How to train an LLM

Principes, Decisions, and Engineering

Li Shang
lishang@slai.edu.cn

Acknowledgement

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

Part of the course material was created by LLM itself.

Who wants to train an LLM?

–Tom Jerry

Yes, you can, 2-3 folks with 10-100 GPUs for a few months

–Tom Jerry

Whitening the journey from “we have a great dataset and GPUs” to “we built a model actually works”.

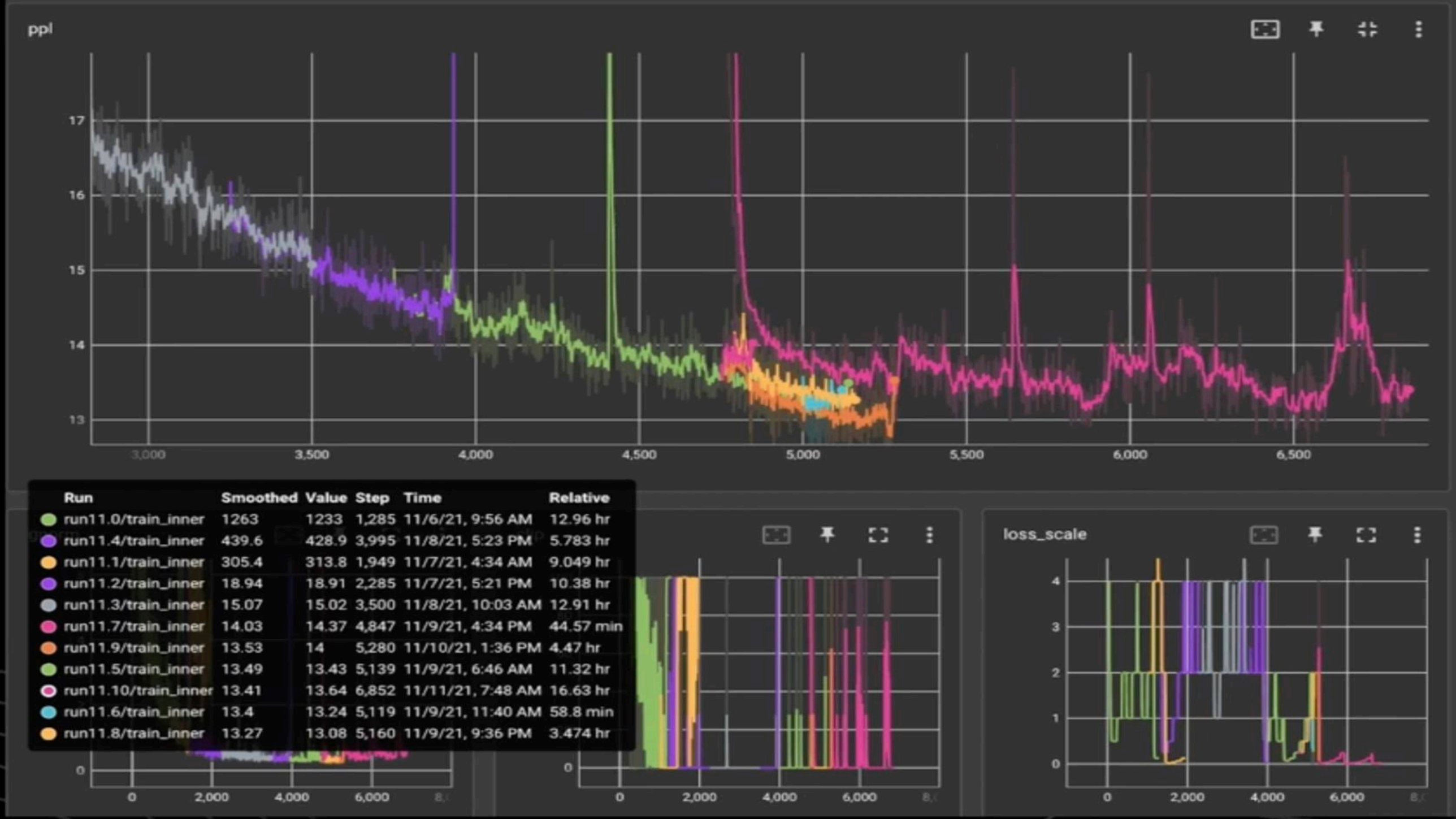
–Tom Jerry

OPT-175B: A true story

- OPT-175B was a large language model with 175 billion parameters, released by Meta AI in May 2022 as part of an effort to democratize access to large-scale models for research purposes.
- During the training process on 992 80GB A100 GPUs, the team encountered various real-world problems that contributed to instability.

OPT-175B: A true story

- **Hardware Failures:** The training was subject to a significant number of hardware failures in the compute cluster, which complicated the continuous training process.
- **Numerical Instabilities:** The team faced numerical instabilities, such as abrupt loss divergence or NaN (Not a Number) values.
- **Scaling Issues:** They observed limitations in transferring results from smaller-scale experiments to the massive scale of the 175B parameter model.
- **Debugging Difficulties:** Some incidents were "implicit failures" (e.g., a communication hang caused by non-deterministic CUDA errors) that were difficult to detect, diagnose, and reproduce, sometimes taking hours of manual effort to resolve.


ppl
17
16
15
14
13
3,000
3,500
4,000
4,500
5,000
5,500
6,000
6,500
Run Smoothed Value Step Time Relative
run11.0/train_inner 1263 1233 1,285 11/6/21, 9:56 AM 12.96 hr
run11.4/train_inner 439.6 428.9 3,995 11/8/21, 5:23 PM 5.783 hr
run11.1/train_inner 305.4 313.8 1,949 11/7/21, 4:34 AM 9.049 hr
run11.2/train_inner 18.94 18.91 2,285 11/7/21, 5:21 PM 10.38 hr
run11.3/train_inner 15.07 15.02 3,500 11/8/21, 10:03 AM 12.91 hr
run11.7/train_inner 14.03 14.37 4,847 11/9/21, 4:34 PM 44.57 min
run11.9/train_inner 13.53 14 5,280 11/10/21, 1:36 PM 4.47 hr
run11.5/train_inner 13.49 13.43 5,139 11/9/21, 6:46 AM 11.32 hr
run11.10/train_inner 13.41 13.64 6,852 11/11/21, 7:48 AM 16.63 hr
run11.6/train_inner 13.4 13.24 5,119 11/9/21, 11:40 AM 58.8 min
run11.8/train_inner 13.27 13.08 5,160 11/9/21, 9:36 PM 3.474 hr
0
0
2,000
4,000
6,000
loss_scale
4
3
2
1
0
0
2,000
4,000
6,000

Yes, building things at scale is hard

- Building high-quality data at scale.
- Orchestrating thousands of GPUs at scale.
- Reasoning and evaluating decisions throughout the process.

Schedule

- **Part 1:** Why train an LLM? Concepts, goals, and design decisions.
- **Part 2:** How to train an LLM? The technical pipeline and ablation framework.
- **Part 3:** Training, engineering reality, debugging, and post-training.

Part 1: Why train an LLM?

Concepts, goals, and design decisions

Before we train an LLM, we need to understand why people train LLMs and what decisions matter before training begins.

Motivation → Architecture → Data → Scaling

Part 2: How to train an LLM?

The technical pipeline & ablation framework

We learn how to design experiments, choose components, and prepare for full-scale training.

Ablations → Architecture choices → Data recipes → Optimizers → LR schedules → Stability

Part 3: Full-scale training

Engineering reality, debugging, and post-training

We learn what really happens at scale, and what makes an LLM useful after pretraining.

Distributed training → Infrastructure → Monitoring → Failures → Safety → SFT/DPO → Deployment.

Part 1

Introduction

What is an LLM?

- A LLM is a neural network trained to predict the next token in a sequence. Along the way, somehow, the model learns representations of language, reasoning, code, mathematics, and multimodal structure purely from token prediction.
- What makes them “large” is not just parameter count, but the amount of data, compute, and training discipline behind them.

Introduction

Why are we talking about training LLMs now?

LLMs have become central infrastructure

- LLMs are the foundation of AI-empowered tasks, e.g., search, copilots, robotics, and scientific discovery. We need to understand how training decisions affect task ability.

Training has become more accessible

- Open-source projects like Qwen, LLaMA, and Gemma show that high-quality LLMs can be trained outside Big Tech.

The ecosystem needs experts who understand fundamentals

- Students in AI today must understand not just how to use LLMs but how to design and train them, which involves scientific, engineering, and decision-making thinking.

Introduction

This is not a GPU tutorial, instead this is a thinking framework

Success comes from a systematic framework of decisions, validation, ablation, and monitoring—not a collection of engineering tricks.

- Why should we train an LLM at all?
- What decisions matter before training starts?
- What experiments validate our choices?
- How do we avoid catastrophic mistakes at scale?

Why train your own LLM?

Why does this model need to exist?

Research: You have a scientific question to answer.

Product: You have a practical need that existing models cannot satisfy.

Open source: You aim to fill a strategic gap in the open-source ecosystem.

Research motivation

Train an LLM to understand something fundamental

- Test a hypothesis about architecture (e.g., Does GQA improve long-context performance?)
- Investigate data effects (Does curriculum learning improve reasoning?)
- Explore optimization methods (How stable is training with Muon vs AdamW?)
- Study emergent behavior (At what scale does reasoning appear and why?)

Product motivation

Train an LLM if the requirements cannot be met by existing models

A. Domain expertise

- Medicine, law, finance, biology, engineering
- Existing models lack domain-knowledge or hallucinate too much

B. Data privacy / security

- Sensitive company data cannot be sent to proprietary models
- Full control over training and deployment is required

C. Hardware constraints

- Need an LLM that runs on edge devices or low-cost GPU clusters
- Require FP8/FP4, quantization friendliness, small KV cache, etc.

Open Source

Train an LLM to enrich the ecosystem

A. There is a missing open-source model with certain properties:

- Small size, high quality
- Long context
- Specific languages (e.g., Chinese, Arabic, Hindi)
- Efficient runtime (MLA, GQA, low-memory footprint)

B. Open-source reinforcement:

- Enables downstream instruction tuning, agents, robotics, multimodal systems
- Community adoption amplifies impact

Decision flow

Decide shall not made by the availability of GPUs

- First try prompting.
- If insufficient → Fine-tune an existing model.
- If still insufficient → Continue pretraining (domain adaptation).
- Only if all else fails → Train from scratch.

Now we have decided to train a model, what is next?

–Tom Jerry

Pick a training framework

–Tom Jerry

Training framework

Installation…

1. Installing CUDA
2. Setting up Python, PyTorch, Megatron-LM, and DeepSpeed
3. Preparing a dataset in the Megatron indexed format
4. Writing the Megatron + DeepSpeed training script
5. Launching multi-GPU training
6. Checking training results (loss, perplexity, inference)

Training framework

Pick the framework to support ablations and model training

Megatron-LM	Mature, feature-rich, battle-tested (used for Kimi-K2, Nemotron). Excellent performance but heavy and hard to modify.
DeepSpeed	Similarly mature; pioneer of ZeRO and used in BLOOM/GLM. Also complex and difficult to customize.
TorchTitan	Lightweight, modular, easier to understand; still new and less stable; good for experimentation but not yet feature-complete.
Nanotron	Minimal, simple, built specifically by the authors for HF-style pretraining. Fully hackable but requires more engineering investment to reach maturity.

Megatron-LM

Nvidia’s large-scale training framework: TP/PP/DP

Originally created to train models like GPT-3-class transformers, it is NVIDIA’s large-scale training framework designed to push transformer models to hundreds of billions of parameters through highly optimized 3D parallelism.

Tensor parallelism (TP): Splitting individual weight matrices across GPUs.

Pipeline parallelism (PP): Splitting model layers across GPUs.

Data parallelism (DP): Splitting training data across GPUs.

Megatron-LM provides provides fused CUDA kernels, custom attention and layer-norm implementations, and optimized scheduling of forward/backward microbatches, enabling unified, communication-efficient system that keeps all GPUs busy while minimizing memory overhead.

Megatron-LM

Step 1: Estimate the memory usage

Training memory = Model states + Activations + overheads

- Model states: parameters + gradients + optimization states.
- Activations: layer outputs kept for backward.
- Overheads: cuBLAS/cudnn workspaces, CUDA context, fragmentation (20-30%).

Megatron-LM

Step 1: Estimate the memory usage

Model states (8 Billion model)

- Weights in FP16/BF16: 2B/param
- Gradients in FP16/BF16: 2B/param
- FP32 master weights (for Adam): 4B/param
- Adam moments m/v in FP32: 8B/param

$$\theta_{t+1} = (1 - \eta\lambda)\theta_t \; - \; \eta \left(\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon} \right)$$

Model state memory = 8 x 10^9 x (2+2+4+8) = 128GB

Megatron-LM

Step 1: Estimate the memory usage

Activations (8 Billion model)

Layers	Hidden size	Sequence length	Micro-batch	Data format	Activations per layer
L=48	D=4096	S=2048	B=8	2B	c=3

ActMem = c x L x B x S x d x 2B = 19GB

Megatron-LM

Step 1: Estimate the memory usage

Training memory = Model states + Activations + overheads

- Model states: 128GB
- Activations: 20GB
- Overheads: 10-20GB

Total peak: 150 -170GB

We cannot train an 8B model on a single 80GB GPU.

We need a parallel training infrastructure.

Megatron-LM

Step 2: Determine the training configuration

- TP (Tensor Parallel)
- PP (Pipeline Parallel)
- DP (Data Parallel)

Megatron-LM

Step 2: Determine the training configuration

- **TP (Tensor Parallel)**
  - Splits big matmuls within a layer across GPUs.
  - + Helps fit larger layers & increases FLOP throughput.
  - - Adds significant intra-layer communication (per attention/MLP).

- **PP (Pipeline Parallel)**
  - Splits layers across GPUs (stages).
  - + Helps fit deeper models & reduces per-GPU memory.
  - - Adds pipeline bubbles; you need enough micro-batches to keep stages busy.

- **DP (Data Parallel)**
  - Replicates model shard; each replica trains on a different mini-batch slice.
  - + Easiest way to scale throughput.
  - - Needs gradient all-reduce / ZeRO comm across replicas.

Megatron-LM

Step 2: Determine the training configuration

Increase TP until a single layer fits

TP is best used to:

- Shrink the memory per layer.
- keep matmuls large and efficient.
- stay within a node (use NVLink/NVSwitch).

Choose TP so that:

- Hidden size / TP is still a “nice” matmul shape (e.g., 4096 / 4 = 1024 OK).
- Head count divisible by TP (e.g., 32 heads, TP=4 → 8 heads per GPU).
- Keep TP within a node if possible (TP ≤ #GPUs per node).

Megatron-LM

Step 2: Determine the training configuration

If still memory tight, add PP to split depth

PP splits layers across GPUs. Use when:

- Model is deep (many layers, big activations), and
- Even after TP, GPUs are still near memory limit.

Choose PP so that:

- Keep PP factor small at first: 2 → 4 → 8.
- Try to balance layers per stage (e.g., 48 layers → 24+24 for PP=2).

Megatron-LM

Step 2: Determine the training configuration

Whatever GPUs remain -> DP

We usually do not want to reduce TP/PP just to increase DP, because:

- TP/PP are chosen to make memory fit and matmuls efficient.
- Once that's done, fill remaining dimensionality with DP.

Further optimization (to reduce training steps):

- Increase micro-batch size if memory allows.
- Increase DP if we have more GPUs.

