram, the thick vertical blue residual highway on the left still contains *only* `Add` nodes. On each branch, however, the signal passes through **Pre-Norm**, then into **Attention** (or **FFN**), and then through a second `Post-Norm (branch)` box *before* rejoining the residual spine at `Add`:

$$x_{l+1} = x_l + \mathrm{Norm}\Big(F\big(\mathrm{Norm}(x_l)\big)\Big)$$

Reading from inside out: $\mathrm{Norm}(x_l)$ standardizes the input to the sublayer, $F(\cdot)$ computes the attention or FFN update, the outer $\mathrm{Norm}(\cdot)$ bounds the magnitude of that branch update *before* it touches the spine, and the final addition preserves the unobstructed identity path $x_l + (\cdots)$.

> **DEFINITION**
> **Non-Residual Post-Norm (Double Norm)**: A normalization layout that places a second normalization layer on the output of each sublayer branch *prior* to the residual addition, $x_{l+1} = x_l + \mathrm{Norm}(F(\mathrm{Norm}(x_l)))$, bounding every sublayer's update while keeping normalization off the main residual spine.

Grok, Gemma 2, and Gemma 3 use both pre-norm and non-residual post-norm together. OLMo 2 uses non-residual post-norm *only*—dropping the pre-norm while keeping the branch post-norm off the spine—demonstrating the overarching engineering rule of thumb: **everyone now keeps normalization off the main residual spine**.

> **COMMON MISCONCEPTION**
> You might think that because Gemma 2, Gemma 3, and OLMo 2 are described as using "post-norm," the field is reverting to the 2017 Post-LN Transformer. That breaks because 2017 Post-LN wraps the *sum* $x_l + F(x_l)$ inside $\mathrm{Norm}(\cdot)$, placing normalization directly on the residual spine and damaging gradient flow. Non-residual post-norm wraps only the *branch output* $F(\cdot)$ before the addition, leaving the identity highway completely intact.

Widening the Residual Stream: mHC Hyper-Connections and Attention Residuals

Once normalization is safely off the spine, a deeper structural limitation appears in frontier-scale models: a single residual vector of width d_{model} acts as a narrow communication bottleneck across dozens of layers. Every layer must read from and overwrite the exact same d_{model}-dimensional channel, forcing early lexical features, intermediate syntactic structures, and late predictive features to compete for space in one vector.

In 2026, **DeepSeek-V4** addresses this bottleneck with **Manifold-Constrained Hyper-Connections (mHC)**. Reconstructing its diagram: instead of a single vertical residual highway of width d_{model}, the model maintains $n \approx 4$ **parallel residual streams**, giving a total residual state of size $4 \times d_{\text{model}}$. At each layer l, two things happen in parallel:

1. **Stream mixing**: The 4 vertical streams pass through a **Learned Mix** H_l box—an $n \times n$ learned mixing matrix projected toward the set of **doubly stochastic matrices** via the **Sinkhorn–Knopp algorithm** (meaning every entry is non-negative, every row sums to 1, and every column sums to 1).
2. **Sublayer branch**: The Attention or FFN sublayer reads a learned weighted sum of the 4 streams, computes its update, and distributes its output back via an **Add to 4 streams** block on top of the mixed streams.

> **DEFINITION**
> **Manifold-Constrained Hyper-Connections (mHC)**: A multi-stream residual architecture (used in DeepSeek-V4) that widens the residual state into $n \approx 4$ parallel streams mixed at each layer by a learned $n \times n$ matrix projected toward doubly stochasticity via Sinkhorn–Knopp, expanding cross-layer communication capacity while preserving signal conservation.

Why insist on a doubly stochastic mixing matrix rather than an unconstrained learned $n \times n$ matrix H_l? Because an unconstrained matrix repeated across L layers multiplies signals by $\prod_{l=1}^{L} H_l$, whose eigenvalues can either explode exponentially or collapse to zero—reintroducing the exact gradient pathology that Pre-LN eliminated! A doubly stochastic matrix has spectral norm 1 and preserves the total sum (and mean) across the n streams, generalizing the scalar identity mapping of a standard residual highway to n parallel lanes.

> **WORKED EXAMPLE**
> Check how Sinkhorn–Knopp projection enforces signal conservation on a 2-stream ($n = 2$) residual state.
>
> Suppose a layer learns raw positive mixing weights $M = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$ and the two incoming stream values (for a single feature coordinate) are $s = \begin{pmatrix} 4 \\ 10 \end{pmatrix}$ (sum $= 14$).

1. **Normalize rows** (divide each row by its row sum 3 and 4):

$$M^{(1)} = \begin{pmatrix} 2/3 & 1/3 \\ 1/4 & 3/4 \end{pmatrix} \approx \begin{pmatrix} 0.667 & 0.333 \\ 0.250 & 0.750 \end{pmatrix}$$

   Column sums are now 0.917 and 1.083 (not yet 1).

2. **Normalize columns** (divide each column by its column sum):

$$M^{(2)} = \begin{pmatrix} 0.667/0.917 & 0.333/1.083 \\ 0.250/0.917 & 0.750/1.083 \end{pmatrix} \approx \begin{pmatrix} 0.727 & 0.308 \\ 0.273 & 0.692 \end{pmatrix}$$

3. **Iterate to convergence**: Alternating row and column normalizations converges to the doubly stochastic matrix

$$H_l \approx \begin{pmatrix} 0.7101 & 0.2899 \\ 0.2899 & 0.7101 \end{pmatrix}$$

   where every row sums to 1.0000 and every column sums to 1.0000.

4. **Apply H_l to the 2 streams**:

$$s' = H_l s = \begin{pmatrix} 0.7101(4) + 0.2899(10) \\ 0.2899(4) + 0.7101(10) \end{pmatrix} = \begin{pmatrix} 5.7394 \\ 8.2606 \end{pmatrix}$$

   Check the sum: $5.7394 + 8.2606 = 14.0000$.

Interpretation: The streams exchange information freely without amplifying or attenuating the aggregate signal across depth. However, widening the residual state to n streams is not free: reading and writing n copies of the d_{model} residual tensor incurs $n\times$ **residual memory traffic** (for $n = 4$, $4\times$ the bytes moved per layer along the residual path).

A related late-2026 direction for overcoming the single-stream residual bottleneck appears in **Kimi K3 / K3.1** (July 2026), which replaces standard additive residual accumulation with **Attention Residuals** across depth, allowing deeper layers to selectively attend over representations from earlier layers rather than relying solely on a uniform running sum.

KEY IDEA

Keeping normalization off the main residual spine (via Pre-LN or non-residual post-norm) guarantees uniform gradient flow and stability across depth, while late-2026 multi-stream architectures like mHC widen the residual pathway at the cost of $n\times$ residual memory traffic by constraining cross-stream mixing to doubly stochastic matrices.

That warning about memory traffic in mHC points directly to our next question: even inside a standard single-stream Pre-LN Transformer, *which* normalization formula should we use on the branch—LayerNorm or RMSNorm—and why would changing an operation that accounts for less than 0.2% of total FLOPs measurably speed up training?

✎ Check Your Understanding

1. **What would happen if** you switched a deep Transformer from Pre-LN back to 2017 Post-LN at initialization without changing the learning rate?
   Answer: Gradients would become severely imbalanced across depth—exploding near the output layers and attenuating near the earliest layers—causing large gradient norm spikes and training instability.
2. **Yes or No (with justification):** Does projecting the $n \times n$ mixing matrix H_l in mHC to be doubly stochastic eliminate the extra memory traffic caused by using n parallel residual streams?
   Answer: No. Doubly stochastic projection preserves signal conservation and gradient stability across depth, but the model still must read and write an $n \times d_{\text{model}}$ residual state at every layer, incurring $n\times$ residual memory traffic.

Section 3: Normalization Mechanics, Data Movement, and Bias Removal

Once normalization lives on the sublayer branches, the 2017 Transformer's default choice is **LayerNorm**. Given a token's hidden vector $x \in \mathbb{R}^d$ (where $d = d_{\text{model}}$), LayerNorm normalizes both the mean and the variance across the d coordinates, then applies a learned element-wise scale $\gamma \in \mathbb{R}^d$ and shift $\beta \in \mathbb{R}^d$:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \odot \gamma + \beta$$

Decompose this equation from the innermost terms outward:

- $\mathbb{E}[x] = \frac{1}{d}\sum_{i=1}^{d} x_i$ computes the scalar mean of the vector's coordinates.
- $x - \mathbb{E}[x]$ re-centers the vector so its mean is zero.
- $\mathrm{Var}[x] = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mathbb{E}[x])^2$ computes the mean squared deviation from that mean.
- Adding a small constant $\epsilon > 0$ prevents division by zero, and dividing by $\sqrt{\mathrm{Var}[x] + \epsilon}$ scales the centered vector to unit variance.
- Finally, $\odot\gamma$ performs element-wise multiplication by a learned gain vector, and $+\beta$ adds a learned bias vector so the network can restore any mean and scale it needs.

Historical models—GPT-1, GPT-2, GPT-3, OPT, GPT-J, and BLOOM—used LayerNorm. Modern models—the LLaMA family, PaLM, Chinchilla, T5, the 2025–early-2026 wave (Llama 4, Nemotron 3, DeepSeek-V3.2), and the late-2026 frontier—almost universally replace it with **RMSNorm** (Root Mean Square Normalization):

$$y = \frac{x}{\sqrt{\frac{1}{d}\|x\|_2^2 + \epsilon}} \odot \gamma$$

DEFINITION

RMSNorm (Root Mean Square Normalization): A simplified normalization layer that rescales a hidden vector $x \in \mathbb{R}^d$ by its root-mean-square magnitude $\sqrt{\frac{1}{d}\|x\|_2^2 + \epsilon}$ and multiplies by a learned gain γ, omitting both mean subtraction and the additive bias parameter β.

Break RMSNorm apart term by term:

- $\|x\|_2^2 = \sum_{i=1}^d x_i^2$ is the squared Euclidean norm across the d features.
- Dividing by d yields the mean square $\frac{1}{d}\|x\|_2^2$.
- Taking $\sqrt{\frac{1}{d}\|x\|_2^2 + \epsilon}$ produces the root-mean-square (RMS) magnitude of x.
- Dividing x by this scalar rescales the vector to unit RMS length without subtracting its mean, and $\odot\gamma$ applies the learned per-coordinate gain without any additive bias β.

WORKED EXAMPLE

Apply both LayerNorm and RMSNorm to the same 4-dimensional hidden vector $x = [2,\ 4,\ 6,\ 8]^T$ with $\gamma = [1,\ 1,\ 1,\ 1]^T$, $\beta = [0,\ 0,\ 0,\ 0]^T$, and $\epsilon = 0$.

1. **LayerNorm**:
   - Mean: $\mathbb{E}[x] = \frac{2+4+6+8}{4} = 5$.
   - Centered vector: $x - 5 = [-3,\ -1,\ 1,\ 3]^T$.
   - Variance: $\mathrm{Var}[x] = \frac{(-3)^2+(-1)^2+1^2+3^2}{4} = \frac{20}{4} = 5$, so $\sqrt{\mathrm{Var}[x]} = \sqrt{5} \approx 2.236$.
   - Output: $y_{\mathrm{LN}} = [-3,\ -1,\ 1,\ 3]^T / 2.236 \approx [-1.342,\ -0.447,\ 0.447,\ 1.342]^T$. (Mean $= 0$, variance $= 1$.)
2. **RMSNorm**:
   - Mean square: $\frac{1}{4}\|x\|_2^2 = \frac{4+16+36+64}{4} = \frac{120}{4} = 30$, so $\mathrm{RMS}(x) = \sqrt{30} \approx 5.477$.
   - Output: $y_{\mathrm{RMS}} = [2,\ 4,\ 6,\ 8]^T / 5.477 \approx [0.365,\ 0.730,\ 1.095,\ 1.461]^T$.
   - Check RMS of y_{RMS}: $\sqrt{(0.365^2 + 0.730^2 + 1.095^2 + 1.461^2)/4} = \sqrt{4/4} = 1.000$.

Interpretation: LayerNorm strips out the vector's $+5$ offset and centers it around zero, requiring a mean pass and a variance pass (plus loading/storing β). RMSNorm preserves the direction of x strictly—scaling it onto a sphere of radius $\sqrt{d}$—using only a single sum-of-squares reduction and no β parameter.

