Machine Learning and Optimization

Lecture 19: Why Modern LLMs Look the Way They Do

CSCI 4140/6960 — Spring 2026

Course website: `cs.rpi.edu/.../mlandopt.html`

Learning Objectives

By the end of this lecture, you should be able to:

1. **Articulate** why decoder-only transformers dominate modern LLMs (generative modeling argument + scaling simplicity).
2. **Identify** the four bottlenecks that appear when scaling a vanilla decoder-only transformer, and state which resource (memory, compute, stability) each constrains.
3. **Explain** how pre-norm, RMSNorm, and SwiGLU together stabilize training at 80+ layers, connecting to variance arguments from the training deep networks lecture.
4. **Describe** why large context windows are important and how sinusoidal PE fails to provide them.
5. **Explain** RoPE: how rotation encodes relative position in the attention dot product, and why this enables context extension.

Learning Objectives (continued)

6. **Explain** the MHA $\rightarrow$ GQA progression as a practical solution to the KV cache bottleneck (MLA in Appendix B).
7. **Describe** the dense compute bottleneck and how MoE addresses it by decoupling capacity from per-token cost (see Appendix C).
8. **Read** a modern model card (Llama 4, DeepSeek V3) and map every architectural choice to the bottleneck it addresses.
9. **Describe** what the model *is* at the end of pretraining — a density estimator with no alignment — setting up the next lecture.

Scaling law derivations: see Appendix E.

Lecture Roadmap

1. Why Decoder-Only?
2. The Four Bottlenecks (overview)
3. B1: Training Stability $\rightarrow$ Pre-Norm + RMSNorm + SwiGLU
4. B2: Position Extrapolation $\rightarrow$ RoPE
5. B3: KV Cache $\rightarrow$ GQA (MLA in appendix)
6. B4: Dense Compute $\rightarrow$ Mixture of Experts (in appendix)
7. The Modern LLM Template
8. Handoff to Next Lecture

Scaling law derivations: see Appendix E.

The Question

Vanilla decoder-only transformer LLMs

Multiple decoder-only transformer blocks. Causal masking. Cross-entropy objective. Train with teacher forcing. Done.

The Question

Vanilla decoder-only transformer LLMs

Multiple decoder-only transformer blocks. Causal masking. Cross-entropy objective. Train with teacher forcing. Done.

But ChatGPT, Claude, and Gemini are *not* the architecture from that lecture. Every component has been replaced or modified.

The Question

Vanilla decoder-only transformer LLMs

Multiple decoder-only transformer blocks. Causal masking. Cross-entropy objective. Train with teacher forcing. Done.

But ChatGPT, Claude, and Gemini are *not* the architecture from that lecture. Every component has been replaced or modified.

Today's question: What changed and why?

Each modification exists because something **broke when these models were scaled up**.

The Question

Vanilla decoder-only transformer LLMs

Multiple decoder-only transformer blocks. Causal masking. Cross-entropy objective. Train with teacher forcing. Done.

But ChatGPT, Claude, and Gemini are *not* the architecture from that lecture. Every component has been replaced or modified.

Today's question: What changed and why?

Each modification exists because something **broke when these models were scaled up**.

What does "scale" mean here? Three things grow simultaneously:

- **Parameters** N: from millions to hundreds of billions.
- **Training data** D: from gigabytes to tens of terabytes of text.
- **Compute** C: from GPU-days to millions of GPU-hours.

At each jump, a new failure mode becomes the binding constraint.

The Question

Vanilla decoder-only transformer LLMs

Multiple decoder-only transformer blocks. Causal masking. Cross-entropy objective. Train with teacher forcing. Done.

But ChatGPT, Claude, and Gemini are *not* the architecture from that lecture. Every component has been replaced or modified.

Today's question: What changed and why?

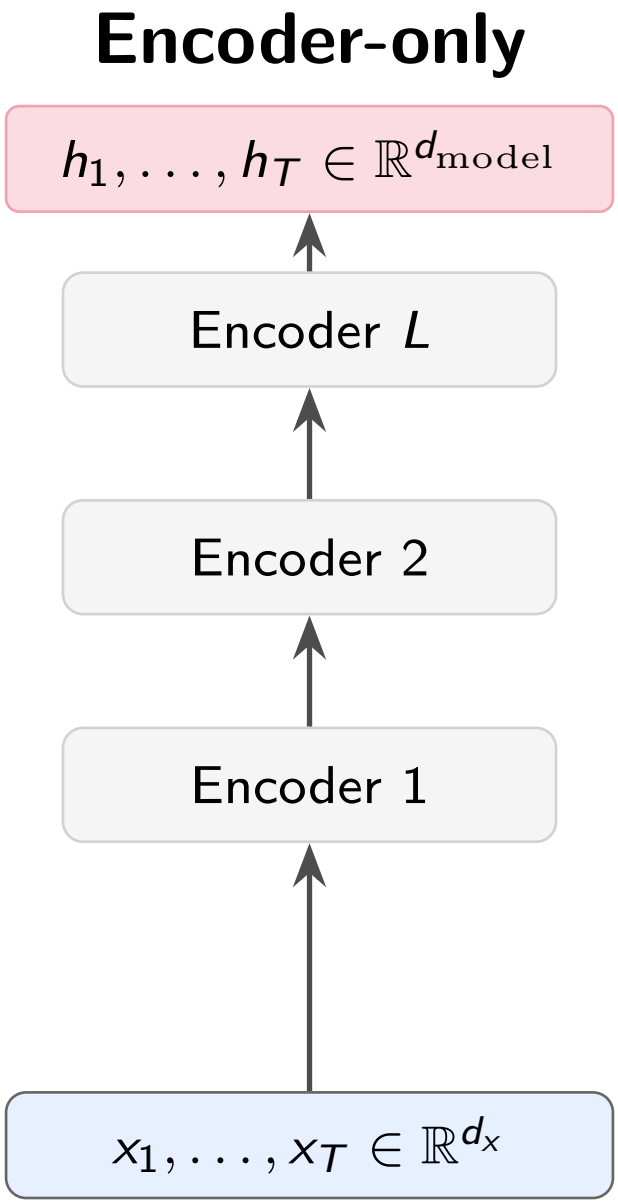
Each modification exists because something **broke when these models were scaled up**.

What does "scale" mean here? Three things grow simultaneously:

- **Parameters** N: from millions to hundreds of billions.
- **Training data** D: from gigabytes to tens of terabytes of text.
- **Compute** C: from GPU-days to millions of GPU-hours.

At each jump, a new failure mode becomes the binding constraint.
Organizing principle: Identify bottleneck $\rightarrow$ Engineer fix.

Architecture 1: Encoder-Only [Devlin et al.2019]



Input: Full sequence (bidirectional self-attention).

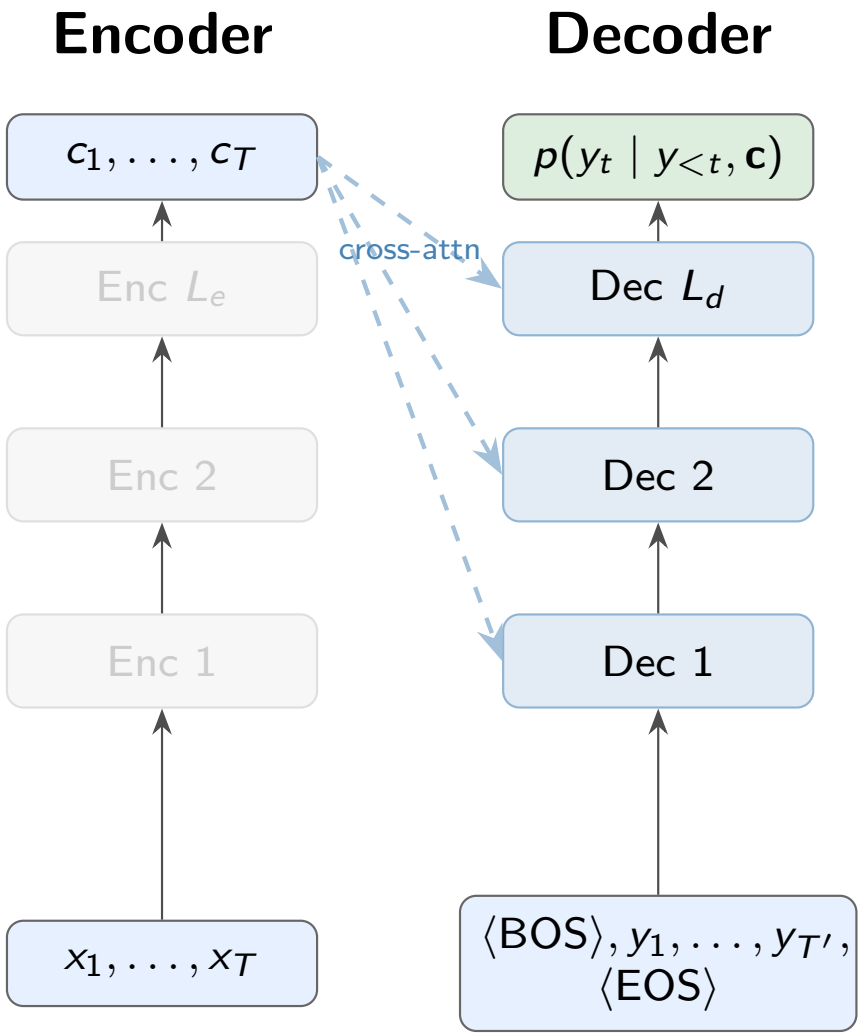
Objective: Masked Language Modeling (MLM) — predict randomly masked tokens from surrounding context.

Output: Contextual representations $h_i \in \mathbb{R}^{d_{\text{model}}}$ per token.

Use: Classification, NER, sentence similarity.

Cannot generate text. MLM learns to fill blanks, not to model the joint distribution $p(x_1, \ldots, x_T)$. You cannot sample new sequences from an encoder-only model.

Architecture 2: Encoder-Decoder [Raffel et al.2020]



Input: Source → encoder; target → decoder.

Objective: Seq-to-seq (e.g. translate, summarize).

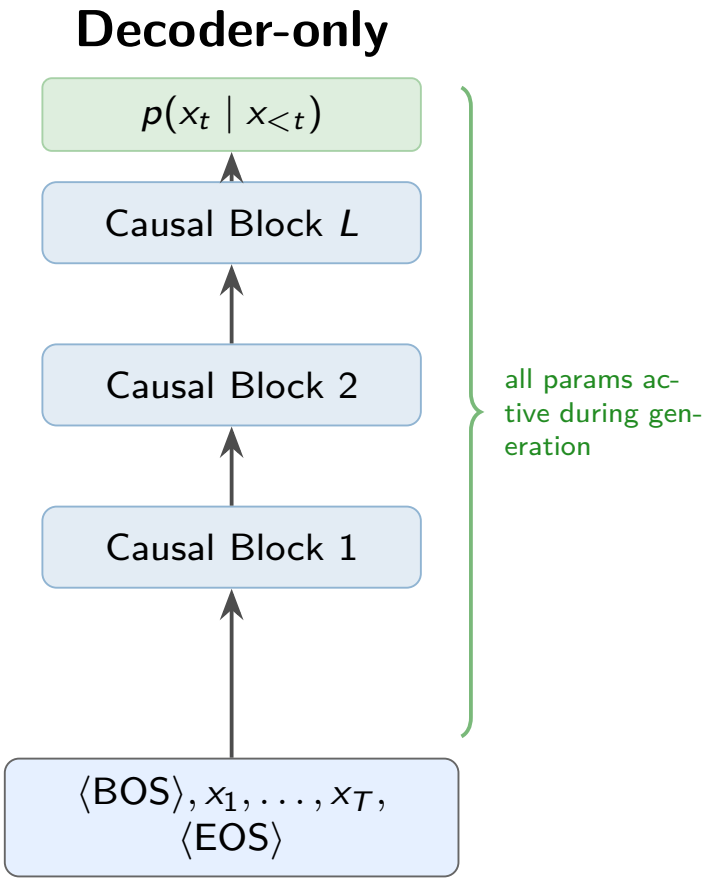
Output: Autoregressive generation from decoder.

Can generate text — but the encoder is **idle during generation**. Two separate stacks means more parameters per unit of useful compute.

Requires explicit source/target framing of every task.

Diagram reflects training: target sequence is provided via teacher forcing.

Architecture 3: Decoder-Only [Radford et al.2018, 2019, Brown et al.2020]



Input: Single token sequence (causal mask: each token attends only to previous tokens).

Objective: Next-token prediction: $p(x_1, \ldots, x_T) = \prod_{i=1}^{T} p(x_i \mid x_{<i})$.

Output: Full generative model of text.

Fully generative. One stack — every parameter is active during both training and generation.

Task-general: any task can be cast as text completion via prompt formatting.

Diagram reflects training: input sequence is provided via teacher forcing.

What Makes an Architecture Scale?

Before comparing the three architectures, we need to define what "scales well" means. Three desiderata:

1. Generative. The model must be able to *sample new text* — i.e., model the joint distribution $p(x_1, \ldots, x_T)$. This is required for open-ended tasks like dialogue, code generation, and creative writing. An architecture that only produces representations (not probability distributions over sequences) cannot do this.

2. Scale-efficient. Every parameter and every FLOP should contribute to the task at hand. If large parts of the model sit idle during the most expensive phase (generation), you are paying for capacity you cannot use.

3. Task-general. The architecture should handle *any* NLP task without structural changes — translation, QA, summarization, reasoning — via prompt formatting alone, not via task-specific encoder/decoder splits.

Why Decoder-Only Wins

	Generative?	**Scale-efficient?**	**Task-general?**
Encoder-only	**No** — fills blanks, cannot sample	N/A	**No** — representations only
Enc-Dec	Yes	**Partial** — encoder idle during generation	**Partial** — needs source/target split
Decoder-only	Yes — $\prod_i p(x_i \mid x_{<i})$	Yes — one stack, all tokens in loss	Yes — prompt formatting

Why Decoder-Only Wins

	Generative?	**Scale-efficient?**	**Task-general?**
Encoder-only	**No** — fills blanks, cannot sample	N/A	**No** — representations only
Enc-Dec	Yes	**Partial** — encoder idle during generation	**Partial** — needs source/target split
Decoder-only	Yes — $\prod_i p(x_i \mid x_{<i})$	Yes — one stack, all tokens in loss	Yes — prompt formatting

Key claim [Brown et al.2020]

Decoder-only is the *minimal scalable architecture* for autoregressive generation. It won because simplicity scales.

Why Decoder-Only Wins

	Generative?	**Scale-efficient?**	**Task-general?**
Encoder-only	**No** — fills blanks, cannot sample	N/A	**No** — representations only
Enc-Dec	Yes	**Partial** — encoder idle during generation	**Partial** — needs source/target split
Decoder-only	Yes — $\prod_i p(x_i \mid x_{<i})$	Yes — one stack, all tokens in loss	Yes — prompt formatting

Key claim [Brown et al.2020]

Decoder-only is the *minimal scalable architecture* for autoregressive generation. It won because simplicity scales.

The scaling laws tell us *how much* to scale. The next question: *what breaks* when we do?

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	Llama 3 (70B)	Llama 4 Maverick (17B act./400B tot.)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	Llama 3 (70B)	Llama 4 Maverick (17B act./400B tot.)
Training tokens	15T	∼22T

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	∼22T
Interpretable scale	∼11T words ∼44M books	∼16T words ∼65M books

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	∼22T
Interpretable scale	∼11T words ∼44M books	∼16T words ∼65M books
GPU count	16,384 H100 GPUs	(not disclosed)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	∼22T
Interpretable scale	∼11T words ∼44M books	∼16T words ∼65M books
GPU count	16,384 H100 GPUs	(not disclosed)
GPU-hours	∼6.4M H100-hours	(est. >10M H100-hours)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	$\sim$22T
Interpretable scale	$\sim$11T words $\sim$44M books	$\sim$16T words $\sim$65M books
GPU count	16,384 H100 GPUs	(not disclosed)
GPU-hours	$\sim$6.4M H100-hours	(est. $>$10M H100-hours)
Compute (FLOPs)	$\sim 6.3 \times 10^{24}$	$\sim 5.3 \times 10^{25}$

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	$\sim$22T
Interpretable scale	$\sim$11T words $\sim$44M books	$\sim$16T words $\sim$65M books
GPU count	16,384 H100 GPUs	(not disclosed)
GPU-hours	$\sim$6.4M H100-hours	(est. $>$10M H100-hours)
Compute (FLOPs)	$\sim 6.3 \times 10^{24}$	$\sim 5.3 \times 10^{25}$
Electricity	$\sim$8.3 GWh ($\sim$770 US homes/yr)	(est. $>$15 GWh) ($\sim$1,400+ homes/yr)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	∼22T
Interpretable scale	∼11T words ∼44M books	∼16T words ∼65M books
GPU count	16,384 H100 GPUs	(not disclosed)
GPU-hours	∼6.4M H100-hours	(est. >10M H100-hours)
Compute (FLOPs)	$\sim 6.3 \times 10^{24}$	$\sim 5.3 \times 10^{25}$
Electricity	∼8.3 GWh (∼770 US homes/yr)	(est. >15 GWh) (∼1,400+ homes/yr)
Carbon footprint	∼2,300 t CO_2 (∼500 cars/yr)	(est. >4,000 t) (∼870+ cars/yr)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

The Scale of Modern LLM Training

To appreciate why architectural efficiency matters, consider the resources consumed by a single pre-training run:

Resource	**Llama 3 (70B)**	**Llama 4 Maverick (17B act./400B tot.)**
Training tokens	15T	∼22T
Interpretable scale	∼11T words ∼44M books	∼16T words ∼65M books
GPU count	16,384 H100 GPUs	(not disclosed)
GPU-hours	∼6.4M H100-hours	(est. >10M H100-hours)
Compute (FLOPs)	$\sim 6.3 \times 10^{24}$	$\sim 5.3 \times 10^{25}$
Electricity	∼8.3 GWh (∼770 US homes/yr)	(est. >15 GWh) (∼1,400+ homes/yr)
Carbon footprint	∼2,300 t CO_2 (∼500 cars/yr)	(est. >4,000 t) (∼870+ cars/yr)
Dollar cost	(est. $50–100M)	(est. $100–200M+)

Sources: Dubey et al.2024], Meta AI2025], Patterson et al.2021].

Scaling Forces Specific Failures

Scaling laws (Appendix K) tell us that training loss decreases as $A/N^{0.34} + B/D^{0.28} + L_\infty$, so we want both N and D large. When we scale, four specific failure modes emerge:

Bottleneck	**Scaling dimension that makes it binding**

Scaling Forces Specific Failures

Scaling laws (Appendix K) tell us that training loss decreases as $A/N^{0.34} + B/D^{0.28} + L_\infty$, so we want both N and D large. When we scale, four specific failure modes emerge:

Bottleneck	**Scaling dimension that makes it binding**
B1: Training instability	Depth L: gradient pathologies compound across 80+ layers.

Resource classification: B1 is an **optimization stability** problem; B2, B3 are **memory** problems; B4 is a **compute** problem.

Scaling Forces Specific Failures

Scaling laws (Appendix K) tell us that training loss decreases as $A/N^{0.34} + B/D^{0.28} + L_\infty$, so we want both N and D large. When we scale, four specific failure modes emerge:

Bottleneck	**Scaling dimension that makes it binding**
B1: Training instability	Depth L: gradient pathologies compound across 80+ layers.
B2: Position extrapolation	Context length T: users demand 128K+, but pre-training at such lengths is prohibitively expensive.

Resource classification: B1 is an **optimization stability** problem; B2, B3 are **memory** problems; B4 is a **compute** problem.

Scaling Forces Specific Failures

Scaling laws (Appendix K) tell us that training loss decreases as $A/N^{0.34} + B/D^{0.28} + L_\infty$, so we want both N and D large. When we scale, four specific failure modes emerge:

Bottleneck	**Scaling dimension that makes it binding**
B1: Training instability	Depth L: gradient pathologies compound across 80+ layers.
B2: Position extrapolation	Context length T: users demand 128K+, but pre-training at such lengths is prohibitively expensive.
B3: KV cache explosion	Context $\times$ depth $\times$ heads: at 128K context, per-user cache exceeds model weights.

Resource classification: B1 is an **optimization stability** problem; B2, B3 are **memory** problems; B4 is a **compute** problem.

Scaling Forces Specific Failures

Scaling laws (Appendix K) tell us that training loss decreases as $A/N^{0.34} + B/D^{0.28} + L_\infty$, so we want both N and D large. When we scale, four specific failure modes emerge:

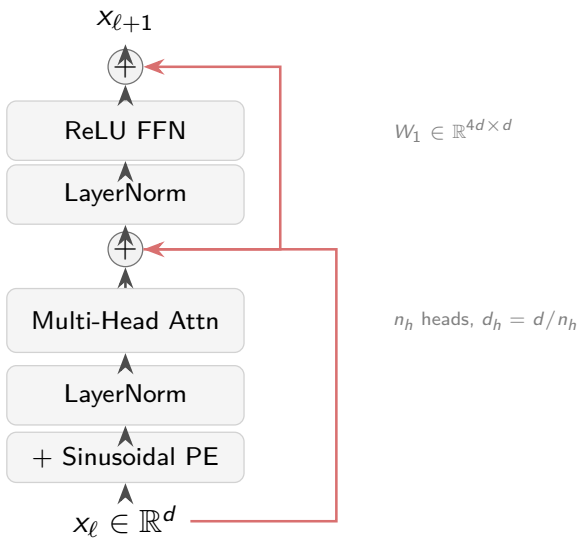
Bottleneck	**Scaling dimension that makes it binding**
B1: Training instability	Depth L: gradient pathologies compound across 80+ layers.
B2: Position extrapolation	Context length T: users demand 128K+, but pre-training at such lengths is prohibitively expensive.
B3: KV cache explosion	Context $\times$ depth $\times$ heads: at 128K context, per-user cache exceeds model weights.
B4: Dense compute	Parameter count N: $\sim$2/3 of params are FFN; per-token cost is $O(N)$ regardless of what the token needs.

Resource classification: B1 is an **optimization stability** problem; B2, B3 are **memory** problems; B4 is a **compute** problem.

The Modern LLM Block — Preview

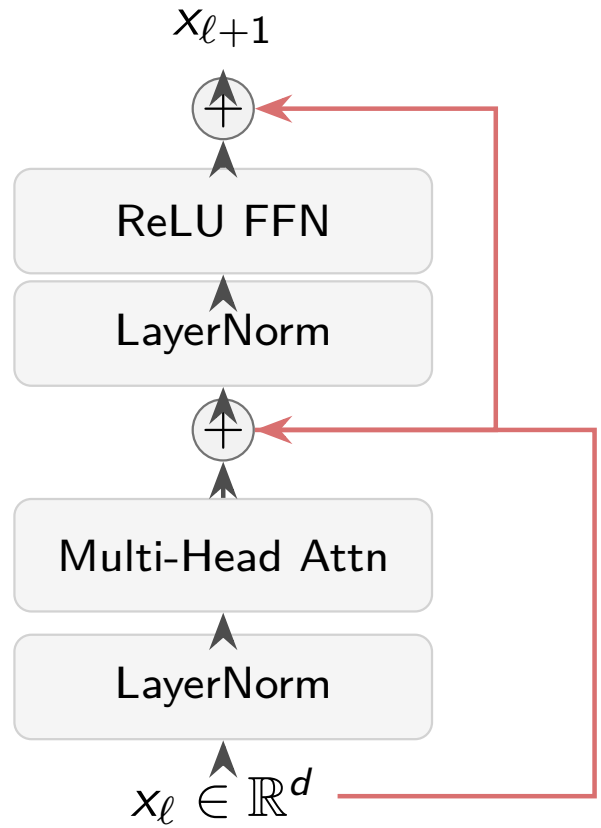
Vanilla (Lecture 18):

Standard decoder block



Modern (2024–2026):

Components change as we address each bottleneck.