Megatron-LM

Step 2: Determine the training configuration

Considering the 8 Billion mixed-precision model with 300B tokens, 32 GPUs with 80GB/GPU

TP = 4 (within-node, 4096/4 = 1024 per shard)

PP = 2 (24 layers per stage)

TP x PP = 8: each GPU sees 1/8 of params/activations

Each GPU requires 25GB storage < 80GB margin

DP = 32/(4x2) = 4

Megatron-LM

Step 2: Determine the training configuration

Choose micro-batch, grad accumulation, global batch

Micro-batch per GPU: 4 sequences

Gradient accumulation: 4

Batch-sequence = 4 x 4 x 4 (DP) = 64 sequences

Tokens per step: 64 x 2048 = 131,072

Steps = 300 x 10^9 /1.21 x 10^5 = 2.3 x 10^6 steps

Megatron-LM

Summary

1. **Start from memory & topology**
   - Pick TP as smallest value that makes a layer shard fit and plays nice with matmul shapes, staying within a node.
   - If still memory-constrained: add PP to split depth until each stage fits.
2. **Compute DP from remaining GPUs**
3. **Design global batch**
   - Choose micro-batch per GPU and grad accumulation K that, fits memory, and #micro-batches is greater than @PP to keep pipeline busy.
4. **Estimate training steps and training time.**
5. **Iterate**
   - if memory too high (increase TP/PP, reduce batch).
   - If training too slow (increase DP or global batch), or more GPU.

ZeRO 1-2-3

From a “single-GPU memory limitation” to a “cluster-wide pooled memory resource”

ZeRO (Zero Redundancy Optimizer), introduced by Microsoft in 2020, addresses the memory redundancy and scalability bottlenecks in LLM training.

Traditional data parallelism replicates the full set of parameters, gradients, and optimizer states on every GPU, causing memory consumption to grow linearly with model size and making it impractical to train models at the tens- or hundreds-of-billions scale.

ZeRO’s core value proposition is that without changing the model architecture or the forward/backward computation, it shards training states across GPUs and reconstructs them on demand, causing memory usage per GPU to scale nearly inversely with the number of GPUs, thus enabling extremely large model training.

ZeRO 1-2-3

From a “single-GPU memory limitation” to a “cluster-wide pooled memory resource”

ZeRO’s key techniques include a staged approach to state partitioning:

Stage 1 shards optimizer states (m, v),

Stage 2 additionally shards gradients, and

Stage 3 shards parameters themselves.

These stages rely on efficient communication primitives such as reduce-scatter and all-gather to perform “on-demand reconstruction, distributed summation, and local updates.”

ZeRO 1-2-3

ZeRO 1: Shard optimizer states only

Mixed-precision training: 2B parameters, 2B gradients, 12B optimization states.

All GPUs hold full model parameters, compute full gradients, and get full averaged gradients after all-reduce.

Only the optimizer states (m, v) are sharded, and parameter shards exist only temporarily during sharded update, but not during forward/backward.

ZeRO 1-2-3

ZeRO 1: Shard optimizer states only

ZeRO-1 = shard optimizer states only.

Forward: no change.
Backward:

- all GPUs compute local gradients
- ALL-REDUCE → all GPUs obtain full average gradient vector G

Update:

- each GPU updates ONLY its optimizer-state shard (m, v) and parameter shard (θ)
  Example:
  GPU0 updates θ1
  GPU1 updates θ2
  GPU2 updates θ3
  GPU3 updates θ4

$$\theta_{t+1} = (1 - \eta\lambda)\theta_t \; - \; \eta \left(\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon} \right)$$

After update:

- ALL-GATHER updated parameter shards so every GPU has full θ again.

This preserves exact DP behavior with 75% less optimizer memory.

ZeRO 1-2-3

ZeRO 1: Shard optimizer states only

- It works well with tensor parallelism.
- It works with pipeline parallelism.
- It introduces low communication overhead.
- It is stable, predictable, robust.
- It doesn’t fight with fused kernels.
- It doesn’t break backward compatibility.
- It’s easier to get fast training throughput.

Now we have the training framework ready, what is next?

–Tom Jerry

Evaluation

–Tom Jerry

Model baseline

No, we do not start from scratch

There are many proven good model architectures and training receipts out there.

A proven baseline matching the specific needs is a good starting point: model size, data scale, documentation, framework support.

Architecture Type	Model Family	Sizes
Dense	Llama 3.1	8B, 70B
Dense	Llama 3.2	1B, 3B
Dense	Qwen3	0.6B, 1.7B, 4B, 14B, 32B
Dense	Gemma3	12B, 27B
Dense	SmolLM2, SmolLM3	135M, 360M, 1.7B, 3B
MoE	Qwen3 MoE	30B-A3B, 235B-A122B
MoE	GPT-OSS	21B-A3B, 117B-A5B
MoE	Kimi Moonlight	16B-A3B
MoE	Kimi-k2	1T-A32B
MoE	DeepSeek V3	671B-A37B
Hybrid	Zamba2	1.2B, 2.7B, 7B
Hybrid	Falcon-H1	0.5B, 1.5B, 3B, 7B, 34B
MoE + Hybrid	Qwen3-Next	80B-A3B
MoE + Hybrid	MiniMax-01	456B-A46B

Modify the baseline

The baseline is not optimized for your constraints or targets

Opportunities: Modification shall bring benefits, e.g., faster inference, lower memory, better stability.

Challenges: Multiple interactive decision variables, attention, positional encoding, activations, optimizers, hyperparameters, normalization, layout, etc.

Rules: Never change anything unless you have tested that it helps.

So, we need to design ablation studies to empirically validate.

Why Evaluation Matters

Loss: How good is the model at predicting the next token on this dataset?

- Loss may improve while capabilities worsen.
- Wikipedia → lower loss but worse general ability (easier prediction tasks).
- Some capabilities (math, reasoning) don’t show in average loss.
- Models often keep improving downstream even after loss converges.

Need fine-grained downstream evaluations:

- Knowledge, understanding, reasoning, common sense

What makes a good early-signal task?

Monotonicity, low noise, above-random performance, ranking consistency

- **Monotonicity:** Performance consistently improves as training progresses.
- **Low noise:** Different seeds shouldn't produce wildly different scores.
- **Above-random performance:** If capabilities only emerge later in training, tasks that show random-level performance for extended periods aren't useful for ablations. (unlike MMLU-MCF)
- **Consistency:** If method A > B early, that ranking should remain stable later.

Task formulations

MCF, CF, FG

MCF — Multiple Choice Format (Model selects A/B/C/D): May be struggle with early training, useful for mid-training, example: MMLU.

CF — Cloze Formulation (No explicit choices; compare log-likelihoods across options): Best for early training stages with high SNR, normalized log-probability per character.

FG — Free-Form Generation (Evaluate greedy generation accuracy):Too hard for early/mid ablations, mainly in post-training evaluation, as we do need the model to generate useful response by then.

Format	Definition	Best Training Stage	Why	Pros	Cons	Typical Benchmarks
CF — Cloze Formulation	Compute **normalized log-likelihood** over each candidate answer (no A/B/C/D choices shown to model).	**Early training** (0–40% of tokens)	Early checkpoints cannot generate or follow MC structure; CF tracks basic LM ability.	Most stable; correlates with LM loss; high SNR; extremely fast.	Must normalize per token/char; not comparable to human baselines; not a “real task”.	Early MMLU-Ablation, Winograd, BoolQ (likelihood mode)
MCF — Multiple Choice Format	Present **question + A/B/C/D options**, select option with highest likelihood.	**Mid-training** (40–90% of training)	Model now understands structure; MCF gives clean, discrete accuracy.	Easy scoring; interpretable; strong mid-training signal.	Overestimates early checkpoints; short options distort likelihood; sensitive to formatting.	MMLU, ARC-Challenge, PIQA, HellaSwag
FG — Free-Form Generation	Model **generates full answer** (greedy or sampled); evaluate correctness.	**Late training / post-training** (after SFT/RLHF or stable base model)	Requires coherent reasoning, formatting, instruction ability.	Most realistic; matches user-facing behavior; captures long-range coherence.	Very noisy; formatting errors; decoding variance; useless for early ablations.	GSM8K, HumanEval, BigBench tasks requiring reasoning

Benchmark	Domain	Type	#Q	Tests
MMLU	Knowledge	MC	14k	Knowledge across 57 subjects
ARC	Science & reasoning	MC	7k	Grade-school science
HellaSwag	Commonsense	MC	10k	Everyday reasoning
WinoGrande	Commonsense	Binary	1.7k	Pronoun resolution
CommonSenseQA	Commonsense	MC	1.1k	Commonsense concepts
OpenBookQA	Science	MC	500	Elementary facts + reasoning
PIQA	Physical commonsense	Binary	1.8k	Physical reasoning
GSM8K	Math	Free-form	1.3k	Math word problems
HumanEval	Coding	Free-form	164	Python synthesis

What these tasks actually test

Diversity is critical

- **MMLU / ARC** → factual & academic knowledge.
- **HellaSwag, WinoGrande, CSQA** → commonsense reasoning.
- **PIQA** → physical reasoning.
- **GSM8K** → math reasoning & intermediate steps.
- **HumanEval** → code generation & correctness.

Diversity provides broad coverage of early capability shifts.

What these tasks actually test

Diversity provides broad coverage of early capability shifts

Benchmark	Domain / What It Measures	Example Question	Correct Answer
MMLU	Academic & factual knowledge	*A mutation that introduces a premature stop codon is known as:* A. Missense B. **Nonsense** C. Silent D. Frameshift	**B**
ARC	Elementary-level science reasoning	*Which object conducts electricity?* A. Rubber band B. Plastic spoon C. **Copper wire** D. Wood	**C**
HellaSwag	Commonsense completion	*A man cuts onions and starts crying because…* A. **onion chemicals irritate eyes** B. he's sad C. allergic to knives	**A**
WinoGrande	Coreference + commonsense	*The trophy didn't fit in the suitcase because **it** was too big.* What was too big?	**The trophy**
CSQA	Commonsense Q&A	*What helps you see at night?* A. candle B. telescope C. microscope D. **flashlight** E. sunglasses	**D**
PIQA	Physical reasoning	*To stop glass from shattering when cutting it, you should…* A. cool rapidly B. **heat before scoring**	**B**
GSM8K	Math reasoning (step-by-step)	*8 apples + 5, then give away 3. How many left?*	**10**
HumanEval	Code generation & correctness	*Write add(a,b) that returns a+b.*	return a + b

Ablation principles

Architecture vs. data ablations

- Loss is not enough → need downstream tasks.
- Use high-signal, low-noise benchmarks.
- Prefer cloze formulation for early ablations.
- Use a broad mixture of domains (knowledge, reasoning, math, code).
- Ablations are expensive → plan for them.
- Evaluate early, often, and rigorously.

Ablation is important but also costly

May account for more than half of the main training cost

Phase	GPUs	Days	GPU-hours
Main pretrain run	384	30	**276,480**
Ablations (pre)	192	15	**69,120**
Ablations (mid)	192	10	**46,080**
Debugging	384/192	3/4	**46,080**
TOTAL	—	—	**437,760 GPU-hours**

Practical takeaways

- Build a small ablation harness before scaling up.
- Use CF for early architectures / tokenizer / optimization changes.
- Use ~1k samples/task to get signal fast.
- Establish stable ranking consistency early.
- Expect ≥50% of total compute to go into ablations.
- Treat eval pipelines as mission-critical infrastructure.

Now we know how to train and evaluate a model, what is next?

–Tom Jerry

Yes, build our own model

–Tom Jerry

Model baseline

No, we do not start from scratch

There are many proven good model architectures and training receipts out there.

A proven baseline matching the specific needs is a good starting point: Model size, data scale, documentation, framework support.

Architecture Type	Model Family	Sizes
Dense	Llama 3.1	8B, 70B
Dense	Llama 3.2	1B, 3B
Dense	Qwen3	0.6B, 1.7B, 4B, 14B, 32B
Dense	Gemma3	12B, 27B
Dense	SmolLM2, SmolLM3	135M, 360M, 1.7B, 3B
MoE	Qwen3 MoE	30B-A3B, 235B-A122B
MoE	GPT-OSS	21B-A3B, 117B-A5B
MoE	Kimi Moonlight	16B-A3B
MoE	Kimi-k2	1T-A32B
MoE	DeepSeek V3	671B-A37B
Hybrid	Zamba2	1.2B, 2.7B, 7B
Hybrid	Falcon-H1	0.5B, 1.5B, 3B, 7B, 34B
MoE + Hybrid	Qwen3-Next	80B-A3B
MoE + Hybrid	MiniMax-01	456B-A46B

Model architecture design for LLMs

Why architecture matters

Architecture determines:

- model capability
- training stability
- inference efficiency
- long-context performance
- memory footprint

Modern LLMs differ:

- attention mechanism
- positional encoding
- masking strategy
- depth / width allocation
- embedding design
- sparse vs dense layout

Architecture design starts with the WHY

What do we really want

What does the model need to be good at?

- Multilingual?
- On-device inference?
- Long context?
- Math/code capability?

Goal determine:

- model size
- attention mechanism
- positional encoding
- data mixture
- timeline constraints

Architecture design starts with the WHY

What do we really want

Model size: Multilingual and reasoning-heavy models need larger or deeper networks; on-device deployment forces small models; long-context models are limited by KV-cache memory.

Attention mechanism:

- Long-context ⇒ FlashAttention-2, block-sparse, RWKV, or linear attention.
- On-device ⇒ memory-efficient attention.
- Reasoning ⇒ multi-head, grouped-query, or multi-query attention.

Positional encoding: RoPE for general use; extended RoPE for 128K–1M context; ALiBi for long-range extrapolation; absolute encodings for compact mobile models.

Data mixture: Code/math models use curated technical data; multilingual models need broad language diversity; on-device models rely on small, focused datasets.

The transformer architecture landscape

So many variants

Model	Arch.	Params	Train Tokens	Attention	Context	Pos Enc.	Precision	Init	Optimizer	Max LR	LR Schedule	Warmup	Batch Size
DeepSeek LLM 7B	Dense	7B	2T	GQA	4K	RoPE	BF16	0.006	AdamW	4.2e-4	Multi-Step	2K	9.4M
DeepSeek LLM 67B	Dense	67B	2T	GQA	4K	RoPE	BF16	0.006	AdamW	3.2e-4	Multi-Step	2K	18.9M
DeepSeek V2	MoE	236B (21B)	8.1T	MLA	128K	Partial RoPE	-	0.006	AdamW	2.4e-4	Multi-Step	2K	9.4M→37.7 M
DeepSeek V3	MoE	671B (37B)	14.8T	MLA	129K	Partial RoPE	FP8	0.006	AdamW	2.2e-4	Multi+Cos	2K	12.6M→62.9 M
MiniMax-01	MoE+Hyb rid	456B (45.9B)	11.4T	LinAttn+GQA	4M	Partial RoPE	-	Xavier	AdamW	2e-4	Multi-Step	500	16M→128M
Kimi K2	MoE	1T (32B)	15.5T	MLA	128K	Partial RoPE	BF16	~0.006	MuonClip	2e-4	WSD	500	67M
OLMo 2 7B	Dense	7B	5T	MHA	4K	RoPE	BF16	0.02	AdamW	3e-4	Cosine	2K	4.2M
SmolLM3	Dense	3B	11T	GQA	128K	NoPE	BF16	0.02	AdamW	2e-4	WSD	2K	2.3M

The transformer architecture landscape

So many variants

- **Attention**: MHA / GQA / MLA / Linear Attention
- **Context window**: 4k → 128k → 4M tokens
- **Position encoding**: RoPE / NoPE / Partial RoPE / Parallax
- **Architecture**: Dense vs MoE vs Hybrid
- **Training tokens**: from 2T → 15T+
- **Model Active Parameters**: MoE activates only a small fraction

Attention mechanism

Why attention matters:

- Training: compute ≈ dominated by FFN
- Inference: main bottleneck = KV Cache memory

Attention mechanism options:

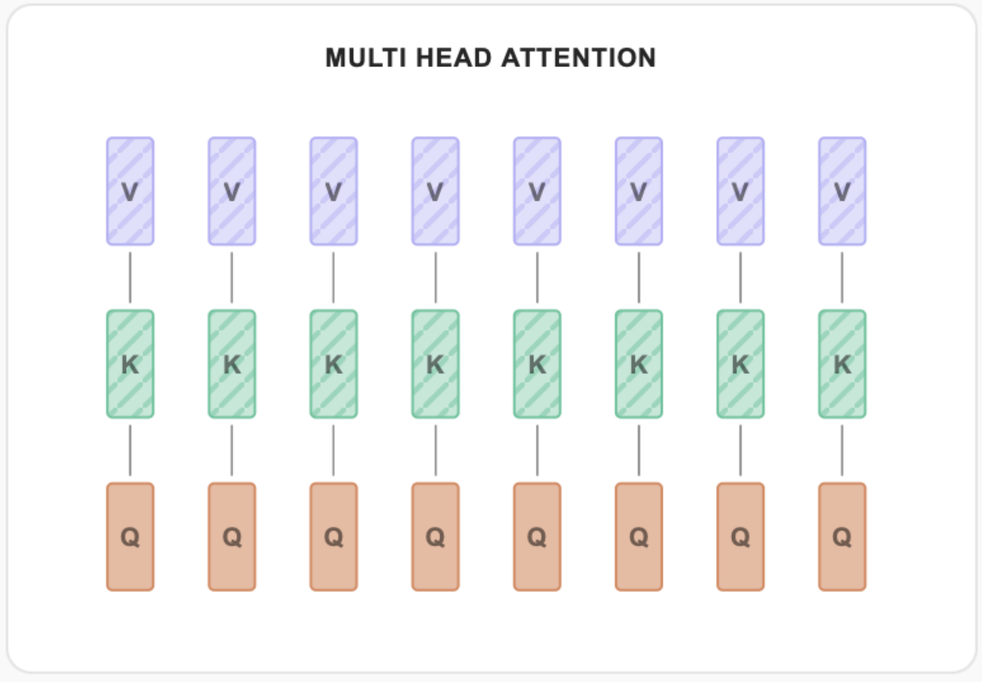
$$\text{KV memory} = n\_layers \times n\_heads \times seq\_len \times head\_dim \times 2 \text{ (K and V)}$$

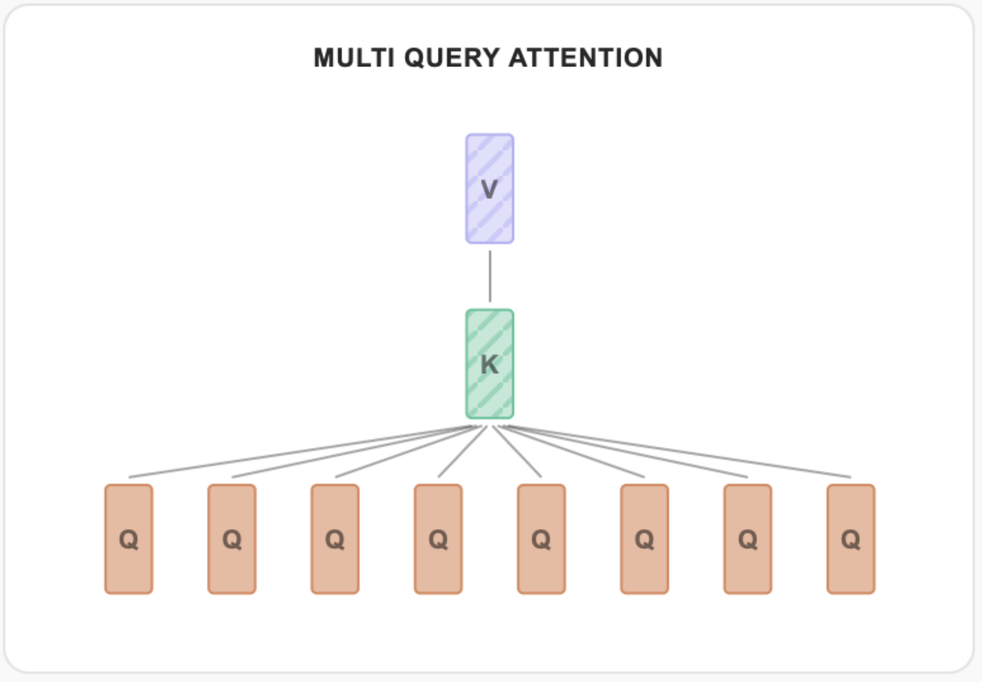
- **MHA**: classic, many KV heads
- **MQA**: 1 KV head for all Q heads
- **GQA**: shared KV per group (middle ground)
- **MLA** (DeepSeek v2/v3): latent-compressed KV cache

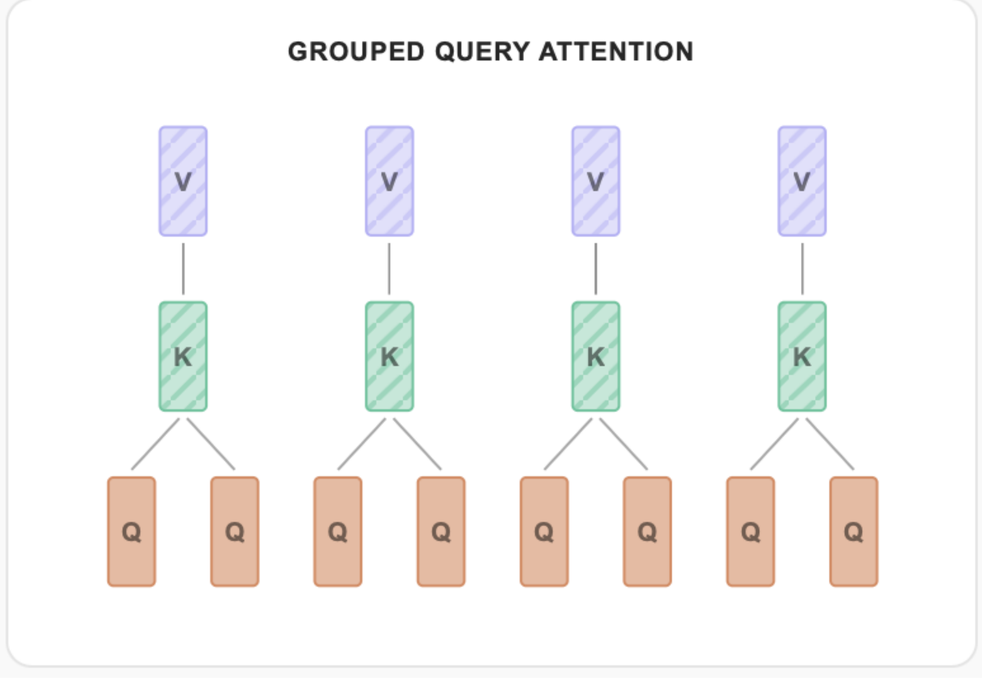
Attention mechanism

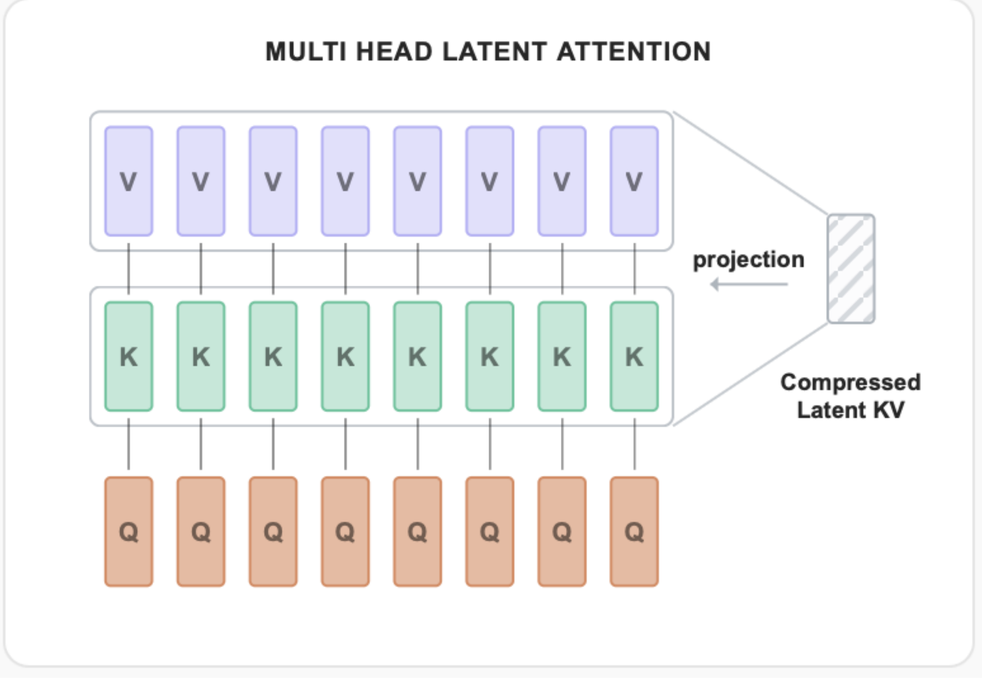
Multi-Head Attention (MHA): Multiple heads independently compute their own queries, keys, and values to retrieve information from past tokens. During inference, previously computed keys and values do not need to be recomputed and are stored in the KV-cache, which allows fast autoregressive decoding. However, as context length increases, the KV-cache grows linearly with sequence length, number of layers, number of heads, and head dimension. This makes the KV-cache a major memory bottleneck in long-context inference.

$$
\begin{aligned}
s_{KV} &= 2 \times n_{bytes} \times seq \times n_{layers} \times n_{heads} \times dim_{heads} \\
&= 2 \times 2 \times 8192 \times 32 \times 32 \times 128 = 4 \text{ GB } \textit{(Llama 3 8B)} \\
&= 2 \times 2 \times 8192 \times 80 \times 64 \times 128 = 20 \text{ GB } \textit{(Llama 3 70B)}
\end{aligned}
$$


MULTI HEAD ATTENTION
V V V V V V V V
K K K K K K K K
Q Q Q Q Q Q Q Q


MULTI QUERY ATTENTION
V
K
Q Q Q Q Q Q Q Q

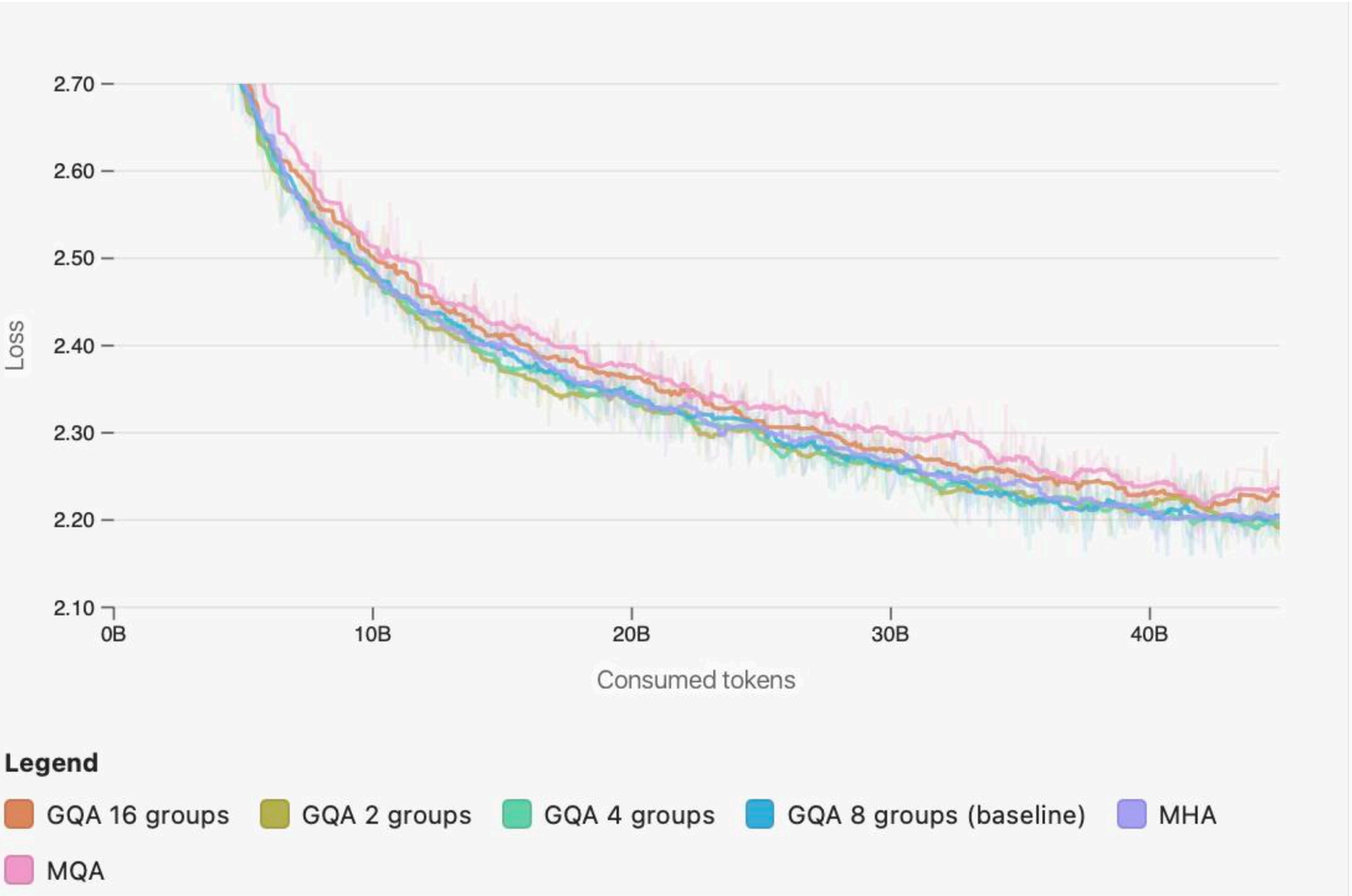

GROUPED QUERY ATTENTION
V V V V
K K K K
Q Q Q Q Q Q Q Q

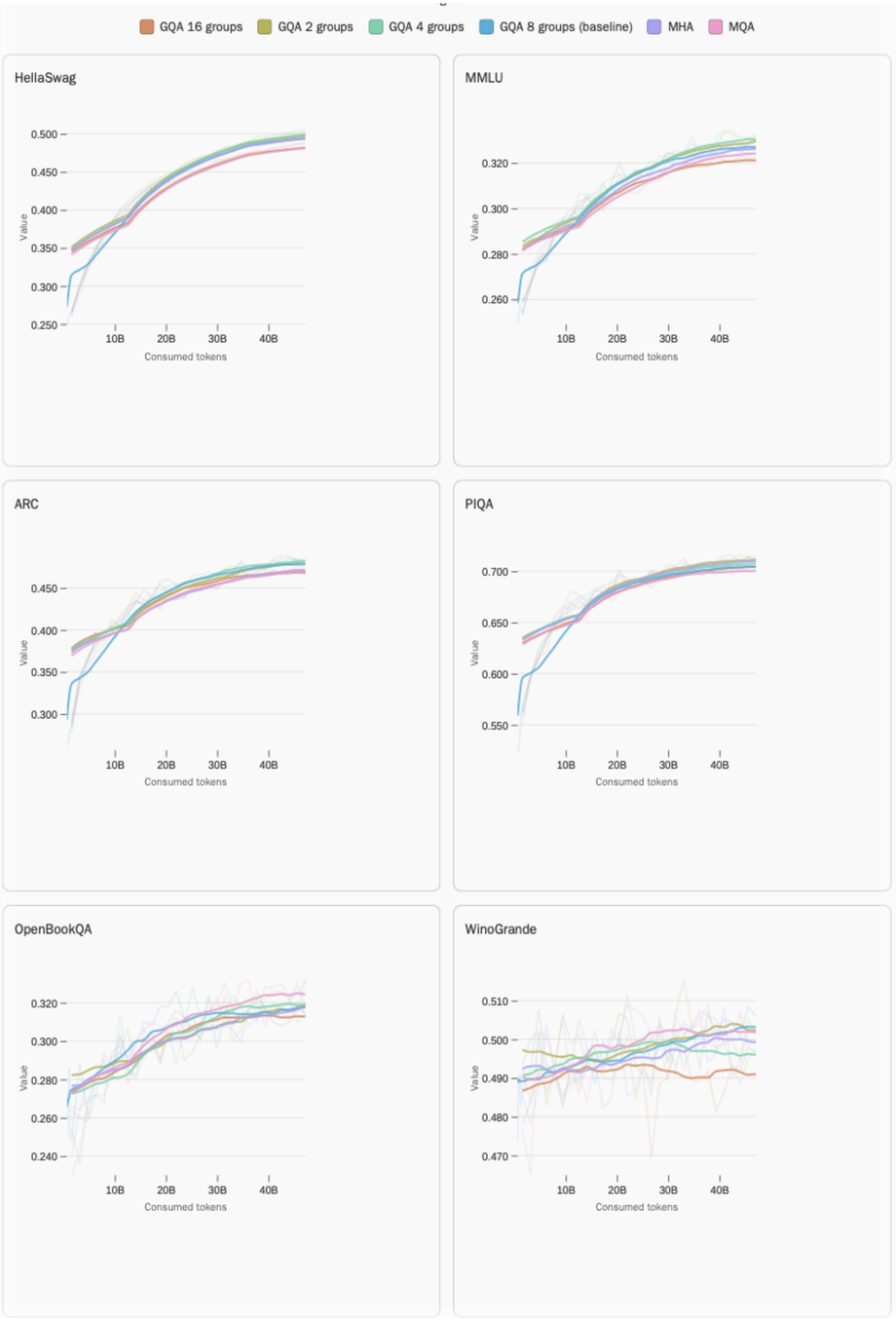

MULTI HEAD LATENT ATTENTION
V V V V V V V V
projection
K K K K K K K K
Compressed
Latent KV
Q Q Q Q Q Q Q Q