Why RMSNorm Matters: FLOPs Are Not Runtime

At first glance, dropping a mean subtraction and a d-element bias vector looks irrelevant to training speed. Matrix multiplications (tensor contractions) inside attention and FFNs perform nearly all the floating-point arithmetic in a Transformer. So why does simplifying normalization improve wall-clock throughput at all?

The answer comes from **Ivanov et al. ("Data Movement Is All You Need", MLSys 2021)**, who profiled a Transformer's breakdown between floating-point operations (FLOPs) and actual wall-clock runtime:

Operator Class	% of Total FLOPs	% of Total Runtime
Tensor contraction (matrix multiplies)	99.80%	61.0%
Statistical normalization	**0.17%**	**25.5%**
Element-wise ops	**0.03%**	**13.5%**

Read those numbers carefully. Statistical normalization and element-wise operations together represent only 0.20% **of total FLOPs**, yet they consume 39.0% **of total runtime**!

To see why, reconstruct Ivanov et al.'s dataflow diagram of an attention sublayer block, where every operation carries two badges—a left badge showing total **FLOPs** and a right badge showing its **FLOP/byte ratio** (arithmetic intensity):

> **DEFINITION**
> **Arithmetic Intensity (FLOP/byte ratio)**: The number of floating-point operations an operator performs for every byte of data it reads from or writes to memory; operators with low arithmetic intensity stall compute units waiting on memory bandwidth and are **memory-bound**.

Tracing the diagram from top to bottom:

- **MHA (Multi-Head Attention)**: Performs **43G FLOPs** at an arithmetic intensity of **153 FLOPs/byte**. Each byte loaded from memory is reused across hundreds of multiply-adds, keeping compute units busy.
- **Dropout**: Performs only **4M FLOPs** at an arithmetic intensity of **1/3 FLOP/byte**.
- **Residual Add (`+`)**: Combines the skip path with the branch output; performs **4M FLOPs** at **1/3 FLOP/byte**.
- **LayerNorm**: Performs **29M FLOPs** at an arithmetic intensity of **3.5 FLOPs/byte**.

> **COMMON MISCONCEPTION**
> You might think that optimizing an operation accounting for only 0.17% of a model's FLOPs can improve training speed by at most 0.17%. That breaks because FLOPs do not equal runtime when operators are memory-bound: statistical normalization takes 25.5% of wall-clock runtime due to its tiny 3.5 FLOP/byte ratio, so removing memory passes inside normalization yields a large real-world speedup.

WORKED EXAMPLE

Use Ivanov et al.'s runtime breakdown to compare two hypothetical engineering optimizations: (A) reducing matrix-multiply FLOPs by 2%, versus (B) reducing normalization data-movement time by 25%.

- **Optimization A**: Matmuls account for 61.0% of runtime, so a 2% speedup on matmuls saves $0.02 \times 61.0\% = 1.22\%$ of total wall-clock time.
- **Optimization B**: Statistical normalization accounts for 25.5% of runtime (despite being 0.17% of FLOPs), so a 25% reduction in normalization runtime saves $0.25 \times 25.5\% = 6.38\%$ of total wall-clock time—more than $5\times$ the wall-clock savings of Optimization A!

Interpretation: Eliminating the mean-computation pass and the reads/writes of the bias vector β in RMSNorm directly attacks a quarter of total training time. We will encounter this exact same arithmetic-intensity bottleneck again when we analyze incremental decoding and KV caches in Section 9.

Empirical Validation and Dropping Bias Terms

Does removing mean-centering and β hurt model quality? Controlled ablations by **Narang et al. (2020)** on a 223M-parameter model trained for 11.1T operations answer that decisively:

Model (223M, 11.1T ops)	Step/s	Early Loss	Final Loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
RMSNorm	**3.68**	**2.167 ± 0.008**	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
ReZero	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
ReZero + RMSNorm	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31

RMSNorm did not merely tie LayerNorm; it **improved both throughput** ($3.50 \to 3.68$ steps/s, a $+5.1\%$ wall-clock speedup) **and quality across every single metric**—lower final loss ($1.838 \to 1.821$) and higher scores on SuperGLUE ($71.66 \to 75.45$), XSum, WebQ, and WMT EnDe. Meanwhile, attempts to remove normalization entirely via initialization/scaling tricks (ReZero and Fixup) degraded loss and downstream quality severely.

The exact same data-movement and stability principle motivated **dropping bias terms** throughout linear projections and normalizations:

$$\text{FFN}(x) = \max(0,\, xW_1 + b_1)W_2 + b_2 \;\longrightarrow\; \text{FFN}(x) = \sigma(xW_1)W_2$$

Adding a bias vector b_1 or b_2 is a low-arithmetic-intensity element-wise operation that requires extra memory reads and writes without adding meaningful capacity at large hidden dimensions, and removing biases also empirically improves optimization stability.

> **KEY IDEA**
> Because normalization and element-wise operations consume nearly 39% of wall-clock runtime despite representing only 0.20% of FLOPs, switching from LayerNorm to RMSNorm and dropping bias terms simultaneously speeds up training and improves both stability and downstream quality.

With normalization and biases streamlined, the largest share of parameters and compute in a dense Transformer sits inside the feed-forward block. Next, we examine how replacing ReLU with gated activations, scaling FFNs via sparse Mixtures of Experts, and choosing between serial and parallel block wiring reshape model capacity and throughput.

> **✎ Check Your Understanding**
>
> 1. **Quick computation:** Compute the RMSNorm output (with $\gamma = \mathbf{1}$ and $\epsilon = 0$) for the 2-element vector $x = [3,\, 4]^T$.
> **Answer:** The mean square is $(3^2 + 4^2)/2 = 25/2 = 12.5$, so $\text{RMS}(x) = \sqrt{12.5} \approx 3.536$, and $y = [3,\, 4]^T / 3.536 \approx [0.849,\, 1.131]^T$.
> 2. **Fill in the blank:** In Ivanov et al.'s Transformer profile, statistical normalization and element-wise operations make up only 0.20% of total FLOPs yet consume 39.0% of runtime because their low _____ causes them to bottleneck on data movement.
> **Answer:** FLOP-to-memory ratio (or arithmetic intensity).

Section 4: Feed-Forward Blocks, Sparse MoE Scaling, and Block Topology

Inside each Transformer layer, attention mixes information *across* token positions, while the feed-forward network (FFN) applies a nonlinear transformation independently to *each* token's hidden vector. Because the FFN accounts for roughly two-thirds of the parameters in a dense Transformer block, even small improvements in how the FFN uses its parameters translate into large reductions in language modeling loss.

Activations: From ReLU and GeLU to Gated Linear Units (`*GLU`)

Start by comparing the three scalar activation functions plotted on the activation curve diagram (horizontal axis x, vertical axis $\sigma(x)$):

1. **ReLU (dashed grey line)**: $\max(0, x)$. Flat at 0 for all $x < 0$, with a sharp corner at $x = 0$ and slope 1 for $x > 0$. Used in the original 2017 Transformer, T5, Gopher, Chinchilla, and OPT.
2. **GeLU (solid blue curve)**: $\mathrm{GELU}(x) = x\,\Phi(x)$, where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution. It behaves like x for large positive x and approaches 0 for large negative x, but smooths out the corner at 0 with a slight negative dip near $x = -0.75$. Used in GPT-1/2/3, GPT-J, GPT-NeoX, and BLOOM.
3. **Swish (solid red curve)**: $\mathrm{Swish}_1(x) = x \cdot \mathrm{sigmoid}(x) = \frac{x}{1+e^{-x}}$. Very similar in shape to GeLU, with a slightly deeper smooth negative dip near $x = -1.28$.
4. **Outlier**: Nemotron-4 340B used **Squared ReLU**, $\max(0, x)^2$.

Non-gated activations like ReLU and GeLU certainly work (GPT-3 used GeLU), but they apply a fixed scalar nonlinearity to a single linear projection xW_1. **Gated Linear Units (`*GLU`)** improve expressivity by introducing a multiplicative interaction between two separate projections of the input: one branch passes through an activation function $\sigma(xW)$ to act as a learned, input-dependent gate, and the other branch xV stays linear, combining element-wise before the output projection W_2:

$$\max(0, xW_1) \longrightarrow \max(0, xW_1) \odot (xV)$$

> **DEFINITION**
> **Gated Linear Unit (`*GLU`) FFN**: A feed-forward architecture that replaces a single up-projection with two parallel projections—one passed through an activation function and multiplied element-wise with a linear gating projection xV—before projecting back down via W_2.

Three specific variants arise depending on which activation sits on the left branch:

- **ReGLU**: $\mathrm{FFN}_{\mathrm{ReGLU}}(x) = \big(\max(0, xW_1) \odot (xV)\big)W_2$
- **GeGLU**: $\mathrm{FFN}_{\mathrm{GeGLU}}(x) = \big(\mathrm{GELU}(xW) \odot (xV)\big)W_2$ — used in T5 v1.1, mT5, LaMDA, Phi-3, Gemma 2, Gemma 3, and Gemma 4.
- **SwiGLU**: $\mathrm{FFN}_{\mathrm{SwiGLU}}(x) = \big(\mathrm{Swish}_1(xW) \odot (xV)\big)W_2$, where $\mathrm{Swish}_1(x) = x \cdot \mathrm{sigmoid}(x)$ — used in LLaMA 1/2/3, PaLM, Mistral, OLMo, and most post-2023 models.

Decompose $\mathrm{FFN}_{\mathrm{SwiGLU}}(x)$ hierarchically for a single token vector $x \in \mathbb{R}^{d_{\text{model}}}$:

- $xW \in \mathbb{R}^{d_{\text{ff}}}$ computes the gate pre-activations using weight matrix $W \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$.
- $\mathrm{Swish}_1(xW)$ applies $u \mapsto u \cdot \mathrm{sigmoid}(u)$ element-wise, smoothly opening positive channels and suppressing strongly negative ones.
- In parallel, $xV \in \mathbb{R}^{d_{\text{ff}}}$ computes a second linear projection of the same input using $V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$.
- The Hadamard (element-wise) product $\odot$ modulates each feature in xV by the corresponding gate value from $\mathrm{Swish}_1(xW)$, producing a bilinear-like quadratic interaction in x.
- Finally, multiplying by $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ projects the gated hidden vector back to the residual dimension d_{model}.

Notice an immediate accounting consequence: a standard FFN has **2** weight matrices (W_1, W_2), whereas a `*GLU` FFN has **3** weight matrices (W, V, W_2). If you kept $d_{\text{ff}} = 4\,d_{\text{model}}$, you would increase FFN parameters and FLOPs by 50%. To hold parameters and FLOPs constant when switching to `*GLU`, shrink d_{ff} by a factor of $\frac{2}{3}$:

$$d_{\text{ff}} = \frac{2}{3} \cdot 4\,d_{\text{model}} = \frac{8}{3}\,d_{\text{model}} \approx 2.67\,d_{\text{model}}$$

WORKED EXAMPLE

First verify parameter equivalence for $d_{\text{model}} = 3072$, then evaluate one hidden channel of SwiGLU.

1. **Parameter count comparison**:
   - Standard 2-matrix ReLU FFN at $d_{\text{ff}} = 4 \times 3072 = 12{,}288$:

     $$\text{Params} = 2 \times d_{\text{model}} \times d_{\text{ff}} = 2 \times 3072 \times 12{,}288 = 75{,}497{,}472$$

   - 3-matrix SwiGLU FFN at $d_{\text{ff}} = \frac{8}{3} \times 3072 = 8192$:

     $$\text{Params} = 3 \times d_{\text{model}} \times d_{\text{ff}} = 3 \times 3072 \times 8192 = 75{,}497{,}472$$

     Both blocks have the exact same 75.5M parameters and identical matrix-multiply FLOPs per token.

2. **Single-channel SwiGLU evaluation**: Suppose for one hidden coordinate, the linear branch gives $v = (xV)_j = -1.5$.
   - When the gate pre-activation is open, $u = (xW)_j = +2.0$: $\text{sigmoid}(2.0) = \frac{1}{1+e^{-2}} \approx 0.8808$, so $\text{Swish}_1(2.0) = 2.0 \times 0.8808 = 1.7616$. The gated value is $1.7616 \times (-1.5) = -2.6424$.
   - When the gate pre-activation is closed, $u = (xW)_j = -2.0$: $\text{sigmoid}(-2.0) \approx 0.1192$, so $\text{Swish}_1(-2.0) = -2.0 \times 0.1192 = -0.2384$. The gated value shrinks in magnitude to $(-0.2384)(-1.5) = +0.3576$.