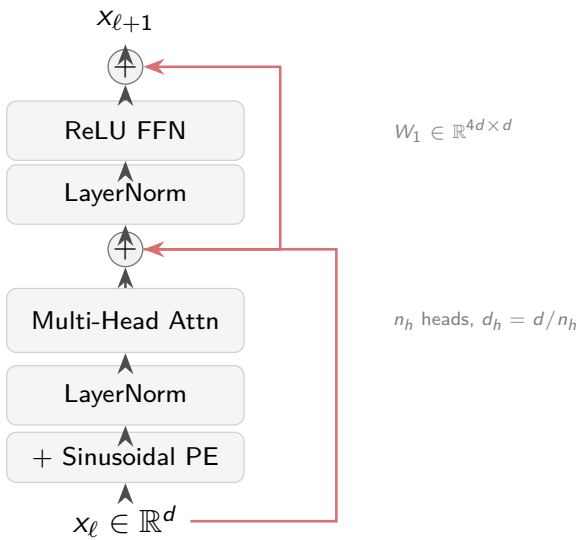
We address the bottlenecks in order:

Starting point: identical to vanilla.

The Modern LLM Block — Preview

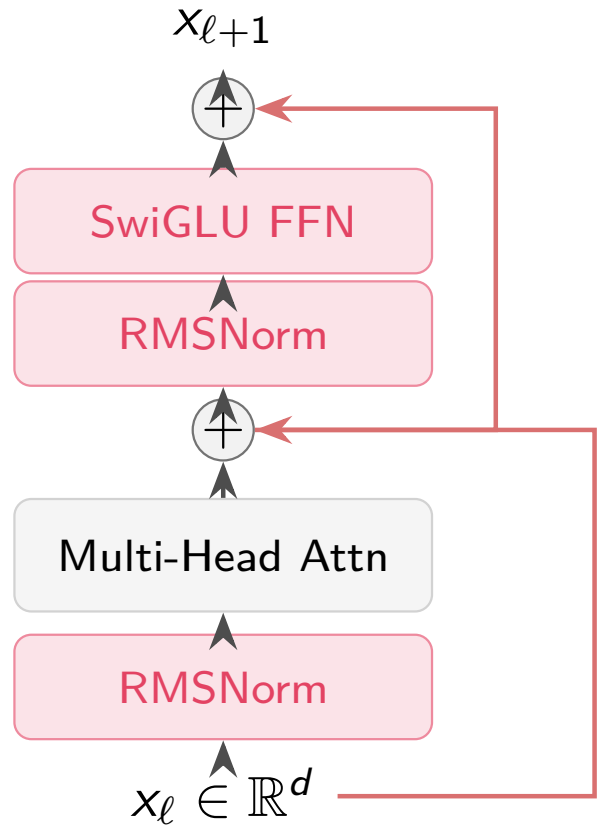
Vanilla (Lecture 18):

Standard decoder block



Modern (2024–2026):

Components change as we address each bottleneck.



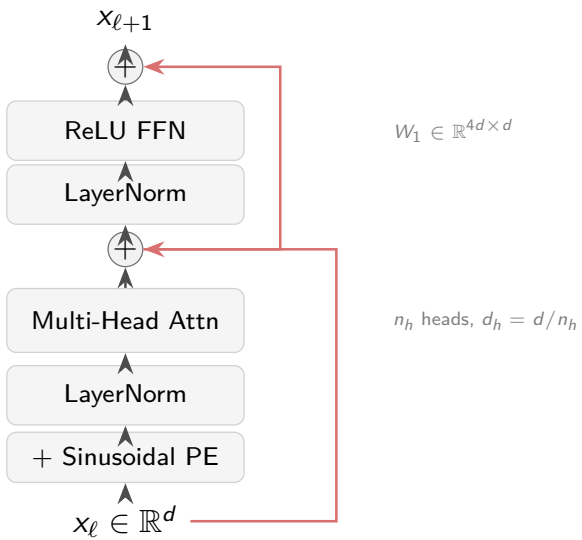
We address the bottlenecks in order:

B1 (training stability): LayerNorm → RMSNorm, ReLU FFN → SwiGLU, post-norm → pre-norm.

The Modern LLM Block — Preview

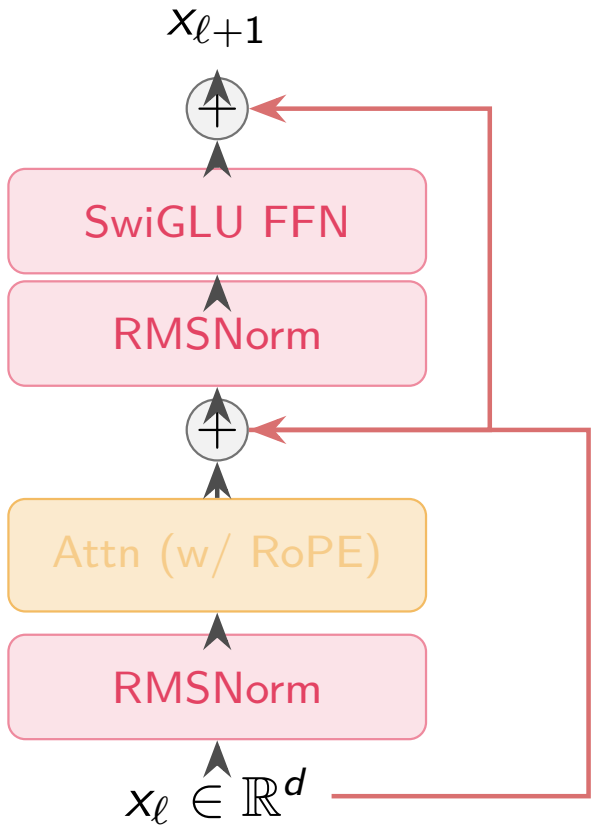
Vanilla (Lecture 18):

Standard decoder block



Modern (2024–2026):

Components change as we address each bottleneck.



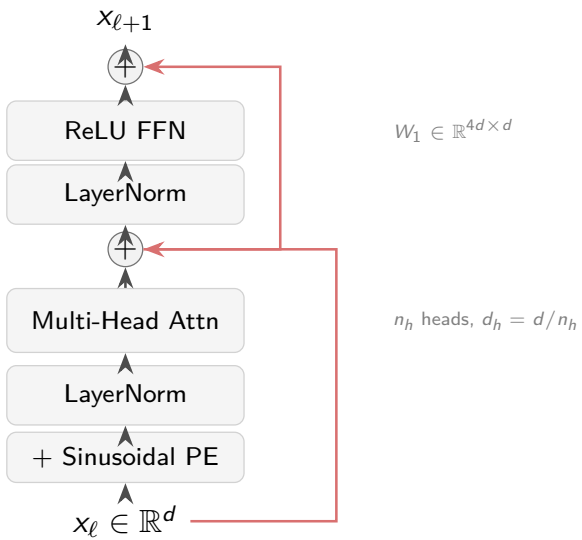
We address the bottlenecks in order:

B2 (position encoding): Multi-Head Attn → Attn (w/ RoPE).

The Modern LLM Block — Preview

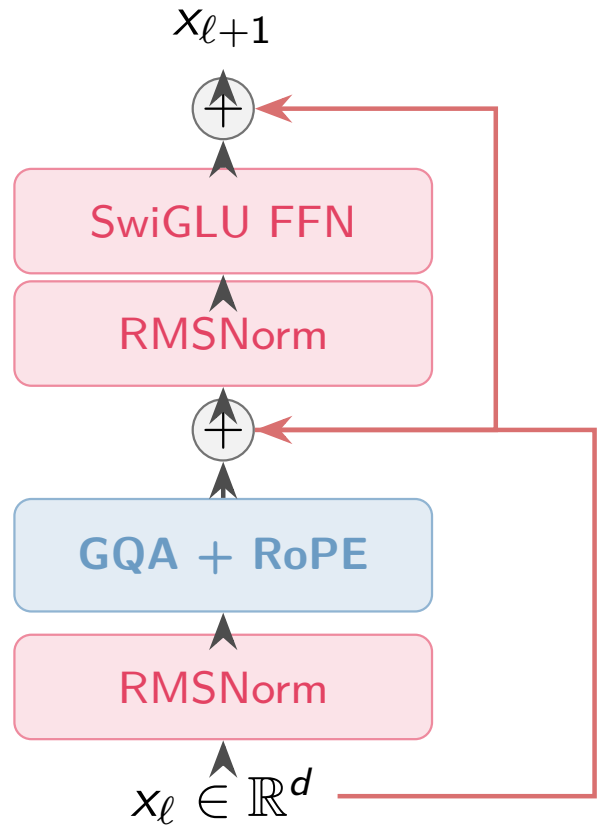
Vanilla (Lecture 18):

Standard decoder block



Modern (2024–2026):

Components change as we address each bottleneck.



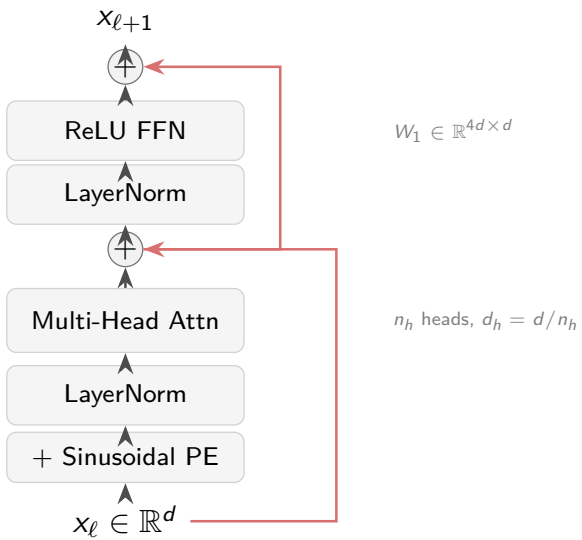
We address the bottlenecks in order:

B3 (KV cache): Multi-Head Attn → GQA/MLA.

The Modern LLM Block — Preview

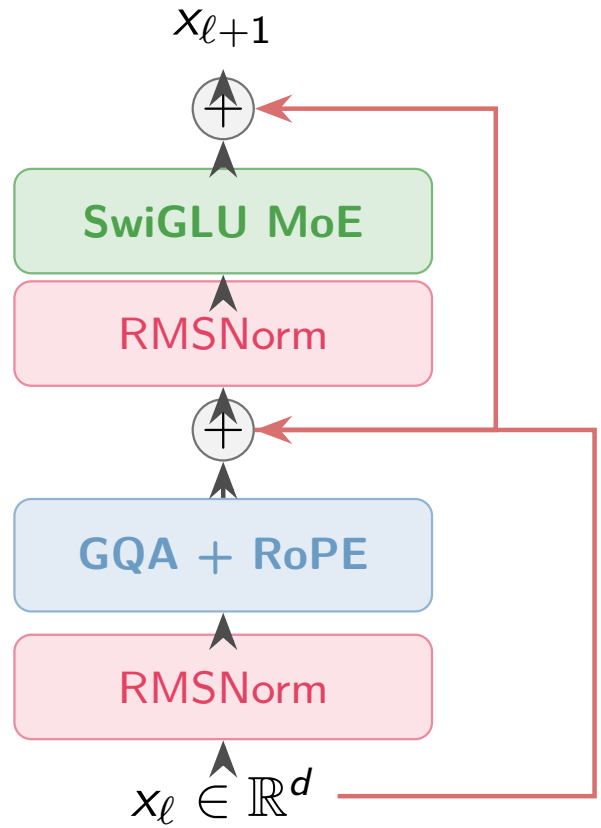
Vanilla (Lecture 18):

Standard decoder block



Modern (2024–2026):

Components change as we address each bottleneck.



We address the bottlenecks in order:

B4 (dense compute): Dense FFN → SwiGLU MoE.

The First Bottleneck — Training Instability at Depth

Bottleneck: At 80+ layers (Llama 3 70B, DeepSeek V3), the Jacobian $\partial x_L / \partial x_\ell$ compounds layer-by-layer. Gradient pathologies (exploding/vanishing gradients, loss spikes) make optimization stability the binding constraint.

The First Bottleneck — Training Instability at Depth

Bottleneck: At 80+ layers (Llama 3 70B, DeepSeek V3), the Jacobian $\partial x_L / \partial x_\ell$ compounds layer-by-layer. Gradient pathologies (exploding/vanishing gradients, loss spikes) make optimization stability the binding constraint.

Core failure: Post-norm residual blocks force the gradient through LayerNorm at every layer; ReLU FFNs create dead neurons. Neither forward propagation (activation scale) nor backward flow (Jacobian conditioning) is controlled.

The First Bottleneck — Training Instability at Depth

Bottleneck: At 80+ layers (Llama 3 70B, DeepSeek V3), the Jacobian $\partial x_L / \partial x_\ell$ compounds layer-by-layer. Gradient pathologies (exploding/vanishing gradients, loss spikes) make optimization stability the binding constraint.

Core failure: Post-norm residual blocks force the gradient through LayerNorm at every layer; ReLU FFNs create dead neurons. Neither forward propagation (activation scale) nor backward flow (Jacobian conditioning) is controlled.

Design goal: Ensure stable forward propagation (controlled activation scale) and clean backward gradient flow (near-identity layer-to-layer Jacobians $I + J_k$).

The First Bottleneck — Training Instability at Depth

Bottleneck: At 80+ layers (Llama 3 70B, DeepSeek V3), the Jacobian $\partial x_L / \partial x_\ell$ compounds layer-by-layer. Gradient pathologies (exploding/vanishing gradients, loss spikes) make optimization stability the binding constraint.

Core failure: Post-norm residual blocks force the gradient through LayerNorm at every layer; ReLU FFNs create dead neurons. Neither forward propagation (activation scale) nor backward flow (Jacobian conditioning) is controlled.

Design goal: Ensure stable forward propagation (controlled activation scale) and clean backward gradient flow (near-identity layer-to-layer Jacobians $I + J_k$).

Roadmap: Post-norm pathology $\rightarrow$ pre-norm (residual highway) $\rightarrow$ RMSNorm (cheaper, no mean drift) $\rightarrow$ SwiGLU (gated activation, no dead neurons) $\rightarrow$ progress check.

This is infrastructure: every other fix (RoPE, GQA, MoE) assumes stable training at depth. B1 enables everything that follows.

B1: Pre-Norm vs. Post-Norm — Gradient Analysis

Post-norm (original transformer):

$$\text{out} = \text{Norm}(x + \text{SubLayer}(x))$$

Gradient must pass through normalization at every layer.

Pre-norm (all modern LLMs):

$$\text{out} = x + \text{SubLayer}(\text{Norm}(x))$$

Residual is unimpeded. Gradient flows directly through x.

B1: Pre-Norm vs. Post-Norm — Gradient Analysis

Post-norm (original transformer):

$$\text{out} = \text{Norm}(x + \text{SubLayer}(x))$$

Gradient must pass through normalization at every layer.

Pre-norm (all modern LLMs):

$$\text{out} = x + \text{SubLayer}(\text{Norm}(x))$$

Residual is unimpeded. Gradient flows directly through x.

Layerwise Jacobian (pre-norm). Let $f_k(x) = \text{SubLayer}_k(\text{Norm}(x))$ and $J_k = \frac{\partial f_k}{\partial x}\big|_{x=x_k}$. Each pre-norm residual update gives:

$$x_{k+1} = x_k + f_k(x_k) \implies \frac{\partial x_{k+1}}{\partial x_k} = I + J_k$$

B1: Pre-Norm vs. Post-Norm — Gradient Analysis

Post-norm (original transformer):

$$\text{out} = \text{Norm}(x + \text{SubLayer}(x))$$

Gradient must pass through normalization at every layer.

Pre-norm (all modern LLMs):

$$\text{out} = x + \text{SubLayer}(\text{Norm}(x))$$

Residual is unimpeded. Gradient flows directly through x.

Layerwise Jacobian (pre-norm). Let $f_k(x) = \text{SubLayer}_k(\text{Norm}(x))$ and $J_k = \frac{\partial f_k}{\partial x}\big|_{x=x_k}$. Each pre-norm residual update gives:

$$x_{k+1} = x_k + f_k(x_k) \implies \frac{\partial x_{k+1}}{\partial x_k} = I + J_k$$

The end-to-end Jacobian from layer ℓ to layer L is:

$$\frac{\partial x_L}{\partial x_\ell} = \prod_{k=\ell}^{L-1} (I + J_k)$$

B1: Pre-Norm vs. Post-Norm — Gradient Analysis

Post-norm (original transformer):

$$\text{out} = \text{Norm}(x + \text{SubLayer}(x))$$

Gradient must pass through normalization at every layer.

Pre-norm (all modern LLMs):

$$\text{out} = x + \text{SubLayer}(\text{Norm}(x))$$

Residual is unimpeded. Gradient flows directly through x.

Layerwise Jacobian (pre-norm). Let $f_k(x) = \text{SubLayer}_k(\text{Norm}(x))$ and $J_k = \frac{\partial f_k}{\partial x}\big|_{x=x_k}$. Each pre-norm residual update gives:

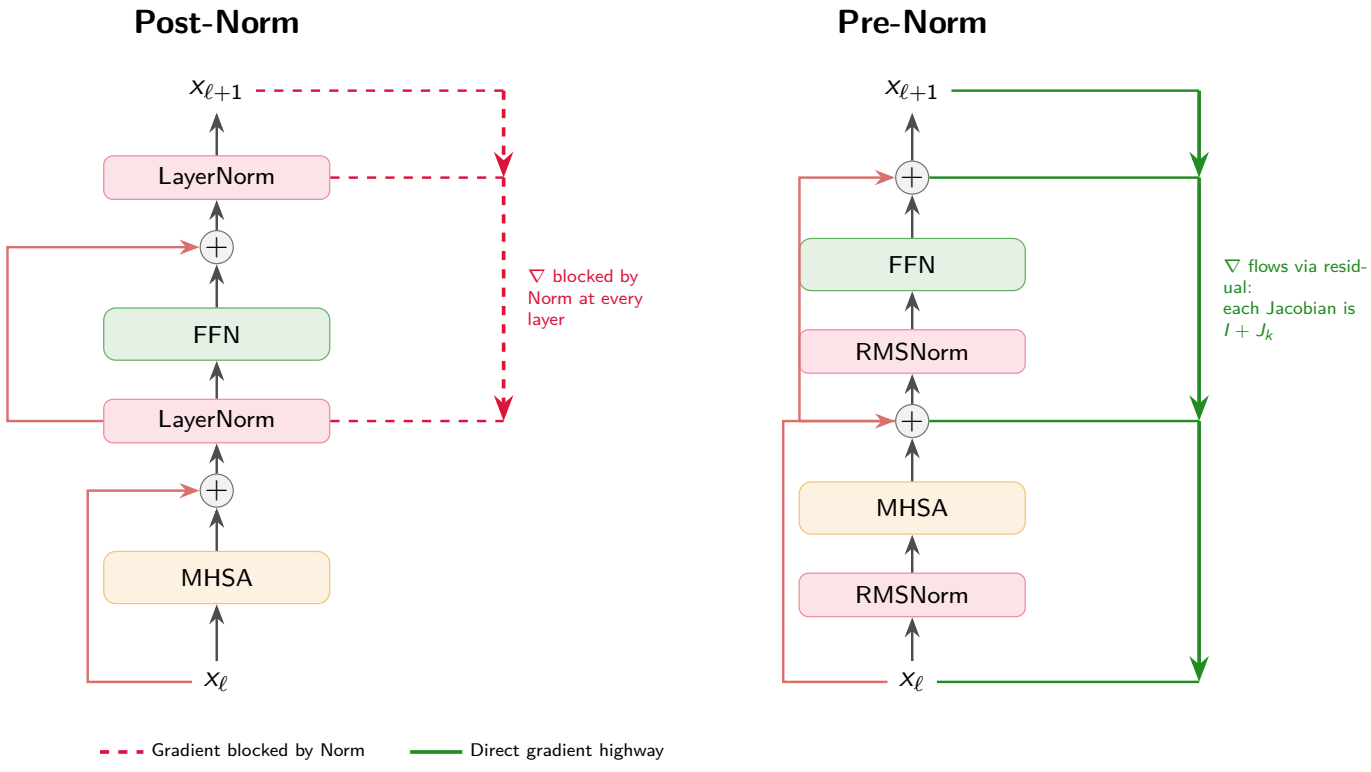
$$x_{k+1} = x_k + f_k(x_k) \implies \frac{\partial x_{k+1}}{\partial x_k} = I + J_k$$

The end-to-end Jacobian from layer ℓ to layer L is:

$$\frac{\partial x_L}{\partial x_\ell} = \prod_{k=\ell}^{L-1} (I + J_k)$$

Key insight: *Every* layer-to-layer Jacobian $I + J_k$ is a near-identity perturbation (when $\|J_k\|$ is small). This is the layerwise analogue

B1: Pre-Norm vs. Post-Norm Gradient Path



Post-norm: gradient must traverse LayerNorm at every layer — Jacobian products compound across 80+ layers, causing instability. Pre-norm: each layer-to-layer Jacobian is $I + J_k$; the residual stream provides a **gradient highway** that prevents vanishing.

B1: RMSNorm — Cheaper Normalization

Fix 2: RMSNorm [Zhang and Sennrich2019].

LayerNorm (original):

$$\text{LN}(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta$$

Center then scale; $2d$ parameters (γ, β).

RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x)}$$

$$\text{RMS}(x) = \sqrt{\frac{1}{d} \sum_i x_i^2}$$

Scale only; d parameters (γ).
No centering.

B1: RMSNorm — Cheaper Normalization

Fix 2: RMSNorm [Zhang and Sennrich2019].

LayerNorm (original):

$$\text{LN}(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta$$

Center then scale; $2d$ parameters (γ, β).

RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x)}$$

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_i x_i^2}$$

Scale only; d parameters (γ). No centering.

What changed? Mean-centering is dropped. **Why is this justified?**

Emperically, what matters for downstream dot products (attention scores, FFN activations) is the *scale* (norm) of the vector, not its center. RMSNorm controls scale without wasting computation on a mean that carries no useful signal. Saves a meaningful fraction of the normalization cost.

B1: Why ReLU Fails at Scale

Fix 3: Replace the FFN activation.

Standard FFN: $\text{FFN}(x) = W_2 \cdot \text{ReLU}(W_1 x) \in \mathbb{R}^d$, where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, typically $d_{ff} = 4d$.

B1: Why ReLU Fails at Scale

Fix 3: Replace the FFN activation.

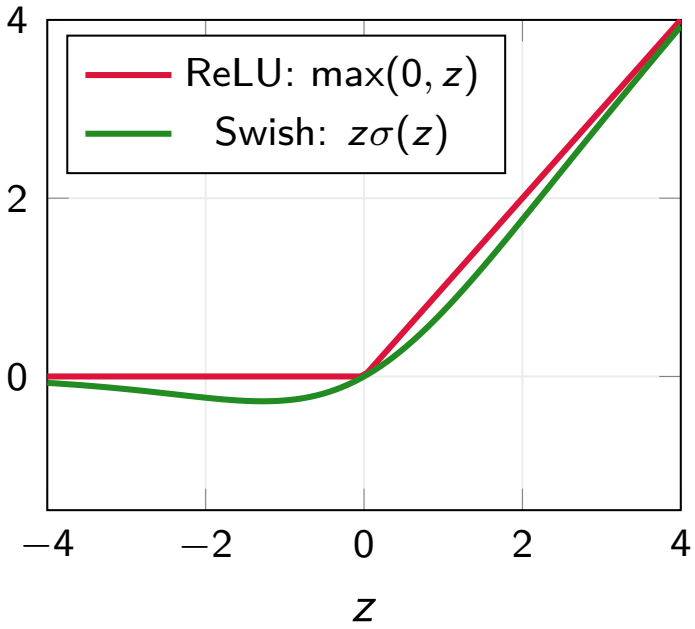
Standard FFN: $\text{FFN}(x) = W_2 \cdot \text{ReLU}(W_1 x) \in \mathbb{R}^d$, where $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, typically $d_{ff} = 4d$.