Attention mechanism

Mechanism	Q shared?	K shared?	V shared?	K/V compressed?
MHA	❌ No	❌ No	❌ No	❌ No
MQA	❌ No	✔ Yes (1 head)	✔ Yes	❌ No
GQA	❌ No	✔ Within groups	✔ Within groups	❌ No
MLA	❌ No	✔ Compressed latent	✔ Compressed latent	✔ Yes

Attention Mechanism	KV-Cache parameters per token
MHA	$= 2 \times n_{heads} \times n_{layers} \times dim_{head}$
MQA	$= 2 \times 1 \times n_{layers} \times dim_{head}$
GQA	$= 2 \times g \times n_{layers} \times dim_{head}$ (typically g=2,4,8)
MLA	$= 4.5 \times n_{layers} \times dim_{head}$





MQA and GQA with 16 groups (leaving only 1 and 2 KV heads respectively), underperform MHA significantly. On the other hand, GQA configurations with 2, 4, and 8 groups roughly match MHA performance.

Most modern models (DeepSeek, Qwen, Kimi, OLMo, SmolLM) use GQA.

Document Masking

Problem:

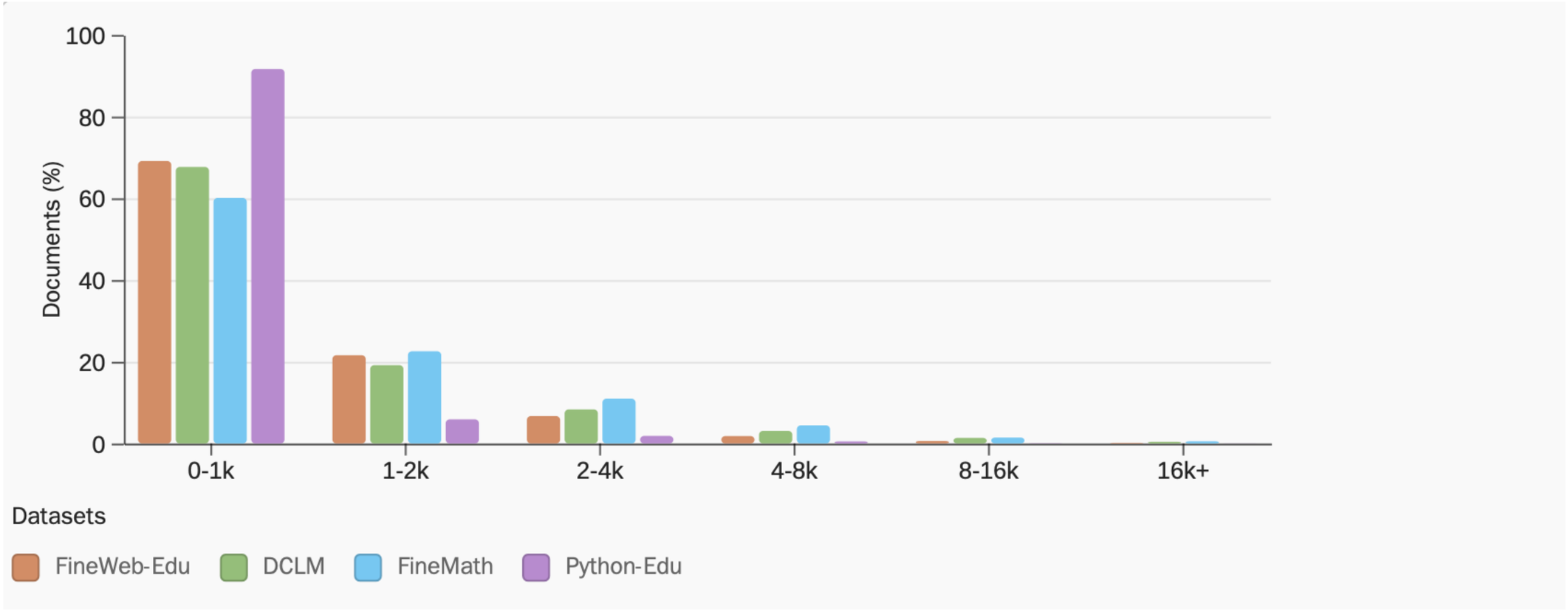
- Document lengths significantly vary, yet training on fixed-length training sequence.
- Training sequences pack many short documents.
- Standard causal mask → tokens attend across unrelated docs.
- 80–90% of real web/code docs < 2k tokens → lots of noise (FineWeb0Edu, DCLM, FineMath)

Solution: Intra-Document Masking

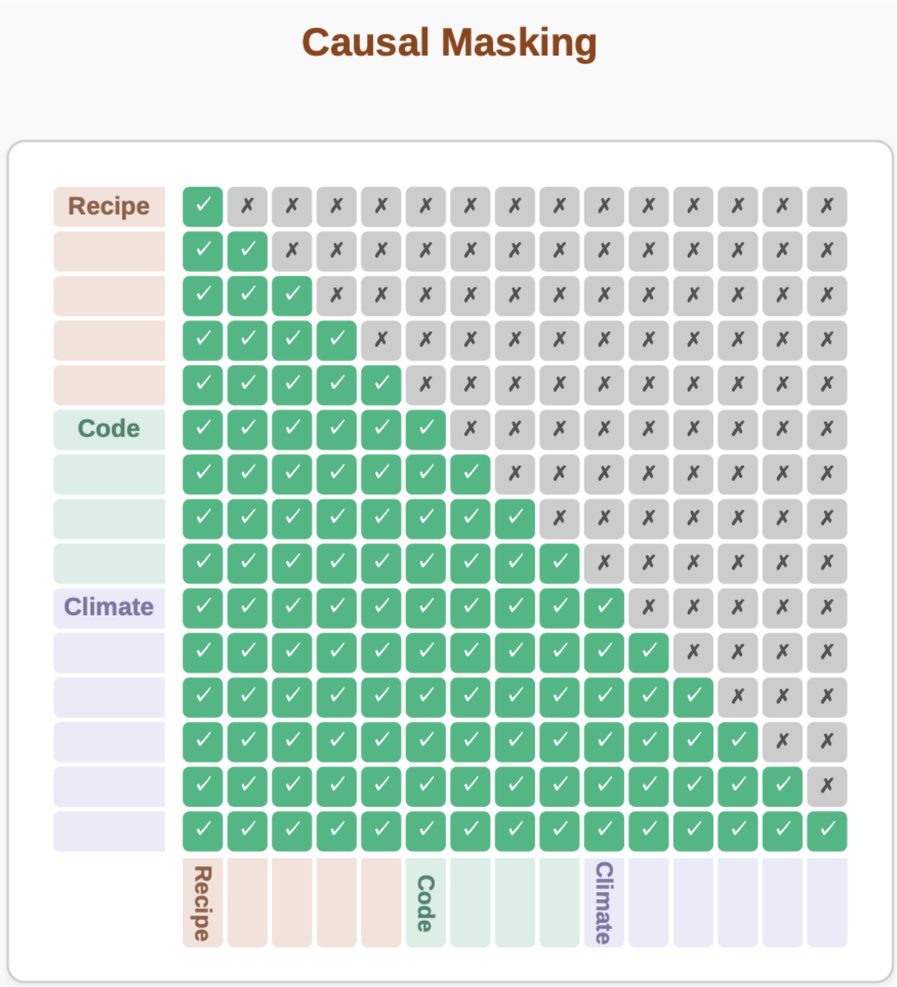
- Token can attend only inside its own document, and reduces noise
- Improves long-context scaling (Llama3, SkyLadder, SmolLM3)
- For short context (4k), little effect

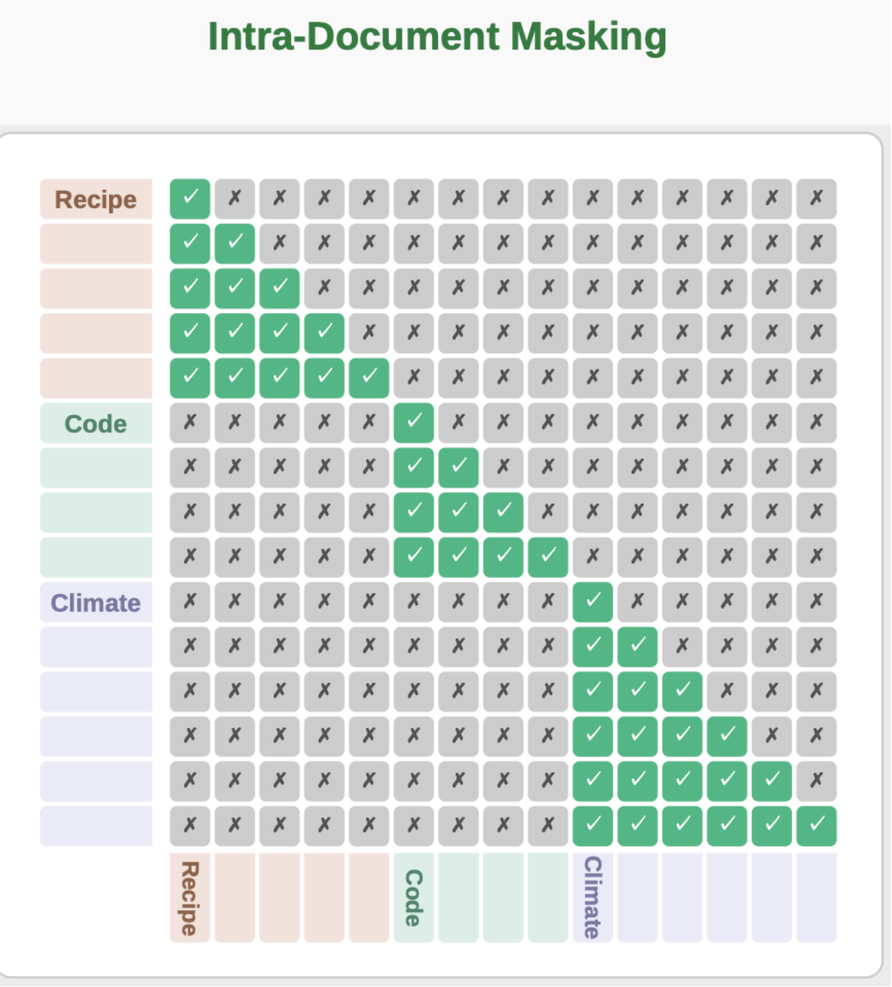
```
File 1: "Recipe for granola bars..." (400 tokens) <EOS>
File 2: "def hello_world()..." (300 tokens) <EOS>
File 3: "Climate change impacts..." (1000 tokens) <EOS>
File 4: "import numpy as np..." (3000 tokens) <EOS>
...

After concatenation and chunking into 4k sequences:
Sequence 1: [File 1] + [File 2] + [File 3] + [partial File 4]
Sequence 2: [rest of File 4] + [File 5] + [File 6] + ...
```



Document Masking





Position Encoding

Transformers have no inherent sense of word order because attention operates in parallel over all tokens, so without positional information “Adam beats Muon” is indistinguishable from “Muon beats Adam.”

Early models solved this using Absolute Position Embeddings (APE), which assign each token a learned positional vector but fail to generalize beyond the training context length.

As context windows grew, the field shifted to relative position encodings that model distances rather than absolute positions, enabling better extrapolation.

RoPE, the dominant modern approach, uses rotations in Q/K space to encode relative positions in a flexible, scalable way, making it the standard for contemporary long-context LLMs.

$$\text{dot\_product}(\text{RoPE}(x, m), \text{RoPE}(y, n)) = \Sigma_k \left[x_k * y_k * \cos((m-n) * \theta_k)\right]$$

Position Encoding

RoPE: Encodes positions by rotating Q/K vectors in 2D subspaces so attention depends on relative token distance, yielding strong short-context performance.

RoPE Frequency Extensions (ABF, YaRN): ABF and YaRN modify RoPE's rotation frequencies to slow angle growth, enabling stable attention and effective inference at very long context lengths (32k→1M).

NoPE / RNoPE: NoPE removes explicit positional encoding to enable implicit, highly extrapolatable position learning, while RNoPE mixes NoPE and RoPE layers to retain strong short-context accuracy.

Partial RoPE: Partial RoPE applies RoPE to only part of each head's dimensions, preserving essential positional signal while enabling latent KV compression and more efficient long-context attention.