Interpretation: Even with a narrower intermediate dimension (8192 vs. $12{,}288$), multiplicative gating lets one projection dynamically route or suppress features computed by the other projection, giving higher representational power per parameter.

Does `*GLU` actually win when parameters and compute are matched? Two independent benchmark suites—**Shazeer (2020)** on T5-style evaluations (where the Raffel et al., 2019 baseline scored 83.28 ± 0.235) and **Narang et al. (2020)** at 223M parameters and 11.1T ops—show a clean separation:

Variant	Shazeer (2020) Avg	Shazeer CoLA	Shazeer SST-2	Narang (2020) Step/s	Narang Final Loss	Narang SGLUE	Narang XSum
$\mathrm{FFN}_{\mathrm{ReLU}}$ (Vanilla)	83.80	51.32	94.04	3.50	1.838	71.66	17.78
$\mathrm{FFN}_{\mathrm{GELU}}$	83.86	53.48	94.04	3.58	1.838	75.79	17.86
$\mathrm{FFN}_{\mathrm{Swish}}$	83.60	49.79	93.69	3.62	1.847	73.77	17.74
$\mathrm{FFN}_{\mathrm{GLU}}$	84.20	49.16	94.27	3.59	1.814	74.20	17.42
$\mathrm{FFN}_{\mathrm{GEGLU}}$	84.12	53.65	93.92	3.55	1.792	75.96	18.27
$\mathrm{FFN}_{\mathrm{ReGLU}}$	**84.67**	**56.16**	**94.38**	3.57	1.803	**76.17**	**18.36**
$\mathrm{FFN}_{\mathrm{SwiGLU}}$	84.36	51.59	93.92	3.53	**1.789**	76.00	18.20

Every single `*GLU` row beats every non-gated row (ReLU, GeLU, Swish) on final loss at matched compute, with SwiGLU reaching the lowest final loss (1.789 vs. 1.838) and GeGLU/ReGLU/SwiGLU all landing within a narrow band at the top of downstream evaluations.

> **COMMON MISCONCEPTION**
> You might think SwiGLU beats ReLU simply because Swish is a smoother activation than ReLU. That breaks when you check the `Swish` row in both tables: non-gated Swish actually scores *worse* than ReLU (83.60 vs. 83.80 average in Shazeer; 1.847 vs. 1.838 final loss in Narang). The gain comes from **multiplicative gating** (`*GLU`), not from the scalar activation curve alone.

Scaling FFNs Today: Sparse Mixture of Experts (MoE) and Multi-Token Prediction (MTP)

Even with SwiGLU, a dense FFN forces every token to execute every FFN parameter. Doubling model capacity doubles training and inference FLOPs per token. To decouple total knowledge capacity from per-token compute, modern frontier architectures replace the single dense FFN with a **Sparse Mixture of Experts (MoE)** layer while keeping each individual expert a SwiGLU-style FFN.

Reconstruct the MoE block diagram from bottom to top:

- A token's hidden vector x enters at the bottom and splits along two paths.
- On the left path, a lightweight **Router (top-k)** scores a large pool of specialized expert FFNs ($E_1, E_2, \ldots, E_N$) and activates only the top-k highest-scoring experts (for example, E_1 and E_k, highlighted in blue, while inactive experts like E_2 are skipped completely).
- On the right path, x passes unconditionally through one or more **Shared Expert(s)** that run for every token, capturing common linguistic or syntactic transformations so routed experts do not waste capacity duplicating shared knowledge.
- At the top, a **Weighted Sum** combines the shared expert output with the router-weighted outputs of the k active routed experts.

DEFINITION

Sparse Mixture of Experts (MoE): An FFN architecture that replaces a single dense MLP with many parallel expert FFNs plus a lightweight router that activates only the top-k experts per token (often alongside always-on shared experts), scaling total parameter count far beyond per-token compute.

Two additional FFN-era scaling techniques pair closely with MoE in recent waves:

- **LatentMoE**: Introduced in the early-2026 wave by **Nemotron 3** (and adopted in stabilized form as **Stable LatentMoE** in Kimi K3 / K3.1), which projects tokens into a lower-dimensional latent space for expert computation.
- **Multi-Token Prediction (MTP)**: Introduced by DeepSeek (DeepSeek-V3 and extended in DeepSeek-V4.1-Flash and GLM-5.3), which adds auxiliary prediction heads/layers that train the model to predict multiple future tokens per position rather than only the single next token, densifying the training signal and aiding speculative decoding.

DEFINITION

Multi-Token Prediction (MTP): A training and architectural extension (pioneered by DeepSeek) that predicts several upcoming tokens at each sequence position using lightweight additional prediction layers, increasing training signal density per token.

Compare how aggressively sparsity has scaled from the late-2024 historical baseline through the July 2025 wave to the late-2026 frontier:

Model (Wave)	Total / Active Params	Experts (Active + Shared) & Additions
DeepSeek-V3 (Dec 2024, historical)	671B / 37B	256 routed (8 active + 1 shared), MTP
Kimi K2 (July 2025, historical)	1T total	384 routed (8 active)
DeepSeek-V4.1-Flash (Sept 2026)	552B / 16B decode, 8B prefill	384 routed (6 active + 1 shared), 3 MTP layers
Kimi K3 / K3.1 (July 2026)	2.8T / 104B	896 routed (16 active + 2 shared), Stable LatentMoE, MXFP4 QAT on experts
GLM-5.3 (2026)	78 layers, $d = 6144$	256 routed (8 active + 1 shared), 1 MTP layer
Qwen3.8-Flash-Next (Aug 2026)	125B / 6B	512 routed (10 active + 1 shared)

WORKED EXAMPLE

Compute the active parameter fractions and routed-expert sparsity ratios for two late-2026 frontier models from the table above.

1. **Kimi K3 / K3.1** (July 2026):

- Active parameter fraction: $104\text{B}/2800\text{B} \approx 3.71\%$.
- Routed experts active per token: $16/896 \approx 1.79\%$ of routed experts (plus 2 always-on shared experts).

2. **Qwen3.8-Flash-Next** (Aug 2026):
   - Active parameter fraction: $6\text{B}/125\text{B} = 4.80\%$.
   - Routed experts active per token: $10/512 \approx 1.95\%$ of routed experts (plus 1 always-on shared expert).

Interpretation: Late-2026 frontier models activate fewer than 5% of their total parameters and fewer than 2% of their routed experts on any given token, achieving multi-trillion-parameter capacity at the per-token FLOP budget of a medium-sized dense model. Notice also that Kimi K3 applies **MXFP4 quantization-aware training (QAT)** specifically to its MoE experts to keep storing 2.8T weights practical.

Serial vs. Parallel Transformer Blocks

Beyond what happens *inside* the FFN, one topological choice remains within each Transformer layer: should the Attention sublayer and the MLP/FFN sublayer run sequentially or side by side?

Reconstruct the side-by-side topology diagram:

- **Serial (Standard)** (left panel): The blue residual spine runs straight up. First, a branch goes through `Norm` → `Attn` and adds back into the spine. *Then*, using that updated residual state, a second branch goes through a second `Norm` → `MLP` and adds back into the spine:

$$y = x + \text{MLP}\Big(\text{Norm}\big(x + \text{Attn}(\text{Norm}(x))\big)\Big)$$

- **Parallel (GPT-J / PaLM)** (right panel): The residual spine forks once into a single **`Shared Norm`** block. The normalized output splits immediately into two side-by-side branches— `Attn` on the left and `MLP` on the right—which execute simultaneously on the exact same input and are both added back onto the residual spine:

$$y = x + \text{MLP}\big(\text{Norm}(x)\big) + \text{Attn}\big(\text{Norm}(x)\big)$$

DEFINITION

Parallel Transformer Block: A layer topology that computes attention and the feed-forward MLP simultaneously from a single shared pre-normalization of x, $y = x + \text{MLP}(\text{Norm}(x)) + \text{Attn}(\text{Norm}(x))$, allowing their input projections to be fused into one matrix multiply.

Why did GPT-J and PaLM adopt parallel blocks? Fusing the query/key/value input projection matrix of Attention with the up-projection matrix of the MLP—plus eliminating one of the two memory-bound normalization passes per layer—delivered **~15% faster training** in PaLM.

However, PaLM's own ablation showed a trade-off: parallel blocks caused a **small quality drop at 8B parameters** and were **quality-neutral at 62B**, while the 540B scale was extrapolated rather than directly ablated. Because a serial block lets the MLP immediately react to what Attention just retrieved within the same layer (effectively doubling the depth of nonlinear composition per layer), **serial blocks remain the default** in modern models, with parallel blocks used in a minority of architectures (GPT-J, GPT-NeoX, PaLM, Falcon 2 11B, Command R+, and Command A).

> **KEY IDEA**
> Multiplicative gating (`*GLU`, with d_{ff} scaled by $2/3$ to match parameters) consistently beats ReLU, GeLU, and Swish at equal compute; frontier models scale these SwiGLU experts via ultra-sparse MoE (activating under 5% of parameters per token) and MTP while keeping serial attention-then-FFN wiring as the default.

So far, every operation we have built—RMSNorm, SwiGLU, MoE, and residual additions—acts identically on every token regardless of where it sits in the sequence. Without positional information, self-attention is completely permutation-equivariant: "dog bites man" and "man bites dog" would produce the exact same attention scores. Next, we examine how positional encodings evolved from additive sines to Rotary Position Embeddings (RoPE) and its modern long-context hybrids.

> **✎ Check Your Understanding**
>
> 1. **Quick computation:** You convert a dense Transformer with $d_{\text{model}} = 4096$ from a standard 2-matrix ReLU FFN ($d_{\text{ff}} = 4\,d_{\text{model}} = 16{,}384$) to a 3-matrix SwiGLU FFN. What value of d_{ff} keeps parameter count and FLOPs identical (to the nearest integer)?
> **Answer:** $d_{\text{ff}} = \frac{2}{3} \times 16{,}384 = \frac{8}{3} \times 4096 \approx 10{,}923$ (often rounded to a hardware-friendly multiple such as $10{,}944$ or $11{,}008$).
>
> 2. **What would happen if** you converted a standard serial Transformer block into a parallel block? Name one throughput advantage and one modeling trade-off.
> **Answer:** Throughput improves (~15% faster training in PaLM) because the two sublayers share one normalization pass and can fuse their input matrix multiplies; the trade-off is that the MLP can no longer act on the current layer's attention output, which caused a small quality drop at 8B scale in PaLM's ablation.

Section 5: Positional Encodings and Rotary Position Embeddings (RoPE)

Self-attention computes similarity scores via inner products between query vectors and key vectors, $\langle q_i, k_j \rangle$. Because those projections apply the same matrices W^Q and W^K at every token position, attention by itself has no awareness of token order.

What mathematical property *should* a positional encoding satisfy? In language, the syntactic and semantic relationship between two words (“we” followed immediately by “know”) depends on their **relative offset** $i - j$, not on whether the phrase starts at token index 0 or token index $2{,}000$. Formally, we want a position-encoding function $f(x, i)$ applied to a token vector x at position i such that the attention inner product between token x at position i and token y at position j depends *only* on the two token contents and their relative displacement $i - j$:

$$\langle f(x, i),\ f(y, j) \rangle = g(x, y,\ i - j)$$

Prior approaches failed to achieve this clean property for three distinct reasons:

Type	Formulation	Historical Models	Why It Misses Pure Relative Inner Products
Sinusoidal	$\mathrm{Embed}(x, i) = v_x + \mathrm{PE}_i$	Original Transformer (2017)	Adding PE_i creates unwanted position-content cross terms $\langle \mathrm{PE}_i, v_y \rangle$ and $\langle v_x, \mathrm{PE}_j \rangle$ that depend on absolute positions
Learned absolute	$\mathrm{Embed}(x, i) = v_x + u_i$	GPT-1/2/3, OPT	Learned vectors u_i encode absolute index only; cannot generalize past maximum training length and also produce cross terms
Relative (additive)	$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$	T5, Gopher, Chinchilla	Injects a learned relative bias/vector a_{ij}^K inside attention logits, which is not a pure inner product $\langle f(x, i), f(y, j) \rangle$ and complicates fast attention kernels
RoPE (rotary)	Rotate Q_i, K_j by angles $i\Theta, j\Theta$	GPT-J, PaLM, LLaMA, **modern consensus**	Exact relative invariance via pure inner product, zero cross terms

RoPE Intuition: Shift Invariance via Rotation

Recall from linear algebra that rotating two vectors in a 2D plane by the *exact same angle* leaves their Euclidean inner product completely unchanged, because the angle between them and their lengths do not change. **Rotary Position Embeddings (RoPE)** (Su et al., 2021) turns that geometric fact into a positional encoding.