ReLU has two problems at scale:

1. Dead neurons. For any input x where $(W_1 x)_j < 0$, neuron j outputs exactly zero and receives exactly zero gradient. Once dead, it stays dead — a permanent capacity loss. At 80+ layers with billions of parameters, a significant fraction of FFN neurons die during training.

2. No gating. The output magnitude of $\text{ReLU}(W_1 x)$ is unbounded above, and there is no learned mechanism to suppress large activations before they enter W_2. This contributes to variance growth through the FFN.

B1: Swish — Smooth Activation



$\text{Swish}(z) = z \cdot \sigma(z)$, where σ is the sigmoid.

Key properties:

- **Smooth:** Differentiable everywhere.
- **Non-monotone:** Allows small negative signals through ($\text{Swish}(z) < 0$ for $z < 0$), so neurons never fully die.
- **Gradient:** $\text{Swish}'(z)$ is nonzero everywhere, maintaining gradient flow through all neurons.

Swish addresses dead neurons, but not gating. Gating is a separate idea.

B1: SwiGLU — Gating for Per-Token Selectivity

Gating gives the network a *learned input-dependent switch*: the gate $\sigma(W_g x)$ can suppress dimensions of $W_1 x$ that are irrelevant for the current input.

Why this matters: The FFN is shared across all tokens. A token about organic chemistry and one about contract law both pass through the same W_1, W_2. Gating provides per-token selectivity within a shared parameter set: $\sigma(W_g x)$ can suppress the "legal dimensions" for the chemistry token.

B1: SwiGLU — Gating for Per-Token Selectivity

Gating gives the network a *learned input-dependent switch*: the gate $\sigma(W_g x)$ can suppress dimensions of $W_1 x$ that are irrelevant for the current input.

Why this matters: The FFN is shared across all tokens. A token about organic chemistry and one about contract law both pass through the same W_1, W_2. Gating provides per-token selectivity within a shared parameter set: $\sigma(W_g x)$ can suppress the "legal dimensions" for the chemistry token.

SwiGLU FFN [Shazeer2020]:

$$\text{SwiGLU}(x) = W_2\big(\underbrace{W_1 x}_{\text{value}} \odot \underbrace{\text{Swish}(W_g x)}_{\text{gate}}\big)$$

Three weight matrices: $W_1, W_g \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$.

B1: SwiGLU — Gating for Per-Token Selectivity

Gating gives the network a *learned input-dependent switch*: the gate $\sigma(W_g x)$ can suppress dimensions of $W_1 x$ that are irrelevant for the current input.

Why this matters: The FFN is shared across all tokens. A token about organic chemistry and one about contract law both pass through the same W_1, W_2. Gating provides per-token selectivity within a shared parameter set: $\sigma(W_g x)$ can suppress the "legal dimensions" for the chemistry token.

SwiGLU FFN [Shazeer2020]:

$$\text{SwiGLU}(x) = W_2\big(\underbrace{W_1 x}_{\text{value}} \odot \underbrace{\text{Swish}(W_g x)}_{\text{gate}}\big)$$

Three weight matrices: $W_1, W_g \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$.

Bias removal: Modern SwiGLU FFNs set all biases to zero ($b_1, b_2, b_g = 0$). Bias is redundant because (a) RMSNorm before the FFN already adjusts scale via γ, and (b) the residual connection provides an additive pathway. Removing biases

B1: SwiGLU Parameter Count

Parameter budget: Standard FFN uses $d_{ff} = 4d$:

Standard: $2 \times 4d \times d = 8d^2$ parameters (two matrices).

B1: SwiGLU Parameter Count

Parameter budget: Standard FFN uses $d_{ff} = 4d$:

$$\text{Standard:} \quad 2 \times 4d \times d = 8d^2 \ \text{ parameters (two matrices).}$$

SwiGLU uses *three* matrices of size $d_{ff} \times d$ or $d \times d_{ff}$:

$$\text{SwiGLU:} \quad 3 \times d_{ff} \times d \ \text{ parameters.}$$

B1: SwiGLU Parameter Count

Parameter budget: Standard FFN uses $d_{ff} = 4d$:

$$\text{Standard:} \quad 2 \times 4d \times d = 8d^2 \ \text{ parameters (two matrices).}$$

SwiGLU uses *three* matrices of size $d_{ff} \times d$ or $d \times d_{ff}$:

$$\text{SwiGLU:} \quad 3 \times d_{ff} \times d \ \text{ parameters.}$$

To match the parameter budget: Set $3\, d_{ff}\, d = 8d^2$, giving $d_{ff} = \frac{8d}{3} \approx 2.67d$.

Verification: $3 \times \frac{8d}{3} \times d = 8d^2$. ✓

B1: SwiGLU Parameter Count

Parameter budget: Standard FFN uses $d_{ff} = 4d$:

$$\text{Standard:} \quad 2 \times 4d \times d = 8d^2 \ \text{ parameters (two matrices).}$$

SwiGLU uses *three* matrices of size $d_{ff} \times d$ or $d \times d_{ff}$:

$$\text{SwiGLU:} \quad 3 \times d_{ff} \times d \ \text{ parameters.}$$

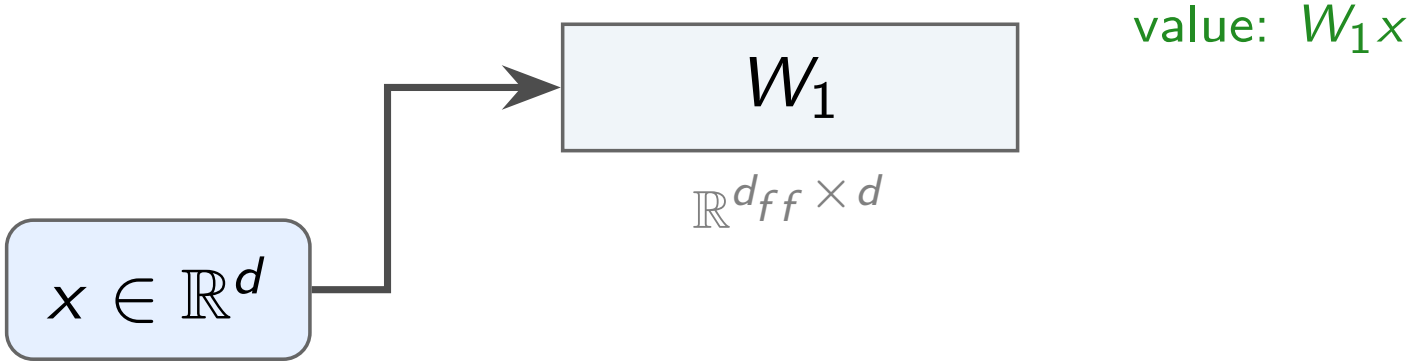
To match the parameter budget: Set $3\, d_{ff}\, d = 8d^2$, giving $d_{ff} = \frac{8d}{3} \approx 2.67d$.

Verification: $3 \times \frac{8d}{3} \times d = 8d^2$. ✓

Empirical result: At matched parameter count, SwiGLU achieves 1–3% lower loss than ReLU FFN across model scales [Shazeer2020].

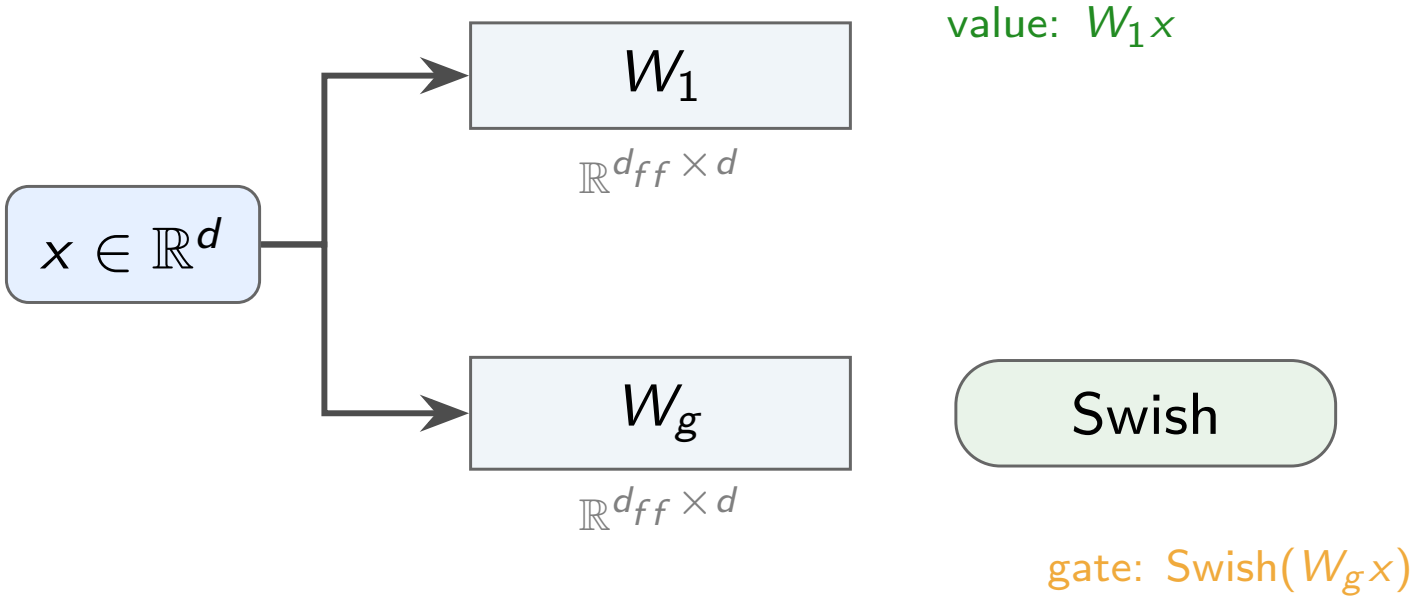
Together: Pre-norm + RMSNorm + SwiGLU stabilize gradient flow across 80+ layers. The residual stream preserves signal; RMSNorm controls scale; SwiGLU eliminates dead neurons and controls FFN variance via gating.

B1: SwiGLU Dataflow



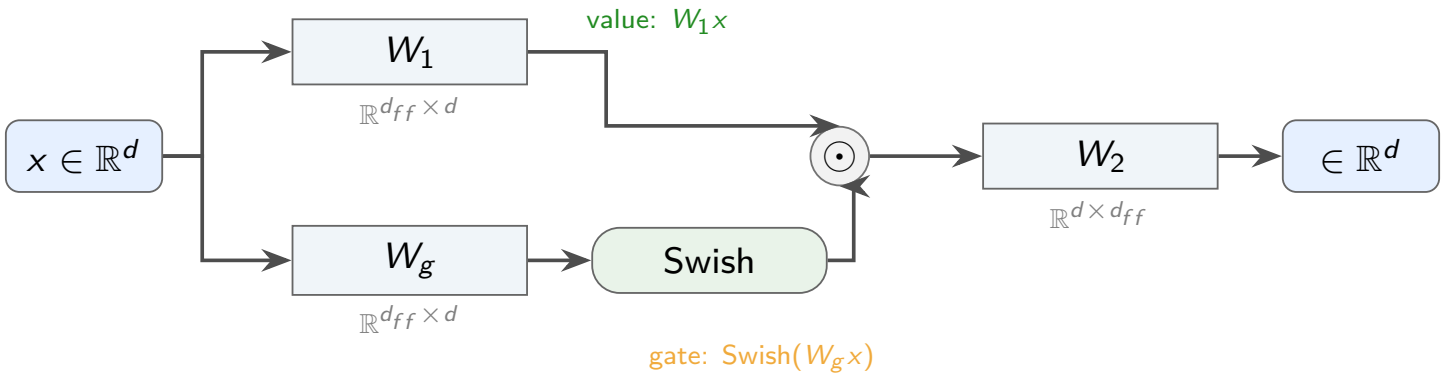
Step 1: Value projection $W_1 x$ — same as standard FFN.

B1: SwiGLU Dataflow



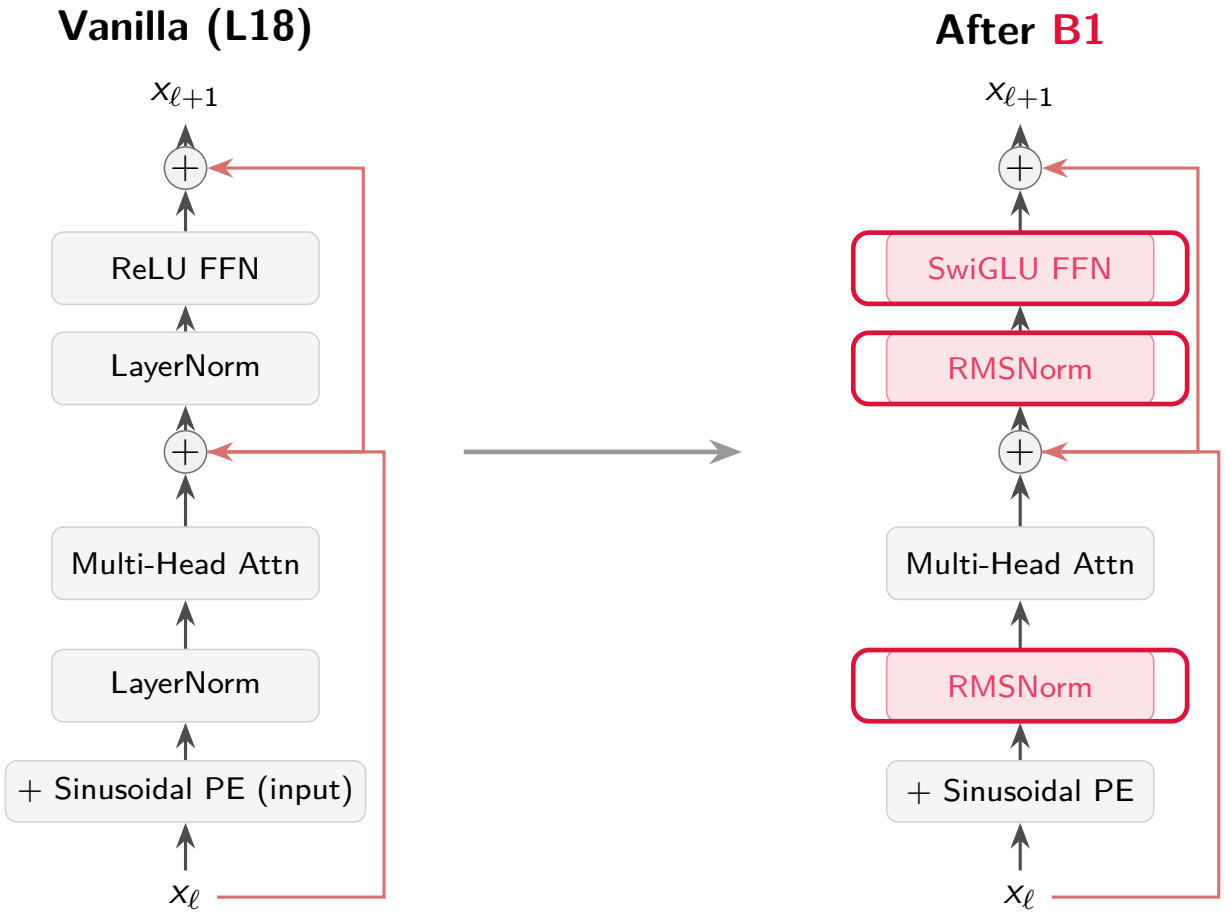
Step 2: Gate projection Swish($W_g x$) *— learned input-dependent switch.*

B1: SwiGLU Dataflow



Step 3: Element-wise product, then output projection W_2. Full formula: $SwiGLU(x) = W_2(W_1 x \odot \text{Swish}(W_g x))$.

Progress: After B1 (Training Stability)



B1 fixes: **Post-norm** $\rightarrow$ **Pre-norm** (unimpeded gradient highway). **LayerNorm** $\rightarrow$ **RMSNorm** (cheaper, equally effective). **ReLU FFN** $\rightarrow$ **SwiGLU FFN** (smooth activation + gating controls variance).

Result: stable training at 80+ layers. Attention and position encoding are unchanged — that's next.

The Second Bottleneck — Long Contexts

Bottleneck: Context length T is the binding dimension. Users demand 128K+ tokens ($\sim$200 pages), but attention costs $O(T^2)$ and pretraining at such lengths is prohibitively expensive.

The Second Bottleneck — Long Contexts

Bottleneck: Context length T is the binding dimension. Users demand 128K+ tokens ($\sim$200 pages), but attention costs $O(T^2)$ and pretraining at such lengths is prohibitively expensive.

Core failure: Standard position embeddings (learned or sinusoidal) do not extrapolate beyond the training context length. The model either has no representation for unseen positions (learned PE) or the additive position–content entanglement produces out-of-distribution attention scores (sinusoidal PE).

The Second Bottleneck — Long Contexts

Bottleneck: Context length T is the binding dimension. Users demand 128K+ tokens ($\sim$200 pages), but attention costs $O(T^2)$ and pretraining at such lengths is prohibitively expensive.

Core failure: Standard position embeddings (learned or sinusoidal) do not extrapolate beyond the training context length. The model either has no representation for unseen positions (learned PE) or the additive position–content entanglement produces out-of-distribution attention scores (sinusoidal PE).

Design requirement: A **relative position encoding** that is re-injected every layer, depends only on the offset $t_k - t_q$, and enables post-training context extension via frequency rescaling.

The Second Bottleneck — Long Contexts

Bottleneck: Context length T is the binding dimension. Users demand 128K+ tokens ($\sim$200 pages), but attention costs $O(T^2)$ and pretraining at such lengths is prohibitively expensive.

Core failure: Standard position embeddings (learned or sinusoidal) do not extrapolate beyond the training context length. The model either has no representation for unseen positions (learned PE) or the additive position–content entanglement produces out-of-distribution attention scores (sinusoidal PE).

Design requirement: A **relative position encoding** that is re-injected every layer, depends only on the offset $t_k - t_q$, and enables post-training context extension via frequency rescaling.

Roadmap: Why additive PE fails $\rightarrow$ mathematical requirement (relative, per-layer) $\rightarrow$ RoPE construction $\rightarrow$ efficient implementation $\rightarrow$ empirical comparison.

B2: Why Standard Position Embeddings Fail

At scale: Llama 3 (8B/70B) was trained with context length up to 8K tokens, yet users demand 128K+ for document-level tasks. Context length T becomes the binding dimension — standard positional schemes cannot bridge this gap.

Recall from the transformer lecture: Standard attention computes scores as $q_i^\top k_j$ where position information is added to the embedding: $x_i \leftarrow x_i + p_i$.

Two standard approaches and why they break:

1. **Learned positional embeddings:** A lookup table $P \in \mathbb{R}^{T_{\max} \times d}$.
   - Position i gets a learned vector p_i.
   - **Hard cutoff at $T_{\max}$**: the model literally has no representation for positions beyond its training length. Cannot extrapolate at all.

2. **Sinusoidal positional embeddings:** Fixed frequencies $p_i^{(k)} = \sin(i/10000^{k/d})$.
   - Defined for any position, so extrapolation is *theoretically* possible.
   - **Problem:** The position information is *added* to the content embedding. The attention score $q_i^\top k_j = (x_i + p_i)^\top W_Q^\top W_K (x_j + p_j)$ mixes content–content, content–position, and position–position

B2: Why Inject Position into Attention Itself?

Root cause: Additive PEs entangle content and position in the attention score. The model cannot separate "what is at this position" from "how far apart are these positions."

B2: Why Inject Position into Attention Itself?

Root cause: Additive PEs entangle content and position in the attention score. The model cannot separate "what is at this position" from "how far apart are these positions."

A deeper problem with additive PE: Position is injected *once* at the embedding layer, then transformed by L decoder blocks of nonlinear maps. As representations propagate, the positional signal becomes **attenuated, entangled with content**, and progressively harder to recover.

B2: Why Inject Position into Attention Itself?

Root cause: Additive PEs entangle content and position in the attention score. The model cannot separate "what is at this position" from "how far apart are these positions."

A deeper problem with additive PE: Position is injected *once* at the embedding layer, then transformed by L decoder blocks of nonlinear maps. As representations propagate, the positional signal becomes **attenuated, entangled with content**, and progressively harder to recover.

Principled fix: Make the model **position-aware at the point where relevance is computed** — inside the attention score itself.

Two requirements for extrapolation

1. The attention score between positions t_q and t_k should depend only on the *relative* offset $t_k - t_q$, not on absolute positions.
2. Position should be *re-injected every layer*, not added once and hoped to survive.

B2: Why Inject Position into Attention Itself?

Root cause: Additive PEs entangle content and position in the attention score. The model cannot separate "what is at this position" from "how far apart are these positions."

A deeper problem with additive PE: Position is injected *once* at the embedding layer, then transformed by L decoder blocks of nonlinear maps. As representations propagate, the positional signal becomes **attenuated, entangled with content**, and progressively harder to recover.

Principled fix: Make the model **position-aware at the point where relevance is computed** — inside the attention score itself.

Two requirements for extrapolation

1. The attention score between positions t_q and t_k should depend only on the *relative* offset $t_k - t_q$, not on absolute positions.
2. Position should be *re-injected every layer*, not added once and hoped to survive.

RoPE — Multi-Frequency Rotation Structure

Goal: Define $R_\Delta \in \mathbb{R}^{d_k \times d_k}$ so that $q^\top R_{t_k - t_q}\, k$ encodes relative position at multiple granularities.

RoPE — Multi-Frequency Rotation Structure

Goal: Define $R_\Delta \in \mathbb{R}^{d_k \times d_k}$ so that $q^\top R_{t_k - t_q}\, k$ encodes relative position at multiple granularities.

Construction. Partition each head's $q, k \in \mathbb{R}^{d_k}$ into $d_k/2$ coordinate pairs (q_{2i-1}, q_{2i}). Assign frequency $\theta_i = 10000^{-2(i-1)/d_k}$ to the i-th pair.

RoPE — Multi-Frequency Rotation Structure

Goal: Define $R_\Delta \in \mathbb{R}^{d_k \times d_k}$ so that $q^\top R_{t_k - t_q}\, k$ encodes relative position at multiple granularities.

Construction. Partition each head's $q, k \in \mathbb{R}^{d_k}$ into $d_k/2$ coordinate pairs (q_{2i-1}, q_{2i}). Assign frequency $\theta_i = 10000^{-2(i-1)/d_k}$ to the i-th pair.

Each pair is rotated by a 2×2 block:

$$R_{\Delta\theta_i} = \begin{pmatrix} \cos(\Delta\,\theta_i) & -\sin(\Delta\,\theta_i) \\ \sin(\Delta\,\theta_i) & \cos(\Delta\,\theta_i) \end{pmatrix}, \qquad \Delta = t_k - t_q.$$

RoPE — Multi-Frequency Rotation Structure

Goal: Define $R_\Delta \in \mathbb{R}^{d_k \times d_k}$ so that $q^\top R_{t_k - t_q}\, k$ encodes relative position at multiple granularities.

Construction. Partition each head's $q, k \in \mathbb{R}^{d_k}$ into $d_k/2$ coordinate pairs (q_{2i-1}, q_{2i}). Assign frequency $\theta_i = 10000^{-2(i-1)/d_k}$ to the i-th pair.

