Large Language Models

Lecture 15 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. State the three **scaling laws** (compute, data, params) as power-law exponents.
2. Derive the **Chinchilla** D/N ≈ 20 optimum and explain why over-training is OK.
3. Describe **RoPE** (rotary position embedding) geometrically.
4. Explain **GQA** · fewer KV heads, same speed, same quality.
5. Contrast **DP / TP / PP** and know when to combine them.
6. Articulate **emergent abilities** · what they are and why they're contested.

Where we are

- **Transformer block** (L13) · stack of attention + FFN.
- **Tokenization + pretraining** (L14) · BPE, BERT / GPT / T5.

The **Transformer** stack is the architecture. The tokenizer is the input. Now let's scale.

> REFERENCE
>
> Today maps to the **Chinchilla paper** (Hoffmann 2022), HuggingFace course Ch 1, Karpathy's *State of GPT*. UDL Ch 12 supports this at a high level; LLM scale details come from the literature.

Four questions

1. Why do LLMs keep getting **better with more compute**?
2. What changed in positional encoding — and what is **RoPE**?
3. How do 100B+ models fit on GPUs — **GQA, distributed training**?
4. What are **emergent abilities** and why are they surprising?

What changed · 2018 to 2026

Architecture

Unchanged. Still a decoder-only Transformer. A student from 2018 who squinted at GPT-1 would recognize Llama-3.

Scale & engineering

Params · 117M → 1T+ (10,000×)
Data · 1B → 15T tokens (15,000×)
Compute · 10^{20} → 10^{25} FLOPs (100,000×)
Context · 512 → 1M tokens

KEY IDEA

"Bitter lesson" (Sutton) · general methods that leverage computation dominate specialized ones. LLMs are Exhibit A.

PART 1

Scaling laws

The empirical backbone of the LLM era

Scaling laws · the cake-baking analogy

KEY IDEA

You're baking a cake (the model's performance) with a fixed budget for ingredients (compute).

Should you spend on fancy flour (more **parameters**), or on sugar/eggs (more **data**)?

Scaling laws are the **recipe** · they tell you the optimal flour-to-sugar ratio for your specific budget.

The Chinchilla paper figured out that recipe. It's: roughly **20 tokens per parameter**. Spend more on flour than that, you're "overparameterized." Spend less, "undertrained." The sweet spot maximizes the cake.

The Chinchilla result

KEY IDEA

Hoffmann et al. 2022 · *"Training Compute-Optimal Large Language Models"*

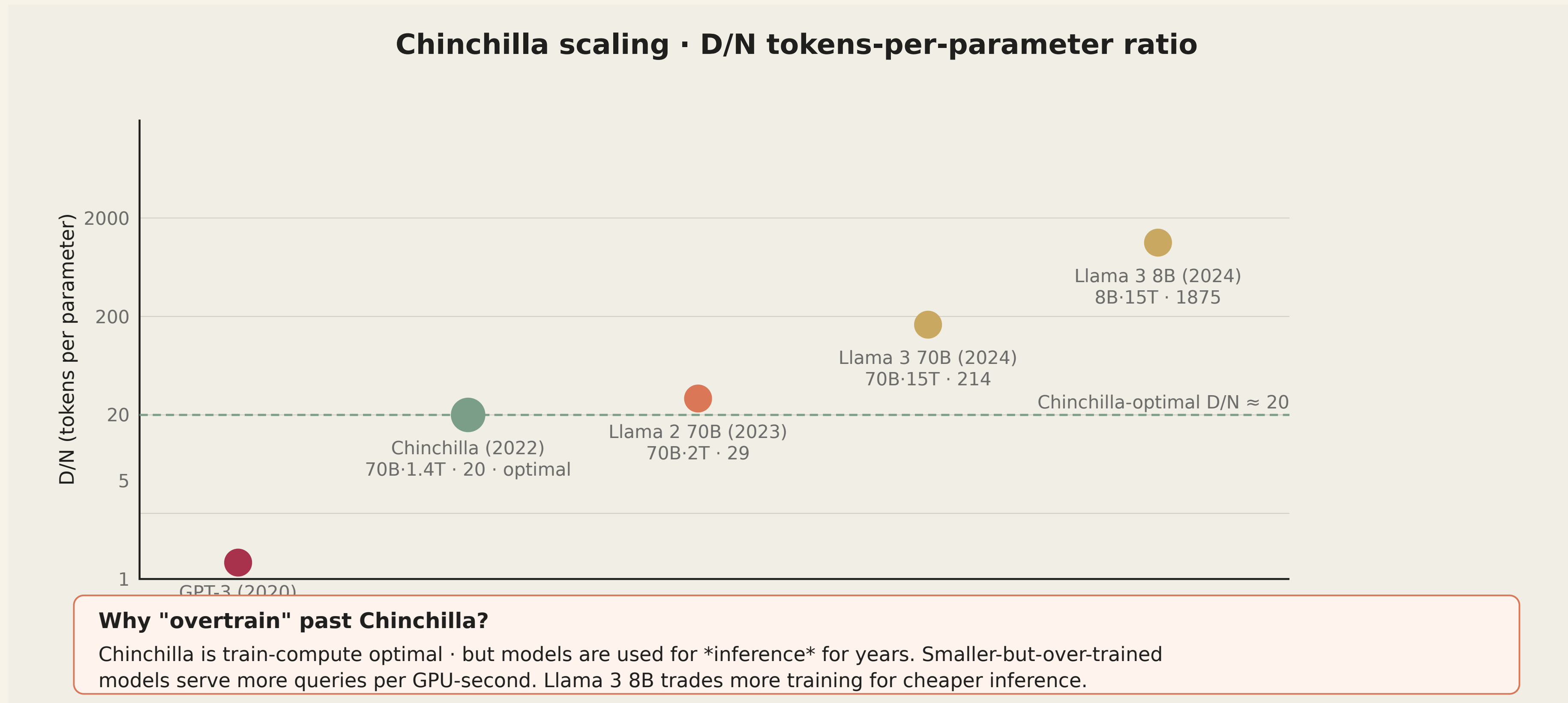
For a fixed compute budget C, performance is optimized when you scale **model size N** and **training tokens D** roughly **proportionally**.