Reconstruct the three-panel RoPE diagram to see this geometrically:

- **Panel 1 (Unrotated Embeddings)**: Two vectors labeled `we` (pointing nearly straight up) and `know` (pointing down and to the right) radiate from the origin. There is no position information; only their intrinsic angle separates them.
- **Panel 2 (`"we know that"`, positions 0 and 1)**: We rotate each token's vector counterclockwise by $m\theta$, where m is its integer position. Token `we (0)` rotates by 0θ (stays put). Token `know (1)` rotates counterclockwise by 1θ from its faded dashed original position toward `we`. The relative rotational offset between them is $+1\theta$.
- **Panel 3 (`"of course we know"`, positions 2 and 3)**: Now `we (2)` rotates counterclockwise by 2θ, and `know (3)` rotates counterclockwise by 3θ. Both vectors have swung far around the plane compared to Panel 2—yet the angle *between* `we (2)` and `know (3)` is identical to Panel 2! Their inner product is **exact to the last bit**.

DEFINITION

Rotary Position Embedding (RoPE): A multiplicative positional encoding that pairs the d coordinates of each query and key vector into $d/2$ independent 2D planes and rotates plane k at token position m by angle $m\theta_k$, guaranteeing that query-key inner products depend strictly on relative offset $n - m$.

RoPE Mathematics

To extend 2D rotation to a d-dimensional head vector (where d is even), RoPE splits the d coordinates into $d/2$ independent 2D pairs and rotates pair $k \in \{1, \ldots, d/2\}$ by angle $m\theta_k$ using a block-diagonal orthogonal matrix $R^d_{\Theta,m} \in \mathbb{R}^{d\times d}$:

$$f_{\{q,k\}}(x_m, m) = R^d_{\Theta,m} W_{\{q,k\}} x_m$$

where

$$R^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

Break this expression down piece by piece:

- $W_{\{q,k\}} x_m$ is the standard unrotated query vector $q \in \mathbb{R}^d$ (or key vector $k \in \mathbb{R}^d$) for token m.
- Each 2×2 diagonal block $\begin{pmatrix} \cos m\theta_j & -\sin m\theta_j \\ \sin m\theta_j & \cos m\theta_j \end{pmatrix}$ is a standard 2D rotation matrix rotating coordinate pair $(2j-1, 2j)$ by angle $m\theta_j$.

- Just like sinusoidal encodings, the base angular frequencies θ_j span a geometric range (from fast-rotating low pairs that resolve nearby tokens to slow-rotating high pairs that resolve distant tokens).
- Now take the inner product between a query at position m and a key at position n. Because rotation matrices satisfy $(R_{\Theta,m})^T = R_{\Theta,-m}$ and $R_{\Theta,-m}R_{\Theta,n} = R_{\Theta,n-m}$:

$$\left(R^d_{\Theta,m}\, q\right)^T \left(R^d_{\Theta,n}\, k\right) = q^T \left(R^d_{\Theta,m}\right)^T R^d_{\Theta,n}\, k = q^T R^d_{\Theta,n-m}\, k$$

Notice what disappeared: there are **zero position-content cross terms** and no dependence on the absolute positions m or n individually—only on their difference $n - m$. Because the matrix $R^d_{\Theta,m}$ is sparse (block-diagonal 2×2), implementations never materialize the $d \times d$ matrix; they compute the rotation in $O(d)$ element-wise operations using precomputed $\cos$ and $\sin$ tables.

WORKED EXAMPLE

Verify exact shift invariance in 2D ($d = 2$) with $q = [1,\, 0]^T$, $k = [0,\, 1]^T$, and angular step $\theta = \pi/6 = 30^\circ$, and contrast the result against additive positional encoding.

1. **RoPE at positions** $(m, n) = (0, 1)$ (relative offset $n - m = +1$):
   - $R_0 q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.
   - $R_1 k = \begin{pmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.5 \\ \sqrt{3}/2 \end{pmatrix}$.
   - Inner product: $(R_0 q)^T (R_1 k) = (1)(-0.5) + (0)(\sqrt{3}/2) = -0.5$.
2. **RoPE at positions** $(m, n) = (2, 3)$ (same relative offset $n - m = +1$, with $2\theta = 60^\circ$ and $3\theta = 90^\circ$):
   - $R_2 q = \begin{pmatrix} \cos 60^\circ & -\sin 60^\circ \\ \sin 60^\circ & \cos 60^\circ \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.5 \\ \sqrt{3}/2 \end{pmatrix}$.
   - $R_3 k = \begin{pmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$.
   - Inner product: $(R_2 q)^T (R_3 k) = (0.5)(-1) + (\sqrt{3}/2)(0) = -0.5$.
   - Direct identity check: $q^T R_{3-2} k = [1,\, 0] \begin{pmatrix} -0.5 \\ \sqrt{3}/2 \end{pmatrix} = -0.5$.
3. **Contrast with additive positional encoding** $p_m = [\cos m\theta,\, \sin m\theta]^T$:
   - Expanding $(q + p_m)^T (k + p_n) = q^T k + q^T p_n + p_m^T k + p_m^T p_n$.
   - At $(0, 1)$: $0 + \cos 30^\circ + \sin 0^\circ + \cos 30^\circ = \sqrt{3} \approx +1.732$.
   - At $(2, 3)$: $0 + \cos 90^\circ + \sin 60^\circ + \cos 30^\circ = 0 + \frac{\sqrt{3}}{2} + \frac{\sqrt{3}}{2} = \sqrt{3}$ here only by coincidence of orthogonal q, k; change $k = [1, 0]^T$ and at $(0, 1)$ you get $1 + \cos 30^\circ + \cos 0^\circ + \cos 30^\circ = 2 + \sqrt{3} \approx 3.732$, whereas at $(2, 3)$ you get $1 + \cos 90^\circ + \cos 60^\circ + \cos 30^\circ = 1.5 + \frac{\sqrt{3}}{2} \approx 2.366$!

> **Interpretation**: Additive encodings corrupt the similarity score with absolute-position cross terms ($q^T p_n$ and $p_m^T k$). RoPE achieves exact shift invariance by construction.

Implementing RoPE in Attention and 2025–2026 Variants

Inspecting the actual attention forward pass clarifies three crucial implementation rules:

```
# 1. Project hidden_states to Q, K, V and reshape to (bsz, heads, seq_len, head_dim)
q = self.q_proj(x).view(bsz, q_len, self.num_heads, self.head_dim).transpose(1, 2)
k = self.k_proj(x).view(bsz, q_len, self.num_kv_heads, self.head_dim).transpose(1, 2)
v = self.v_proj(x).view(bsz, q_len, self.num_kv_heads, self.head_dim).transpose(1, 2)

# 2. Compute per-position cos/sin tables and rotate Q and K (never V)
cos, sin = self.rotary_emb(v, position_ids)
q, k = apply_rotary_pos_emb(q, k, cos, sin)

# 3. Proceed with standard scaled dot-product attention (q, k, v)
```

1. **Applied inside every attention layer**, not once at the input embedding: if you rotated token embeddings only at the bottom, subsequent nonlinearities and linear projections would mix coordinates and destroy the exact block-diagonal relative-position identity.
2. **Only Q and K are rotated; never V**: queries and keys exist to compute positional and semantic alignment scores via QK^T. Values V carry content forward into the weighted average; rotating V by position would distort content representations needlessly.
3. **2025–2026 RoPE variants** refine how much of the head and which layers receive rotation—a distinction we will need again when we examine Multi-Head Latent Attention (MLA) and hybrid long-context attention in Section 9:
   - **Partial RoPE (`p-RoPE`)**: Rotate only the first d_{rope} dimensions of each head and leave the remaining semantic dimensions unrotated. Used in Gemma 4 and Qwen 3.5; in the late-2026 frontier, **GLM-5.3** uses `rope_dim = 64` decoupled inside its MLA layers.
   - **Hybrid RoPE + NoPE**: Mix positional schemes across layers. **Command A** applies RoPE on local sliding-window attention layers (where fine relative distance matters) and **NoPE** (no positional embedding at all) on full-attention layers, which improves long-range length generalization.
   - **Context extension via YaRN**: Scaling RoPE frequencies on full-attention layers—for example, **OLMo 3** uses YaRN RoPE scaling with base frequency $\theta = 5 \times 10^5$—stretches the phase wavelengths so models trained on shorter windows generalize cleanly to much longer contexts.

COMMON MISCONCEPTION

You might think RoPE should be applied to Q, K, and V—or applied once right after the token embedding table like 2017 sinusoidal encodings. Both break: applying RoPE once at the input loses exact relative invariance after the first linear layer and nonlinearity, while rotating V corrupts value vectors with position-dependent phase shifts even though values never enter the QK^T inner product.

KEY IDEA

RoPE encodes position multiplicatively by rotating 2D coordinate pairs of Q and K (never V) inside every attention layer, achieving exact relative-offset invariance with zero cross terms; modern long-context models refine this via partial RoPE, RoPE/NoPE layer hybrids, and YaRN frequency scaling.

Now that every component inside a Transformer block is specified—Pre-RMSNorm, bias-free SwiGLU/MoE, serial wiring, and RoPE—we face a different class of design choices: how to size the dimensions of those blocks (FFN width, head count, depth vs. width, and vocabulary size) to balance modeling quality against multi-device systems efficiency.

✎ Check Your Understanding

1. **Yes or No (with justification):** Should `apply_rotary_pos_emb` be called on the value tensor V in addition to Q and K?
   Answer: No. RoPE is only applied to Q and K so that the attention score inner product $q_m^T k_n$ depends on relative offset $n - m$; values V carry content without rotational distortion.
2. **Quick computation:** Under RoPE, if $(R_{\Theta,10}q)^T(R_{\Theta,15}k) = 4.2$ for two specific query and key vectors q and k, what is the exact value of $(R_{\Theta,1000}q)^T(R_{\Theta,1005}k)$?
   Answer: Exactly 4.2, because $(R_{\Theta,m}q)^T(R_{\Theta,n}k) = q^T R_{\Theta,n-m}k$ depends only on the relative offset $n - m = 5$.

Section 6: Architectural Dimensions and Vocabulary Sizing

Once the qualitative architecture of a Transformer block is fixed, you must pick concrete integer dimensions: the FFN intermediate width d_{ff}, the number of attention heads n_{head} and head dimension d_{head}, the balance between hidden width d_{model} and layer count n_{layer}, and the vocabulary size $|V|$. A recurring empirical finding from scaling studies is that **language modeling loss sits in wide, flat basins** across many of these ratios—which means the winning choices in practice are dictated as much by parallelization and accelerator efficiency as by raw perplexity.

Hyperparameter 1: Feed-Forward Ratio ($d_{\text{ff}}/d_{\text{model}}$)

From Section 4, recall the two baseline rules:

- **Non-gated FFN default**: $d_{\text{ff}} = 4\, d_{\text{model}}$.
- **Gated (`*GLU`) default** (applying the $\frac{2}{3}$ scaling for 3 matrices): $d_{\text{ff}} = \frac{8}{3}\, d_{\text{model}} \approx 2.67\, d_{\text{model}}$.

In practice, dense GLU models span a noticeable range around and above 2.67:

Model (GLU)	$d_{\text{ff}}/d_{\text{model}}$
PaLM (540B)	4.0
Mistral 7B	3.5
Llama 2 / Llama 3 70B	**3.5**
Yi 34B	2.85
LLaMA 1 65B	**2.68**
DeepSeek 67B	2.68
Qwen 14B	2.67
T5 v1.1	2.5

Notice the distinction between LLaMA generations: LLaMA 1 65B sat right at the $\frac{8}{3}$ rule (2.68), whereas Llama 2 and Llama 3 70B deliberately pushed the GLU ratio up to 3.5 (and other models pushed further: SmolLM, Gemma 3, and Gemma 4 use $4\times$ GLU, while Gemma 2 uses $8\times$).