Each pair is rotated by a 2×2 block:

$$R_{\Delta\theta_i} = \begin{pmatrix} \cos(\Delta\,\theta_i) & -\sin(\Delta\,\theta_i) \\ \sin(\Delta\,\theta_i) & \cos(\Delta\,\theta_i) \end{pmatrix}, \qquad \Delta = t_k - t_q.$$

Collect all blocks into a block-diagonal matrix:

$$R_\Delta = \operatorname{diag}\big(R_{\Delta\theta_1},\ R_{\Delta\theta_2},\ \ldots,\ R_{\Delta\theta_{d_k/2}}\big) \ \in\ \mathbb{R}^{d_k \times d_k}.$$

RoPE — Multi-Frequency Rotation Structure

Goal: Define $R_\Delta \in \mathbb{R}^{d_k \times d_k}$ so that $q^\top R_{t_k - t_q}\, k$ encodes relative position at multiple granularities.

Construction. Partition each head's $q, k \in \mathbb{R}^{d_k}$ into $d_k/2$ coordinate pairs (q_{2i-1}, q_{2i}). Assign frequency $\theta_i = 10000^{-2(i-1)/d_k}$ to the i-th pair.

Each pair is rotated by a 2×2 block:

$$R_{\Delta\theta_i} = \begin{pmatrix} \cos(\Delta\,\theta_i) & -\sin(\Delta\,\theta_i) \\ \sin(\Delta\,\theta_i) & \cos(\Delta\,\theta_i) \end{pmatrix}, \qquad \Delta = t_k - t_q.$$

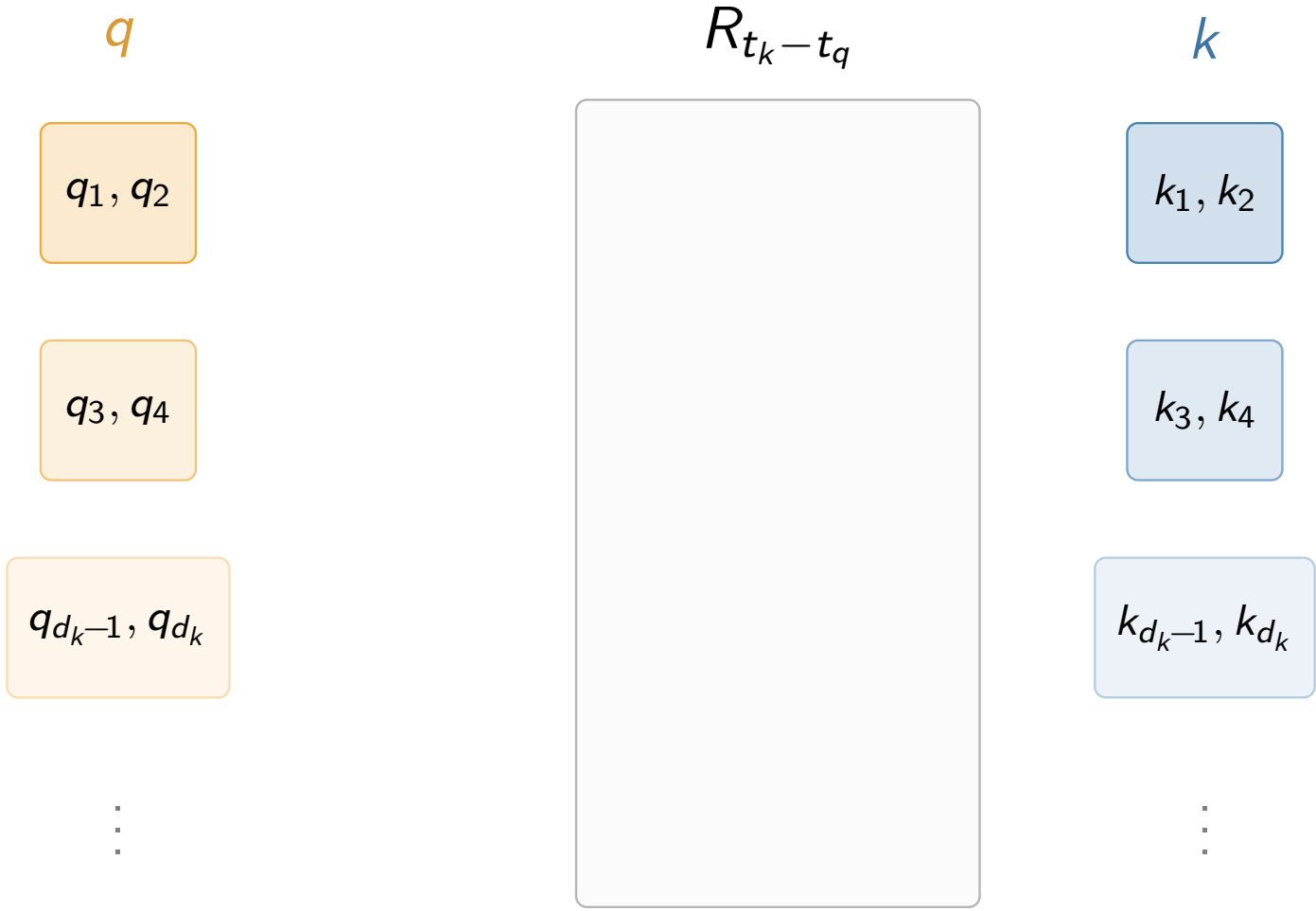
Collect all blocks into a block-diagonal matrix:

$$R_\Delta = \operatorname{diag}\big(R_{\Delta\theta_1},\ R_{\Delta\theta_2},\ \ldots,\ R_{\Delta\theta_{d_k/2}}\big) \ \in\ \mathbb{R}^{d_k \times d_k}.$$

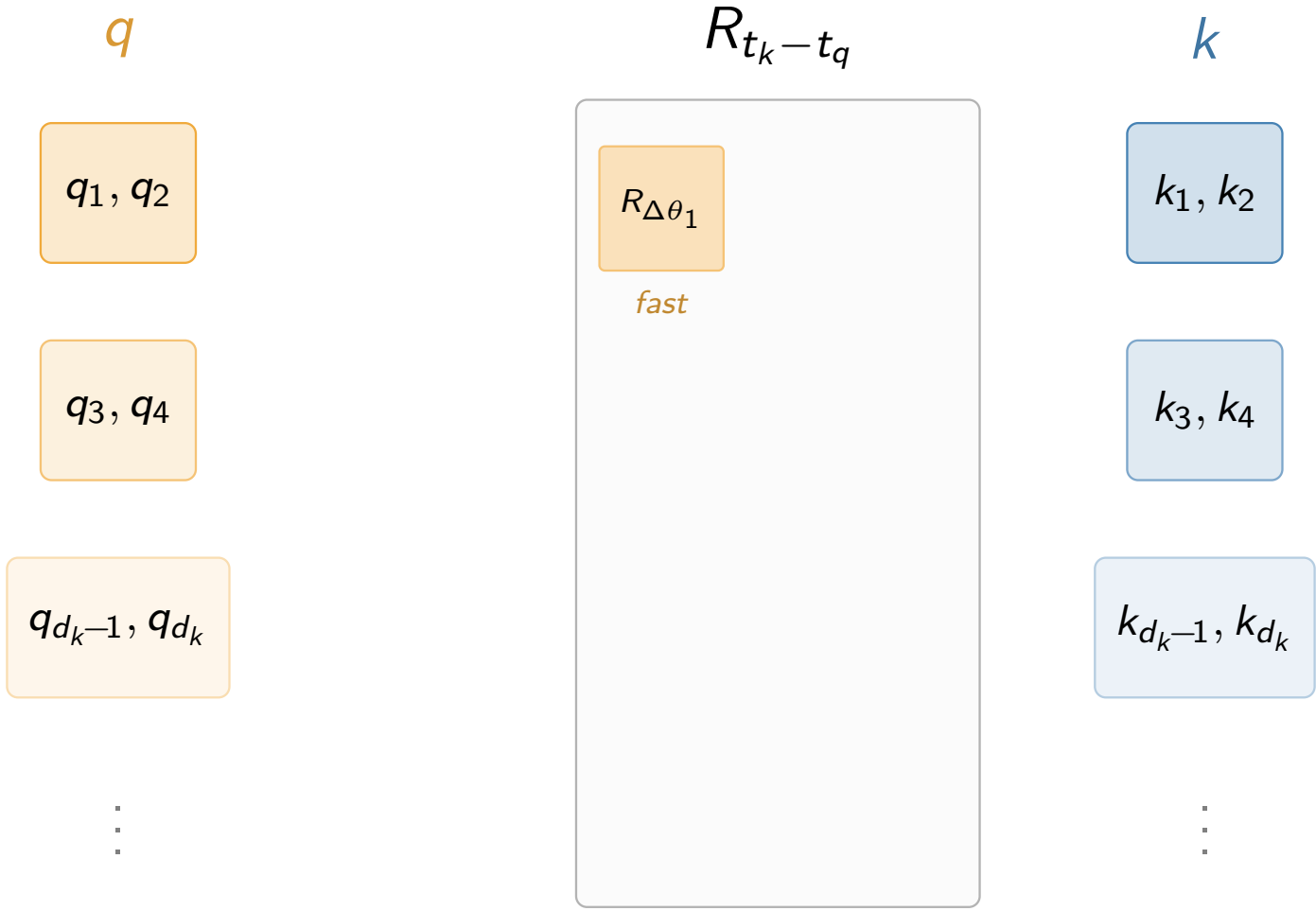
Frequency schedule. Since $\theta_i = 10000^{-2(i-1)/d_k}$:

- **Early pairs** (i small) $\Rightarrow$ θ_i large $\Rightarrow$ fast rotation $\Rightarrow$ sensitive to *fine-grained* (nearby) position differences.
- **Later pairs** (i large) $\Rightarrow$ θ_i small $\Rightarrow$ slow rotation $\Rightarrow$ sensitive to *coarse-grained* (distant) position differences.

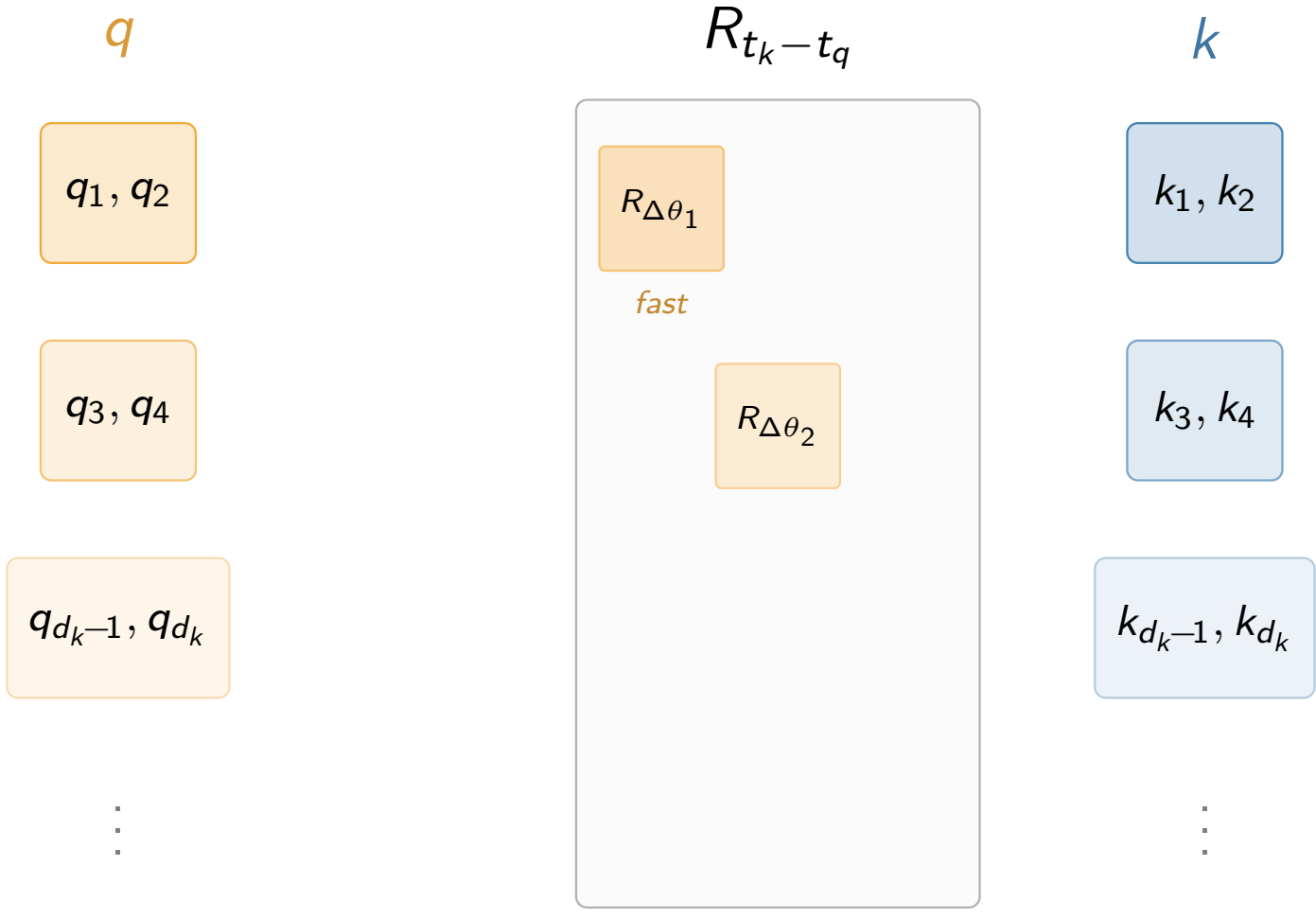
RoPE — Visualizing the Rotated Score



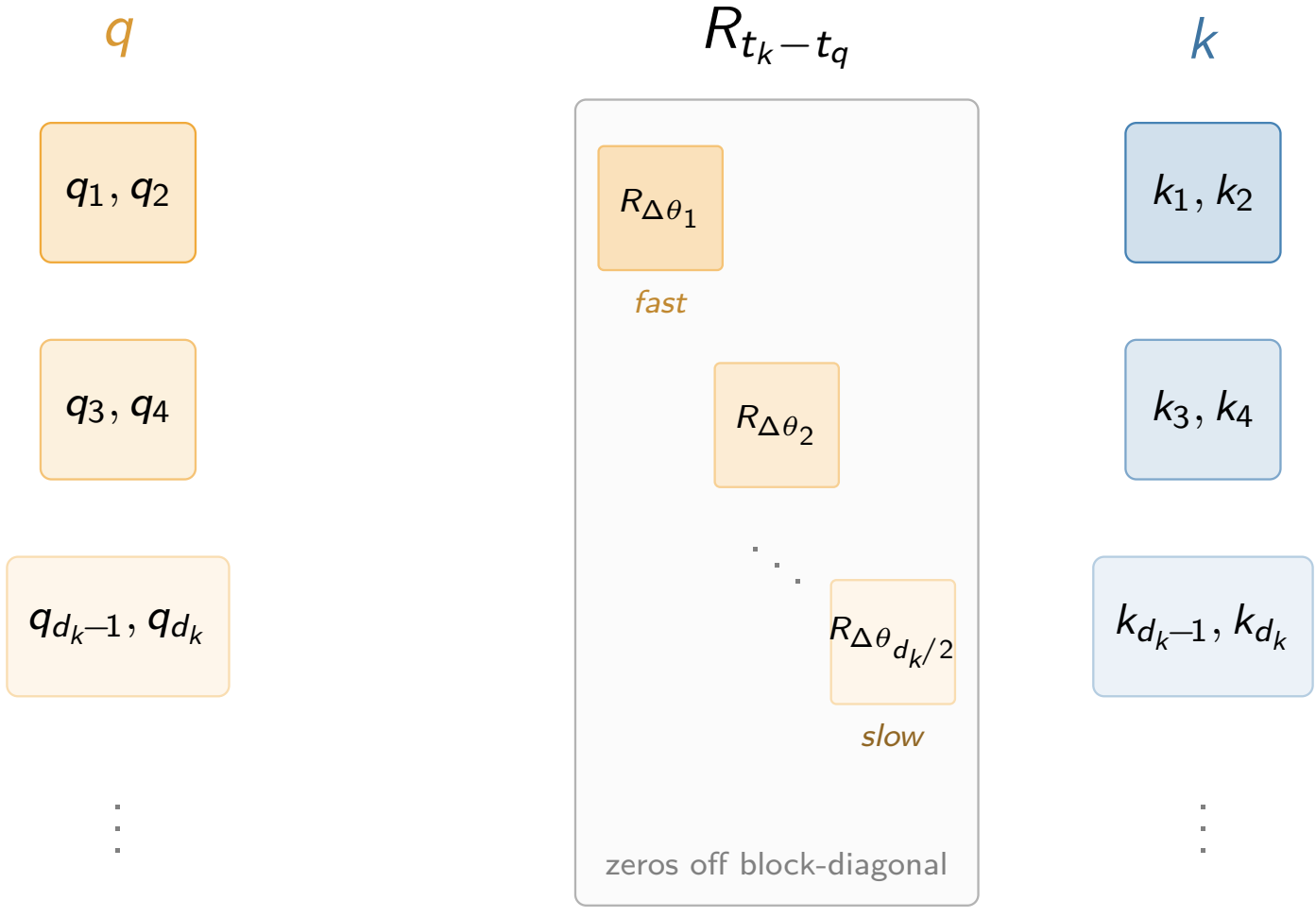
RoPE — Visualizing the Rotated Score



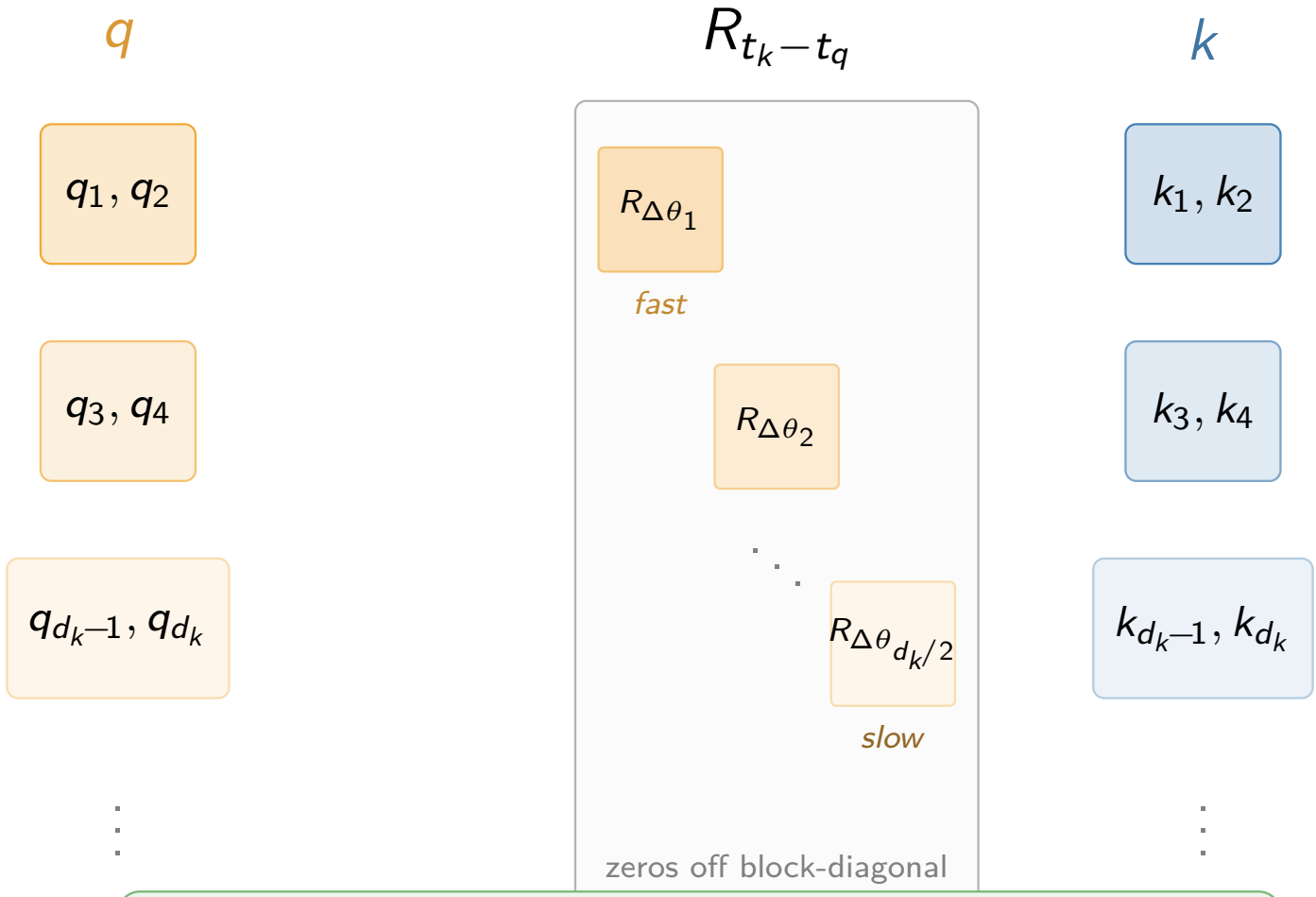
RoPE — Visualizing the Rotated Score



RoPE — Visualizing the Rotated Score



RoPE — Visualizing the Rotated Score



$$\text{score}(t_q, t_k) \quad = \quad q^\top R_{t_k - t_q}\, k \quad = \quad \sum_{i=1}^{d_k/2} \left(q^{(i)}\right)^\top R_{\Delta\theta_i}\, k^{(i)}$$

RoPE — Efficient Implementation via Rotation Composition

Key identity. For any 2×2 rotation matrices:

$$R_\alpha^\top R_\beta = R_{\beta-\alpha}.$$

RoPE — Efficient Implementation via Rotation Composition

Key identity. For any 2×2 rotation matrices:

$$R_\alpha^\top R_\beta = R_{\beta-\alpha}.$$

Derivation for one 2D block. The rotated score for the i-th coordinate pair is:

$$\begin{aligned}\left(q^{(i)}\right)^\top R_{(t_k-t_q)\theta_i}\, k^{(i)} &= \left(q^{(i)}\right)^\top \underbrace{R_{t_q\theta_i}^\top R_{t_k\theta_i}}_{=R_{(t_k-t_q)\theta_i}}\, k^{(i)} \\ &= \left(R_{t_q\theta_i}\, q^{(i)}\right)^\top \left(R_{t_k\theta_i}\, k^{(i)}\right).\end{aligned}$$

RoPE — Efficient Implementation via Rotation Composition

Key identity. For any 2×2 rotation matrices:

$$R_\alpha^\top R_\beta = R_{\beta-\alpha}.$$

Derivation for one 2D block. The rotated score for the i-th coordinate pair is:

$$\begin{aligned}\left(q^{(i)}\right)^\top R_{(t_k-t_q)\theta_i}\, k^{(i)} &= \left(q^{(i)}\right)^\top \underbrace{R_{t_q\theta_i}^\top R_{t_k\theta_i}}_{=R_{(t_k-t_q)\theta_i}}\, k^{(i)} \\ &= \left(R_{t_q\theta_i}\, q^{(i)}\right)^\top \left(R_{t_k\theta_i}\, k^{(i)}\right).\end{aligned}$$

Extending to the full vector. Stack all $d_k/2$ blocks:

$$q^\top R_{t_k-t_q}\, k \;=\; \left(R_{t_q}\, q\right)^\top \left(R_{t_k}\, k\right) \;=\; \langle \tilde{q}_{t_q},\; \tilde{k}_{t_k} \rangle$$

where $R_t = \mathrm{diag}(R_{t\theta_1}, \ldots, R_{t\theta_{d_k/2}})$ and $\tilde{q}_{t_q} = R_{t_q}\, q,\;\; \tilde{k}_{t_k} = R_{t_k}\, k.$