$$C \approx 6 \cdot N \cdot D$$

Optimal ratio: **D / N ≈ 20 tokens per parameter**.

GPT-3 (175B params, 300B tokens · D/N ≈ 1.7) was hugely **undertrained** by this standard. Chinchilla (70B params, 1.4T tokens · D/N = 20) got better results with far fewer parameters.

Chinchilla · D/N across models

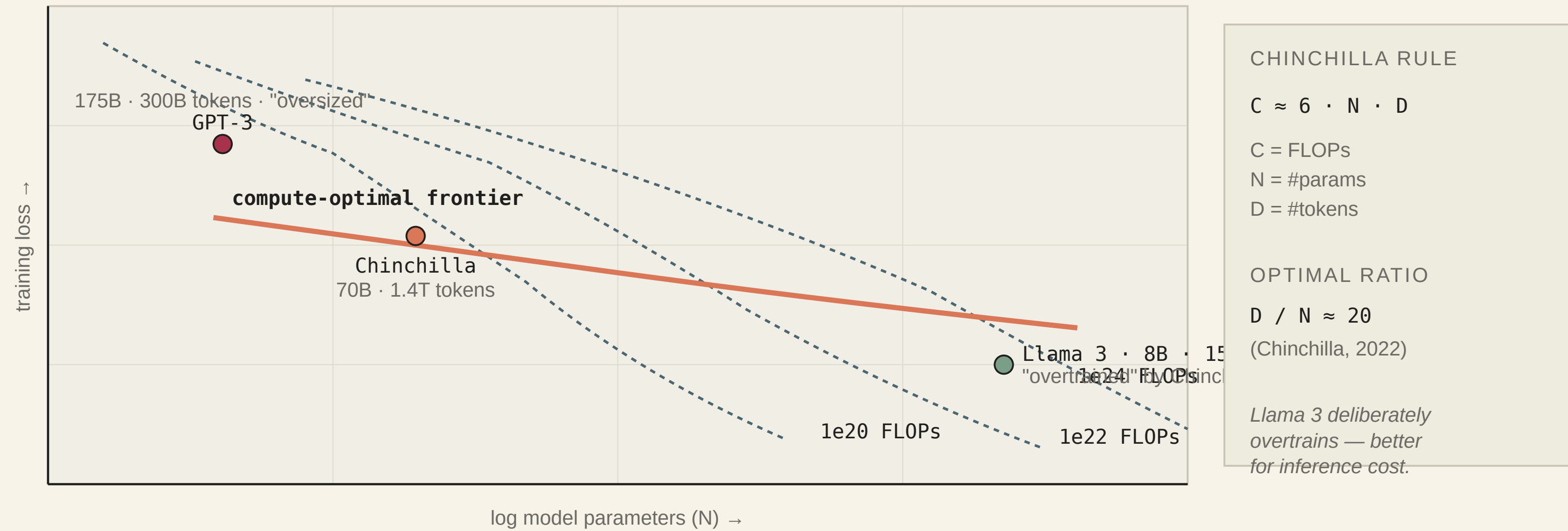


Why "overtrain" past Chinchilla?