Why can models vary $d_{\text{ff}}/d_{\text{model}}$ from $2.5\times$ to $8\times$ without collapsing? Reconstruct the empirical curve from **Kaplan et al. (2020)** on 50M-parameter models plotting **Loss Increase (%)** on the vertical axis (0% to 8%) against **Feed-Forward Ratio** $d_{\text{ff}}/d_{\text{model}}$ on a logarithmic horizontal axis (from below $10^0 = 1$ through $10^1 = 10$ out to 40):

- Between $1\times$ **and** $10\times$, there is a broad, flat **near-optimal green basin** where loss stays within roughly 1% of the minimum.
- Past $10\times$, the curve bends sharply upward, climbing past 4% around $20\times$ and reaching $\sim 8\%$ loss degradation by $40\times$.

History provides a famous cautionary outlier right past the edge of that basin: **T5 11B** (Raffel et al., 2020) set $d_{\text{model}} = 1024$ and $d_{\text{ff}} = 65{,}536$—an extreme $64\times$ **multiplier**—specifically to maximize efficiency on large dense matrix multiplications on accelerators. When the authors revisited the design in **T5 v1.1**, they reverted to a $2.5\times$ GeGLU ratio, confirming that pushing far outside the $[1, 10]$ basin wastes parameter capacity.

Hyperparameter 2: Attention Head Dimensions

In standard multi-head attention, the hidden state $X \in \mathbb{R}^{n \times d_{\text{model}}}$ is projected to Q, K, V and split across n_{head} heads of dimension d_{head}. The standard convention sets:

$$n_{\text{head}} \cdot d_{\text{head}} = d_{\text{model}} \qquad (\text{head-to-model ratio} = 1)$$

This convention is algebraically clean: a single projection $XW^Q \in \mathbb{R}^{n \times d}$ reshapes directly into $\mathbb{R}^{n \times h \times (d/h)}$, treating the head dimension as an extra batch axis.

However, ratio $= 1$ is a convention, **not a hard mathematical requirement**. Nothing stops the projection W^Q from mapping d_{model} to a total attention width $n_{\text{head}} \cdot d_{\text{head}}$ that differs from d_{model} (with the output projection W^O mapping back to d_{model}):

Model	Heads (n_{head})	d_{head}	d_{model}	Ratio
GPT-3	96	128	12,288	1
LLaMA 2	64	128	8,192	1
T5 v1.1	64	64	4,096	1
T5 11B	128	128	1,024	**16**
LaMDA	128	128	8,192	**2**
PaLM	48	258	18,432	**1.48**
Qwen 3.5 (27B)	24	256	5,120	**1.2**

Notice that several Google models (T5 11B, LaMDA, PaLM) and Qwen 3.5 (27B, where $24 \times 256 = 6144 = 1.2 \times 5120$) decouple total head width from d_{model} (ratio $= 1$ remains the clean default, though direct empirical ablations comparing ratio $= 1$ against ratio > 1 are sparse). In the late-2026 frontier, decoupling goes even further: **DeepSeek-V4.1-Flash** (Sept 2026) uses **64 query heads with `head_dim = 512`** paired with a single compressed KV head, showing that query head dimensionality is now chosen independently of both d_{model} and KV storage size.

Hyperparameter 3: Aspect Ratio ($d_{\text{model}}/n_{\text{layer}}$) and Systems Parallelism

Given a fixed total parameter budget, should you build a **wide, shallow** network (large d_{model}, small n_{layer}) or a **narrow, deep** network (small d_{model}, large n_{layer})? We summarize this shape using the **aspect ratio** $d_{\text{model}}/n_{\text{layer}}$:

Model	Aspect Ratio ($d_{\text{model}}/n_{\text{layer}}$)
BLOOM	205
T5 v1.1	171
PaLM (540B)	**156**
GPT-3 / OPT / Mistral / Qwen / OLMo 3	**128**
LLaMA / LLaMA 2	**102**
Gemma 3	87
Gemma 4	61
T5 (11B)	33

Reconstruct the **Kaplan et al. (2020)** plot of **Test Loss** versus **Aspect Ratio** $d_{\text{model}}/n_{\text{layer}}$ (log scale from 10^1 to 10^3) across three parameter scales (50M in blue, 274M in orange, 1.5B in green): all three curves trace shallow U-shapes that stay nearly flat across a wide green middle band spanning roughly 30 to 300, climbing steeply only when a model becomes extremely deep and narrow (below ~ 15) or extremely wide and shallow (above ~ 500). Most large models cluster in a sweet spot around 100–200, while some newer models run somewhat deeper (such as **GLM-5.3** at 78 layers and hidden size $6{,}144$, giving an aspect ratio of $6144/78 \approx 78.8$).

If test loss is nearly flat across a huge range of aspect ratios—and **Tay et al. (2021)** even found that deeper models can show strong quality per FLOP at small and medium scales—why don't frontier models simply stack hundreds of narrow layers?

Reconstruct the four-device pipeline diagram to see why **aspect ratio is fundamentally a systems choice**:

- Four boxes sit left to right: `Layer 0 (Device 0)`, `Layer 1 (Device 1)`, `Layer 2 (Device 2)`, `Layer 3 (Device 3)`.
- Red `forward` arrows chain strictly left to right from Device $0 \to 1 \to 2 \to 3$.
- Blue `backward` arrows chain strictly right to left from Device $3 \to 2 \to 1 \to 0$.

Because **depth is inherently sequential**—Layer $l+1$ cannot begin until Layer l finishes—an overly deep model has higher inference latency per token and forces you to split layers across devices using **pipeline parallelism**, which introduces sequential dependencies and bubble overhead. **Width, by contrast, parallelizes easily** across devices via **tensor parallelism** (splitting large matrix multiplies column-wise and row-wise across devices concurrently). Balanced aspect ratios around 100–200 (or moderately deep designs like 60–80) keep matrix multiplies wide enough to parallelize efficiently while capping sequential latency.

COMMON MISCONCEPTION

You might think the $d_{\text{model}}/n_{\text{layer}} \approx 100$–$200$ sweet spot exists because language modeling loss has a sharp mathematical minimum right at 128. That breaks against Kaplan et al.'s flat U-curve: loss varies minimally across an entire order of magnitude of aspect ratios. The 100–200 band is selected primarily by systems parallelism—trading off sequential latency across layers against tensor-parallel efficiency within wide layers.

Hyperparameter 4: Vocabulary Size

The vocabulary size $|V|$ determines how text is segmented into subword tokens and sets the size of the input embedding table ($|V| \times d_{\text{model}}$) and output LM head ($d_{\text{model}} \times |V|$). Practice splits cleanly into two regimes:

- **Monolingual English models (~30k–50k)**: LLaMA ($32{,}000$), T5 / T5 v1.1 ($32{,}128$), Original Transformer ($37{,}000$), GPT-1 ($40{,}478$), GPT-2 / GPT-3 ($50{,}257$).
- **Multilingual, code, and modern production models (~100k–262k)**: Yi ($64{,}000$), GPT-4 `cl100k` ($\approx 100{,}277$), DeepSeek LLM ($102{,}400$), Qwen 1.5 / 2 / 3 ($151{,}936$–$152{,}064$), mT5 (250k), PaLM (256k), and Gemma 4 ($262{,}144$).

Why expand vocabulary size by $3\times$ to $8\times$ for multilingual and production models? A 32k English-centric tokenizer splits non-English scripts and programming syntax into tiny byte-level fragments, inflating sequence length n—which multiplies both attention cost and decode steps per sentence. A 100k–262k vocabulary compresses multilingual text and code into far fewer tokens per document.

The late-2026 frontier pushes vocabulary-like lookup capacity even further: **Qwen3.8-Flash-Next** (Aug 2026) pairs its 125B backbone (6B active) with a **51B-parameter n-gram embedding table kept in host memory**, offloading massive lexical lookup capacity out of the active Transformer layers.

WORKED EXAMPLE

Size a complete Transformer layer configuration using GLM-5.3's hidden dimensions ($d_{\text{model}} = 6144$, $n_{\text{layer}} = 78$, $n_{\text{head}} = 64$) and check the parameter cost of vocabulary scaling.

1. **Aspect ratio**: $d_{\text{model}}/n_{\text{layer}} = 6144/78 \approx 78.8$ (moderately deeper than the historical 100–200 cluster, sitting between Gemma 4 at 61 and Gemma 3 at 87).
2. **Standard ratio-1 head dimension**: $d_{\text{head}} = d_{\text{model}}/n_{\text{head}} = 6144/64 = 96$.
3. **Standard SwiGLU FFN width** ($\frac{8}{3}$ rule): $d_{\text{ff}} = \frac{8}{3} \times 6144 = 16{,}384$.
4. **Embedding table parameter cost** ($|V| \times d_{\text{model}}$):
   - At monolingual $|V| = 32{,}000$: $32{,}000 \times 6144 = 196.6\text{M}$ parameters (393.2M if input and output embeddings are untied).
   - At multilingual $|V| = 256{,}000$: $256{,}000 \times 6144 = 1.573\text{B}$ parameters (3.15B if untied).

> **Interpretation**: Raising vocabulary size to 256k adds over 1.5B parameters per embedding matrix —tiny as a fraction of a 500B+ model, yet substantial for a 3B–8B model—which is why large production models gladly pay the embedding parameter cost to gain shorter sequence lengths across languages and code.

> **KEY IDEA**
> Both the FFN ratio ($[1, 10]$) and the aspect ratio ($d_{\text{model}}/n_{\text{layer}}$) exhibit wide, flat loss basins, so practical choices—such as keeping aspect ratio around 100–200 to favor tensor parallelism over sequential depth, and expanding vocabularies to 100k–262k for multilingual compression—are governed primarily by systems and inference efficiency.

With architectural dimensions settled, we turn to a surprising empirical puzzle in training hyperparameters: when pretraining runs over tens of trillions of tokens in a single pass—making classic overfitting nearly impossible—why do modern LLMs drop dropout to zero yet still keep weight decay?

> **✎ Check Your Understanding**
>
> 1. **Yes or No (with justification):** If you double a model's depth n_{layer} while shrinking its width d_{model} to keep total parameters fixed, can tensor parallelism across devices keep per-token inference latency unchanged?
>   **Answer:** No. Depth is inherently sequential—each layer must wait for the previous layer's output —so making a model much deeper increases step latency and forces pipeline parallelism rather than pure tensor parallelism.
> 2. **Fill in the blank:** Empirically (Kaplan et al., 2020), the FFN ratio $d_{\text{ff}}/d_{\text{model}}$ sits in a flat near-optimal basin roughly between $1\times$ and _____, beyond which loss climbs steeply (as confirmed when T5 v1.1 reverted T5 11B's extreme $64\times$ ratio).
>   **Answer:** $10\times$.

Section 7: Regularization in Single-Pass Pretraining: Dropout and Weight Decay

In traditional supervised learning on small datasets, models see the same training examples for dozens or hundreds of epochs. Regularization techniques—specifically **dropout** (randomly zeroing activations during training) and **weight decay** (shrinking parameter magnitudes on every step)—were invented to stop models from memorizing the training set and hurting validation generalization.

Now examine how modern language model pretraining actually runs. Training corpora span **trillions of tokens**: historical Llama 3 (July 2024) trained on **15T tokens**, historical Qwen3 (May 2025) trained on **36T tokens**, and late-2026 **DeepSeek-V4.1-Flash** (Sept 2026) pretrained on **45T tokens**. Because

pretraining operates mostly as **single-pass stochastic gradient descent**—each sequence is seen roughly once—it is very difficult for the model to memorize the training corpus and open a train-versus-validation generalization gap.

Look at how real models responded in practice:

Model	Dropout*	Weight Decay (λ_{WD})
Original Transformer	0.1	0
GPT-2	0.1	0.1
T5	0.1	0
GPT-3	0.1	0.1
OPT	0.1	0.1
T5 v1.1	**0**	0
PaLM	**0**	(variable)
LLaMA / Llama 3	**0**	**0.1**
Qwen 14B	0.1	0.1

**Papers rarely discuss dropout explicitly; open models usually omit it.*

Older models routinely set dropout to 0.1. Modern open models almost uniformly set **dropout** $= 0$—which makes perfect sense in single-pass pretraining—yet they **still keep weight decay around** $\lambda_{\text{WD}} = 0.1$! That leaves a sharp puzzle: if we are not overfitting, why does weight decay still help?