RoPE — Efficient Implementation via Rotation Composition

Key identity. For any 2×2 rotation matrices:

$$R_\alpha^\top R_\beta = R_{\beta-\alpha}.$$

Derivation for one 2D block. The rotated score for the i-th coordinate pair is:

$$\left(q^{(i)}\right)^\top R_{(t_k-t_q)\theta_i}\, k^{(i)} = \left(q^{(i)}\right)^\top \underbrace{R_{t_q\theta_i}^\top R_{t_k\theta_i}}_{=R_{(t_k-t_q)\theta_i}}\, k^{(i)}$$

$$= \left(R_{t_q\theta_i}\, q^{(i)}\right)^\top \left(R_{t_k\theta_i}\, k^{(i)}\right).$$

Extending to the full vector. Stack all $d_k/2$ blocks:

$$q^\top R_{t_k-t_q}\, k \;=\; \left(R_{t_q}\, q\right)^\top \left(R_{t_k}\, k\right) \;=\; \langle \tilde{q}_{t_q},\; \tilde{k}_{t_k} \rangle$$

where $R_t = \mathrm{diag}(R_{t\theta_1}, \ldots, R_{t\theta_{d_k/2}})$ and $\tilde{q}_{t_q} = R_{t_q}\, q,\;\; \tilde{k}_{t_k} = R_{t_k}\, k.$

Implementation recipe

RoPE vs. Sinusoidal PE — Why RoPE Wins in Practice

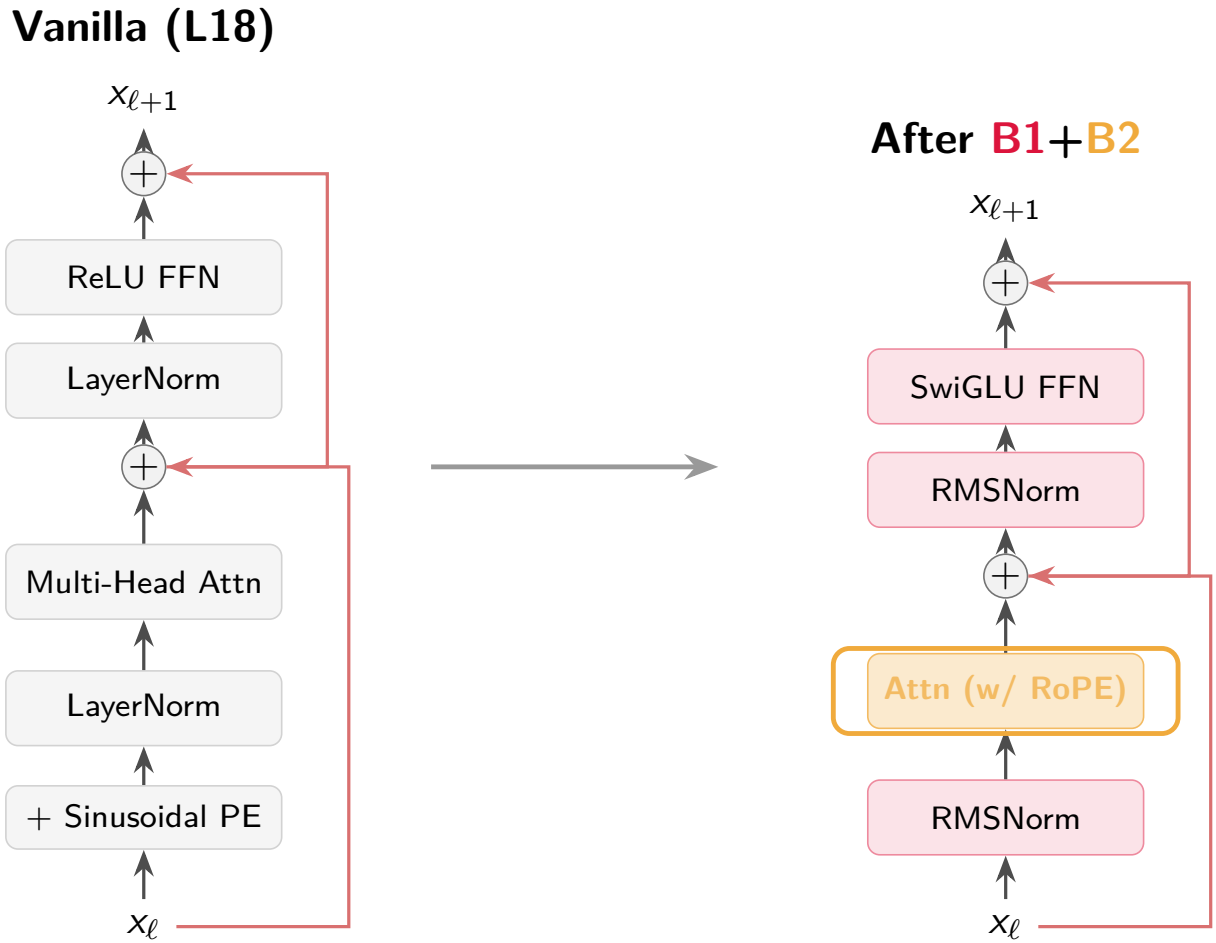
All other things being equal (same model, same data, same hyperparameters), RoPE enables significantly better extrapolation beyond training context length.

	Train length	**Eval at** $2\times$	**Eval at** $4\times$
Sinusoidal PE	2K	degrades	collapses
Learned PE	2K	undefined	undefined
RoPE	2K	mild degradation	moderate
RoPE + freq. rescaling	2K $\rightarrow$ 128K	matches	matches

RoPE's practical impact: Frequency rescaling (Appendix I) enables post-training context extension. This is what allows Llama 3 to go from 8K training context to 128K deployment context. Sinusoidal PE cannot do this because the additive entanglement of content and position means the model sees out-of-distribution inputs at extrapolated positions.

References: Su et al.2024] (RoPE); Press et al.2022] (ALiBi comparison).

Progress: After B2 (Position Encoding)



B2 fix: **Multi-Head Attn** $\rightarrow$ **Attn (w/ RoPE)** (rotation applied to Q, K inside attention; score depends on relative position $t_k - t_q$; enables context extension via frequency rescaling).

Attention is now position-aware and extrapolation-capable, but still uses full MHA — the memory cost at inference is next.

The Third Bottleneck — Inference-Time Memory (KV Cache)

Bottleneck: The KV cache grows as $O(T \cdot L \cdot n_h \cdot d_h)$. At 128K context with 80 layers, the per-user cache exceeds the model weights themselves ($\sim$320 GB vs. $\sim$140 GB for Llama 3 70B in FP16).

The Third Bottleneck — Inference-Time Memory (KV Cache)

Bottleneck: The KV cache grows as $O(T \cdot L \cdot n_h \cdot d_h)$. At 128K context with 80 layers, the per-user cache exceeds the model weights themselves ($\sim$320 GB vs. $\sim$140 GB for Llama 3 70B in FP16).

Key asymmetry: Training is *parallel* (full $T \times T$ attention matrix computed at once via teacher forcing). Inference is *sequential and memory-bound* — at step t, attend over all $t-1$ cached keys/values, sample one token, repeat. The KV cache is what makes this $O(Td)$ instead of $O(T^2 d)$.

The Third Bottleneck — Inference-Time Memory (KV Cache)

Bottleneck: The KV cache grows as $O(T \cdot L \cdot n_h \cdot d_h)$. At 128K context with 80 layers, the per-user cache exceeds the model weights themselves ($\sim$320 GB vs. $\sim$140 GB for Llama 3 70B in FP16).

Key asymmetry: Training is *parallel* (full $T \times T$ attention matrix computed at once via teacher forcing). Inference is *sequential and memory-bound* — at step t, attend over all $t-1$ cached keys/values, sample one token, repeat. The KV cache is what makes this $O(Td)$ instead of $O(T^2 d)$.

Design goal: Reduce the per-token cache footprint without degrading attention quality.

The Third Bottleneck — Inference-Time Memory (KV Cache)

Bottleneck: The KV cache grows as $O(T \cdot L \cdot n_h \cdot d_h)$. At 128K context with 80 layers, the per-user cache exceeds the model weights themselves ($\sim$320 GB vs. $\sim$140 GB for Llama 3 70B in FP16).

Key asymmetry: Training is *parallel* (full $T \times T$ attention matrix computed at once via teacher forcing). Inference is *sequential and memory-bound* — at step t, attend over all $t-1$ cached keys/values, sample one token, repeat. The KV cache is what makes this $O(Td)$ instead of $O(T^2 d)$.

Design goal: Reduce the per-token cache footprint without degrading attention quality.

Roadmap: Quantify KV explosion $\rightarrow$ structural redundancy in MHA $\rightarrow$ MQA (too aggressive) $\rightarrow$ GQA (practical middle ground) $\rightarrow$ MLA (learned compression) $\rightarrow$ comparison.

B3: The KV Cache Trades Memory for Compute

Without a KV cache: at step t, recompute $K_1, \ldots, K_t$ and $V_1, \ldots, V_t$ from scratch. Total cost over T steps: $O(T^2 d)$.

With a KV cache: at step t, only compute K_t, V_t for the new token; read $K_{1:t-1}, V_{1:t-1}$ from cache. Total cost: $O(T d)$.

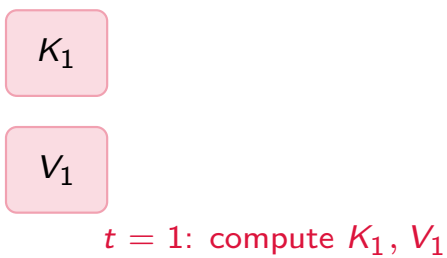
Without cache **With KV cache**

B3: The KV Cache Trades Memory for Compute

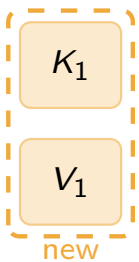
Without a KV cache: at step t, recompute $K_1, \ldots, K_t$ and $V_1, \ldots, V_t$ from scratch. Total cost over T steps: $O(T^2 d)$.

With a KV cache: at step t, only compute K_t, V_t for the new token; read $K_{1:t-1}, V_{1:t-1}$ from cache. Total cost: $O(T d)$.

Without cache



With KV cache

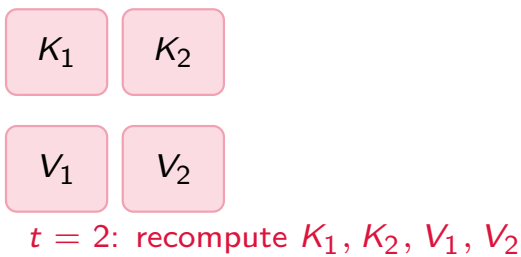


B3: The KV Cache Trades Memory for Compute

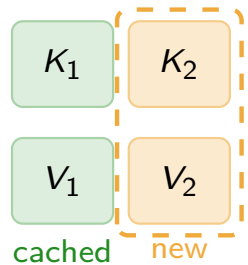
Without a KV cache: at step t, recompute $K_1, \ldots, K_t$ and $V_1, \ldots, V_t$ from scratch. Total cost over T steps: $O(T^2 d)$.

With a KV cache: at step t, only compute K_t, V_t for the new token; read $K_{1:t-1}, V_{1:t-1}$ from cache. Total cost: $O(T d)$.

Without cache



With KV cache

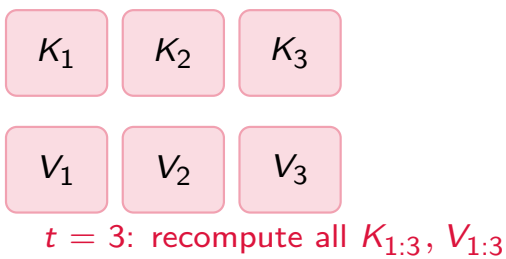


B3: The KV Cache Trades Memory for Compute

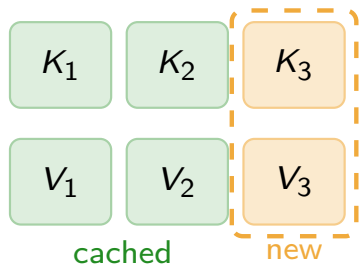
Without a KV cache: at step t, recompute $K_1, \ldots, K_t$ and $V_1, \ldots, V_t$ from scratch. Total cost over T steps: $O(T^2 d)$.

With a KV cache: at step t, only compute K_t, V_t for the new token; read $K_{1:t-1}, V_{1:t-1}$ from cache. Total cost: $O(Td)$.

Without cache



With KV cache

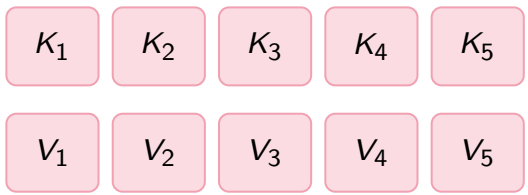


B3: The KV Cache Trades Memory for Compute

Without a KV cache: at step t, recompute $K_1, \ldots, K_t$ and $V_1, \ldots, V_t$ from scratch. Total cost over T steps: $O(T^2 d)$.

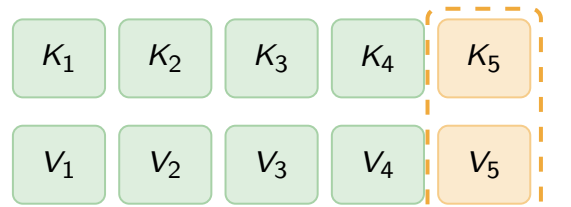
With a KV cache: at step t, only compute K_t, V_t for the new token; read $K_{1:t-1}, V_{1:t-1}$ from cache. Total cost: $O(Td)$.

Without cache



Cost: $O(T^2 d)$ total

With KV cache



Cost: $O(Td)$ total

Summary: Without cache: $O(T^2 d)$. With cache: $O(Td)$. But the cache grows as $O(T \cdot L \cdot n_h \cdot d_h)$.

B3: KV Cache Size Calculation

KV cache size per token per layer (standard multi-head attention):

$$\underbrace{2}_{\text{K and V}} \times \underbrace{n_h}_{\text{heads}} \times \underbrace{d_h}_{\text{head dim}} \quad \text{values}$$

B3: KV Cache Size Calculation

KV cache size per token per layer (standard multi-head attention):

$$\underbrace{2}_{\text{K and V}} \times \underbrace{n_h}_{\text{heads}} \times \underbrace{d_h}_{\text{head dim}} \quad \text{values}$$

Example: Llama 3 70B ($n_h = 64$, $d_h = 128$, $L = 80$ layers, FP16):

$$2 \times 64 \times 128 \times 80 \times 2\,\text{bytes} = \mathbf{2.6}\ \text{MB per token}$$

B3: KV Cache Size Calculation

KV cache size per token per layer (standard multi-head attention):

$$\underbrace{2}_{\text{K and V}} \times \underbrace{n_h}_{\text{heads}} \times \underbrace{d_h}_{\text{head dim}} \quad \text{values}$$

Example: Llama 3 70B ($n_h = 64$, $d_h = 128$, $L = 80$ layers, FP16):

$$2 \times 64 \times 128 \times 80 \times 2\,\text{bytes} = \mathbf{2.6}\ \text{MB per token}$$

At 128K context: $128{,}000 \times 2.6\,\text{MB} \approx \mathbf{320}$ GB per user.

*Order-of-magnitude estimate assuming FP16, no tensor parallelism, no KV quantization. Real deployments use sharding + quantization to reduce this, but the **scaling** (linear in $T \times L$) is the point.*

B3: KV Cache Size Calculation

KV cache size per token per layer (standard multi-head attention):

$$\underbrace{2}_{\text{K and V}} \times \underbrace{n_h}_{\text{heads}} \times \underbrace{d_h}_{\text{head dim}} \quad \text{values}$$

Example: Llama 3 70B ($n_h = 64$, $d_h = 128$, $L = 80$ layers, FP16):

$$2 \times 64 \times 128 \times 80 \times 2\,\text{bytes} = \mathbf{2.6}\ \text{MB per token}$$

At 128K context: $128{,}000 \times 2.6\,\text{MB} \approx \mathbf{320}$ GB per user.

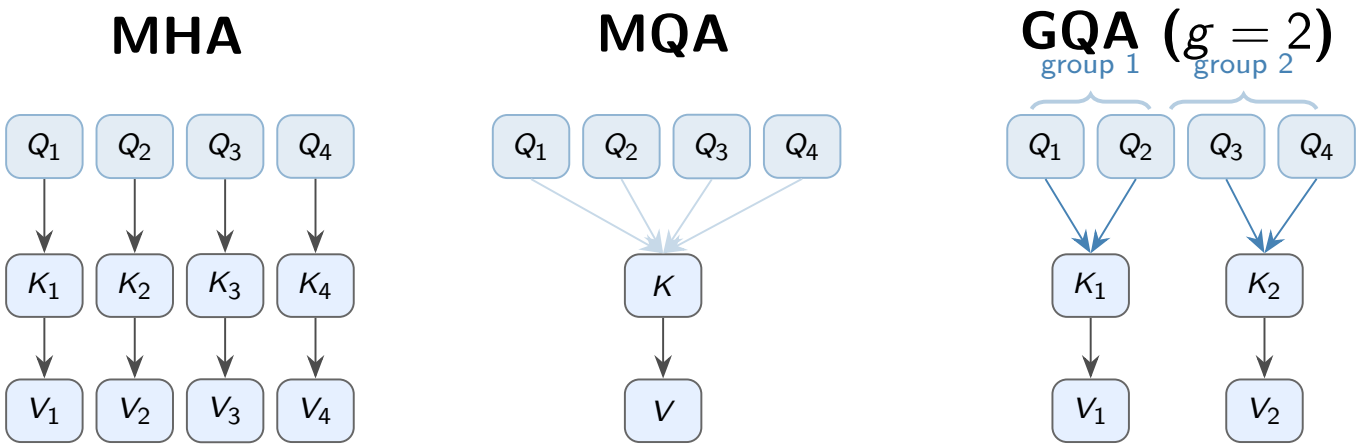
*Order-of-magnitude estimate assuming FP16, no tensor parallelism, no KV quantization. Real deployments use sharding + quantization to reduce this, but the **scaling** (linear in $T \times L$) is the point.*

This exceeds the model weights themselves (~140 GB in FP16).

The three quantities that drive cache size are **context length** T, **number of layers** L, and **heads per layer** n_h. All three grow as models scale, making the cache the dominant memory cost at inference.

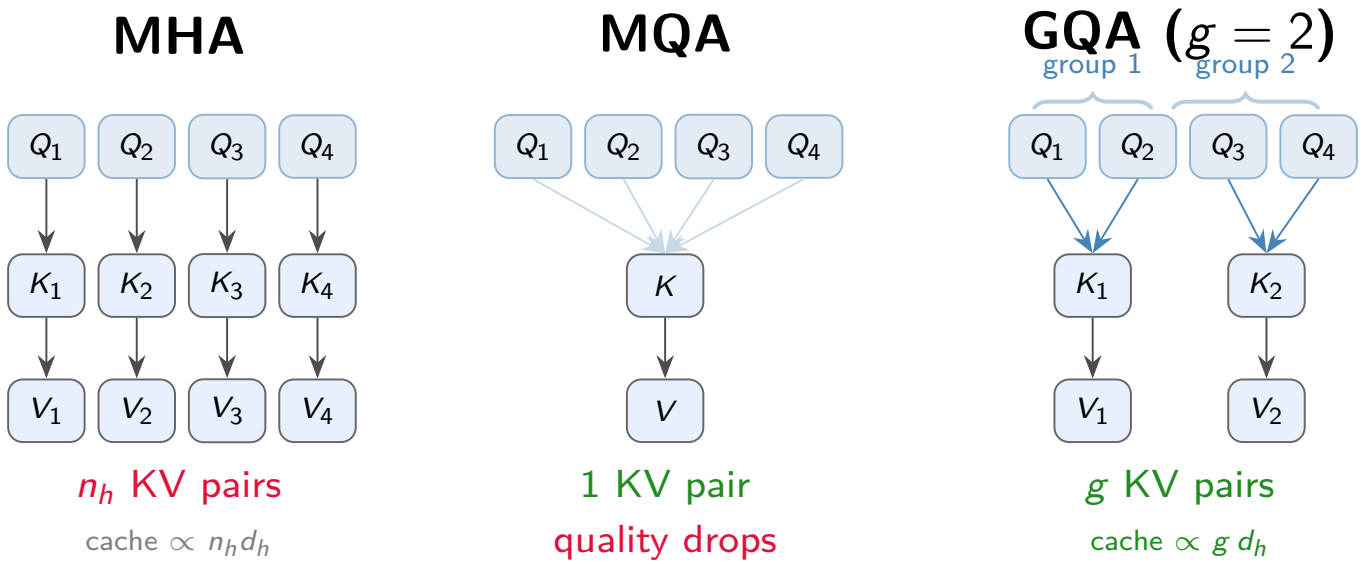
B3 Fix 1: Grouped-Query Attention (GQA)

Idea [Ainslie et al.2023]: Reduce the number of distinct KV pairs per layer.



B3 Fix 1: Grouped-Query Attention (GQA)

Idea [Ainslie et al.2023]: Reduce the number of distinct KV pairs per layer.



How it works: In standard MHA, each of n_h heads has its own $W_K^{(i)}, W_V^{(i)}$ matrices. In GQA with g KV groups, heads are partitioned into g groups; all heads within a group share the *same* K and V projections. Head i uses KV group $\lfloor i \cdot g / n_h \rfloor$.

Llama 3 70B: $n_h = 64$ query heads, $g = 8$ KV groups $\Rightarrow$ $8\times$ **KV cache reduction** [Dubey et al.2024].

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Each of the 4 KV groups is shared by $32/4 = 8$ query heads. KV cache shrinks by $32/4 = 8\times$ relative to MHA.

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Each of the 4 KV groups is shared by $32/4 = 8$ query heads. KV cache shrinks by $32/4 = 8\times$ relative to MHA.

Q2. Compute the KV cache size *per token* for Llama 3 70B (GQA, $n_h = 64$, $g = 8$ KV groups, $d_h = 128$, $L = 80$).

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Each of the 4 KV groups is shared by $32/4 = 8$ query heads. KV cache shrinks by $32/4 = 8\times$ relative to MHA.

Q2. Compute the KV cache size *per token* for Llama 3 70B (GQA, $n_h = 64$, $g = 8$ KV groups, $d_h = 128$, $L = 80$).

Per token per layer: $2 \times g \times d_h = 2 \times 8 \times 128 = 2{,}048$ values. For $L = 80$ layers: $2{,}048 \times 80 = 163{,}840$ values per token. In FP16: $163{,}840 \times 2$ bytes ≈ 320 KB/token. At 128K context: $\sim$41 GB total.

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Each of the 4 KV groups is shared by $32/4 = 8$ query heads. KV cache shrinks by $32/4 = 8\times$ relative to MHA.

Q2. Compute the KV cache size *per token* for Llama 3 70B (GQA, $n_h = 64$, $g = 8$ KV groups, $d_h = 128$, $L = 80$).

Per token per layer: $2 \times g \times d_h = 2 \times 8 \times 128 = 2{,}048$ values. For $L = 80$ layers: $2{,}048 \times 80 = 163{,}840$ values per token. In FP16: $163{,}840 \times 2$ bytes ≈ 320 KB/token. At 128K context: $\sim$41 GB total.

Q3. RMSNorm drops mean-centering. Why is this justified in pre-norm architectures?

Pause & Check 1

Q1. A model has $n_h = 32$ query heads and uses GQA with $g = 4$ KV groups. By what factor does the KV cache shrink relative to MHA?

Each of the 4 KV groups is shared by $32/4 = 8$ query heads. KV cache shrinks by $32/4 = 8\times$ relative to MHA.