Chinchilla is train-compute optimal · but models are used for *inference* for years. Smaller-but-over-trained models serve more queries per GPU-second. Llama 3 8B trades more training for cheaper inference.

Chinchilla · in one chart

Chinchilla scaling — for compute budget C, train ~20 tokens per parameter



The 2023+ twist · overtraining

Modern LLMs often train *well past* Chinchilla optimal:

MODEL	PARAMS	TOKENS	D/N	NOTES
Chinchilla	70B	1.4T	20	training-compute optimal
Llama 2 70B	70B	2T	29	slightly over
Llama 3 8B	8B	15T	1875	**wildly over**
Llama 3 70B	70B	15T	214	heavily over

Why overtrain?

INTUITION

Chinchilla optimizes *training compute*. But **inference** is where models earn their keep. A smaller, over-trained model has lower inference cost per query — you get back the extra training compute many times over.

Compute budget · derivation from two rules

Two rules from Chinchilla:

1. Budget · $C = 6\,N\,D$.
2. Recipe · $D = 20\,N$.

Substitute (2) into (1) to solve for N given a budget C:

$$C = 6\,N \cdot (20\,N) = 120\,N^2 \implies N = \sqrt{\frac{C}{120}}$$

Then $D = 20\,N$.

Worked example · spending $C = 1.2 \times 10^{24}$ FLOPs

1. **Solve for N.**
   $N^2 = (1.2 \times 10^{24})/120 = 10^{22}$
   $N = 10^{11}$ → **100 billion parameters.**
2. **Solve for D.**
   $D = 20 \cdot 10^{11} = 2 \times 10^{12}$ → **2 trillion tokens.**

For 10× more compute · N grows by $\sqrt{10} \approx 3.16$, D grows by $\sqrt{10}$ too. **Both scale as $\sqrt{C}$.** This is why GPT-4 (~1.8T) isn't 10× GPT-3 (175B) — Chinchilla's recipe says spread the budget across both axes.

Sub-optimal training · a table

SCENARIO	PARAMS	TOKENS	STATUS
GPT-3 (2020) · undertrained	175B	300B	too big, too few tokens
Chinchilla (2022) · optimal	70B	1.4T	train-compute sweet spot
Llama-3 8B (2024) · overtrained	8B	15T	inference-optimal for serving
A large startup's "bigger = better" model	500B	200B	wastes compute

WATCH OUT

The undertrained regime is more wasteful than the overtrained. GPT-3 used 10× Chinchilla's compute for similar final loss. Modern LLMs carefully size N, D together.

PART 2

RoPE · rotary positional encoding

The 2021 fix that stuck

Problems with sinusoidal / learned PE

Both inject position by **adding** a position vector to the token embedding:

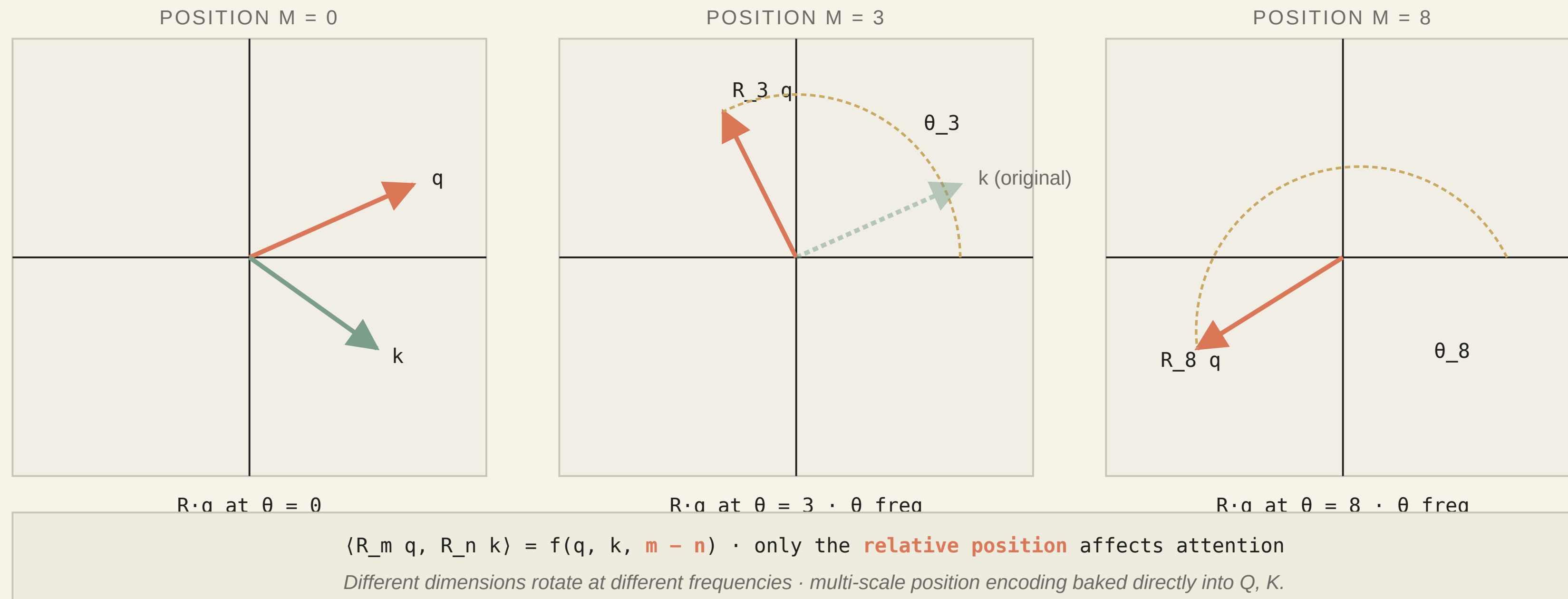
$$x_{\text{pos}}(t) = \text{emb}(token) + \text{pe}(t)$$

Problems:

- **Absolute** position only — model can't easily learn "2 positions apart" as a primitive.
- **Extrapolation** fails — trained on ≤ 4k tokens, breaks at 10k.

RoPE · rotation in pictures

RoPE · rotate Q and K by an angle proportional to position



RoPE · three key properties

DERIVATION

1. **Relative positions encoded naturally** · inner product after rotation depends only on $m - n$ (query-minus-key position), not absolute positions.
2. **Extrapolates beyond training length** · rotation frequencies are fixed; a model trained at 4k context can extend to 32k without re-training (with minor fixes).
3. **Zero added parameters** · rotation matrices are deterministic given position; no `nn.Embedding(max_len, d_model)` allocation.

KEY IDEA

Llama, Mistral, Qwen, GPT-NeoX all use RoPE in 2026. A 2021 paper (Su et al.) that took ~2 years to catch on is now the default.

Context length · the scaling wall

YEAR	FRONTIER MODEL	CONTEXT
2018	BERT	512 tokens
2020	GPT-3	2,048
2023	GPT-4	32k
2023	Claude 2	100k
2024	Gemini 1.5	**1,000,000**
2026	frontier	2-10M

INTUITION

What unlocked 1M? · **RoPE extrapolation**, **FlashAttention** (O(N) memory), **GQA** (smaller KV cache), and training on long documents from the start. No single trick; the stack compounds.

RoPE · the spinning-pointer intuition

INTUITION

Imagine Q and K as pointers on a clock face. Instead of *adding* a position vector, **rotate** the pointer by an angle that depends on its position.

- Token at position $m = 1$ → rotate by 10°.
- Token at position $n = 3$ → rotate by 30°.

Attention score = dot product of Q and K → depends on the **angle between them** = $30^\circ - 10^\circ = 20^\circ$, which is the relative offset $n - m = 2$.

The model learns about **relative positions directly**.

RoPE · derivation in 2D

For position m, define angle $\theta_m = m \cdot \Theta$ for some base Θ. Rotation matrix:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Apply to query (at position m) and key (at position n):
$q'_m = R_{\theta_m} q, \quad k'_n = R_{\theta_n} k$

Attention score:
$(q'_m)^\top k'_n = q^\top R_{\theta_m}^\top R_{\theta_n}\, k$

Two properties:

- $R_\theta^\top = R_{-\theta}$
- $R_{-\theta_m} R_{\theta_n} = R_{\theta_n - \theta_m}$

Substitute:
$(q'_m)^\top k'_n = q^\top R_{(n-m)\Theta}\, k$

The score depends only on the relative position $n - m$, not on absolutes. In high-dim, group dimensions into pairs; rotate each pair with a different frequency Θ_i. That's "block-diagonal."