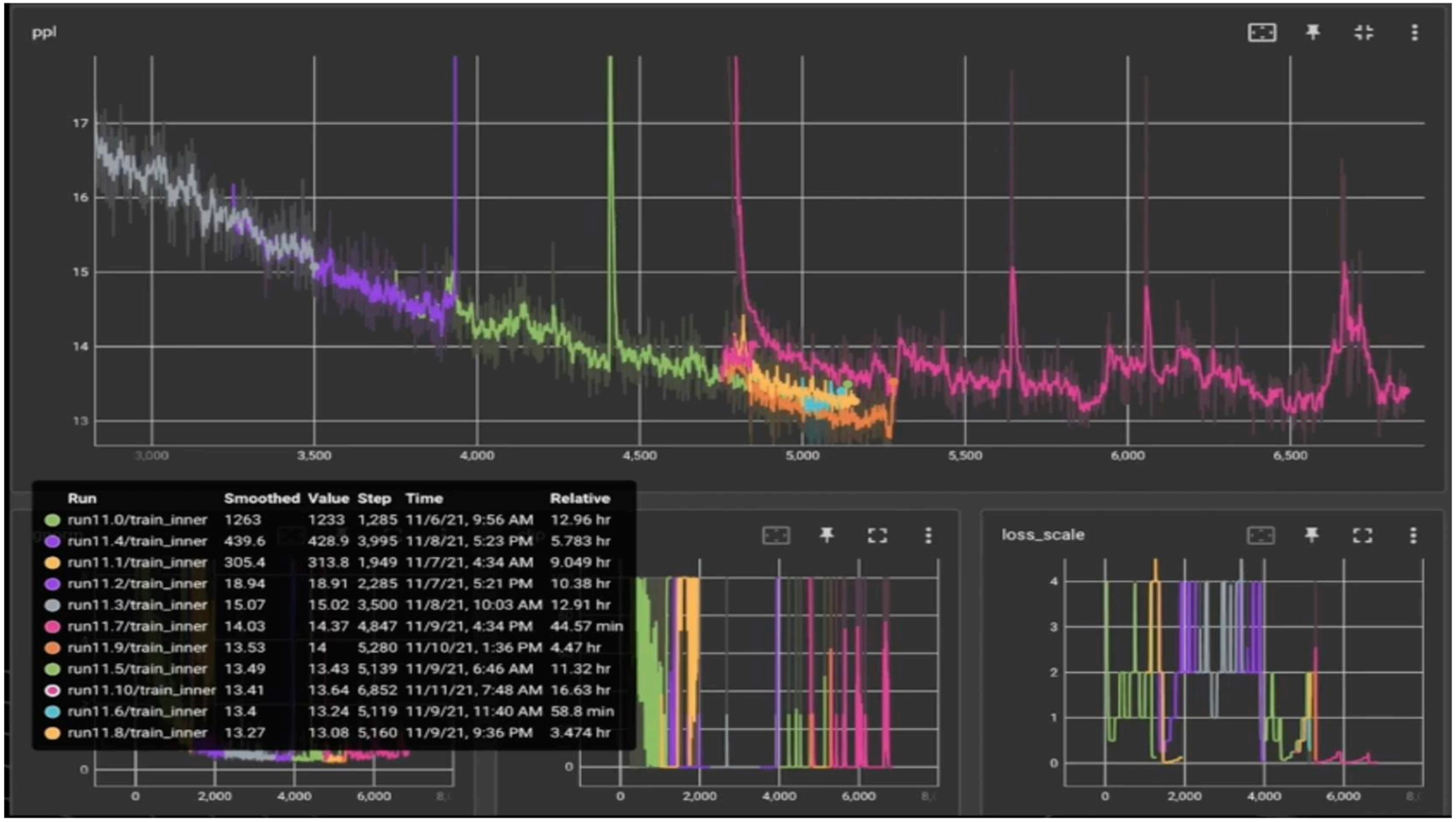
Position Encoding

The loss and evaluation results show similar performance across all three configurations, indicating that NoPE maintains strong short-context capabilities while providing the foundation for better long-context handling.

Technique	Key Idea	How It Works	Strengths	Weaknesses
RoPE (Rotary Position Embedding)	Encode position via rotations in 2-D subspaces	Rotate Q/K by angles proportional to token index; each dimension has its own base frequency	Strong short-context performance; clean relative-position modeling; mathematically elegant	Breaks down at long contexts due to fast-growing rotation angles (phase saturation / aliasing)
RoPE Frequency Extensions (NTK, YaRN, ABF, etc.)	Modify RoPE's frequency schedule to extend usable context	Rescale or remap rotational frequencies so angle growth slows at long positions; piecewise or smooth scaling	Enables 32k → 128k → 1M-token contexts; generally plug-and-play with base models	Requires tuning; different tasks/models prefer different scaling schedules
NoPE / RNoPE	Remove explicit positional embeddings; rely on implicit positional learning	NoPE drops position encodings entirely; RNoPE alternates between NoPE and RoPE layers to combine implicit + explicit signals	More stable behavior at unseen long lengths; simpler architecture; avoids RoPE's extreme-length failures	Slightly weaker short-context reasoning/ knowledge; limited absolute-position information
Partial RoPE	Apply RoPE only to part of Q/K dimensions	Split head dimensions: rotate a subset, leave the rest unrotated	Works well with MLA/latent-KV systems; offers stable long-context behavior	Provides weaker explicit positional signal; design requires choosing rotation split ratios

Stability techniques

Stability problems: loss spikes, divergence, “nan-ing.”

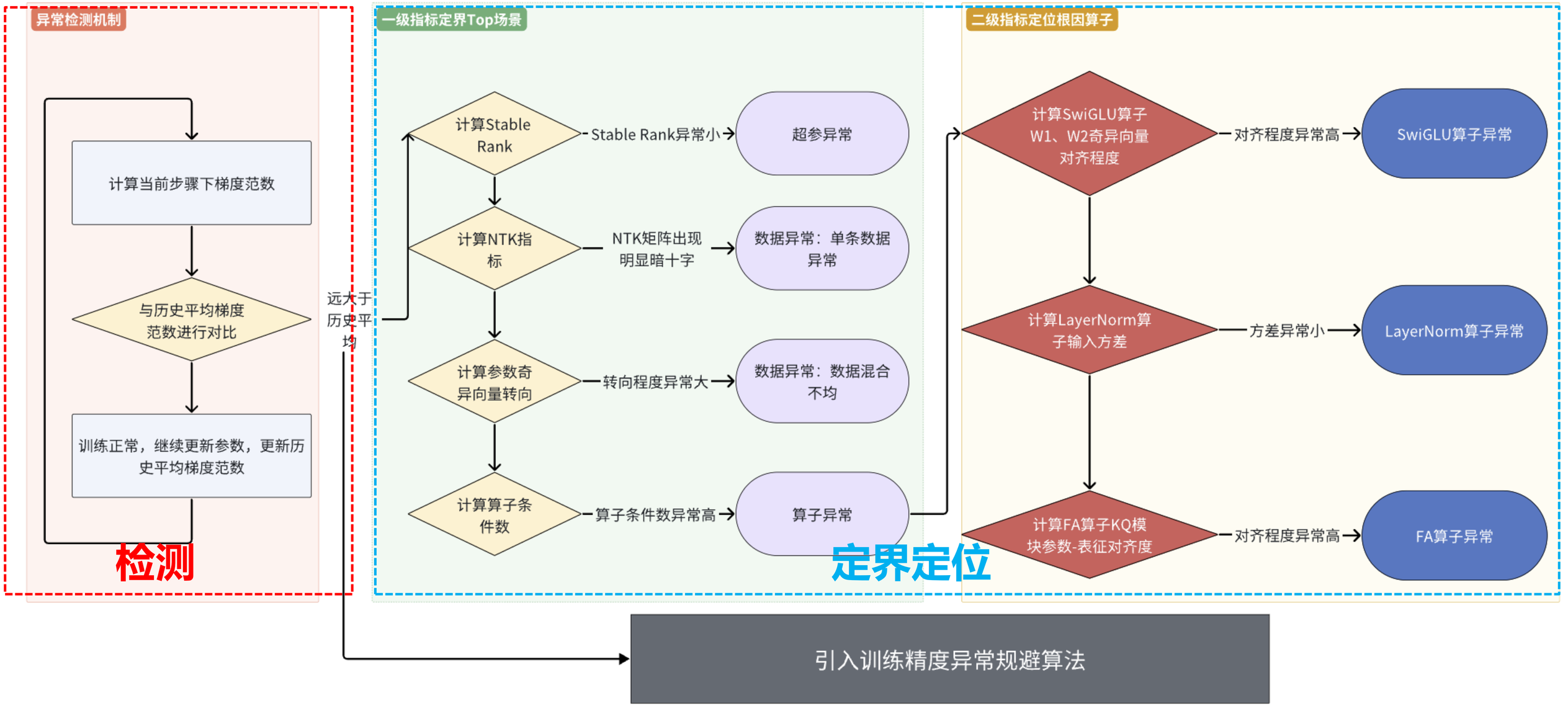


The cause of training instability

Training instability in large language models typically arises from bad data, mis-tuned hyperparameters, faulty or unstable operators, or numerical fragility introduced by low-precision formats such as BF16/FP8/FP4.

Category	Description	Examples / Failure Modes
Data Issues	Low-quality, noisy, or poorly mixed data can destabilize gradients and cause sudden loss spikes.	Duplicates, HTML garbage, malformed sequences; imbalanced data mixture; distribution shifts.
Hyperparameter Issues	Incorrect optimization settings lead to divergence or NaNs.	LR too large, improper warmup/decay schedule, mismatched LR–batch-size ratio, incorrect momentum/betas.
Operator / Kernel Issues	Unstable or incorrect implementations cause silent numerical errors.	Precision mismatch in fused ops, unstable softmax/layernorm, custom CUDA/Triton bugs, wrong backward pass.
Low-Precision Numerical Issues	Lower formats reduce numerical stability and increase sensitivity to scale and initialization.	BF16/FP8/FP4 underflow/overflow, mantissa loss, incorrect dynamic scaling, unstable FP8 kernels.

The saver


异常检测机制
计算当前步骤下梯度范数
与历史平均梯度范数进行对比
训练正常，继续更新参数，更新历史平均梯度范数
检测
远大于历史平均
一级指标定界Top场景
计算Stable Rank
Stable Rank异常小
超参异常
计算NTK指标
NTK矩阵出现明显暗十字
数据异常：单条数据异常
计算参数奇异向量转向
转向程度异常大
数据异常：数据混合不均
计算算子条件数
算子条件数异常高
算子异常
定界定位
二级指标定位根因算子
计算SwiGLU算子W1、W2奇异向量对齐程度
对齐程度异常高
SwiGLU算子异常
计算LayerNorm算子输入方差
方差异常小
LayerNorm算子异常
计算FA算子KQ模块参数-表征对齐度
对齐程度异常高
FA算子异常
引入训练精度异常规避算法

Depth vs Width Tradeoff

Height vs. width

Deeper > wider for:

- language modeling
- reasoning
- compositionality

Shallower + wider gives:

- higher throughput
- faster inference

Empirical rule:

- Sub-billion models → deep & thin
- Large models → moderate depth, wide hidden

Deep models reason better; wide models run faster—so small models should be deep and thin for expressivity, while large models are built moderate-depth and wide to balance stability, parallelism, and efficiency.

Dimension	Deeper (More Layers, Smaller Width)	Wider (Fewer Layers, Larger Width)
Reasoning Ability	Stronger multi-step reasoning; better compositionality	Weaker reasoning; limited sequential transformations
Language Modeling Loss	Better perplexity per parameter	Needs more parameters to match deep models
Compositional Generalization	Significantly better	Poorer; tends to memorize instead of compose
Training Throughput	Lower throughput (more sequential ops)	Higher throughput (more parallelizable per layer)
Inference Latency	Higher latency per token	Lower latency (fewer layers to traverse)
Hardware Utilization	Less efficient for GPUs/TPUs (more kernel launches)	Better utilization (large GEMMs per layer)
Numerical Stability	Harder to optimize when extremely deep	Easier to train; more robust at scale
Parameter Efficiency	Very high (best use of limited parameters)	Lower efficiency; needs larger parameter count
Best For (Sub-1B Models)	✓ Deep & thin recommended	✗ Not optimal
Best For (7B–400B Models)	✗ Too deep becomes unstable	✓ Moderate-depth, wide-hidden best
Examples	TinyLlama, MobileLLM, Longformer	GPT-3, LLaMA-2/3, DeepSeek V3

MoE (Mixture of Experts)

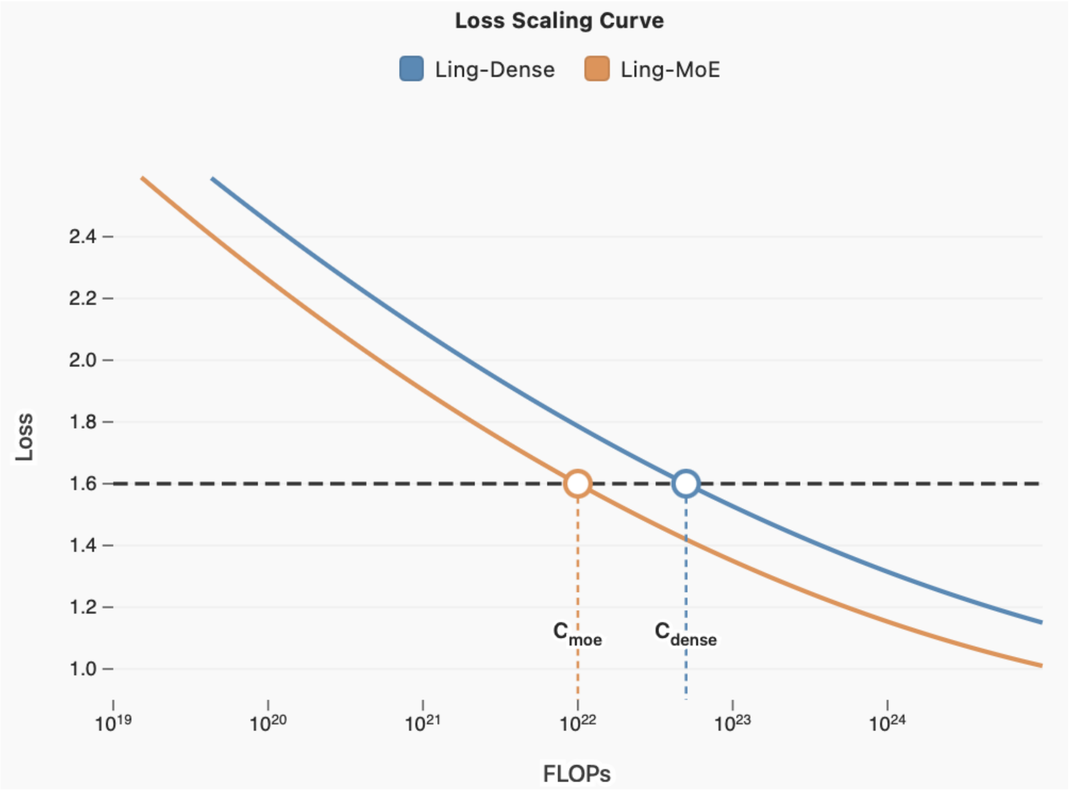
To be or not to be

MoE intuition:

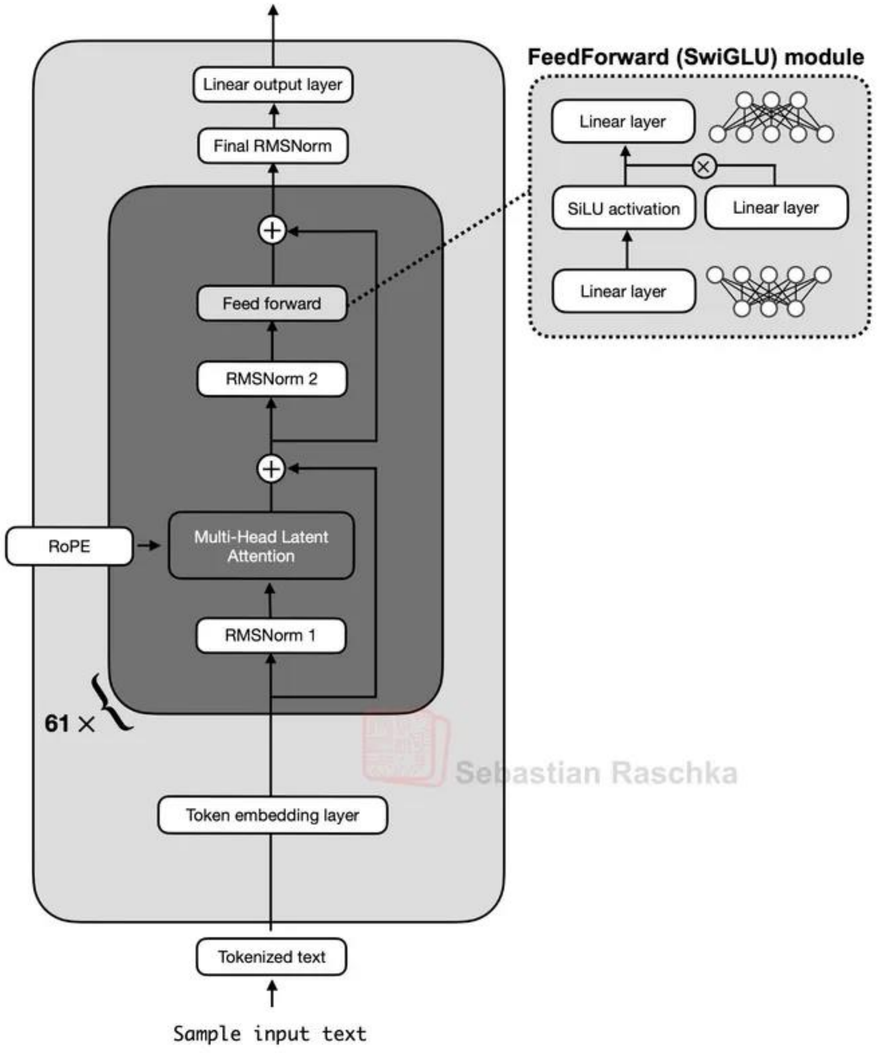
- Grow total parameters without increasing FLOPs (#active parameters per token)
- Only activate small expert subset per token
- Enables trillion-parameter models with fast inference