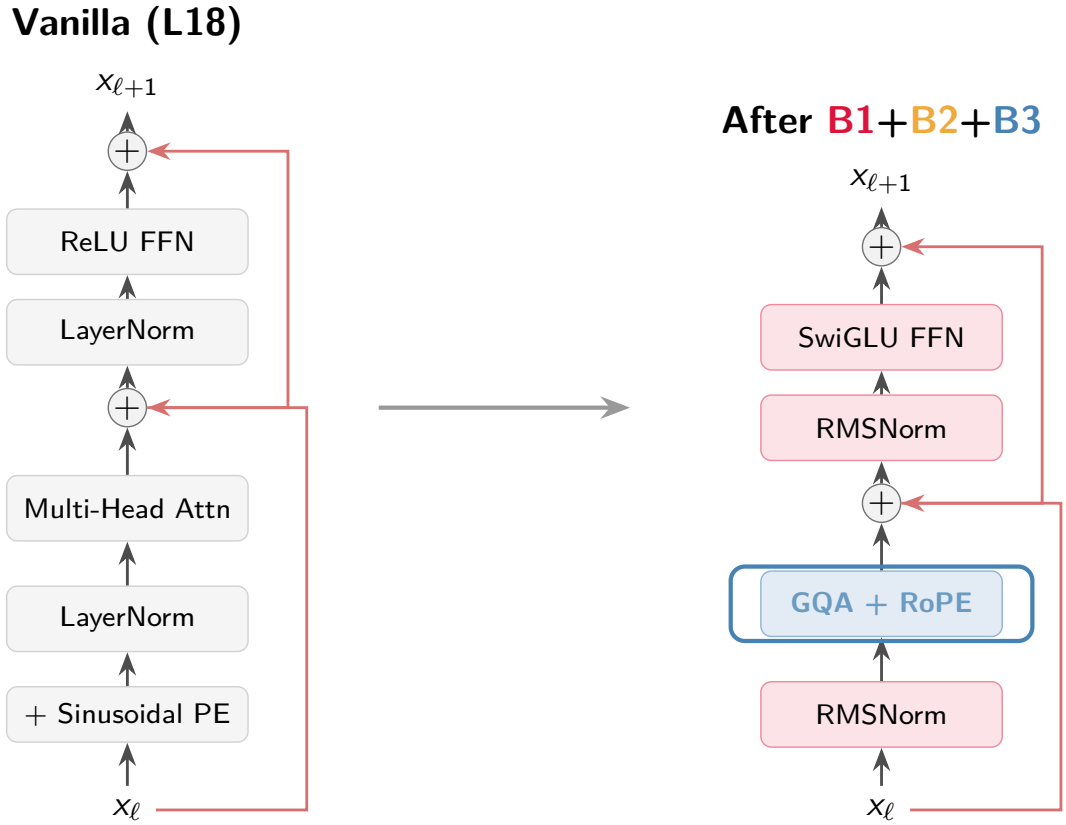
Q2. Compute the KV cache size *per token* for Llama 3 70B (GQA, $n_h = 64$, $g = 8$ KV groups, $d_h = 128$, $L = 80$).

Per token per layer: $2 \times g \times d_h = 2 \times 8 \times 128 = 2{,}048$ values. For $L = 80$ layers: $2{,}048 \times 80 = 163{,}840$ values per token. In FP16: $163{,}840 \times 2$ bytes ≈ 320 KB/token. At 128K context: $\sim$41 GB total.

Q3. RMSNorm drops mean-centering. Why is this justified in pre-norm architectures?

In pre-norm architectures, the residual stream carries a running

Progress: After B3 (KV Cache)



B3 fix: **Multi-Head Attn** $\rightarrow$ **GQA + RoPE** (GQA shares KV heads across query groups for 8$\times$ cache reduction; RoPE applied inside attention).

MLA (learned compression, used by DeepSeek) is covered in Appendix B.

Only the FFN remains unchanged from vanilla. The last bottleneck addresses it: can we decouple capacity from per-token cost? (See Appendix C.)

The Fourth Bottleneck — Dense Compute

Bottleneck: Per-token compute scales as $O(N)$. The FFN sublayer accounts for $\sim 2/3$ of all parameters ($\sim$47B of Llama 3 70B's 70B parameters).

The Fourth Bottleneck — Dense Compute

Bottleneck: Per-token compute scales as $O(N)$. The FFN sublayer accounts for $\sim 2/3$ of all parameters ($\sim$47B of Llama 3 70B's 70B parameters).

Core inefficiency: Every token — chemistry, contract law, or code — activates the *entire* FFN. If you want more model capacity (knowledge, reasoning ability), you increase N, but per-token compute grows proportionally. **Capacity and compute are locked together.**

The Fourth Bottleneck — Dense Compute

Bottleneck: Per-token compute scales as $O(N)$. The FFN sublayer accounts for $\sim 2/3$ of all parameters ($\sim$47B of Llama 3 70B's 70B parameters).

Core inefficiency: Every token — chemistry, contract law, or code — activates the *entire* FFN. If you want more model capacity (knowledge, reasoning ability), you increase N, but per-token compute grows proportionally. **Capacity and compute are locked together.**

Design objective: Decouple total model capacity from per-token compute. Increase the parameter count by a factor of E while keeping the per-token cost at $\sim k/E$ of the equivalent dense model ($k \ll E$).

The Fourth Bottleneck — Dense Compute

Bottleneck: Per-token compute scales as $O(N)$. The FFN sublayer accounts for $\sim 2/3$ of all parameters ($\sim$47B of Llama 3 70B's 70B parameters).

Core inefficiency: Every token — chemistry, contract law, or code — activates the *entire* FFN. If you want more model capacity (knowledge, reasoning ability), you increase N, but per-token compute grows proportionally. **Capacity and compute are locked together.**

Design objective: Decouple total model capacity from per-token compute. Increase the parameter count by a factor of E while keeping the per-token cost at $\sim k/E$ of the equivalent dense model ($k \ll E$).

Answer: Mixture of Experts (MoE). Replace the single dense FFN with E expert FFNs plus a learned router. Each token is routed to only $k \ll E$ experts. **Capacity and per-token compute are decoupled.**

DeepSeek V3: 256 routed experts + 1 shared, top-1 routing. Only $\sim$0.78% of

The Modern LLM Template (2024–2026)

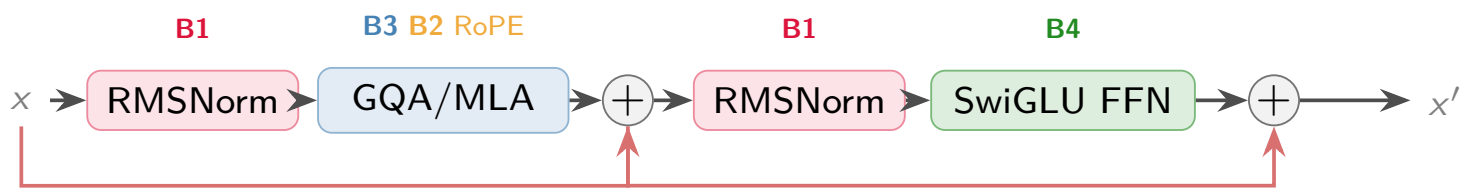
A modern LLM is a decoder-only transformer with:

- **Normalization:** Pre-norm with **RMSNorm**
- **Position encoding:** **RoPE** (or iRoPE / NTK-scaled)
- **Attention:** **GQA** or **MLA** (Appendix B) (for KV cache efficiency)
- **FFN:** **SwiGLU** activation (gated linear unit)
- **Optional sparsity:** **MoE** (Appendix C) (decouple capacity from compute)
- **Training objective:** Next-token prediction (autoregressive cross-entropy)

The Modern LLM Template (2024–2026)

A modern LLM is a decoder-only transformer with:

- **Normalization:** Pre-norm with **RMSNorm**
- **Position encoding:** **RoPE** (or iRoPE / NTK-scaled)
- **Attention:** **GQA** or **MLA** (Appendix B) (for KV cache efficiency)
- **FFN:** **SwiGLU** activation (gated linear unit)
- **Optional sparsity:** **MoE** (Appendix C) (decouple capacity from compute)
- **Training objective:** Next-token prediction (autoregressive cross-entropy)



Comparative Anatomy: DeepSeek V3 vs Llama 4 Maverick

	DeepSeek V3	**Llama 4 Maverick**
Architecture choices		
Attention	**MLA ($d_c = 512$)**	**GQA**
Position encoding	**RoPE**	**iRoPE (interleaved)**
FFN activation	SwiGLU	SwiGLU
Normalization	RMSNorm (pre-norm)	RMSNorm (pre-norm)
MoE experts	**256 routed + 1 shared**	**128 routed + 1 shared**
Scale		
Total params	**671B**	**400B**
Active params/token	**37B**	**17B**
Max context	**128K**	**1M**
Layers	61	92
Training		
Training tokens	14.8T	$\sim$22T

Comparative Anatomy: DeepSeek V3 vs Llama 4 Maverick

	DeepSeek V3	Llama 4 Maverick
Architecture choices		
Attention	**MLA ($d_c = 512$)**	**GQA**
Position encoding	**RoPE**	**iRoPE (interleaved)**
FFN activation	SwiGLU	SwiGLU
Normalization	RMSNorm (pre-norm)	RMSNorm (pre-norm)
MoE experts	**256 routed + 1 shared**	**128 routed + 1 shared**
Scale		
Total params	**671B**	**400B**
Active params/token	**37B**	**17B**
Max context	**128K**	**1M**
Layers	61	92
Training		
Training tokens	14.8T	∼22T

Architecture: agree on FFN + norm; differ on attention (B3) and position encoding (B2). **Scale:** DeepSeek larger overall; Llama activates fewer params per token.

GPT, Claude, Gemini do not publish full details, but all are believed to use the same template (decoder-only, pre-norm, rotary PE, MoE). The architectural differences between frontier models are increasingly small compared to the shared template; the meaningful differences are in **training data quality, training pipeline, and alignment** — where the next lecture picks up.

Sources: DeepSeek-AI2024], Meta AI2025], Dubey et al.2024].

What You Have at the End of Pretraining

At the end of pretraining, you have a model that:

- Computes $p(x_t \mid x_{<t})$ for any continuation of any text sequence.
- Is a powerful **density estimator** trained by maximum likelihood.
- Can continue any sequence — including toxic, incorrect, or harmful ones.
- Has **no notion of what a good response is.** MLE on a web corpus optimizes for "looks like the internet," not "is helpful and honest."

Pause & Check 3 — Synthesis

Q1. Match each architectural choice to the bottleneck it fixes:
(a) RMSNorm + pre-norm; (b) GQA; (c) SwiGLU; (d) RoPE.

Pause & Check 3 — Synthesis

Q1. Match each architectural choice to the bottleneck it fixes:
(a) RMSNorm + pre-norm; (b) GQA; (c) SwiGLU; (d) RoPE.

(a) RMSNorm + pre-norm → **B1** training stability.
(b) GQA → **B3** KV cache.
(c) SwiGLU → **B1** training stability (also **B4** as part of FFN redesign).
(d) RoPE → **B2** position extrapolation.

Pause & Check 3 — Synthesis

Q1. Match each architectural choice to the bottleneck it fixes:
(a) RMSNorm + pre-norm; (b) GQA; (c) SwiGLU; (d) RoPE.

(a) RMSNorm + pre-norm → **B1** training stability.
(b) GQA → **B3** KV cache.
(c) SwiGLU → **B1** training stability (also **B4** as part of FFN redesign).
(d) RoPE → **B2** position extrapolation.

Q2. Why is a pretrained LLM *not* ready to deploy as a conversational assistant, even if its loss is very low?

Pause & Check 3 — Synthesis

Q1. Match each architectural choice to the bottleneck it fixes:
(a) RMSNorm + pre-norm; (b) GQA; (c) SwiGLU; (d) RoPE.

(a) RMSNorm + pre-norm → **B1** training stability.
(b) GQA → **B3** KV cache.
(c) SwiGLU → **B1** training stability (also **B4** as part of FFN redesign).
(d) RoPE → **B2** position extrapolation.

Q2. Why is a pretrained LLM *not* ready to deploy as a conversational assistant, even if its loss is very low?

Pretraining minimizes next-token cross-entropy on a web corpus. A low-loss model imitates the distribution of human text — including harmful, biased, and incorrect text. It has no notion of "helpful" vs. "not helpful." Alignment (SFT, RLHF) is required. That is the subject of the next lecture.

Questions?

Next time: Alignment — SFT, RLHF, and instruction tuning.

References I

J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of EMNLP 2023*, 2023. URL https://arxiv.org/abs/2305.13245.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33: 1877–1901, 2020. URL https://arxiv.org/abs/2005.14165.

DeepSeek-AI. DeepSeek-V3 technical report. Technical report, DeepSeek AI, 2024. URL https://arxiv.org/abs/2412.19437.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT 2019*, 2019. URL https://arxiv.org/abs/1810.04805.

A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

References II

W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://arxiv.org/abs/2101.03961.

J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. Training compute-optimal large language models. *Advances in Neural Information Processing Systems*, 35:30016–30030, 2022. URL https://arxiv.org/abs/2203.15556.

J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361.

A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, et al. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024. URL https://arxiv.org/abs/2405.04434.

Meta AI. Llama 4 technical report. Technical report, Meta AI, 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

References III

N. Muennighoff, A. M. Rush, B. Barak, T. Le Scao, N. Tazi, A. Piktus, S. Pyysalo, T. Wolf, and C. Raffel. Scaling data-constrained language models. *Advances in Neural Information Processing Systems*, 36, 2024. URL https://arxiv.org/abs/2305.16264.

D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean. Carbon emissions and large neural network training. In *arXiv preprint arXiv:2104.10350*, 2021. URL https://arxiv.org/abs/2104.10350.

B. Peng, J. Quesnelle, H. Fan, and E. Shippole. YaRN: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023. URL https://arxiv.org/abs/2309.00071.

O. Press, N. A. Smith, and M. Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL https://arxiv.org/abs/2108.12409.

A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018. URL https://openai.com/research/language-unsupervised.

References IV

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019. URL `https://openai.com/research/better-language-models`.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140): 1–67, 2020. URL `https://arxiv.org/abs/1910.10683`.

N. Shazeer. GLU variants improve transformer. In *arXiv preprint arXiv:2002.05202*, 2020. URL `https://arxiv.org/abs/2002.05202`.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations*, 2017. URL `https://arxiv.org/abs/1701.06538`.

J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. URL `https://arxiv.org/abs/2104.09864`. Preprint 2021.

References V

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. URL `https://arxiv.org/abs/2302.13971`.

B. Zhang and R. Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL `https://arxiv.org/abs/1910.07467`.

Appendix Overview

App.	Content / When to use
A	GPT architecture deep-dive + prefill/decode phases — autoregressive
B	Full MLA derivation (query compression, RoPE interaction) — on stud
C	Mixture of Experts: mechanism, load balancing, expert collapse, trainir
D	Scaling Laws — full Chinchilla derivation
E	RoPE context extension (NTK-aware, YaRN) — on student Q
F	Kaplan vs. Chinchilla Scaling Laws — comparison
G	Architecture choices beyond pretraining — downstream pipeline impact

Appendix A: GPT Architecture — Autoregressive Factorization

Autoregressive factorization:

$$p(x) = \prod_{i=1}^{T} p(x_i \mid x_{<i})$$

Training objective: minimize cross-entropy over the training corpus:

$$\mathcal{L} = -\sum_{i} \log p_\theta(x_i \mid x_{<i})$$

Causal mask: upper-triangular $-\infty$ mask applied before softmax. Position (i, j) with $j > i$ is blocked; token i cannot attend to future token j.

Weight tying: Input embedding matrix $W_e \in \mathbb{R}^{|V| \times d}$ is shared with the output projection (logit layer), reducing parameters by $|V| \times d$.

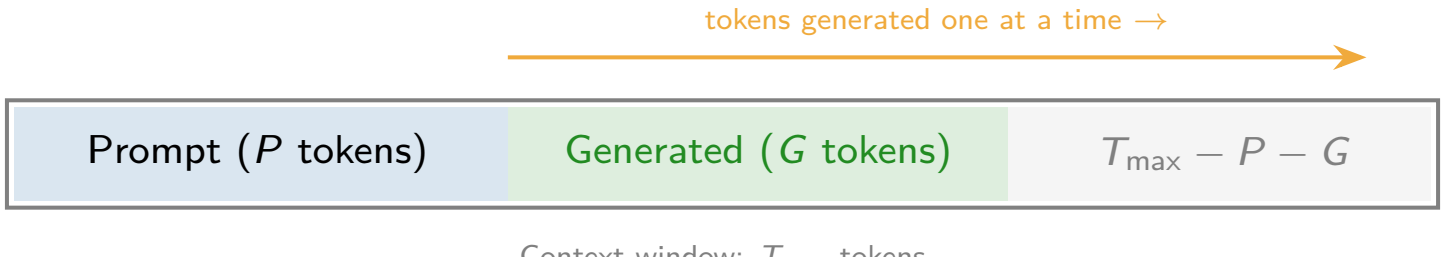
Concrete dimensions (GPT-2): $d = 768$, $L = 12$, $h = 12$, $d_h = 64$, $|V| = 50{,}257$. Total parameters: $\sim$117M.

Appendix A: Context Length and Inference

Context length (also called *context window*): The maximum number of tokens the model can process in a single forward pass. For GPT-2: $T_{\text{max}} = 1{,}024$ tokens. For modern models: 128K to 10M tokens.

Both the prompt and the generated response must fit in the context window. If the prompt is 1,000 tokens and the context window is 4,096 tokens, the model can generate at most 3,096 tokens before running out of space.

Handling short inputs: Inputs shorter than the context window are either left-padded with a special `<PAD>` token (with corresponding attention mask to ignore padding) or packed together with other sequences (more compute-efficient during training).

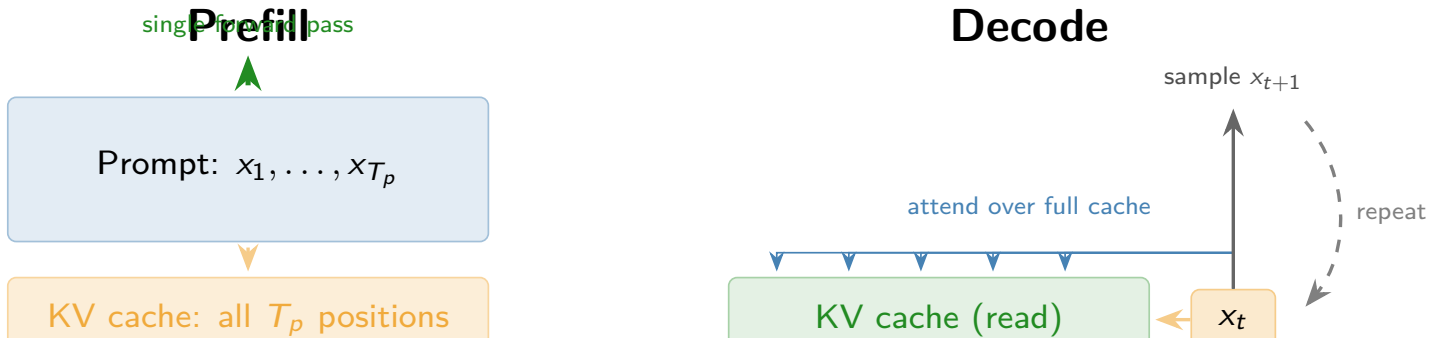


Appendix A: Autoregressive Generation — Prefill and Decode

Modern autoregressive generation splits into two distinct phases:

Prefill phase: A parallel forward pass over the full prompt. All Q, K, V for positions $1, \ldots, T_{\text{prompt}}$ are computed simultaneously under the causal mask. The KV cache is populated for all prompt positions in a *single* forward pass (cost: one forward pass, not T sequential passes).

Decode phase: Sequential token-by-token generation. At each step t: compute Q_t, K_t, V_t for the new token only; append K_t, V_t to the cached keys/values; attend over the full cache ($T_{\text{prompt}} + t - 1$ entries) to produce the next-token distribution; sample; repeat.



Appendix B: KV Cache Footprint — Comparison

Attention Mechanism	**KV Cache / Token / Layer**	**Quality**
MHA	$2\, n_h\, d_h$	Strong
GQA (g groups)	$2\, g\, d_h$	Moderate
MQA ($g = 1$)	$2\, d_h$	Weak
MLA	$d_c + d_h^R \approx 4.5\, d_h$	**Strong**

Notation: n_h = query heads, g = KV groups, d_h = head dimension, d_c = MLA latent dimension, d_h^R = RoPE head dimension.

Key insight: MQA minimizes cache but quality suffers. GQA is the practical middle ground. MLA achieves the smallest cache *while maintaining or exceeding MHA quality* by learning the compression.

Source: DeepSeek V2 [Liu et al.2024].

Appendix B: Multi-Head Latent Attention (MLA) — Intuition

GQA compresses by *discarding* — reduce the number of distinct KV heads. All queries in a group share one K, V pair. Simple, but quality degrades as group size grows (information is thrown away).

Appendix B: Multi-Head Latent Attention (MLA) — Intuition

GQA compresses by *discarding* — reduce the number of distinct KV heads. All queries in a group share one K, V pair. Simple, but quality degrades as group size grows (information is thrown away).

Can we compress by *learning* instead?

Key insight [Liu et al.2024]: If K and V across all heads are approximately low-rank, we can learn a small latent vector c_t^{KV} that captures most of the information. Cache the compressed version; decompress at attention time.

- GQA: discard heads $\Rightarrow$ fixed structure, quality loss.
- MLA: learn a compression map $\Rightarrow$ no fixed structure, quality can be preserved.

Appendix B: Multi-Head Latent Attention (MLA) — Intuition

GQA compresses by *discarding* — reduce the number of distinct KV heads. All queries in a group share one K, V pair. Simple, but quality degrades as group size grows (information is thrown away).

Can we compress by *learning* instead?

Key insight [Liu et al.2024]: If K and V across all heads are approximately low-rank, we can learn a small latent vector c_t^{KV} that captures most of the information. Cache the compressed version; decompress at attention time.

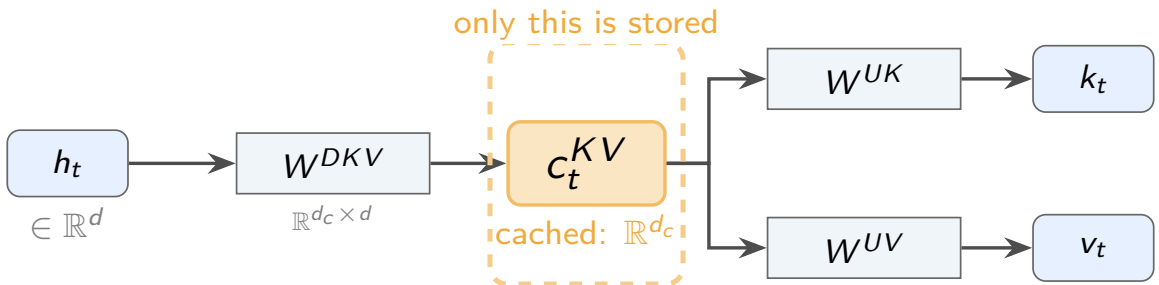
- GQA: discard heads $\Rightarrow$ fixed structure, quality loss.
- MLA: learn a compression map $\Rightarrow$ no fixed structure, quality can be preserved.

Tradeoff: MLA requires learning W^{DKV}, W^{UK}, W^{UV} during pretraining and adds up-projection compute at inference. GQA is simpler to implement and also effective.