Worked numeric · RoPE in 2D

$\Theta = 1$ rad. $q = [1, 2]$ at $m = 2$, $k = [3, 0]$ at $n = 3$.

Rotation matrices. $\theta_m = 2$, $\theta_n = 3$.

$$R_2 \approx \begin{pmatrix} -0.42 & -0.91 \\ 0.91 & -0.42 \end{pmatrix}, \quad R_3 \approx \begin{pmatrix} -0.99 & -0.14 \\ 0.14 & -0.99 \end{pmatrix}$$

Rotate.

$$q'_m = R_2 q \approx [-2.24,\ 0.07]^\top$$
$$k'_n = R_3 k \approx [-2.97,\ 0.42]^\top$$

Dot product. $(-2.24)(-2.97) + (0.07)(0.42) \approx 6.65 + 0.03 = \mathbf{6.68}$.

Verify with the relative-position form ($n - m = 1$): $q^\top R_1 k$ where $R_1 \approx \begin{pmatrix} 0.54 & -0.84 \\ 0.84 & 0.54 \end{pmatrix}$:

$R_1 k = [1.62, 2.52]$, $q \cdot [1.62, 2.52] = 1.62 + 5.04 = \mathbf{6.66}$ ✓ (rounding only).

Used in Llama 1/2/3, Mistral, PaLM, GPT-NeoX. **No extra params**, **extrapolates** beyond training length.

PART 3

Efficient attention

MQA, GQA, and the KV-cache

KV-cache · derivation, piece by piece

During autoregressive decoding we attend to **all previous tokens**. Recomputing K, V each step is wasteful → **cache** them.

Build the size, layer by layer:

1. **One token, one head, one layer** · store K and V vectors, each of size d_h.
   $\text{size} = 2\,d_h$ numbers
2. **One token, one layer, all heads.** Multiply by H:
   $\text{size} = 2\,H\,d_h$
3. **One token, all layers.** Multiply by L:
   $\text{size} = 2\,L\,H\,d_h$
4. **All T tokens.** Multiply by T:
   $\text{size} = 2\,L\,H\,d_h\,T$ numbers
5. **In bytes.** Multiply by 2 (for fp16):

$$\text{KV-cache memory} = 2 \cdot L \cdot H \cdot d_h \cdot T \cdot B \cdot 2 \text{ bytes}$$

Worked numeric · KV-cache for Llama 70B

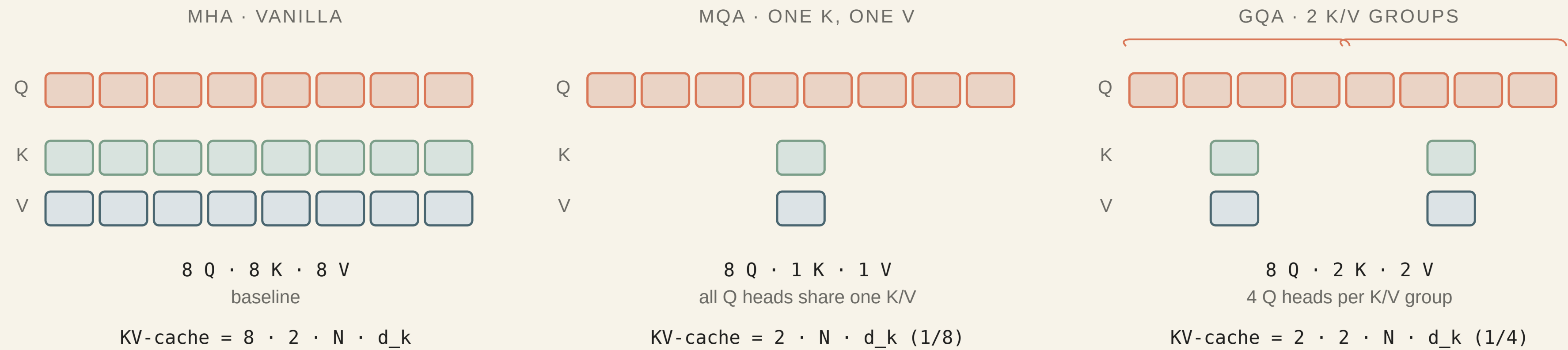
Setup: $L = 80,\ H = 64,\ d_h = 128,\ T = 32{,}000,\ B = 1$, fp16 (2 bytes).