Key design challenges:

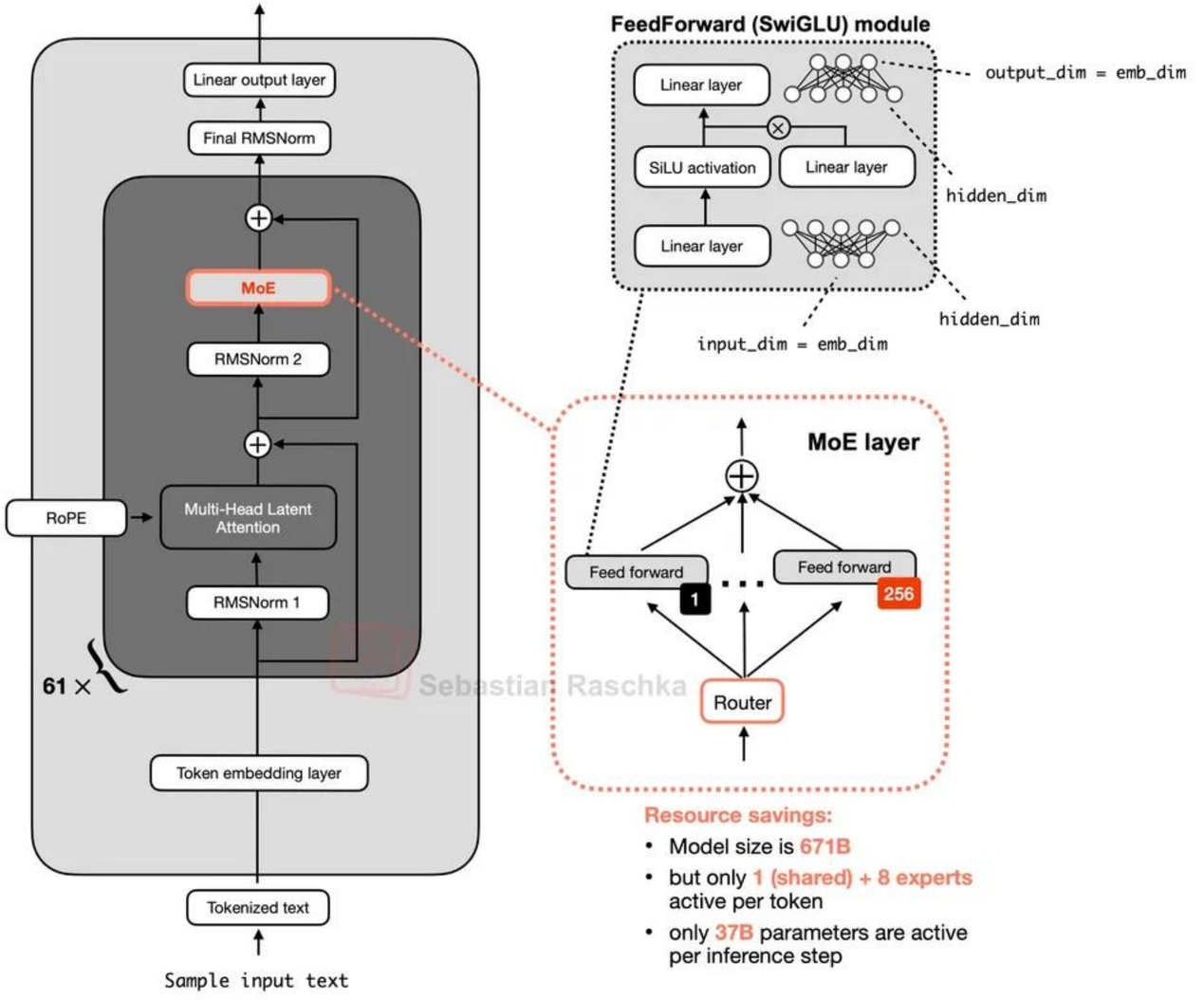
- Expert size, expert counts, active counts, Gating.



Architecture without MoE (“dense”)



DeepSeek V3/R1 with MoE (“sparse”)



Resource savings:

- Model size is 671B
- but only 1 (shared) + 8 experts active per token
- only 37B parameters are active per inference step

MoE (Mixture of Experts)

To MoE or not to MoE

- **Routing Instability:** Early in training, small gradient changes drastically alter expert assignment. Causes oscillations, load imbalance, and high variance across runs.
- **Load Balancing Difficulties:** Router tends to collapse onto a few experts unless strong balancing losses are used. Poor balancing → undertrained experts and degraded specialization.
- **Expert Specialization Chaos:** Specialization emerges unpredictably; different runs produce different expert roles. Leads to inconsistent performance and noisy validation curves.
- **Token–Expert Affinity is Fragile:** Router makes hard top-k decisions; small input shifts change expert selection. Hard routing causes discontinuous behavior and optimization difficulty.
- **Training Instability & Gradient Spikes:** Routed gradients create sparse and uneven updates. Experts receiving too few tokens learn slowly; overloaded experts diverge.
- **Communication & Efficiency Overhead:** Dispatching tokens to selected experts adds significant communication cost. Hard routing complicates batching and increases latency.
- **Sensitivity to Hyperparameters:** Routing temperature, balancing loss weights, expert count, and capacity factor must be finely tuned. Slight deviations can collapse routing or destabilize training.

Tokenizer

One of the most underrated components of an LLM

A tokenizer breaks text into small pieces (tokens) so the model can read it.

It converts:

Text → Tokens → Numbers → Model input

Different tokenizers split text differently:

- Good tokenizer → fewer tokens → faster + cheaper
- Bad tokenizer → many tokens → slow + inefficient

Why it matters:

- A model can only understand what the tokenizer gives it.
- If the tokenizer splits the text badly (especially non-English), the model becomes worse no matter how good the architecture is.

Design choices

One of the most underrated components of an LLM

1. What languages must we support?

- English-only tokenizer → terrible for Arabic, Chinese, etc.
- Example: GPT-2 tokenizer → Arabic sentence becomes 122 tokens
- Gemma3 tokenizer → same sentence becomes 44 tokens

2. Vocabulary size (how many tokens in dictionary?)

- Small vocab → easy for model but produces MANY tokens
- Large vocab → fewer tokens but bigger embedding matrices

3. Tokenization algorithm (usually Byte Pair Encoding)

- BPE builds a vocabulary of common “subword units” by repeatedly merging frequently occurring byte or character pairs into longer tokens. This allows the model to handle rare words, misspellings, multilingual text, and code using a compact vocabulary.

How to evaluate?

One of the most underrated components of an LLM

1. Fertility (tokens per word) – lower is better

- If a word turns into many tokens, training becomes slow and expensive
- Example from PDF:
  - English fertility ~1.4–1.6
  - Arabic fertility ~2.1–2.4 (much harder to tokenize!)

2. Continued words (% of words split into pieces) – lower is better

- Example (English):
  - Llama3: 32% split
  - Gemma3: 26% split (better)

Model decision summary

–Tom Jerry

What are we trying to build?

A 3B model case study

Design decisions:

- Run efficiently on phones and edge devices
- Multilingual ability
- Very long context
- Reasoning
- Stay small enough (≈3B) to fit mobile memory constraints
- Build on a proven recipe, not reinvent everything

Why this matters:

- Design decisions must follow deployment constraints (edge devices) and project goals.

Decision #1: Dense vs MoE

A 3B model case study

Choice: Dense transformer (NOT MoE)

- MoE wasn’t implemented in their training framework yet.
- Team already had strong dense-model training expertise.
- On edge devices → even inactive MoE experts still need to be loaded into memory.
- This is unacceptable for memory-limited deployment.

Why:

- Dense is simpler, more stable, and fits edge-device memory budgets, while MoE inflates memory footprint even when only a few experts are active.

Decision #2: Tokenizer choice (Llama3-128k)

A 3B model case study

Choice: Llama3's 128k tokenizer

- Excellent fertility across their 6 target languages.
- Large enough for multilingual efficiency.
- Small enough to avoid bloating embeddings in a 3B model.
- Performs competitively across English, French, Spanish, Portuguese, Italian.

Why:

- Tokenizer determines compression, training cost, inference speed, and multilingual quality.
- Llama3 achieved the best trade-off for a small, multilingual dense model.

Decision #3: Use grouped query attention (GQA)

A 3B model case study

Choice: GQA with 4 groups

- Validated at 3B scale: matches MHA quality.
- Dramatically reduces KV-cache size.
- KV-cache is the bottleneck in long-context inference, especially on-device.

Why:

- GQA gives almost the same accuracy as Multi-Head Attention but uses much less memory, which is crucial for deployment on phones.

Decision #4: NoPE+Document Masking+Depth Layout

A 3B model case study

NoPE (remove RoPE every 4th layer)

- Improves long-context handling.
- Does not hurt short-context performance.
- → essential for models aiming at long sequences.

Intra-document attention masking

- Prevents tokens from attending across documents.
- Speeds up training.
- Stabilizes very long sequence training.

Model layout optimization (depth > width)

- Tested 3B layouts: Qwen-2.5, Llama3.2, Falcon3-H1.
- All achieved similar loss.

Deeper Qwen-like architecture

- Aligns with evidence that depth improves generalization.

How all decisions fit together

A 3B model case study

- **Dense model** → fits edge devices.
- **Llama3 tokenizer** → efficient multilingual representation.
- **GQA** → memory-efficient long context.
- **NoPE** → stronger long-context generalization.
- **Document masking** → stabler ultra-long sequences.
- **Deeper model layout** → better generalization.

Now we have a model, what is next?

–Tom Jerry

A few more decisions need to be made before training starts

–Tom Jerry

Optimizer

AdamW vs. Muon

AdamW is the standard LLM optimizer because it combines Adam's stable, adaptive updates with a properly separated weight-decay step, preventing weight explosion and improving regularization. This makes it reliable, easy to tune, and effective across model sizes—hence its use in nearly all major models.

Muon is a newer second-order optimizer that updates whole weight matrices using SVD-like steps, enabling better exploration and support for much larger batch sizes. It can train more efficiently than AdamW at scale but is harder to tune and more complex, so adoption is still early.

AdamW

Adaptive updates with decoupled weight-decay

Adam + L2 (Coupled)

The L2 penalty is added to the gradient *before* adaptive scaling.

$$g_t' = g_t + \lambda\theta_{t-1}$$

$$\theta_t = \theta_{t-1} - \left(\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\right) \cdot g_t'$$

Result: The penalty $\lambda \theta$ is scaled, making it ineffective.

AdamW (Decoupled)

The adaptive step and weight decay are applied separately.

$$\Delta\theta_t = \left(\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\right) \cdot g_t$$

$$\theta_t = \theta_{t-1} - \Delta\theta_t - \left(\eta\lambda\theta_{t-1}\right)$$

Result: True weight decay is applied directly to the weights.

AdamW

AdamW vs. Muon

Better Generalization: AdamW separates regularization from optimization, leading to models that generalize better to unseen data.

Effective Regularization: It fixes the core bug in Adam, allowing weight decay to control model complexity a s intended.

The Standard for Transformers: This is the default and recommended optimizer for training large models like BERT, GPT, and Vision Transformers (ViT).

Key Insight: L2 regularization and weight decay are *not* the same thing in adaptive optimizers. AdamW implements true weight decay.

Muon

AdamW vs. Muon

1. Matrix-wise geometry (not parameter-wise)

- AdamW uses diagonal estimates → each weight updates independently.
- Muon uses row/column subspaces of the full matrix → updates reflect matrix structure.

2. Isotropic steps via SVD-like updates

- Muon computes an approximate SVD (using Newton–Schulz).
- It removes the singular values (Σ) and updates using only U and V, giving an isotropic step in the dominant directions.
- This reduces “axis-aligned bias” where AdamW suppresses updates in low-variance directions.

3. Naturally supports larger batches

- Because updates are more global and stable, Muon often tolerates higher batch sizes without divergence—important for trillion-token training.

Muon

AdamW vs. Muon

$$
\begin{aligned}
G_t &= \nabla_\theta \mathcal{L}_t(\theta_{t-1}) \\
B_t &= \mu\, B_{t-1} + G_t \\
O_t &= \mathrm{NewtonSchulz5}(B_t) \;\approx\; UV^\top \quad \text{if } B_t = U\Sigma V^\top \text{ (SVD)} \\
\theta_t &= \theta_{t-1} - \eta\, O_t
\end{aligned}
$$

Muon

AdamW vs. Muon

Why use Muon?

- Can train more efficiently at scale.
- Better at exploring parameter space.
- Higher batch size tolerance → fits large-GPU clusters.
- Already used in high-profile models (e.g., Kimi K2, GLM-4.5).

Why not (yet)?

- Harder to tune than AdamW.
- More computationally expensive (matrix SVD-like operations).
- Less battle-tested for multi-trillion-token runs.
- Frontier labs remain conservative: AdamW is simple, stable, and predictable.

Learning rate

High or low

Learning rate controls how big a step the optimizer takes at every update.

- Too low → model learns painfully slowly, loss stays flat, compute is wasted.
- Too high → updates become chaotic, model overshoots minima, or diverges (“loss shoots to the moon”).
- The optimal learning rate is not constant:
  - Early training needs larger steps to explore.
  - Later training needs smaller steps to stabilize.

- This is why every modern LLM training run uses learning-rate schedules, not a fixed value.