Appendix B: MLA — Mechanism and Dataflow

Three steps. Given input $h_t \in \mathbb{R}^d$ at position t:

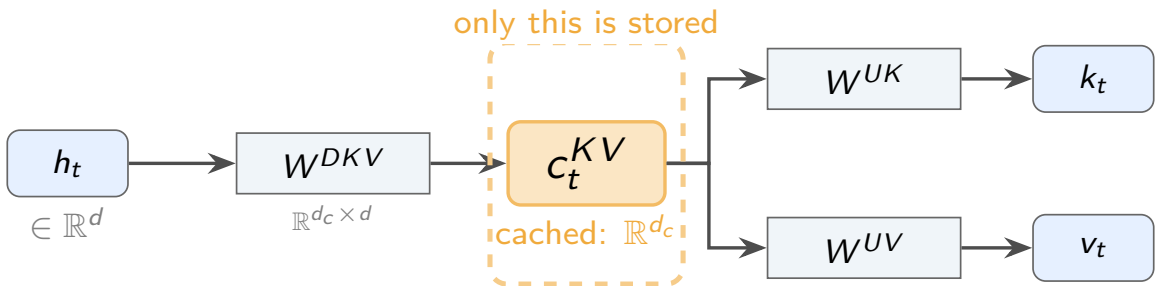
1. **Down-project:** $c_t^{KV} = W^{DKV} h_t \in \mathbb{R}^{d_c}$, $d_c \ll n_h d_h$.
2. **Cache** only c_t^{KV} (a d_c-dim vector).
3. **Up-project:** $k_t = W^{UK} c_t^{KV}$, $v_t = W^{UV} c_t^{KV} \in \mathbb{R}^{n_h d_h}$.



Appendix B: MLA — Mechanism and Dataflow

Three steps. Given input $h_t \in \mathbb{R}^d$ at position t:

1. **Down-project:** $c_t^{KV} = W^{DKV} h_t \in \mathbb{R}^{d_c}$, $d_c \ll n_h d_h$.
2. **Cache** only c_t^{KV} (a d_c-dim vector).
3. **Up-project:** $k_t = W^{UK} c_t^{KV}$, $v_t = W^{UV} c_t^{KV} \in \mathbb{R}^{n_h d_h}$.



DeepSeek V3: $d_c = 512$ vs. $n_h d_h = 16{,}384$ — a **32**× reduction in what is cached per token per layer. MLA *matches or exceeds* MHA quality in ablations.

GQA discards. MLA learns to compress.

Appendix B: Full MLA Derivation (Query Compression)

Query compression (training only — not cached):

$$c_t^Q = W^{DQ} h_t \in \mathbb{R}^{d_c^Q}, \qquad q_t = W^{UQ} c_t^Q \in \mathbb{R}^{n_h d_h}$$

RoPE integration with MLA: RoPE cannot be applied directly to the compressed KV latent (rotation does not commute with the up-projection). Solution: concatenate a small separate RoPE'd component:

$$k_t = \big[\underbrace{W^{UK} c_t^{KV}}_{\text{content}} \;\|\; \underbrace{\text{RoPE}(W^{KR} h_t)}_{\text{position}}\big]$$

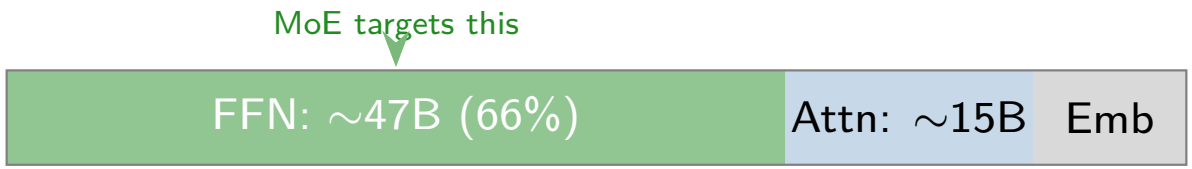
The first part carries content; the second carries position. Only c_t^{KV} is cached; $W^{KR} h_t$ is recomputed.

Appendix C: FFN Dominance and Uniform Activation

Recall: In a standard transformer, the FFN sublayer is:

$$\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2$$

with $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$.

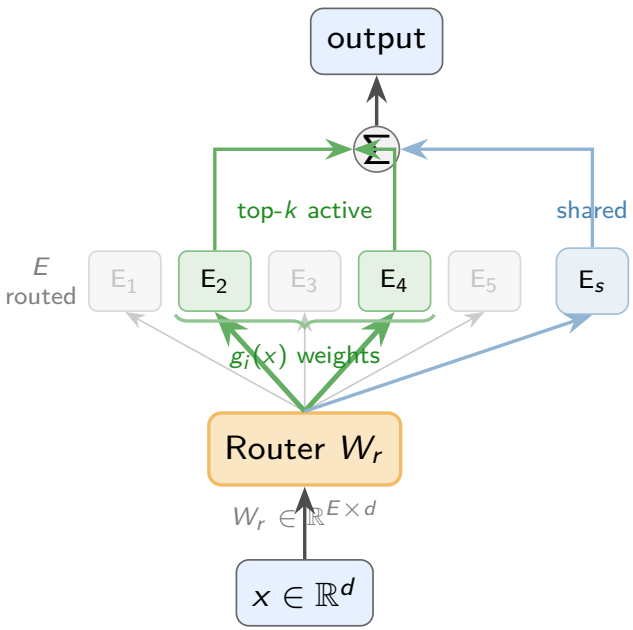


Appendix C: Mixture of Experts — Mechanism

The question: Can we increase capacity (total parameters) *without* proportionally increasing the compute cost per token?

Answer: Mixture of Experts (MoE). Replace the single dense FFN with E *expert* FFNs plus a learned router. Each token is routed to only $k \ll E$ experts.

$$\mathrm{MoE}(x) = \sum_{i \in \mathrm{TopK}(g(x),k)} g_i(x) \cdot \mathrm{FFN}_i(x), \qquad g(x) = \mathrm{softmax}(W_r x) \in \mathbb{R}^E$$

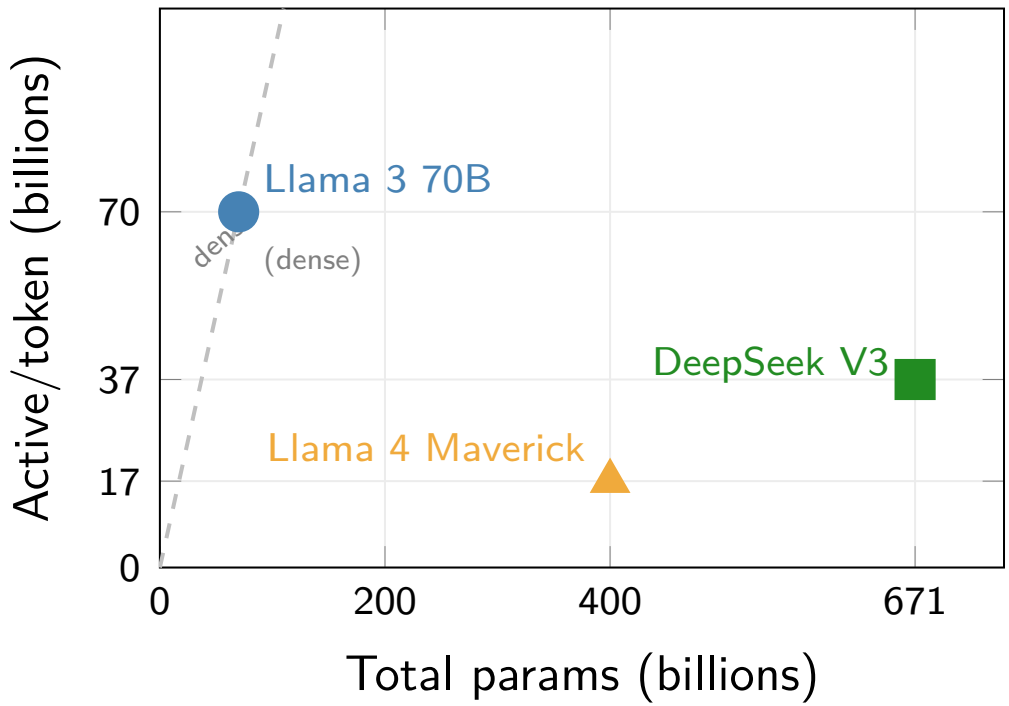


Step by step:

1. Token x enters.
2. Router W_r computes scores $g(x) = \mathrm{softmax}(W_r x)$.
3. Select top-k experts by score.
4. Run token through only those k expert FFNs.
5. Weight outputs by router

Appendix C: MoE — Total vs. Active Parameters

Dense models: active params = total params (on the diagonal).
MoE models: decouple total capacity from per-token compute.



Data requirement scales with *active parameters per token*, not total parameters [Fedus et al.2022]. DeepSeek V3: 671B total, 37B active — training cost comparable to a 37B dense model.

Appendix C: MoE — Training and Inference Efficiency

Does MoE increase training cost proportionally to total parameters?

No. Only k out of E experts are updated per token. The gradient flows *only* through the k active expert FFNs; the remaining $E - k$ experts receive zero gradient for that token.

Per-token compute: $\sim k/E$ of what a dense model with the same total parameters would require.

Appendix C: MoE — Training and Inference Efficiency

Does MoE increase training cost proportionally to total parameters?

No. Only k out of E experts are updated per token. The gradient flows *only* through the k active expert FFNs; the remaining $E - k$ experts receive zero gradient for that token.

Per-token compute: $\sim k/E$ of what a dense model with the same total parameters would require.

Training efficiency concern: Within a mini-batch, tokens with similar content tend to route to the same experts. This is actually *desirable* for GPU utilization — expert activations can be batched. The problem arises when load is *imbalanced*: some experts receive many tokens (high utilization) while others receive few (wasted memory).
This is the **load-balancing problem**.

Appendix C: MoE Training Challenge — Load Balancing

The problem: The router can collapse — sending most tokens to a few "popular" experts while others go unused. This wastes capacity.

Appendix C: MoE Training Challenge — Load Balancing

The problem: The router can collapse — sending most tokens to a few "popular" experts while others go unused. This wastes capacity.

Solution 1: Auxiliary loss [Shazeer et al.2017, Fedus et al.2022]. Add a penalty term to the training loss that encourages equal load across experts:

$$\mathcal{L}_{\text{aux}} = \alpha \sum_{i=1}^{E} f_i \cdot p_i$$

where f_i = fraction of tokens routed to expert i, p_i = average router probability for expert i. Minimized when load is uniform. But the auxiliary loss can distort the primary training signal.

Appendix C: MoE Training Challenge — Load Balancing

The problem: The router can collapse — sending most tokens to a few "popular" experts while others go unused. This wastes capacity.

Solution 1: Auxiliary loss [Shazeer et al.2017, Fedus et al.2022]. Add a penalty term to the training loss that encourages equal load across experts:

$$\mathcal{L}_{\text{aux}} = \alpha \sum_{i=1}^{E} f_i \cdot p_i$$

where f_i = fraction of tokens routed to expert i, p_i = average router probability for expert i. Minimized when load is uniform. But the auxiliary loss can distort the primary training signal.

Solution 2: Auxiliary-loss-free balancing [DeepSeek-AI2024]. Add a *learnable bias* b_i to each expert's router logit: $\tilde{g}_i(x) = (W_r x)_i + b_i$. The bias is updated via running load statistics: increase b_i if expert i is underutilized, decrease if overutilized. The routing *signal* (which expert is best for this

Appendix C: MoE — Expert Collapse

Expert collapse: All experts converge to the same function $\Rightarrow$ no benefit from sparsity.

The **shared expert** addresses this: it handles common-knowledge computation (syntax, frequent patterns), freeing routed experts to specialize.

Appendix C: MoE — Expert Collapse

Expert collapse: All experts converge to the same function $\Rightarrow$ no benefit from sparsity.

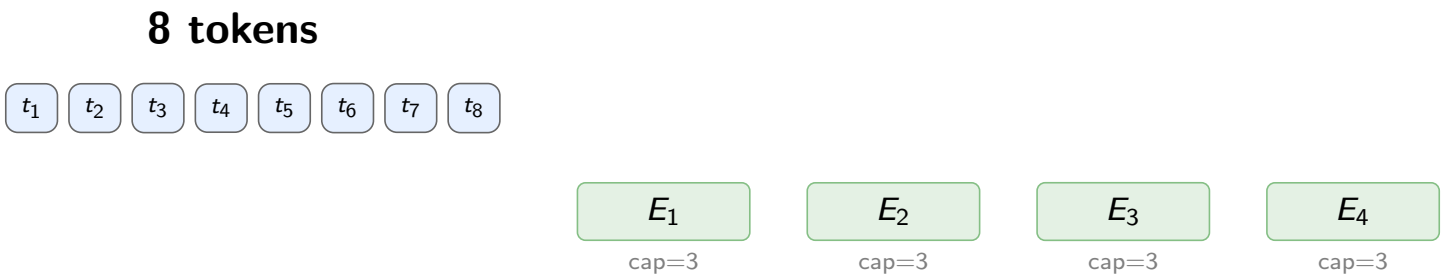
The **shared expert** addresses this: it handles common-knowledge computation (syntax, frequent patterns), freeing routed experts to specialize.

Routing discontinuity: The top-k selection function is non-differentiable — how do gradients flow through the router?

The **straight-through estimator (STE)** is used: treat top-k as differentiable in the backward pass, ignoring the discrete mask. This is a biased gradient estimator, but works well in practice.

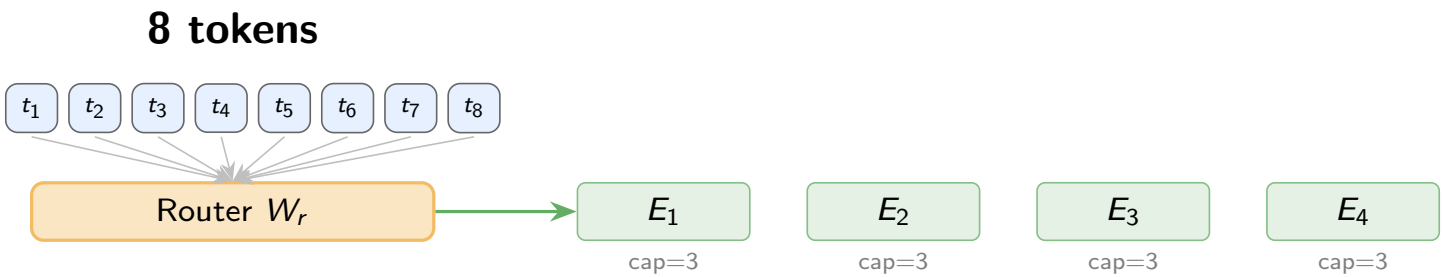
Appendix C: MoE — Capacity Constraints and Token Dropping

With fixed expert capacity (required for batched computation on GPUs), tokens routed to a full expert are *dropped*.



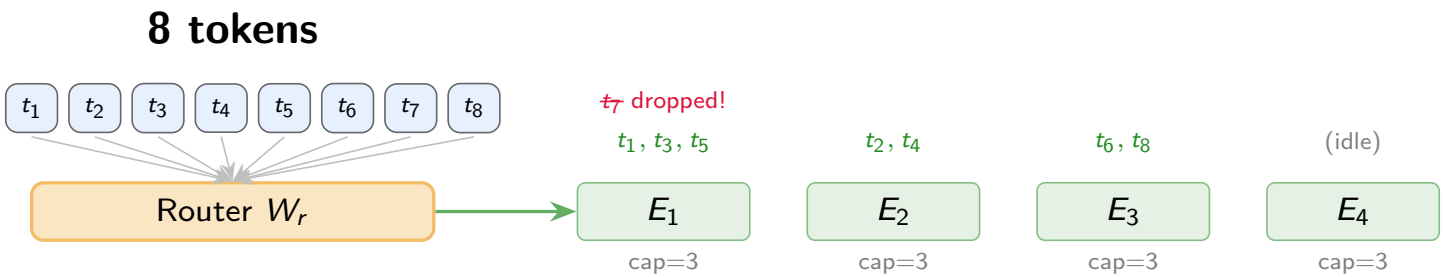
Appendix C: MoE — Capacity Constraints and Token Dropping

With fixed expert capacity (required for batched computation on GPUs), tokens routed to a full expert are *dropped*.



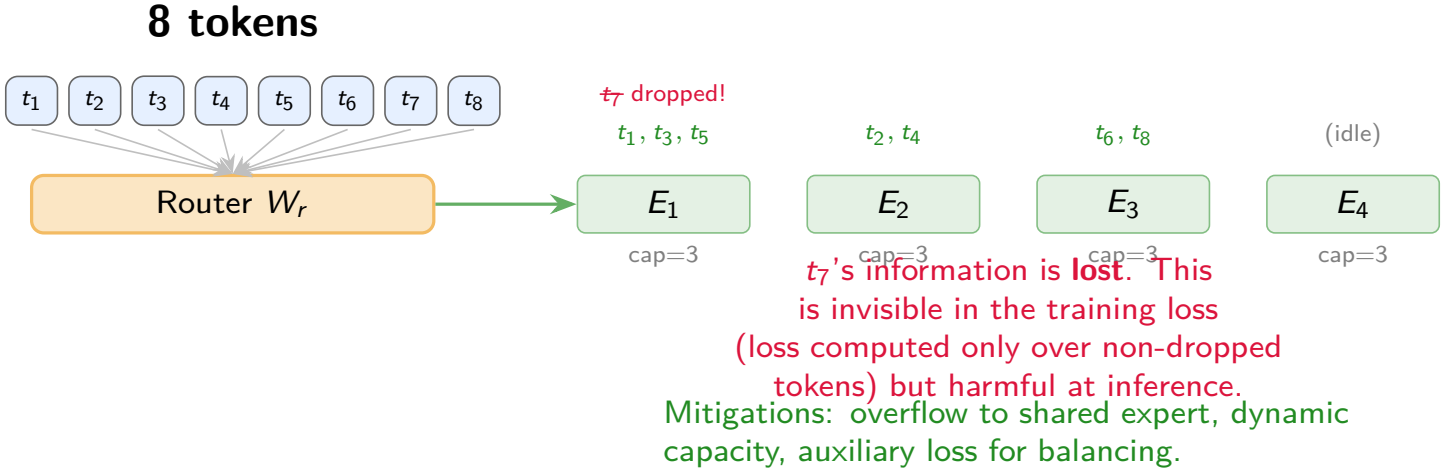
Appendix C: MoE — Capacity Constraints and Token Dropping

With fixed expert capacity (required for batched computation on GPUs), tokens routed to a full expert are *dropped*.



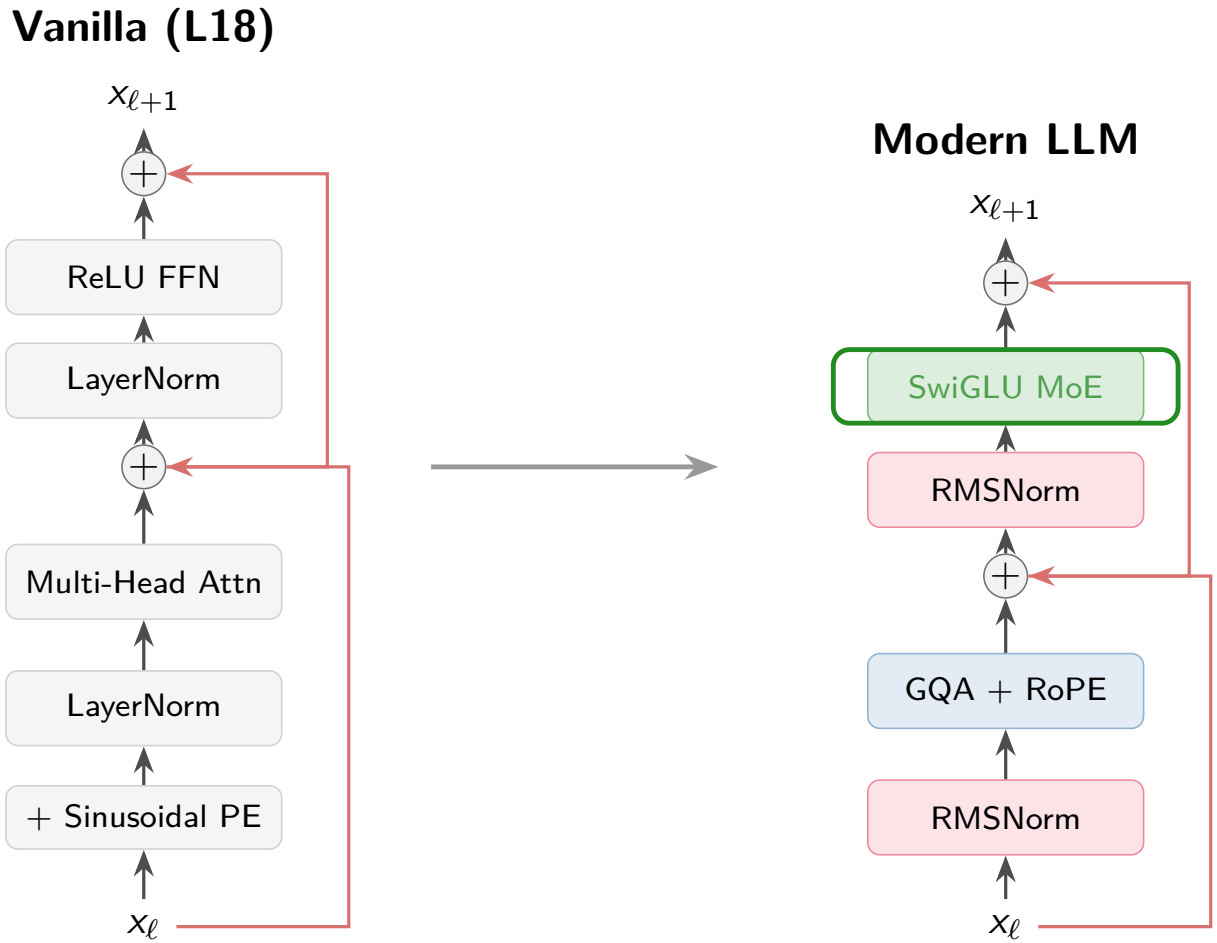
Appendix C: MoE — Capacity Constraints and Token Dropping

With fixed expert capacity (required for batched computation on GPUs), tokens routed to a full expert are *dropped*.



In distributed training, MoE requires all-to-all communication to route tokens to experts on different devices — a systems bottleneck constraining practical expert count.

Appendix C: Progress — After All Bottlenecks



B4 fix: Dense FFN → SwiGLU MoE (replace single FFN with E expert FFNs + learned router; each token activates $k \ll E$ experts; total capacity $\sim E\times$ dense, per-token cost $\sim k\times$ dense).

Every component of the vanilla block has now been replaced.
Nothing in the modern architecture is arbitrary — each change traces to

Appendix C: Pause & Check — MoE

Q1. DeepSeek V3 has 256 routed experts with top-1 routing, plus 1 shared expert. What fraction of total FFN parameters are activated per token?

Appendix C: Pause & Check — MoE

Q1. DeepSeek V3 has 256 routed experts with top-1 routing, plus 1 shared expert. What fraction of total FFN parameters are activated per token?

Per token: 1 routed expert + 1 shared expert = 2 expert FFNs activated. Total FFN capacity: 256 routed + 1 shared = 257 expert FFNs. Fraction activated: $2/257 \approx 0.78\%$ of total FFN parameters.

Appendix C: Pause & Check — MoE

Q1. DeepSeek V3 has 256 routed experts with top-1 routing, plus 1 shared expert. What fraction of total FFN parameters are activated per token?

Per token: 1 routed expert + 1 shared expert = 2 expert FFNs activated. Total FFN capacity: 256 routed + 1 shared = 257 expert FFNs. Fraction activated: $2/257 \approx 0.78\%$ of total FFN parameters.