$$\text{Cache} = 2 \cdot 80 \cdot 64 \cdot 128 \cdot 32{,}000 \cdot 1 \cdot 2$$

$$= 32{,}000 \cdot 80 \cdot 64 \cdot 128 \cdot 4 \approx 8.4 \times 10^{10} \text{ bytes} = \mathbf{84}\ \text{GB}$$

Punchline. The 70B weights (fp16) take $70 \cdot 10^9 \cdot 2 = 140$ GB. The KV-cache for **one** 32k-context sequence adds another **84 GB** — more than half the model size again. This is the biggest bottleneck in long-context LLM serving. Drives GQA (next), FlashAttention (L23), and KV-cache compression.

MHA vs MQA vs GQA



WHY IT MATTERS

During **autoregressive decoding**, the KV-cache has to be kept in memory for every past token.
At long context (100k+ tokens) this dominates inference cost — reducing K/V heads gives near-linear savings.

MQA (Shazeer 2019) was aggressive — lost some quality. GQA (Ainslie 2023) found the sweet spot;
Llama 2 70B and most modern LLMs use GQA.

GQA · the shared-notebook analogy

INTUITION

The KV-cache is the model's **notebook**.

- **MHA** · 64 students each keep a private 100-page notebook → 6400 pages.
- **GQA** · 8 study groups of 8 students share one notebook each → 800 pages. Huge saving, almost no quality drop.
- **MQA** · all 64 share one notebook → 100 pages. Maximum saving, but students may overwrite each other (quality drop).

How GQA shrinks the KV-cache

Refine the formula · let H_q = query heads, H_{kv} = key/value heads.

$$\text{Cache} = T \cdot L \cdot H_{kv} \cdot d_h \cdot 2 \cdot 2$$

Now compare for Llama 2 70B ($T = 32k$, $L = 80$, $d_h = 128$):

VARIANT	H_{kv}	CACHE SIZE
MHA (Llama 1)	64	$32k \cdot 80 \cdot 64 \cdot 128 \cdot 4 \approx \mathbf{84}\ \text{GB}$
GQA (Llama 2, 8 groups)	8	$32k \cdot 80 \cdot 8 \cdot 128 \cdot 4 \approx \mathbf{10.5}\ \text{GB}$
MQA	1	$\approx \mathbf{1.3}\ \text{GB}$ (quality drops)

GQA reduces KV-cache by $H_q/H_{kv} = 64/8 = 8\times$ with negligible quality loss — the modern default.

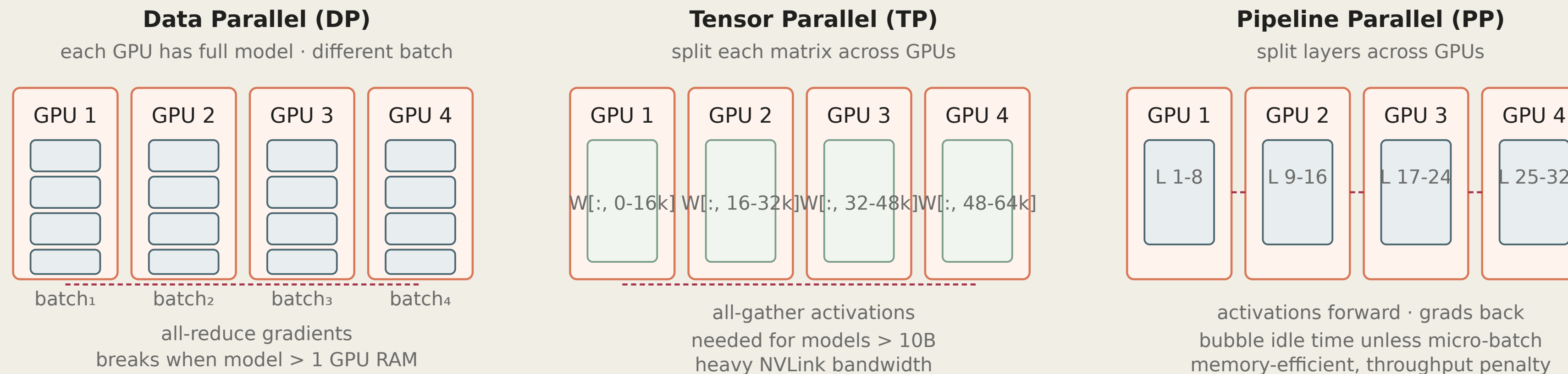
```
# In Llama 2 70B:
n_heads = 64        # query heads
n_kv    = 8         # GQA groups
d_head  = 128
```