Solving the Puzzle: Optimization Dynamics, Not Overfitting

Andriushchenko et al. (2023) directly tested whether weight decay improves generalization in LLM pretraining. Reconstruct their scatter plot (**Validation Loss** on the vertical axis vs. **Training Loss** on the horizontal axis) across three values of weight decay—$\lambda_{\text{WD}} = 0.0$ (red dots), $\lambda_{\text{WD}} = 0.1$ (green dots), and $\lambda_{\text{WD}} = 0.3$ (blue dots):

- Every single point from all three runs—red, green, and blue—lies tightly overlapping along the **exact same narrow diagonal line**.
- For any fixed training loss, a model trained with $\lambda_{\text{WD}} = 0.3$ has the *exact same* validation loss as a model trained with $\lambda_{\text{WD}} = 0.0$.

Weight decay is **not** closing a generalization gap between training and validation loss. Instead, it achieves lower validation loss strictly because it reaches a **lower final training loss**!

> **DEFINITION**
> **Decoupled Weight Decay (λ_{WD})**: An optimizer shrinkage term (as in AdamW) that scales weights toward zero on every step via $w_{t+1} = (1 - \eta_t \lambda_{\text{WD}}) w_t - \eta_t u_t$; in single-pass LLM pretraining, it functions primarily as an optimization-dynamics control rather than an anti-overfitting regularizer.

How can adding a shrinkage penalty—which deliberately pulls parameters away from minimizing the raw training objective—end up producing a *lower* training loss? The mechanism is its interaction with the **cosine learning-rate decay schedule** η_t:

1. **During the high, nearly constant learning-rate phase**: Higher λ_{WD} actually produces *higher* training loss than $\lambda_{\text{WD}} = 0$. Shrinking the weights keeps parameter norms $\|w\|_2$ smaller, which increases the *effective* learning rate relative to weight scale and keeps the optimizer exploring aggressively rather than settling prematurely.
2. **During the cosine learning-rate decay phase**: As η_t smoothly anneals toward zero, the effective step noise drops rapidly. Because the higher-λ_{WD} trajectory maintained better-conditioned weight norms and explored a wider basin during the high-LR phase, its training loss drops sharply, overtakes $\lambda_{\text{WD}} = 0$, and converges to a **lower final training loss** (and therefore lower validation loss along that exact diagonal line).

COMMON MISCONCEPTION

You might think that setting weight decay to 0.1 in LLaMA or Llama 3 is necessary to prevent the model from memorizing its 15-trillion-token training set. That breaks against Andriushchenko et al.'s validation-versus-training-loss plot: models with $\lambda_{\text{WD}} \in \{0.0, 0.1, 0.3\}$ all trace the exact same diagonal line, proving zero difference in generalization gap. Weight decay helps by improving optimization dynamics under a decaying learning-rate schedule.

WORKED EXAMPLE

Trace one parameter update under decoupled weight decay, $w_{t+1} = (1 - \eta_t \lambda_{\text{WD}}) w_t - \eta_t u_t$, where u_t is the AdamW normalized update direction (typically $|u_t| \sim O(1)$), comparing $\lambda_{\text{WD}} = 0$ against $\lambda_{\text{WD}} = 0.1$.

Suppose a weight has grown to $w_t = 5.0$ and the normalized step pushes further outward with $u_t = -1.0$:

1. **High-LR phase** ($\eta_t = 10^{-3}$):
   - With $\lambda_{\text{WD}} = 0$: $w_{t+1} = 5.0 + 0.001 = 5.001$. Nothing caps the weight norm, so over thousands of steps $\|w\|_2$ drifts larger, shrinking the relative update size $|\Delta w|/|w|$.
   - With $\lambda_{\text{WD}} = 0.1$: shrinkage is $-\eta_t \lambda_{\text{WD}} w_t = -(10^{-3})(0.1)(5.0) = -0.0005$, while the gradient step is $+0.001$, giving $w_{t+1} = 5.0005$. Weight norm reaches an equilibrium where shrinkage balances gradient push, keeping the effective relative learning rate high (explaining why training loss stays slightly higher while η_t is large).
2. **Late cosine-decay phase** ($\eta_t = 10^{-5}$, $100\times$ smaller):
   - Both shrinkage (-5×10^{-6}) and step size ($+10^{-5}$) anneal down together, allowing the better-conditioned weights to settle cleanly into a lower-loss minimum than the $\lambda_{\text{WD}} = 0$ run.

> **Interpretation**: Weight decay acts as an implicit norm regulator coupled to the learning-rate schedule—boosting exploration when η_t is high and unlocking lower final loss when η_t decays.

> **KEY IDEA**
> Modern single-pass LLM pretraining drops dropout to zero because overfitting is minimal over trillions of tokens, yet retains weight decay (~ 0.1) because its interaction with cosine learning-rate decay improves optimization dynamics and reaches lower final training (and therefore validation) loss.

Controlling weight norms via weight decay improves smooth convergence, but at multi-billion and trillion-parameter scale, training runs encounter a far more violent failure mode: sudden catastrophic loss spikes and gradient norm explosions. Next, we diagnose the exact mathematical culprits behind those spikes—the Transformer's two softmaxes—and walk through the architectural and optimizer defenses developed through late 2026.

> **✎ Check Your Understanding**
>
> 1. **Yes or No (with justification):** According to Andriushchenko et al. (2023), does increasing weight decay from 0.0 to 0.3 reduce the gap between validation loss and training loss at a given training loss value?
>   **Answer:** No. Across $\lambda_{\text{WD}} \in \{0.0, 0.1, 0.3\}$, validation loss versus training loss falls on the exact same diagonal line; weight decay lowers validation loss by lowering final training loss through optimization dynamics.
> 2. **What would happen to** training loss during a *constant* (non-decaying) learning-rate phase if you increased λ_{WD} from 0.0 to 0.3?
>   **Answer:** Under a constant high learning rate, higher λ_{WD} actually produces *higher* training loss; it only overtakes and beats $\lambda_{\text{WD}} = 0$ once the learning rate decays under a schedule such as cosine decay.

Section 8: Training Stability at Scale: Softmaxes, Normalization, Gating, and Optimizers

At small scale, a well-tuned Pre-RMSNorm Transformer trains smoothly. At large scale—hundreds of billions to trillions of parameters trained across tens of trillions of tokens—runs are routinely derailed by **loss spikes** and **gradient norm explosions**.

Reconstruct the two-panel diagnostic plot comparing **OLMo 0424 7B (spiky blue curve)** against **OLMo 2 1124 7B (smooth orange curve)** across training steps:

- **Top panel (Loss vs. Training Steps)**: The smooth orange curve descends monotonically and cleanly across the entire run. The blue curve descends overall, but periodically shoots violently straight upward into sharp vertical spikes before dropping back down.
- **Bottom panel (Grad L2 Norm vs. Training Steps)**: The orange curve drops immediately after initialization and hugs a flat, quiet floor near zero for the rest of training. The blue curve erupts repeatedly into massive vertical spikes in global gradient L_2 norm, aligning with and destabilizing the loss trajectory.

Where do these violent spikes originate? While many components interact, the **two primary culprits are the model's two softmax operations**:

1. **The output softmax** over the $|V|$ vocabulary logits at the LM head.
2. **The attention softmax** over the scaled dot products $QK^T/\sqrt{d_k}$ inside every attention head.

Both operations exponentiate unbounded inner products via e^z. Because the exponential function grows extraordinarily fast, even moderate upward drift in logit magnitudes produces numerical overflow, saturation, or near-zero denominators, injecting huge, erratic gradients backward through the entire stack. The engineering goal is to bound logits and activations smoothly without degrading model capacity.

Stabilizing the Output Softmax: Auxiliary z-loss

Recall the log-softmax probability of the target token r given logits $U(x) \in \mathbb{R}^{|V|}$:

$$\log P(x) = U_r(x) - \log Z(x), \qquad \text{where} \quad Z(x) = \sum_{r'=1}^{|V|} e^{U_{r'}(x)}$$

Break this apart: $U_{r'}(x)$ is the raw pre-softmax score (logit) for vocabulary token r', $Z(x)$ is the **partition function** (normalizing sum of exponentials across all $|V|$ tokens), and subtracting $\log Z(x)$ converts the raw score $U_r(x)$ into a valid log-probability.

Notice a dangerous symmetry: softmax is **shift-invariant**. Adding any scalar constant c to every logit ($U_{r'} \leftarrow U_{r'} + c$) multiplies $Z(x)$ by e^c and adds c to both $U_r(x)$ and $\log Z(x)$, leaving $\log P(x)$ completely unchanged! Because cross-entropy loss $-\log P(x)$ exerts **zero restorative force** on the overall shift c, $\log Z(x)$ can drift arbitrarily large during training until floating-point precision collapses.

The **auxiliary z-loss** (Devlin et al., 2014; popularized by PaLM) breaks this shift invariance by adding a tiny quadratic penalty encouraging $\log Z(x)$ to stay near zero:

$$\mathcal{L} = \sum_i \Big[-\log P(x_i) + \alpha \log^2 Z(x_i) \Big]$$

DEFINITION

Auxiliary z-loss: A lightweight stability penalty $\alpha \log^2 Z(x)$ added to the cross-entropy loss that penalizes the output softmax log-partition function $\log Z(x)$ for drifting away from zero, breaking softmax shift invariance with near-zero compute overhead.

Typical coefficient values are tiny—$\alpha = 10^{-4}$ in PaLM and $\alpha = 10^{-5}$ in OLMo 3—so the penalty barely perturbs normal probabilities while exerting a strong restoring force whenever logits begin drifting. It has been widely adopted across PaLM, Baichuan 2 (2023), DCLM (2024), OLMo 2 (2025), and OLMo 3 (2025).

WORKED EXAMPLE

Verify how shift invariance lets cross-entropy miss a $+100$ logit drift, and how z-loss catches it immediately, on a 3-token vocabulary with target token $r = 3$.

1. **Well-behaved logits** $U = [0,\ 1,\ 2]$:
   - Partition function: $Z = e^0 + e^1 + e^2 = 1 + 2.718 + 7.389 = 11.107$, so $\log Z \approx 2.408$.
   - Target log-probability: $\log P(r = 3) = 2 - 2.408 = -0.408$ (probability 0.665).
   - With $\alpha = 10^{-4}$, the z-loss term is $10^{-4} \times (2.408)^2 \approx 0.00058$ (negligible compared to cross-entropy 0.408).
2. **Drifted logits** $U' = U + 100 = [100,\ 101,\ 102]$:
   - Partition function: $\log Z' = 100 + \log Z = 102.408$.
   - Target log-probability: $\log P'(r = 3) = 102 - 102.408 = -0.408$. Cross-entropy loss is **completely blind** to the $+100$ drift!
   - Under z-loss ($\alpha = 10^{-4}$): penalty $= 10^{-4} \times (102.408)^2 \approx 1.049$—more than $1{,}800\times$ **larger** than before and larger than the entire cross-entropy loss, generating a strong gradient that pulls $\log Z$ back toward 0.

Interpretation: Because $\log Z(x)$ is already computed during cross-entropy, z-loss costs essentially zero extra compute while eliminating output-logit drift completely.

Stabilizing the Attention Softmax: QK-Norm and Logit Soft-Capping

The second softmax sits inside every attention head: $\mathrm{softmax}(QK^T/\sqrt{d_k})$. As training progresses, query and key weight norms can grow until specific $q_m^T k_n/\sqrt{d_k}$ dot products explode, driving attention entropy to zero (all probability mass collapses onto one token) and triggering gradient spikes. Two methods directly target attention logits:

1. **QK-Norm** (Dehghani et al., 2023; IDEFICS, Chameleon): Apply LayerNorm or RMSNorm directly to the query vectors Q and key vectors K immediately before forming QK^T. Because normalized q_m and k_n have bounded norms, their dot product $q_m^T k_n/\sqrt{d_k}$ is mathematically bounded before it ever

reaches the exponential. QK-Norm is used in DCLM, OLMo 2, Gemma 2, Qwen3, OLMo 3, and Gemma 4.