Q2. If each routed expert FFN has parameter count $3\, d_{ff}^{(e)} \times d$ (SwiGLU: three matrices), what is the active parameter count per token relative to a dense model with $d_{ff} = E \cdot d_{ff}^{(e)}$?

Appendix C: Pause & Check — MoE

Q1. DeepSeek V3 has 256 routed experts with top-1 routing, plus 1 shared expert. What fraction of total FFN parameters are activated per token?

Per token: 1 routed expert + 1 shared expert = 2 expert FFNs activated. Total FFN capacity: 256 routed + 1 shared = 257 expert FFNs. Fraction activated: $2/257 \approx 0.78\%$ of total FFN parameters.

Q2. If each routed expert FFN has parameter count $3\, d_{ff}^{(e)} \times d$ (SwiGLU: three matrices), what is the active parameter count per token relative to a dense model with $d_{ff} = E \cdot d_{ff}^{(e)}$?

Dense model: $3 \times (E \cdot d_{ff}^{(e)}) \times d = 3E\, d_{ff}^{(e)} d$.
MoE active (1 routed + 1 shared): $2 \times 3\, d_{ff}^{(e)} \times d = 6\, d_{ff}^{(e)} d$.
Ratio: $6\, d_{ff}^{(e)} d \;/\; 3E\, d_{ff}^{(e)} d = 2/E$. For $E = 256$: active params $\approx 0.78\%$ of the equivalent dense model.

Appendix D: Scaling Laws — Experimental Setup

The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

Appendix D: Scaling Laws — Experimental Setup

The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

How this was established:

- **What was measured:** Cross-entropy test loss on held-out web text (WebText2 for Kaplan et al.; MassiveText for Hoffmann et al.).

Appendix D: Scaling Laws — Experimental Setup

The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

How this was established:

- **What was measured:** Cross-entropy test loss on held-out web text (WebText2 for Kaplan et al.; MassiveText for Hoffmann et al.).
- **Models:** Decoder-only transformers ranging from $\sim$1M to $\sim$10B parameters, with varying depths, widths, and data sizes.

Appendix D: Scaling Laws — Experimental Setup

The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

How this was established:

- **What was measured:** Cross-entropy test loss on held-out web text (WebText2 for Kaplan et al.; MassiveText for Hoffmann et al.).
- **Models:** Decoder-only transformers ranging from $\sim$1M to $\sim$10B parameters, with varying depths, widths, and data sizes.
- **Fitting:** Log-log regression. Each scaling variable is varied while the others are held near-optimal, revealing straight lines in log-log space.

Appendix D: Scaling Laws — Experimental Setup

The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

How this was established:

- **What was measured:** Cross-entropy test loss on held-out web text (WebText2 for Kaplan et al.; MassiveText for Hoffmann et al.).
- **Models:** Decoder-only transformers ranging from $\sim$1M to $\sim$10B parameters, with varying depths, widths, and data sizes.
- **Fitting:** Log-log regression. Each scaling variable is varied while the others are held near-optimal, revealing straight lines in log-log space.
- **Scope:** *Pre-training only* (no fine-tuning, no RLHF). Assumes near-optimal hyperparameters (learning rate schedule, batch size).

Appendix D: Scaling Laws — Experimental Setup

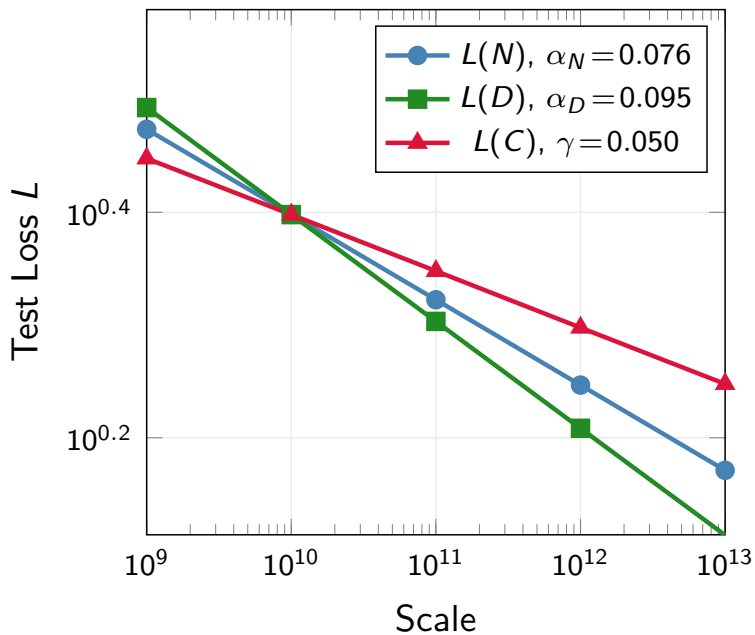
The discovery [Kaplan et al.2020]**:** When decoder-only transformers are trained on large text corpora, the test loss (cross-entropy on held-out text) follows remarkably clean **power laws** in three quantities: model size N, dataset size D, and training compute C.

How this was established:

- **What was measured:** Cross-entropy test loss on held-out web text (WebText2 for Kaplan et al.; MassiveText for Hoffmann et al.).
- **Models:** Decoder-only transformers ranging from $\sim$1M to $\sim$10B parameters, with varying depths, widths, and data sizes.
- **Fitting:** Log-log regression. Each scaling variable is varied while the others are held near-optimal, revealing straight lines in log-log space.
- **Scope:** *Pre-training only* (no fine-tuning, no RLHF). Assumes near-optimal hyperparameters (learning rate schedule, batch size).
- **Generality:** Power-law exponents are consistent across model families — they depend on data distribution and architecture *class* (decoder-only), not on specific hyperparameters or training algorithms.

Appendix D: Scaling Laws — The Empirical Result

$$L(N) \approx \frac{A}{N^{\alpha_N}}, \qquad L(D) \approx \frac{B}{D^{\alpha_D}}, \qquad L(C) \approx C^{-\gamma}$$



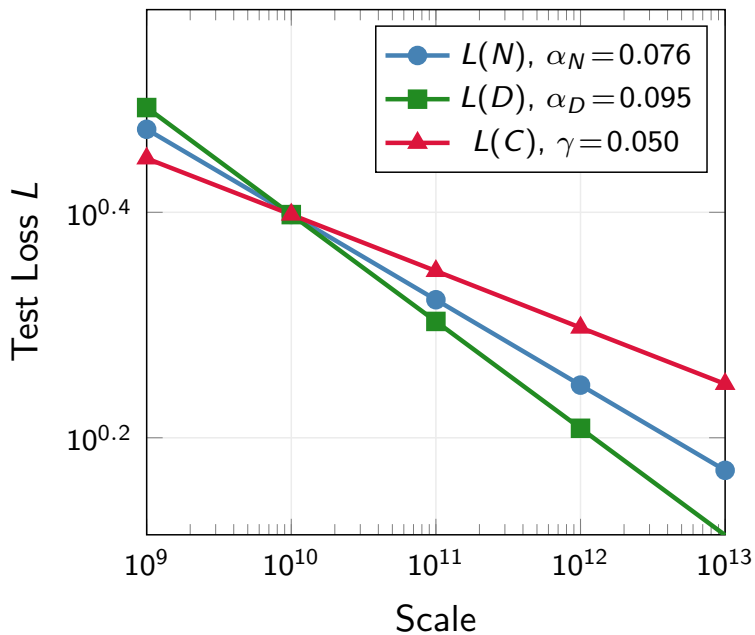
Key takeaways (what decreases loss most?):

1. **Data** ($\alpha_D = 0.095$): Steepest slope — doubling the training corpus decreases loss the most.

"Parameters" concretely: Total trainable weights — set by layers L, embedding dim d, heads n_h, FFN width d_{ff}.

Appendix D: Scaling Laws — The Empirical Result

$$L(N) \approx \frac{A}{N^{\alpha_N}}, \qquad L(D) \approx \frac{B}{D^{\alpha_D}}, \qquad L(C) \approx C^{-\gamma}$$



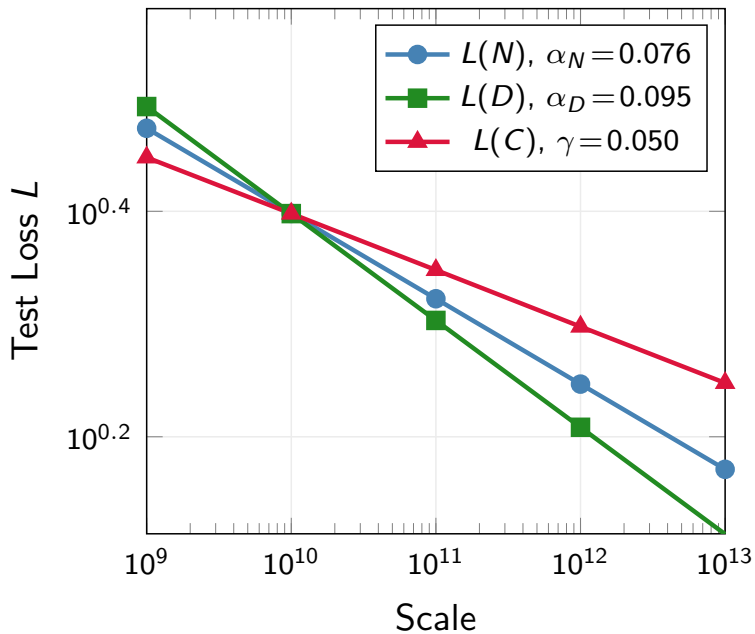
Key takeaways (what decreases loss most?):

1. **Data** ($\alpha_D = 0.095$): Steepest slope — doubling the training corpus decreases loss the most.
2. **Parameters** ($\alpha_N = 0.076$): More layers or wider d is the next most effective lever.

"Parameters" concretely: Total trainable weights — set by layers L, embedding dim d, heads n_h, FFN width d_{ff}.

Appendix D: Scaling Laws — The Empirical Result

$$L(N) \approx \frac{A}{N^{\alpha_N}}, \qquad L(D) \approx \frac{B}{D^{\alpha_D}}, \qquad L(C) \approx C^{-\gamma}$$



Key takeaways (what decreases loss most?):

1. **Data** ($\alpha_D = 0.095$): Steepest slope — doubling the training corpus decreases loss the most.
2. **Parameters** ($\alpha_N = 0.076$): More layers or wider d is the next most effective lever.
3. **Compute** ($\gamma = 0.050$): Shallowest. More FLOPs (for fixed N, D: more epochs) has the smallest marginal effect.

"Parameters" concretely: Total trainable weights — set by layers L, embedding dim d, heads n_h, FFN width d_{ff}.

Appendix D: From Empirical Fits to the Combined Loss Formula

The individual power laws $L(N)$, $L(D)$ describe behavior when one variable is varied while the other is unlimited. When both are finite:

$$L(N, D) = \underbrace{\frac{A}{N^{\alpha_N}}}_{\text{model too small}} + \underbrace{\frac{B}{D^{\alpha_D}}}_{\text{data too small}} + \underbrace{L_\infty}_{\text{irreducible}}$$

Appendix D: From Empirical Fits to the Combined Loss Formula

The individual power laws $L(N)$, $L(D)$ describe behavior when one variable is varied while the other is unlimited. When both are finite:

$$L(N, D) = \underbrace{\frac{A}{N^{\alpha_N}}}_{\text{model too small}} + \underbrace{\frac{B}{D^{\alpha_D}}}_{\text{data too small}} + \underbrace{L_\infty}_{\text{irreducible}}$$

Why additive? The two error sources are approximately *independent*: underfitting due to limited model capacity and underfitting due to limited data.

Appendix D: From Empirical Fits to the Combined Loss Formula

The individual power laws $L(N)$, $L(D)$ describe behavior when one variable is varied while the other is unlimited. When both are finite:

$$L(N, D) = \underbrace{\frac{A}{N^{\alpha_N}}}_{\text{model too small}} + \underbrace{\frac{B}{D^{\alpha_D}}}_{\text{data too small}} + \underbrace{L_\infty}_{\text{irreducible}}$$

Why additive? The two error sources are approximately *independent*: underfitting due to limited model capacity and underfitting due to limited data.

What is L_∞? The **irreducible loss** — the entropy of natural language itself. Even with infinite parameters and infinite data, the model cannot predict language perfectly because language is inherently stochastic. For English web text, $L_\infty \approx 1.69$ nats/token [Hoffmann et al.2022].

Appendix D: The Compute Constraint: $C \approx 6ND$

Why $C \approx 6ND$? Each training token requires approximately $6N$ floating-point operations: $2N$ for the forward pass (one multiply-add per parameter per token) and $4N$ for the backward pass (roughly $2\times$ forward). Over D tokens: $C \approx 6ND$ FLOPs total.

Appendix D: The Compute Constraint: $C \approx 6ND$

Why $C \approx 6ND$? Each training token requires approximately $6N$ floating-point operations: $2N$ for the forward pass (one multiply-add per parameter per token) and $4N$ for the backward pass (roughly $2\times$ forward). Over D tokens: $C \approx 6ND$ FLOPs total.

Why don't attention, embeddings, etc. appear separately? At the scale of frontier models, FFN parameters dominate ($\sim$2/3 of N), and the $6N$ approximation absorbs attention FLOPs as a constant factor. The formula is accurate to within $\sim$10% for models above $\sim$1B parameters.

Appendix D: The Compute Constraint: $C \approx 6ND$

Why $C \approx 6ND$? Each training token requires approximately $6N$ floating-point operations: $2N$ for the forward pass (one multiply-add per parameter per token) and $4N$ for the backward pass (roughly $2\times$ forward). Over D tokens: $C \approx 6ND$ FLOPs total.

Why don't attention, embeddings, etc. appear separately? At the scale of frontier models, FFN parameters dominate ($\sim$2/3 of N), and the $6N$ approximation absorbs attention FLOPs as a constant factor. The formula is accurate to within $\sim$10% for models above $\sim$1B parameters.

Does this count one epoch or multiple? D counts *total tokens seen*, regardless of epochs. Training for 2 epochs on a dataset of size D_0 means $D = 2D_0$. **Modern LLMs typically train for $\sim$1 epoch on unique data** [Muennighoff et al.2024] — repeating data beyond $\sim$4 epochs shows diminishing returns and potential degradation.

Appendix D: Compute-Optimal Scaling (Chinchilla)

Fitted values [Hoffmann et al.2022]**:**

$$A = 406.4, \quad B = 410.7, \quad \alpha_N = 0.34, \quad \alpha_D = 0.28, \quad L_\infty = 1.69$$

Optimization problem: Minimize
$L(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$ subject to $6ND = C$.

Appendix D: Compute-Optimal Scaling (Chinchilla)

Fitted values [Hoffmann et al.2022]**:**

$$A = 406.4, \quad B = 410.7, \quad \alpha_N = 0.34, \quad \alpha_D = 0.28, \quad L_\infty = 1.69$$

Optimization problem: Minimize
$L(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$ subject to $6ND = C$.

Result (Lagrange multipliers):

$$N^* \propto C^{\alpha_D/(\alpha_N+\alpha_D)} \approx C^{0.45}, \qquad D^* \propto C^{\alpha_N/(\alpha_N+\alpha_D)} \approx C^{0.55}$$

Since $\alpha_N \approx \alpha_D$: **scale N and D roughly equally**. Rule of thumb: ≈ 20 tokens per parameter.

Appendix D: Compute-Optimal Scaling (Chinchilla)

Fitted values [Hoffmann et al.2022]**:**

$$A = 406.4, \quad B = 410.7, \quad \alpha_N = 0.34, \quad \alpha_D = 0.28, \quad L_\infty = 1.69$$

Optimization problem: Minimize $L(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$ subject to $6ND = C$.

Result (Lagrange multipliers):

$$N^* \propto C^{\alpha_D/(\alpha_N+\alpha_D)} \approx C^{0.45}, \qquad D^* \propto C^{\alpha_N/(\alpha_N+\alpha_D)} \approx C^{0.55}$$

Since $\alpha_N \approx \alpha_D$: **scale N and D roughly equally**. Rule of thumb: ≈ 20 tokens per parameter.

Impact: Chinchilla (70B params, 1.4T tokens) matched Gopher (280B params, 300B tokens) using 4$\times$ less compute — by allocating more budget to data [Hoffmann et al.2022].

Appendix D: Compute-Optimal Scaling (Chinchilla)

Fitted values [Hoffmann et al.2022]**:**

$$A = 406.4, \quad B = 410.7, \quad \alpha_N = 0.34, \quad \alpha_D = 0.28, \quad L_\infty = 1.69$$

Optimization problem: Minimize $L(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$ subject to $6ND = C$.

Result (Lagrange multipliers):

$$N^* \propto C^{\alpha_D/(\alpha_N+\alpha_D)} \approx C^{0.45}, \qquad D^* \propto C^{\alpha_N/(\alpha_N+\alpha_D)} \approx C^{0.55}$$

Since $\alpha_N \approx \alpha_D$: **scale N and D roughly equally**. Rule of thumb: ≈ 20 tokens per parameter.

Impact: Chinchilla (70B params, 1.4T tokens) matched Gopher (280B params, 300B tokens) using $4\times$ less compute — by allocating more budget to data [Hoffmann et al.2022].

Post-Chinchilla revision: Chinchilla scaling is optimal for minimizing training loss with a given compute budget C, but inference cost scales with N. In practice, one often trains smaller models on more data with the same C (so a higher tokens-per-parameter ratio) to achieve comparable quality at significantly lower inference cost [Touvron et al.2023].

Appendix E: Why RoPE Does Not Generalize for Free

The apparent promise: $\langle R_{t_q} q,\, R_{t_k} k \rangle = q^\top R_{t_k - t_q}\, k$ depends only on the *relative* offset $t_k - t_q$. So can't a trained model just be used at longer contexts?

Appendix E: Why RoPE Does Not Generalize for Free

The apparent promise: $\langle R_{t_q} q,\, R_{t_k} k \rangle = q^\top R_{t_k - t_q}\, k$ depends only on the *relative* offset $t_k - t_q$. So can't a trained model just be used at longer contexts?

No — rotation angles exit the training distribution.
Each frequency $\theta_i = \text{base}^{-2i/d_h}$ defines the angle $t_q \theta_i$ that dimension pair i is rotated at query position t_q.

- **High-frequency pairs** (i small, θ_i large, short period $2\pi/\theta_i$): many full rotations occur within T_{train}, so all angles $[0, 2\pi]$ are well-covered. These dimensions generalize.
- **Low-frequency pairs** ($i \approx d_h/2$, $\theta_i \approx 0$, long period): within T_{train} the angle $t_q \theta_i$ barely advances past 0. Beyond T_{train}, the rotation sweeps into angles **never seen during training**.

Appendix E: Why RoPE Does Not Generalize for Free

The apparent promise: $\langle R_{t_q} q,\, R_{t_k} k \rangle = q^\top R_{t_k - t_q}\, k$ depends only on the *relative* offset $t_k - t_q$. So can't a trained model just be used at longer contexts?

No — rotation angles exit the training distribution.
Each frequency $\theta_i = \text{base}^{-2i/d_h}$ defines the angle $t_q \theta_i$ that dimension pair i is rotated at query position t_q.

- **High-frequency pairs** (i small, θ_i large, short period $2\pi/\theta_i$): many full rotations occur within T_{train}, so all angles $[0, 2\pi]$ are well-covered. These dimensions generalize.
- **Low-frequency pairs** ($i \approx d_h/2$, $\theta_i \approx 0$, long period): within T_{train} the angle $t_q \theta_i$ barely advances past 0. Beyond T_{train}, the rotation sweeps into angles **never seen during training**.

Consequence: At positions $t > T_{\text{train}}$, the rotated query/key vectors $R_t q$ lie in regions of embedding space the model was never trained to attend over. Attention scores collapse or diverge — performance degrades sharply.

Appendix E: Frequency-Scaling Fixes

Strategy: Rescale frequencies so positions up to T_{target} map to the same rotation-angle range seen during training, then fine-tune briefly.

1. **Position interpolation:** Replace $t \to t/s$, $s = T_{\text{target}}/T_{\text{train}}$.
   - Uniformly compresses all frequencies. **Over-compresses** high-frequency dims that already generalize — harms quality.
2. **NTK-aware base scaling** [Peng et al.2023]: base $\to$ base $\cdot\, s^{d_h/(d_h-2)}$.
   - NTK motivation: high-frequency dims (learned first) stay unchanged; low-frequency dims get their period extended to T_{target}. Equivalent to non-uniform per-dimension interpolation.
3. **YaRN** [Peng et al.2023]: three-zone non-uniform scaling.
   - High-frequency ($i < \beta$): no change. Low-frequency ($i > \alpha$): full linear interpolation. Middle: blended. Temperature $\sqrt{1/t}$ preserves attention entropy.

All three require short fine-tuning ($\sim$1K–10K steps) at the target length (e.g. Llama 3.1: 8K $\to$ 128K with NTK-style base scaling [Dubey et al.2024]).

Appendix F: Kaplan vs. Chinchilla — Fitted Exponents

	α_N	α_D	γ	**Recommendation**
Kaplan et al. (2020)	0.076	0.095	0.050	Scale N aggressively; data is cheap
Chinchilla (2022)	0.34	0.28	—	Scale N and D roughly equally

Appendix F: Kaplan vs. Chinchilla — Fitted Exponents

	α_N	α_D	γ	**Recommendation**
Kaplan et al. (2020)	0.076	0.095	0.050	Scale N aggressively; data is cheap
Chinchilla (2022)	0.34	0.28	—	Scale N and D roughly equally

Why the values differ so dramatically:

- **Different fitting methodology:** Kaplan varied one variable at a time with the other "near-optimal." Chinchilla systematically swept both N and D jointly.
- **Undertrained models in Kaplan:** Many of Kaplan's models were not trained to convergence (not enough tokens for their size), biasing the fits toward overestimating the value of more parameters.
- **Broader compute range:** Chinchilla used a broader range of compute budgets and more careful hyperparameter tuning.

Appendix F: Which Scaling Law to Use?

Why Chinchilla is standard: The Chinchilla prediction (scale N and D equally) was validated by the Chinchilla model itself: 70B params trained on 1.4T tokens matched 280B Gopher trained on 300B tokens. Subsequent models (Llama, Mistral, Gemma) all follow Chinchilla-style data scaling. The Kaplan recommendation (few tokens, many parameters) produces undertrained models.

Appendix F: Which Scaling Law to Use?

Why Chinchilla is standard: The Chinchilla prediction (scale N and D equally) was validated by the Chinchilla model itself: 70B params trained on 1.4T tokens matched 280B Gopher trained on 300B tokens. Subsequent models (Llama, Mistral, Gemma) all follow Chinchilla-style data scaling. The Kaplan recommendation (few tokens, many parameters) produces undertrained models.

What Kaplan adds that Chinchilla does not:

- Individual scaling laws in N, D, C separately (Chinchilla focuses on the joint optimum).
- The observation that scaling exponents are consistent across architecture variants within the decoder-only class.
- The $L(C) \sim C^{-\gamma}$ curve, useful for rough compute budgeting.

Shared assumptions: Both assume next-token prediction loss on unique (non-repeated) data with near-optimal hyperparameters. Neither accounts for data quality variation, repeated data, or downstream task performance.

Appendix G: Architecture Choices Shape the Full Training Pipeline

Each bottleneck fix has consequences *beyond* pretraining:

- **Pre-norm (RMSNorm)** **B1**
  Makes 80+ layer training stable; also stabilizes fine-tuning and RLHF.
- **KV cache (GQA/MLA)** **B3**
  Affects RLHF cost: cheaper inference $\Rightarrow$ more parallel generations $\Rightarrow$ better policy.
- **MoE routing** **B4**
  Router is learned on pretraining data; fine-tuning can cause expert-load drift.
- **Autoregressive objective**
  MLE is necessary but *not sufficient* for useful assistants. Alignment (SFT, RLHF) is the subject of the next lecture.