PART 4

Distributed training

How you fit a 70B model on real hardware

Distributed training · three parallelisms



Distributed training · the LEGO-team analogy

KEY IDEA

You need to build a massive LEGO model (the LLM) that won't fit on one person's table (one GPU). Hire a team:

- **Data Parallel** · each person builds a full copy of the model, on different parts of the data.
- **Tensor Parallel** · the whole team works on **one** giant component together (split a matrix multiply).
- **Pipeline Parallel** · set up an assembly line · person 1 does layers 1-10, person 2 does 11-20, etc.

Frontier training combines **all three** · 3D parallelism. Each axis trades off memory, compute, and communication.

Three parallelism strategies

Data parallel (DP)

Each GPU has a **full copy** of the model, trains on different batches. Gradients averaged across GPUs.

Simple. Works for models that fit on one GPU. Breaks at 10B+.

Tensor parallel (TP)

Split each **matrix multiply** across GPUs. Each GPU holds a slice of W.

Megatron-LM. Required for >10B. Heavy all-reduce bandwidth.

Pipeline + 3D parallelism

Pipeline parallel (PP)

Split the **layer stack** across GPUs. Layer 1-10 on GPU 1, layer 11-20 on GPU 2, etc. Bubble of idle time unless you use micro-batching.

> KEY IDEA
>
> **Modern training runs combine all three** (3D parallelism). Add ZeRO (sharded optimizer state) and you get the full picture.

The 2026 reality

Training a 70B from scratch in 2026 · ~10k H100 GPUs for ~2 months.

- Data center cost: ~$100M
- Energy: ~1 GWh
- Engineering team: 50+

> IN PRACTICE
>
> Almost no one trains from scratch. **Everyone fine-tunes** open-weight models (Llama, Mistral, Qwen) with LoRA (next lecture).

PART 5

Emergent abilities

When more params unlock new behaviors

Why "emergence" is surprising

Null hypothesis · smooth scaling. As you make a model bigger, training loss decreases smoothly. A 10B model is a bit better than a 1B; a 100B model is a bit better than a 10B. Intuitive.

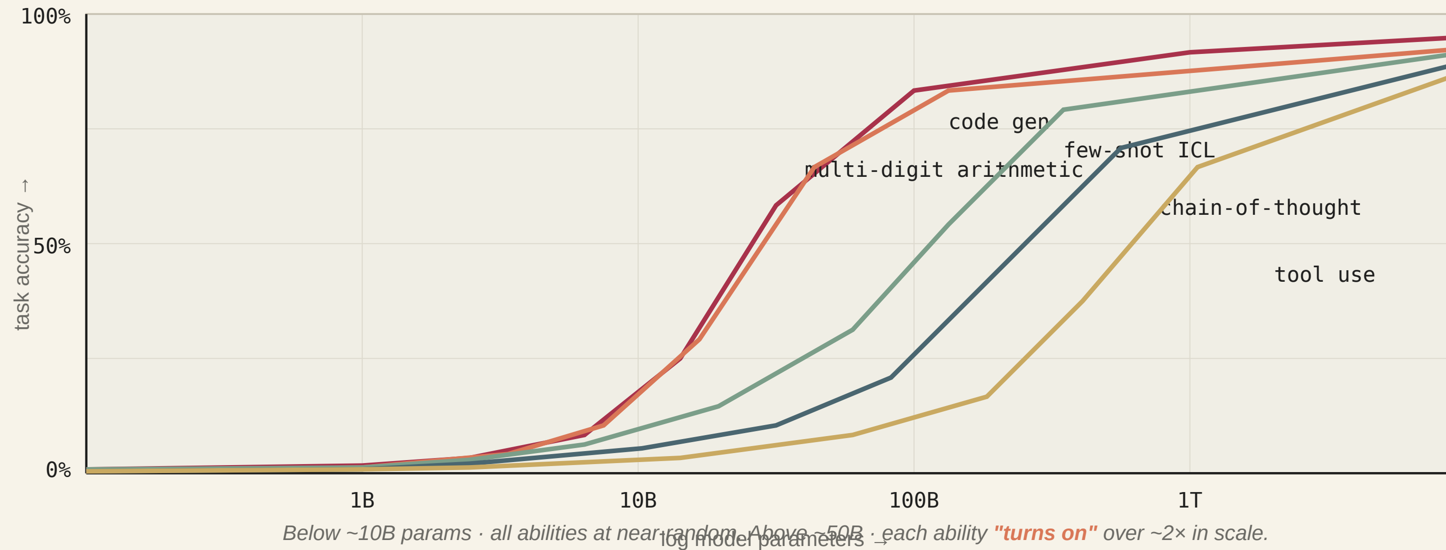
The surprise · for some specific complex tasks, this *doesn't* happen — performance is near-random until a threshold, then takes off.