Warmup + Decay: The LR schedule for LLMs

Modern training uses a three-phase pattern

1. Warmup (0 → peak LR)

- Typically **2000 steps** (constant across models and scales)
- Prevents unstable “early chaos” when weights are random
- For very short trainings, people use **1–5% of total steps**

2. Stable Phase (plateau at high LR)

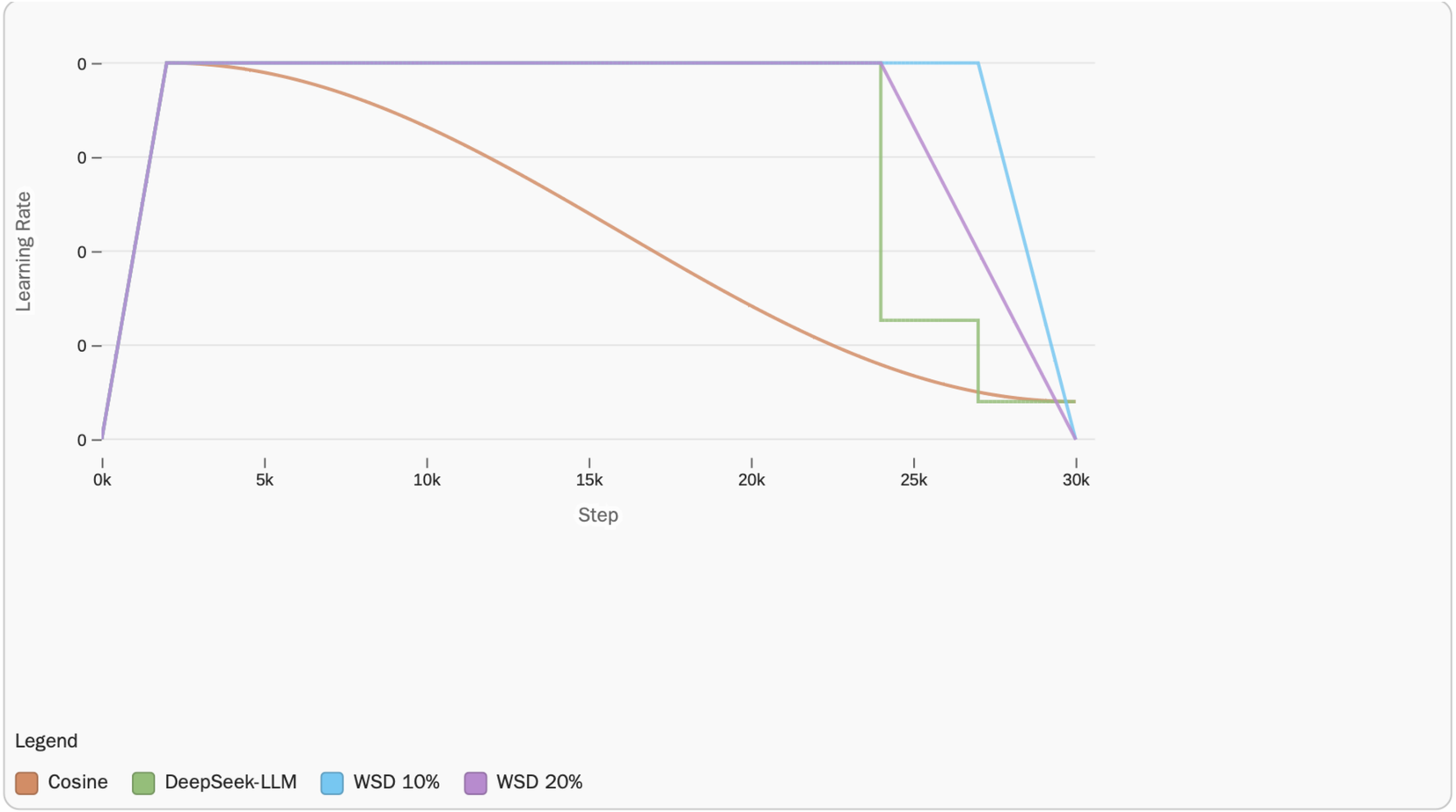
- Keeps learning rate high for most of training
- Favored by WSD and Multi-Step schedules
- Helps rapid learning and efficient compute usage

3. Decay Phase (cooldown) with several variants exist:

- **Cosine decay** → smooth but inflexible (must know total steps ahead of time)
- **WSD (Warmup–Stable–Decay)** → sharp decay in last 10–20% of tokens
- **Multi-Step** (DeepSeek LLM) → two discrete drops (e.g., at 80% and 90% of training)

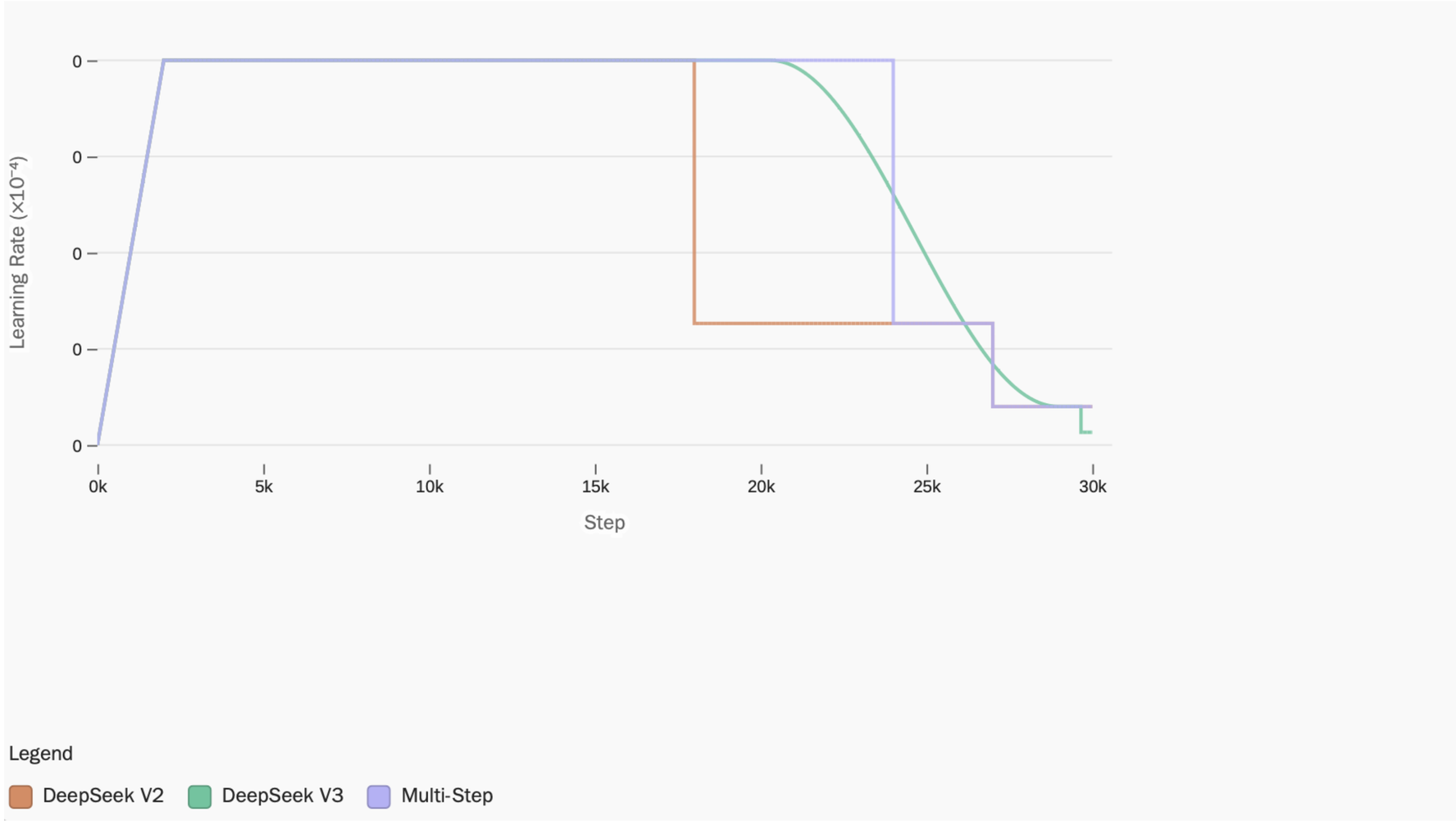
Warmup + Decay: The LR schedule for LLMs

Modern training uses a three-phase pattern



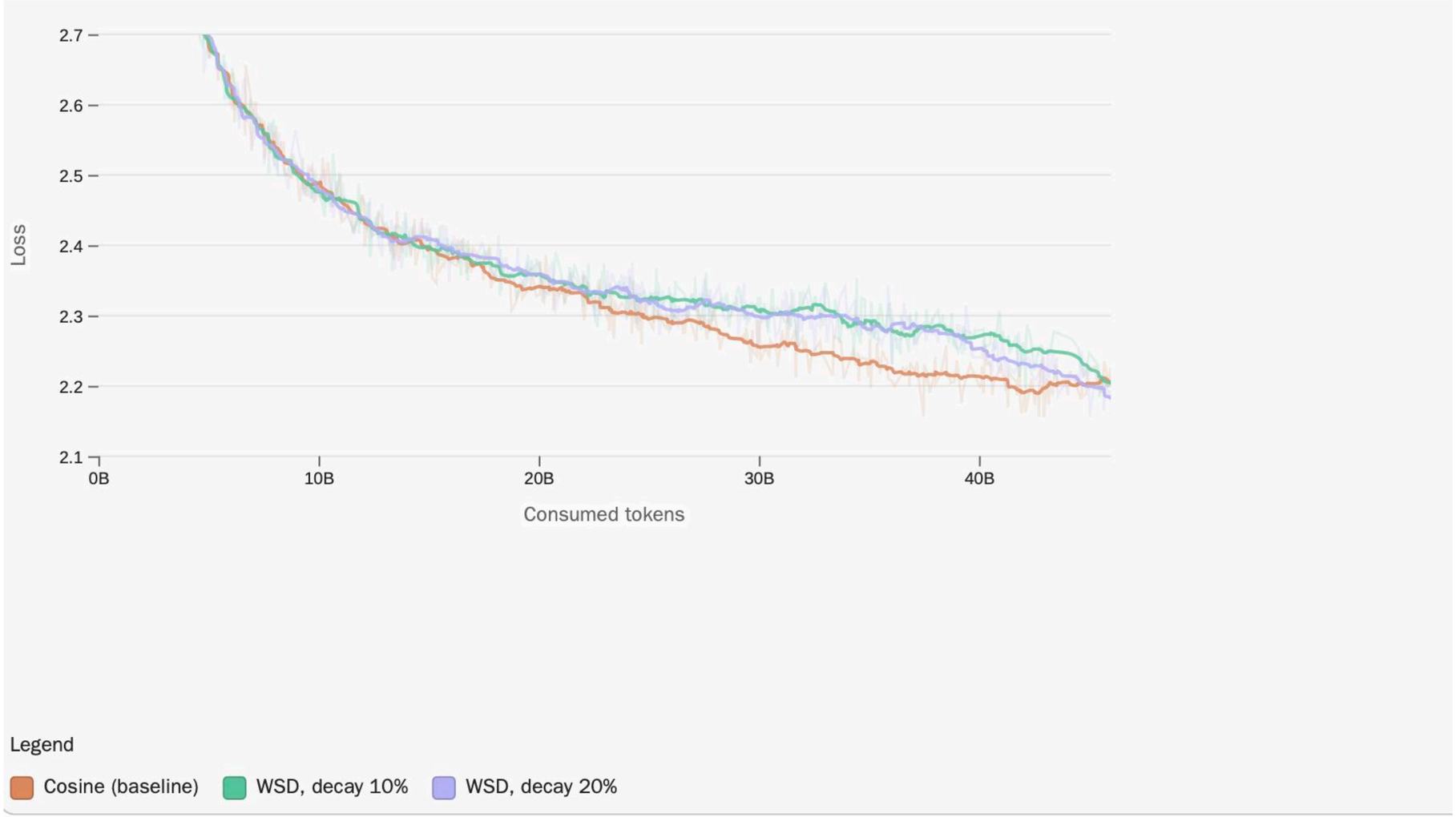
Warmup + Decay: The LR schedule for LLMs

Modern training uses a three-phase pattern



Warmup + Decay: The LR Schedule for LLMs

- Modern training uses a three-phase pattern



Finding the right learning rate

High or low

1. Too high (e.g., 5e–2): catastrophic divergence

- Loss spikes early and never recovers
- Model becomes unusable

2. Too low (e.g., 1e–4): stable but very slow

- Converges far later than other LRs
- Wastes compute

3. Middle-range (5e–4 to 5e–3): fastest convergence

- Best mix of stability + speed
- Used as baselines in many LLMs

- Sweeps showed:
  - 2e–4 = fastest stable convergence
  - 3e–4 = slightly better loss, but higher instability risk
- They chose **2e–4** for the final run

Batch Size

The tradeoff between speed and data efficiency

Batch size is the number of samples processed before the model updates its weights.

Batch size is a tradeoff between speed and data efficiency; larger batches give better throughput but only help until the critical batch size, after which they waste tokens unless you properly rescale the learning rate and verify that training dynamics remain aligned.

This trade-off is central to choosing a good batch size.

Batch Size

If batch size changes, learning rate must change too — otherwise training becomes unstable or too slow.

When batch size increases, the gradient becomes a more accurate estimate of the true direction the model needs to move. This means the optimizer can safely take larger steps, so the learning rate must increase as well.

The general idea is simple: larger batches reduce gradient noise, so you must raise the learning rate to maintain the right level of update "energy."

A common rule of thumb is square-root scaling (LR grows like √batch), but the exact scaling depends on the optimizer. If you don't adjust the learning rate, training may appear stable but will waste compute and converge poorly.

Averaging over B samples

- Batch gradient: $\tilde{g}_B = \frac{1}{B}\sum_{i=1}^{B}\tilde{g}^{(i)}$
- Mean stays the same: $\mathbb{E}[\tilde{g}_B] = g$
- But covariance shrinks: $\mathrm{Cov}(\tilde{g}_B) = \frac{\Sigma}{B}$

The SGD parameter update is:

- $\Delta w = -\eta\,\tilde{g}_B$

The variance of this update is proportional to:

- $\mathrm{Var}(\Delta w) \propto \eta^2\,\frac{\Sigma}{B}$

Batch Size

How to identify the critical batch size

The critical batch size is not fixed; it depends on the model's gradient noise, which changes throughout training.

Early in training the noise comes from the unstable model and loss landscape, not minibatch sampling, so increasing batch size doesn't meaningfully smooth it — hence the critical batch size is small; only later does minibatch noise dominate, making large batches effective.

Practitioners often use a practical method: run two short branches of training—one with the current batch size, and one with a larger batch and rescaled learning rate—and compare their loss curves. If the curves match, the new batch size is safe and data-efficient. If they diverge, the batch is too large. This low-risk technique helps teams find the sweet spot without wasting massive compute.

Batch Size

Batch size warmup and dynamic strategies

Since the critical batch size grows during training, some large models gradually increase batch size over time—a strategy called batch-size warmup.

For example, DeepSeek-V3 starts with a smaller batch size early in training and switches to a much larger batch later, when the gradients become more stable. This is analogous to learning rate warmup and helps maintain optimal training efficiency throughout the run. (first 469B tokens: batch = 12.6M, remaining tokens: batch = 62.9M)

Other models, like Minimax-01, treat the model's loss as a signal for when to increase batch size, even using extremely large batches (128M) in later stages. In their setup, they do not increase the learning rate, so increasing batch size functions similarly to gradually reducing the learning rate.

Batch Size

How to choose batch size and learning rate in practice

The practical recipe is: start with batch size and learning rate values from literature or scaling laws, then explore increasing batch size to improve throughput.

Between your initial batch size and the critical batch size, you can often gain significant hardware efficiency with no loss in model quality. If increasing the batch size yields little throughput improvement or hurts data efficiency (i.e., needs more tokens to reach the same loss), stick with the original settings.

In short, batch size and learning rate must always be tuned together, and the goal is to find the largest batch size that still preserves data efficiency.

Scaling law

Computation bound vs. train as long as you can

“Perfect is the enemy of good” + Compute Constraints

Modern LLM development is always compute-constrained, meaning we never have enough GPUs or time to reach theoretically perfect training. Instead of chasing perfection, we must make pragmatic decisions that deliver the best results within finite compute budgets and deadlines.

This is a shift from early deep learning, where training was not compute-limited and one could simply train the largest model that fit into memory until it overfitted or data ran out.

Scaling law

Computation bound vs. train as long as you can

Kaplan et al. Scaling Laws (Early Takeaway: Bigger Models > More Data)

Kaplan et al.’s original scaling laws showed that LLM performance improves predictably over many orders of scale and suggested allocating most compute to increasing model size, not training duration. This drove the “go bigger” trend, where models like GPT-3 (175B) were trained on relatively modest datasets (300B tokens), reflecting the belief that scaling parameters was the most efficient way to improve performance.