2. **Logit Soft-Capping** (Bello et al., 2016; Gemma 2 and Gemma 4): Pass raw logits through a scaled hyperbolic tangent to clamp them smoothly inside the open interval $(-c, +c)$:

$$\text{logits} \leftarrow c \cdot \tanh\left(\frac{\text{logits}}{c}\right)$$

Break this down: dividing by c rescales the logit; $\tanh(\cdot)$ behaves nearly linearly ($\tanh(u) \approx u$) near 0 and saturates smoothly at ± 1 for large $|u|$; multiplying back by c restores the original scale for normal logits while guaranteeing $|\text{logits}| < c$ everywhere. Gemma 2 set $c = 50.0$ inside attention layers and $c = 30.0$ in the final LM-head layer.

DEFINITION

QK-Norm: Applying RMSNorm or LayerNorm to the projected query and key vectors Q and K immediately prior to the attention dot product QK^T, bounding attention logit magnitudes before the softmax.

DEFINITION

Logit Soft-Capping: Smoothly bounding attention or output logits into $(-c, +c)$ via $\text{logits} \leftarrow c \cdot \tanh(\text{logits}/c)$ so normal logits remain nearly linear while extreme logits saturate safely before exponentiation.

WORKED EXAMPLE

Evaluate Gemma 2's attention soft-cap ($c = 50.0$) on a normal logit versus a runaway logit.

- **Normal logit** $z = 5.0$: $50 \cdot \tanh(5/50) = 50 \cdot \tanh(0.1) \approx 50 \times 0.09967 = 4.983$ (less than 0.35% distortion).
- **Runaway logit** $z = 150.0$: Without capping, $e^{150} \approx 1.39 \times 10^{65}$ overflows low-precision formats; with capping, $50 \cdot \tanh(150/50) = 50 \cdot \tanh(3) \approx 50 \times 0.99505 = 49.75$ (smoothly bounded below 50).

Interpretation: Soft-capping acts as a smooth ceiling on extreme values, though it adds an element-wise $\tanh$ inside the attention score loop.

Which of the two performs better empirically? Check the controlled perplexity ablation (± 0.1 at 95% confidence interval, lower is better):

Configuration	`bf16` baseline	`soft_cap`	`QKV_norm`	`QK_norm_cap`	`QK_norm`	`QK_FC_norm`
Perplexity	11.19	11.24	10.85	11.00	**10.84**	10.87

Soft-capping prevents blow-ups, but slightly hurts perplexity ($11.19 \to 11.24$) and complicates fused attention kernels. **QK-Norm**, by contrast, improved *both* stability and perplexity substantially ($11.19 \to 10.84$), making it the preferred default whenever standard Q, K projections are used.

2025–2026 Stability at Scale: QK-Clip, Gated Attention, GatedNorm, and Muon

QK-Norm works cleanly for standard MHA and GQA, where queries and keys are explicit per-head vectors. However, as we will see in Section 9, **Multi-Head Latent Attention (MLA)** compresses keys and values into a shared low-rank latent vector whose up-projection matrices are algebraically absorbed into the query weights during fast inference—meaning **QK-Norm does not apply cleanly to MLA's compressed latent projections**!

That architectural collision forced three new stability mechanisms in the 2025–2026 wave and late-2026 frontier:

1. **QK-Clip (Kimi K2, July 2025)**: Instead of inserting a runtime normalization layer between Q and K, monitor the maximum attention logit of each head during training; whenever a head's max attention logit exceeds a chosen threshold, directly rescale that head's query and key projection weights downward. Using QK-Clip, **Kimi K2** pretrained **1T parameters on 15.5T tokens with zero loss spikes**.
2. **Gated Attention (Qwen)**: Placing a **learned sigmoid gate** immediately after the attention output suppresses extreme attention activations, improves training stability, and removes the need for some manual clipping. (Kimi K3 in July 2026 likewise adopts **gated MLA** layers.)
3. **Muon-Family Optimizers and GatedNorm (Late 2026)**: Rather than fighting instability only with clipping, late-2026 frontier models (**DeepSeek-V4**, **Kimi K3** with **Per-Head Muon**, and **Qwen3.8-Flash-Next**) pair architectural gating with **Muon-family optimizers** (which orthogonalize matrix updates rather than updating coordinates independently like AdamW).

> **DEFINITION**
> **QK-Clip**: A weight-rescaling stability method (introduced in Kimi K2) compatible with MLA that checks each head's maximum attention logit and rescales its query and key projection weights whenever that logit exceeds a threshold.

Look at the dramatic results from the **Qwen3.8-Next Stability Study (Aug 2026)** stress-testing aggressive learning rates (measuring spikes per 10,000 training steps):

Learning Rate Stress Test	Baseline	Stabilized Setup
$4\times$ **optimal LR**	AdamW: **183 spikes / 10k steps**	**Muon + gated residual: 0 spikes**
$3\times$ **optimal LR**	Standard norm: **32.0 spikes / 10k steps**	**GatedNorm: 3.2 spikes / 10k steps**

Even when pushed to $4\times$ the optimal learning rate—a regime where AdamW explodes 183 times every 10,000 steps—combining Muon with gated residuals produced **zero spikes**, and replacing standard normalization with GatedNorm cut spikes by $10\times$ at $3\times$ learning rate.

COMMON MISCONCEPTION

You might think QK-Norm solves attention instability universally across every modern Transformer. That breaks as soon as you adopt compressed latent attention (MLA), where QK-Norm cannot be inserted cleanly without interfering with latent absorption—which is why MLA-based models introduced QK-Clip and why late-2026 frontier models combine architectural gating (Gated Attention, GatedNorm) with Muon-family optimizers.

KEY IDEA

Large-scale loss spikes stem primarily from the output softmax and attention softmax; controlling them progresses from output z-loss and QK-Norm (which also improves perplexity over soft-capping) to MLA-compatible QK-Clip, sigmoid-gated attention/norms, and Muon-family optimizers that suppress spikes even at 3–$4\times$ learning rates.

We have now solved gradient flow, normalization throughput, FFN capacity, positional encoding, sizing, regularization, and training stability. One massive bottleneck remains—and it hits hardest not during training, but when serving a trained model over long contexts up to 1 million tokens: the memory-bandwidth wall of autoregressive attention.

✎ Check Your Understanding

1. **Fill in the blank:** Output cross-entropy loss cannot prevent output logits from drifting upward by a large constant c because the softmax function is _____; adding the auxiliary z-loss term $\alpha \log^2 Z(x)$ breaks this symmetry.
   Answer: shift-invariant.
2. **Yes or No (with justification):** Based on the perplexity ablation, does logit soft-capping outperform QK-Norm on language modeling perplexity?
   Answer: No. Logit soft-capping slightly degraded perplexity relative to the `bf16` baseline (11.24 vs. 11.19), whereas QK-Norm improved both stability and perplexity (10.84).

Section 9: Attention Efficiency and Long-Context Architecture

During training—and during the **prefill** phase of inference, when a user submits a prompt of n tokens—all n tokens are processed simultaneously in large matrix multiplications. During **incremental decode** (autoregressive text generation), however, the model must generate token t before it can compute token $t+1$. To avoid recomputing past keys and values on every step, inference engines store all previous keys $K_{1:t}$ and values $V_{1:t}$ in a **KV cache** and reload that entire cache from memory to process a single new query vector q_t.

That structural difference between parallel prefill and sequential decode causes arithmetic intensity to collapse.

Prefill vs. Incremental Decode: The Arithmetic Intensity Collapse

Let b be batch size, n be sequence length, d be hidden dimension, h be number of heads, and $k = d/h$ be head dimension (assuming $n \lesssim d$ so projection/attention compute scales as $O(bnd^2)$):

1. **Training / Prefill** (all n tokens in parallel):
   - Total operations: $O(bnd^2)$.
   - Total memory accessed (activations bnd, attention score matrices bhn^2, and weights d^2): $O(bnd + bhn^2 + d^2)$.
   - Dividing operations by memory accessed (and noting $h/d = 1/k$, so for $n \sim d$, $\frac{hn}{d^2} \sim \frac{1}{k}$, while $\frac{1}{d} \ll \frac{1}{k}$):

     $$\text{AI}_{\text{prefill}} = O\left(\left(\frac{1}{k} + \frac{1}{bn}\right)^{-1}\right)$$

     Because head dimension k (e.g., 128) and bn are both large, arithmetic intensity is high—compute units stay saturated.
2. **Incremental Decode** (n sequential generation steps with standard MHA KV cache):
   - Total operations across n generated tokens: still $O(bnd^2)$.
   - Total memory accessed across n steps: at step t, we must read the entire accumulated $K_{1:t}$ and $V_{1:t}$ cache of size $O(btd)$ *plus* reload the $O(d^2)$ projection weights for just one new token per sequence. Summing $t = 1, \ldots, n$ yields:

     $$\text{Mem}_{\text{decode}} = O\Big(\underbrace{bn^2d}_{K,V \text{ cache reads}} + \underbrace{nd^2}_{\text{weight reloads}} \Big)$$

   - Dividing total operations bnd^2 by total memory $bn^2d + nd^2$:

     $$\text{AI}_{\text{decode}} = \frac{bnd^2}{bn^2d + nd^2} = O\left(\left(\frac{n}{d} + \frac{1}{b}\right)^{-1}\right)$$

Look closely at the two terms inside that denominator: $\frac{1}{b}$ can be shrunk by increasing the batch size b, **but the $\frac{n}{d}$ term does not depend on batch size at all!** As context length n grows past d, $\frac{n}{d}$ destroys arithmetic intensity regardless of how large your batch is. Reconstructing the decode dataflow diagram makes the cause visually unmistakable: a single small box `New Query q_t` enters on the left, while two huge red boxes—`K Cache: K_{1:t} (read every step!)` and `V Cache: V_{1:t} (read every step!)`—must be streamed across memory on every single token to produce `Attention Output`. Incremental decode is **severely memory-bound**; to fix it, we **must shrink the KV cache**.

Sharing KV Heads: MQA and GQA

The first major breakthrough was noticing that while having many distinct query heads h helps the model ask diverse questions about the context, storing h separate copies of the key and value caches is largely redundant.

Reconstruct the three-panel head-sharing diagram (with 8 blue query heads `Q` along the bottom row, red key heads `K` in the middle row, and yellow value heads `V` along the top row):

- **Multi-Head Attention (MHA)** (left): 8 `Q` heads, 8 `K` heads, 8 `V` heads, connected by 8 one-to-one dashed vertical lines. Every query head has its own private key and value cache.
- **Grouped-Query Attention (GQA)** (center, Ainslie et al., 2023): 8 `Q` heads, 4 `K` heads, and 4 `V` heads ($g = 4$ KV groups). Each pair of adjacent query heads connects to and shares one `K, V` head, immediately halving KV cache size and bandwidth (in practice, models often use $8:1$ or larger query-to-KV ratios).
- **Multi-Query Attention (MQA)** (right, Shazeer, 2019): 8 `Q` heads, **1** `K` head, and **1** `V` head. Dashed lines fan out from the single shared `K, V` head to all 8 query heads, shrinking the KV cache by a factor of h.

> **DEFINITION**
> **Multi-Query Attention (MQA) & Grouped-Query Attention (GQA)**: Attention variants that reduce KV-cache memory and decode bandwidth by sharing either a single key-value head across all h query heads (MQA) or g key-value heads across groups of h/g query heads (GQA).

Check what MQA does to the decode arithmetic intensity algebra. Because the KV cache now stores dimension $k = d/h$ per token instead of d, summing KV reads over n steps drops from $O(bn^2d)$ to $O(bn^2k)$, giving total memory $O(bnd + bn^2k + nd^2)$. Dividing ops $O(bnd^2)$ by that memory (using $k/d = 1/h$):

$$\text{AI}_{\text{MQA}} = \frac{bnd^2}{bnd + bn^2k + nd^2} = O\left(\left(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b}\right)^{-1}\right)$$

The destructive $\frac{n}{d}$ term is divided by the number of heads h!