Scaling law

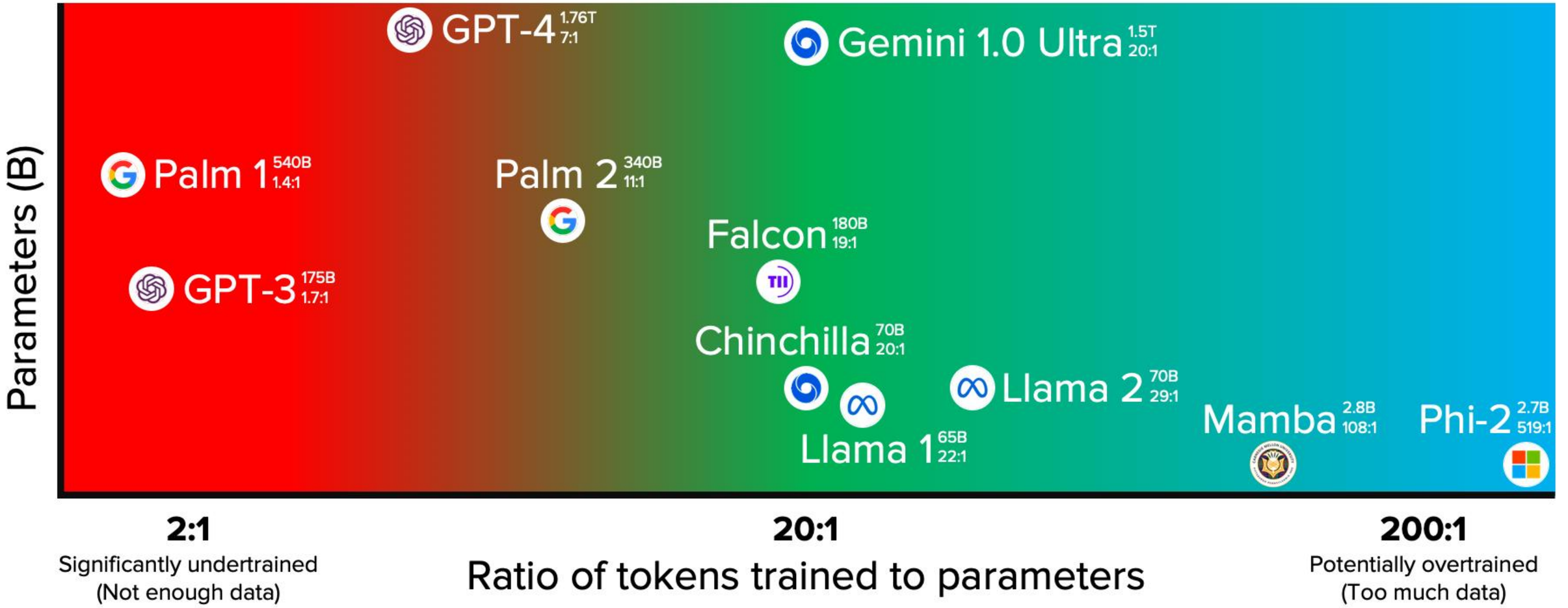
Computation bound vs. train as long as you can

Chinchilla/Hoffman Revisions (Later Takeaway: Train Longer)

Hoffman et al. (Chinchilla laws) reexamined Kaplan's methodology and found that compute-optimal training requires much more data for a given model size. For GPT-3, they estimated the compute-optimal dataset size should have been 3.7 trillion tokens, not 300B. This shifted the field's emphasis from simply making models larger to training them longer, with more tokens, for better efficiency and performance.

DATA-OPTIMAL (CHINCHILLA) MODEL HEATMAP

DEC/ 2023



Selected highlights only. Mostly to scale. Informed estimates for Palm 2, GPT-4, and Gemini. Alan D. Thompson. November 2022, major update December 2023. https://lifearchitect.ai/

Scaling law

Computation bound vs. train as long as you can

Why real models still don’t follow compute-optimal scaling laws?

Even though Chinchilla provides compute-optimal recommendations, real-world teams don’t strictly follow it because compute-optimal training ignores inference cost. Larger models are expensive to deploy, so many teams intentionally train smaller models for longer—an “overtraining” strategy that sacrifices theoretical optimality but produces cheaper, faster models at inference time. This has become industry-standard practice for open and on-device models.

Now we start training model, what is next?

–Tom Jerry

Model architecture defines how your model learns, then data defines what it learns

–Tom Jerry

Data Curation

Data defines what it learns

Even with a perfect architecture, optimized hyperparameters, and strong infrastructure, an LLM can still fail if its training data is poor.

Data determines what a model learns, and no amount of compute or clever tuning can compensate for the wrong mixture of content.

Good data curation means choosing high-quality sources, assembling them into the right proportions, and ensuring the mixture aligns with the skills we want the model to learn (e.g., English, multilingual, math, code).

The challenge of data mixtures

Data defines what it learns

Building a good mixture is unintuitive: increasing one domain (like code) necessarily reduces others under a fixed compute budget, meaning domains compete.

Some data is higher quality, but using only the best data leads to repetition, which harms performance.

So mixtures require balancing high-quality and lower-quality sources, tuning proportions, and running ablations to understand trade-offs across domains.

Data mixtures change over time

Data defines what it learns

Modern LLMs no longer train on a single static mixture. Instead, training uses multi-stage curricula, adding or upweighting different data sources at different points. Research shows a model's final behavior is heavily influenced by data seen near the end of training. This motivates strategies like: using broad, plentiful data early, and reserving scarce, high-quality math or code data for the late stages when it has maximum effect.

- **Performance-driven interventions :** Monitor evaluation metrics on key benchmarks and adapt dataset mixtures to address specific capability bottlenecks. For example, if math performance plateaus while other capabilities continue improving, that's a signal to introduce higher-quality math data.

- **Reserve high-quality data for late stages :** Small high quality math and code datasets are most impactful when introduced during the annealing phase (final stage with learning rate decay).

Case study

3B model with 11 trillion tokens

We wanted a model that can handle English and multiple other languages, and excel in math and code. These domains — web text, multilingual content, code and math.

Case study

English Web Data (Foundation Layer)

Web text is the backbone, built primarily from FineWeb-Edu and DCLM, totaling 5.1T tokens of high-quality data. FineWeb-Edu strengthens STEM and educational benchmarks, while DCLM boosts commonsense reasoning, and ablations showed that a 50/50 or 60/40 mix delivers the best overall performance.

Additional datasets such as Wikipedia, Wikibooks, Pes2o, and StackExchange were included for diversity, although they had minimal isolated impact. This balanced English foundation ensures strong general-purpose language ability.

Case study

Multilingual Web Data

To support multilingual capability, FineWeb2-HQ is included for five target languages (FR, ES, DE, IT, PT) and smaller amounts of ten additional languages (e.g., Chinese, Arabic, Russian) using FineWeb2 where needed.

Ablations showed that 12% multilingual content achieves the best trade-off: it significantly improves multilingual performance without hurting English benchmarks.

Going beyond 12% would require heavy repetition because multilingual data (628B tokens) is much smaller than English data (5.1T tokens), making this a natural upper limit.

Case study

Code Data

Code data came from The Stack v2, StarCoder2Data, GitHub pull requests, Jupyter notebooks, Kaggle scripts, and StackExchange discussions. Although prior work recommends using 25% code for stronger general reasoning, ablations showed that 25% severely degrades English benchmarks.

At 10% code, the model does not gain much, but code remains valuable for real-world capability, so 10% was kept as a safe compromise.

High-quality code (Stack-Edu) was introduced later in training, consistent with the principle of staging high-quality data for maximum late-stage impact.

Case study

Math Data

Math data followed a similar staged strategy. Early training used broad datasets like FineMath3+ and InfiWebMath3+, while later stages incorporated higher-quality datasets such as FineMath4+, InfiWebMath4+, MegaMath, OpenMathInstruct, and OpenMathReasoning.

Math was limited to 3% in Stage 1 because only ~54B high-quality math tokens were available; using more would require too many repeated epochs over the same math content, which can harm performance. Later curriculum stages upsampled higher-quality math sources for stronger reasoning improvements.

Case study

Annealing ablations

Stage 1 mixtures were chosen via from-scratch ablations, but new datasets for later stages were tested using annealing ablations, where a checkpoint (e.g., at 7T tokens) is continued for 50B tokens using a 40% baseline + 60% candidate dataset mix. This method efficiently tests whether new data sources (e.g., MegaMath) yield improvement without restarting training.

We ultimately used a three-stage curriculum: a broad foundation in Stage 1, followed by increased emphasis on high-quality math and code in later stages, ensuring balanced capabilities across domains.

Now we have everything, so we can start model training, yeah!

–Tom Jerry

Suprises, Suprises, Suprises

–Tom Jerry

The training marathon

After finishing architecture, data mixture, and hyperparameters, the real challenge is running a month-long, 11T-token training on 384 H100s. The focus is on what actually happens during full-scale training—pre-flight checks, unexpected issues, and how prior ablations plus solid infrastructure give you a fighting chance when things inevitably go wrong.

What to verify before hitting “train”

Before launching, the team verifies infrastructure (reserved nodes, GPU stress tests, storage behavior), sets up automated evaluation pipelines, tests checkpoint save/resume, confirms logging of all important metrics, and double-checks configs and launch scripts. The goal is to ensure that once training starts, they can trust the system to run, resume, and monitor itself with minimal manual babysitting.

Scaling surprises (Introduction)

Even with good ablations and configs, full runs behave differently: longer duration, larger datasets, and subtle infrastructure differences can introduce new failure modes. For 3B model, moving from short 100B-token ablations to an 11T-token run exposed several surprises that weren’t visible at smaller scale, leading to a sequence of “mysteries” they had to debug.

Mystery #1 – The vanishing throughput

Soon after launch, tokens/sec per GPU collapsed with sharp drops, threatening to double the training time. The team traced this not to the model but to storage: their network filesystem (FSx/Weka) started evicting dataset shards due to space pressure, causing repeated cache misses and stalls when shards were fetched from S3 mid-training.

Fix #1 – Changing Data Storage

To eliminate Weka evictions, they moved the 24TB dataset to each node's local NVMe scratch. This removed the network bottleneck but introduced a new issue: when a node died, its replacement had no data, so they had to copy from S3 or a healthy node (1.5–3 hours). They mitigated this by reserving a spare node preloaded with data, swapping it in instantly on failures and using it for eval/dev when idle.

Mystery #2 – The persisting throughput drops

Even after fixing storage, sharp throughput drops persisted. Hardware monitoring looked clean, and the drops reproduced even on a single node, ruling out flakey GPUs or networking. The only remaining variable was step count: longer planned runs (millions of steps) produced much sharper drops than short ones, pointing to a software issue, likely in the dataloader.

Fix #2 – Bring in TokenizedBytes Dataloader

They discovered that nanotron's default nanosets dataloader built a huge index that grew with the number of steps, eventually blowing up shared memory and causing stalls. By switching to their older, battle-tested TokenizedBytes dataloader inside nanotron, the throughput drops disappeared and performance matched ablations again.

Mystery #3 – The noisy loss

With TokenizedBytes, throughput was stable but the loss curve became noticeably noisier than with nanosets. This reminded them of a past bug where documents were shuffled but sequences inside batches weren't, leading to occasional spikes when a whole batch came from one low-quality or weird long document (e.g., a single bad code file).

Fix #3 – Shuffle at the Sequence Level

They confirmed the new dataloader read sequences sequentially from each document, so one long file could dominate a batch. Instead of rewriting the dataloader for random access (risky under time pressure), they pre-shuffled tokenized sequences offline per node and per epoch. This preserved good shuffling, smoothed loss, and avoided extra runtime complexity.

Launch, Take two

After fixing storage, dataloader, and shuffling, they relaunched with: stable throughput, no step-induced drops, and clean sequence-level shuffling. This time training was smooth, letting them finally focus on model learning rather than firefighting infrastructure issues.

Mystery #4 – Unsatisfactory performance

Around 1T tokens, evaluations showed the 3B model underperforming the old 1.7B at the same training stage, despite better architecture and data. Loss was decreasing, but improvements on benchmarks were slower than expected. Since most changes had been ablated carefully, they suspected one of the few remaining differences: tensor parallelism (TP).

Fix #4 – The Final Fix (TP Seeding Bug)

They trained a 1.7B model with and without TP in the model setup and found TP runs consistently worse. Digging into TP initialization, they found all TP ranks used the same random seed, leading to correlated weight initializations across shards and poor convergence. Fixing the seed to be seed + tp_rank removed this correlation; after that, TP and non-TP runs matched, and SmolLM3 finally outperformed the old 1.7B as expected.

Staying the course (High-level reflection)

This section reflects on the fact that scaling from ablations to full pretraining is never plug-and-play. It requires continuous monitoring, careful interpretation of metrics, and readiness to debug unexpected failures. Solid ablations and validated components made it possible to localize each problem quickly instead of guessing blindly.

Training monitoring: Beyond loss curves

They emphasize that loss alone is not enough: the TP bug was caught only because downstream evaluations lagged historical baselines, and intermediate checkpoints from the old model provided crucial reference points. On the infra side, the key metric is throughput (tokens/sec), monitored alongside GPU temps, utilization, memory, and node health via Grafana and alerts. Sustained throughput drops or eval regressions are treated as red flags.

Fix and restart vs. fix on the fly

Not every issue requires restarting training. In their case, restarting after 1T tokens made sense because the TP seeding bug meant the whole run was effectively handicapped. For other problems, especially loss spikes, you might continue training, rewind to a checkpoint, or skip bad batches. The text categorizes spikes into recoverable vs non-recoverable and lists likely causes: high learning rates, bad data, data–parameter interactions, poor initialization, or precision issues.

Stability techniques and spike mitigation

They list stability techniques: better data filtering and shuffling (e.g., removing documents with repeated n-grams), Z-loss regularization, removing weight decay on embeddings, QK-norm in attention, better initialization, and using BF16 instead of FP16. When spikes happen anyway, practical fixes include skipping problematic batches, tightening gradient clipping, or applying QK-norm mid-run as a “surgical” stabilization trick.

Mid-training: Multi-stage strategy

Modern LLM training is typically multi-stage, with different mixtures and sometimes different context lengths at different phases. Qwen3, for example, uses a general stage, a reasoning stage, and a long-context stage.

We follow the same philosophy: some interventions (e.g., adding higher-quality math and code datasets) are planned, others (e.g., fixing weak math/code performance) are reactive based on monitoring.

Stage 2 and stage 3 mixtures (Curriculum)

Model training is split into three stages:

- **Stage 1 (8T tokens):** base web/code/math mixture at 4k context.
- **Stage 2 (2T tokens):** inject higher-quality datasets (Stack-Edu, FineMath4+, InfiWebMath4+, MegaMath, Q&A data, synthetic rewrites).
- **Stage 3 (1.1T tokens):** during LR decay, further upsample high-quality math & code and add reasoning/instruction Q&A (OpenMathReasoning, OpenCodeReasoning, etc.).

This curriculum lets them exploit abundant data early and concentrate scarce, high-value data near the end.

Long context extension: From 4k to 128k

Model training starts at 4k context and later extends to 32k, then 64k, finally reaching 128k at inference using RoPE ABF + YaRN. Training at long context from the start is too expensive, so they follow a staged approach with separate LR schedules and ablations at each length.

They found that simple RULER evaluation was more reliable than HELMET on base models, and that using NoPE plus naturally occurring long documents in the existing mix was enough—upsampling extra long-doc data didn’t help further.

Wrapping up pretraining

Journey recap: using the “training compass” to choose goals, systematically ablate architecture and data, then surviving the messy realities of large-scale training (throughput collapse, dataloader bugs, TP seeding bug). The main lesson is that training LLMs is as much about disciplined experimentation, monitoring, and debugging as it is about fancy architectures.

This process ultimately produced a 3B model trained on 11T tokens that sits on the Pareto frontier versus Qwen3 models (competitive performance while being faster/cheaper), and sets the stage for the next phase: post-training to turn a raw base model into useful assistants and agents.

How to train an LLM

Principes, Decisions, and Engineering

Li Shang
lishang@slai.edu.cn