> **WORKED EXAMPLE**
> Compare incremental decode arithmetic intensity for MHA versus MQA at $d = 4096$, $h = 32$ ($k = 128$), and batch size $b = 64$, first at $n = 4096$ and then at $n = 1{,}000{,}000$.
>
> 1. **Prefill baseline** ($n = 4096, b = 1$):
>
> $$\text{AI}_{\text{prefill}} \approx \left(\frac{1}{128} + \frac{1}{4096}\right)^{-1} = \left(\frac{33}{4096}\right)^{-1} \approx 124.1 \text{ FLOPs/element}$$

2. **MHA decode** ($n = 4096, b = 64$):

$$\text{AI}_{\text{MHA}} \approx \left(\frac{4096}{4096} + \frac{1}{64}\right)^{-1} = (1.0156)^{-1} \approx 0.98 \text{ FLOPs/element}$$

Even at batch size 64, arithmetic intensity collapsed by over $125\times$! At $n = 1{,}000{,}000$, $n/d \approx 244.1$, crashing MHA decode AI to ≈ 0.004.

3. **MQA decode** ($n = 4096, b = 64, h = 32$):

$$\text{AI}_{\text{MQA}} \approx \left(\frac{1}{4096} + \frac{4096}{4096 \times 32} + \frac{1}{64}\right)^{-1} = (0.00024 + 0.03125 + 0.01563)^{-1} \approx 21.$$

Interpretation: Sharing KV heads yields more than a $21\times$ **jump** in decode arithmetic intensity at $n = 4096$.

Does sharing KV heads ruin quality? **Shazeer (2019)** tested this at matched parameter counts (reinvesting saved KV projection parameters into a wider d_{ff}):

Shazeer (2019) Setup	h	d_k, d_v	d_{ff}	**Dev Perplexity**
Multi-head (MHA)	8	128	8,192	**29.9**
Multi-query (MQA)	8	128	9,088	**30.2**
Smaller MHA	1–8	128–16	9,984	30.9–31.2

MQA lost only 0.3 perplexity compared to full MHA while dramatically outperforming naively shrinking MHA head counts or head sizes (30.9–31.2). **GQA** (Ainslie et al., 2023) bridged the remaining tiny gap, matching full MHA-XXL quality while operating close to MQA latency—which made GQA the default across the Llama 2/3/4 era.

Beyond GQA: Multi-Head Latent Attention (MLA) and Extreme KV Compression

At 1M-token contexts, even GQA caches grow huge. To compress the KV cache far beyond GQA *without* sacrificing the expressivity of many distinct key and value heads, **DeepSeek-V2** introduced **Multi-Head Latent Attention (MLA)** (subsequently used in historical DeepSeek-V3 and Kimi K2, and extended in late-2026 frontier models).

Reconstruct the MLA block diagram from bottom to top:

- A token's hidden state x_t enters at the bottom and projects into **two compact cached vectors**:

1. A single low-rank **KV Latent vector** c_t^{KV} (cached in memory instead of separate full-width K and V tensors across all heads).
2. A small decoupled **RoPE Key** of dimension d_{rope} (cached alongside c_t^{KV}). Why decouple RoPE into its own small vector? Because RoPE applies a position-dependent rotation $R_{\Theta,t}$ that cannot be algebraically merged into static up-projection weight matrices, whereas the unrotated content projections *can* be absorbed into the query and output projections!

- Above the cache, two learned up-projection matrices expand c_t^{KV} into per-head content keys K^C and values V^C, which combine with the decoupled RoPE Key inside **Multi-Head Attention**.

> **DEFINITION**
> **Multi-Head Latent Attention (MLA)**: An attention architecture that compresses each token's keys and values into a single low-rank cached latent vector c_t^{KV} plus a small decoupled RoPE key, restoring full multi-head expressivity via learned up-projections while cutting KV-cache storage dramatically.

Late-2026 frontier models push KV-cache compression to extraordinary extremes:

- **GLM-5.3 (2026)**: Uses MLA with `kv_lora_rank = 512` and `rope_dim = 64` across 64 heads (78 layers, hidden size $6{,}144$).
- **Kimi K3 / K3.1 (July 2026)**: Combines **gated MLA** layers interleaved with Kimi Delta Attention.
- **DeepSeek-V4.1-Flash (Sept 2026)**: Shares **1 KV head across 64 query heads** (`head_dim = 512`) and combines **cross-layer KV reuse** with an **FP4 KV cache**, reducing global KV storage across the entire 552B model to just **890 bytes per token** at 1M context!

> **WORKED EXAMPLE**
> Compare total KV-cache memory per sequence at $n = 1{,}000{,}000$ tokens across three designs for a 78-layer, $d = 6144$, 64-head ($d_{\text{head}} = 96$) model in `bf16` (2 bytes per scalar) versus DeepSeek-V4.1-Flash.
>
> 1. **Standard MHA** (2 tensors K, V of width 6144 per layer, 2 bytes each):
>   - Per token per layer: $2 \times 6144 \times 2 = 24{,}576$ bytes.
>   - Across 78 layers: $78 \times 24{,}576 \approx 1.917$ MB/token.
>   - At 1M tokens: $\approx 1{,}917$ GB (~ 1.9 TB) for a single user sequence!
> 2. **GLM-5.3 MLA** (`kv_lora_rank = 512` , `rope_dim = 64` , 2 bytes each):
>   - Per token per layer: $(512 + 64) \times 2 = 1{,}152$ bytes (a $21.3\times$ **reduction** over MHA!).
>   - Across 78 layers: $78 \times 1{,}152 = 89{,}856$ bytes/token ≈ 89.9 KB/token.
>   - At 1M tokens: ≈ 89.9 GB.
> 3. **DeepSeek-V4.1-Flash** (1 shared KV head, cross-layer KV reuse + FP4 KV cache = **890 bytes/token** globally):

- At 1M tokens: 890×10^6 bytes $= 0.89$ GB—more than a $2{,}100\times$ **reduction** relative to standard MHA!

Interpretation: Without aggressive KV compression, 1M-context serving is physically infeasible; combining low-rank/shared KV heads, cross-layer reuse, and FP4 quantization shrinks a 2-terabyte cache down under a single gigabyte.

Sparse, Interleaved, and Hybrid Attention for 1M Context

Compressing each KV entry solves memory capacity and decode bandwidth, but even with small KV entries, dense attention still compares every query against all n keys—an $O(n^2)$ prefill cost and $O(n)$ per-step decode cost that becomes prohibitive at $n = 1{,}000{,}000$. Two complementary architectural strategies break that quadratic wall, illustrated in the **Sparse, Interleaved, and Hybrid Attention** diagram:

1. **3:1 Interleaved Block Pattern** (top half of the diagram): Instead of running expensive global attention at every layer, stack **three fast local or linear-complexity layers** (`Layers 1–3: Local SWA or Linear / DeltaNet`) for every **one global layer** (`Layer 4: Full / Sparse Global Attn`). Local **Sliding-Window Attention (SWA)** restricts attention to a fixed window of recent tokens (bounding both compute and KV cache to $O(1)$ per token), while **linear-attention / state-space layers** (such as **Gated DeltaNet**, **Kimi Delta Attention**, or **Mamba**) maintain a fixed-size recurrent state rather than a growing KV cache.
2. **Dynamic Top-k Sparse Attention** (bottom half of the diagram): Compare the two causal attention-matrix sparsity squares side by side:
   - **Dense $O(n^2)$** (left square): The entire lower-left triangle is shaded—every token attends to every preceding token.
   - **Sparse / Top-k** (right square): Only a narrow dark-blue diagonal band (recent local tokens) plus a few scattered dark-blue blocks further back in the triangle are active; all remaining entries are skipped. Unlike early fixed sparse patterns (Child et al., 2019; GPT-3's alternating dense and locally banded sparse layers), modern sparse attention selects *content-dependent* top-k blocks or tokens dynamically for each query.

DEFINITION

Interleaved Hybrid Attention: A multi-layer architecture that alternates several constant-state or fixed-window layers (sliding-window attention, DeltaNet, or Mamba) with occasional global full or sparse attention layers (commonly in a 3:1 ratio), capturing long-range retrieval without paying $O(n^2)$ attention cost at every layer.

> **COMMON MISCONCEPTION**
> You might think that interleaving 3 local or linear layers with only 1 global attention layer prevents a model from retrieving facts from 500,000 tokens ago. That breaks because global information routing does not require all 78 layers to attend over all 1M tokens simultaneously: local/linear layers handle local syntax and running state in $O(1)$ cache memory, while the periodic global (or top-k sparse global) layers perform long-range retrieval across the entire context.

Trace how rapidly this pattern moved from the 2025–early-2026 wave into 1M-context late-2026 frontier designs:

- **2025–early-2026 wave (interleaved designs)**:
  - **Command A**: 3:1 Sliding-Window Attention (with RoPE) to Full Attention (with NoPE).
  - **OLMo 3**: 3/4 Sliding-Window Attention (window size $4{,}096$) plus Full Attention with YaRN scaling; Llama 4 and Gemma 3/4 likewise interleave local and global layers.
  - **Qwen 3.5**: 3:1 **Gated DeltaNet** (linear attention) to **Gated Attention**.
- **Late-2026 frontier (1M-context sparse and hybrid designs)**:
  - **MiniMax-M3** (June 2026): **MiniMax Sparse Attention**—blockwise top-k sparse attention built on GQA—matches dense GQA quality on a 109B model while cutting per-token attention compute by $28.4\times$ **at 1M context**.
  - **DeepSeek-V4** (2026) and **DeepSeek-V4.1-Flash** (Sept 2026): Hybrid of **CSA / CSA2** (Compressed Sparse Attention) and **HCA** (Heavily Compressed Attention).
  - **GLM-5.3** (2026): **DSA** (DeepSeek Sparse Attention) with **top-$k = 2048$** on top of MLA.
  - **Qwen3.8-Flash-Next** (Aug 2026): 3:1 **Gated DeltaNet** to full-attention layers, with the full layers replaced by **Qwen Sparse Attention (QSA)** during continued pretraining.
  - **Kimi K3 / K3.1** (July 2026): 3:1 **Kimi Delta Attention** to **gated MLA**.
  - **Nemotron 3.5 Super / Lightning** (Oct 2026): Hybrid **Mamba-Transformer MoE**.

> **KEY IDEA**
> Incremental decode is memory-bound because reloading the KV cache adds an n/d term that collapses arithmetic intensity as context grows; modern 1M-context models overcome this wall by combining KV compression (GQA, MLA, shared/FP4 cross-layer KV) with dynamic top-k sparse attention and 3:1 interleaved linear/local hybrids.

Putting the Whole Architecture Together (Late 2026 Synthesis)

Standing back across all nine sections, the architecture of a late-2026 frontier language model reads as one unified engineering response to four physical constraints—gradient flow, data movement, numerical stability, and long-context memory bandwidth:

- **Dense block consensus**: Normalization strictly off the residual highway (Pre-RMSNorm or non-residual post-norm), no bias terms, gated `*GLU` feed-forward modules, serial attention-then-FFN

ordering, and RoPE / partial RoPE (or RoPE + NoPE hybrids).

- **Hyperparameters and regularization**: $d_{\text{ff}} \approx \frac{8}{3}d$ (or 3.5–$4d$ for dense GLU) and aspect ratio $d_{\text{model}}/n_{\text{layer}} \sim 100$–$200$ sit in flat loss basins guided by systems parallelism; multilingual/code vocabularies span 100k–262k; dropout is set to 0 while weight decay (~ 0.1) improves cosine-schedule optimization dynamics.
- **Stability at scale**: Output z-loss, QK-Norm, and MLA-compatible QK-Clip prevent softmax explosions, while late-2026 models add Gated Attention, GatedNorm, and Muon-family optimizers to train spike-free even under aggressive learning rates.
- **Frontier scaling shifts**: Ultra-sparse MoE (256–896 experts with shared experts and LatentMoE) plus MTP layers, multi-stream or depth-attended residuals (mHC, Attention Residuals), and compressed KV + sparse/linear hybrid attention unlock multi-trillion-parameter capacity and 1M-token contexts.

✎ Check Your Understanding

1. **What would happen to** incremental decode arithmetic intensity $\text{AI} = O\big((\frac{n}{d} + \frac{1}{b})^{-1}\big)$ under standard MHA if you doubled the batch size b when $n = 100\,d$?
   Answer: Almost nothing. When $n/d = 100$, the n/d KV-cache term dominates the $1/b$ weight-reload term, so increasing batch size cannot rescue arithmetic intensity without shrinking the KV cache itself.
2. **Quick computation:** In a 64-head Transformer, switching from standard MHA to MQA reduces the KV-cache term in the decode arithmetic-intensity denominator from $\frac{n}{d}$ to $\frac{n}{dh}$. By what factor does that shrink the KV-cache term?
   Answer: By a factor of $h = 64$.