Why? A multi-step task is a product of step accuracies:

- Small model · 50% per step. 3-step accuracy $= 0.5^3 = 12.5\%$ · barely above random.
- Larger model · 90% per step. 3-step accuracy $= 0.9^3 = 72.9\%$ · competent!

Smooth improvement in per-step accuracy translates to **what looks like a discontinuous jump** in end-to-end task performance.

Emergent abilities · the curves



Below ~10B params · all abilities at near-random. Above ~50B · each ability **"turns on"** over ~2× in scale.

What "emergent" means

An ability is **emergent** if it:

1. Is near-random performance at small scale (< 10B params).
2. Rapidly improves to competent at large scale (> 50B).

No one trained specifically for it. It just appears.

Emergence · the controversy

Emergentists say

- Discontinuous jumps in capability with scale.
- New qualitative behaviors (reasoning, tool use).
- Smaller models CAN'T do these at all.

Skeptics (Schaeffer 2023) say

- Many "emergent" curves are metric artifacts.
- Use a smoother metric (per-token log-prob vs exact match) and the curve becomes smooth.
- Still a real capability gap, but gradual.

KEY IDEA

Resolution · both sides are partially right. Capability improves continuously in log-probability, but certain *thresholded* tasks (match or fail) look discontinuous. The user experience is still of qualitative leaps.

Chain-of-thought · prompting unlocks reasoning

DERIVATION

Standard prompt · "Q: 23 × 47 = ?" → A: "1081" (often wrong)

CoT prompt · "Q: 23 × 47 = ? Let's think step by step." →
A: "23 × 47 = 23 × (50 − 3) = 1150 − 69 = 1081"

CoT unlocks **multi-digit arithmetic, commonsense, logic** at 60B+. Below that, CoT adds nothing (the model can't reason in steps either).

INTUITION

The prompt itself is a learnable control · "let's think step by step" (Kojima 2022) can add 15 points on GSM8K. No fine-tuning. This thread becomes reasoning models (o1, Claude thinking) in 2024.

ABILITY	ROUGHLY WHERE IT EMERGES

REFERENCE

Wei et al. 2022 · *"Emergent Abilities of Large Language Models."* Contested (Schaeffer et al. 2023 argue it's a metric artifact) but the phenomena are real.

In-context learning · the most surprising one

At pretraining, the model only learns next-token prediction. But at 100B+ params, it starts to **learn at inference time** from examples in the prompt:

```
Translate to French:
sea otter → loutre de mer
cheese → fromage
banana → banane
carrot → ???
```

The model has never seen the word "carrot" in its French dictionary. But given three examples, it figures out the task and produces "carotte".

KEY IDEA

This is **few-shot learning without weight updates**. Emergent at scale; the foundation of modern prompting.

Chain of thought

Prompting the model to "think step by step" dramatically improves multi-step reasoning:

```
Q: Roger has 5 tennis balls. He buys 2 more cans of 3
   tennis balls each. How many tennis balls?

Without CoT: "11 tennis balls." ← often wrong at small scale
With CoT:    "He starts with 5. 2 cans × 3 = 6 more.
              Total: 5 + 6 = 11." ← reliably correct
```

CoT *emerges* at scale. At 10B params, adding "let's think step by step" doesn't help. At 100B+, it adds 20+ percentage points on math benchmarks.

Reasoning models (2024+)

The latest generation — **o1, o3, Claude extended thinking, DeepSeek R1** — explicitly trains the model to produce long internal chain-of-thought *before* answering:

- Trained via RL with process rewards.
- Spends 10×–100× more compute per answer.
- Often dramatically better on math, code, logic.

This is where 2026 LLMs are. We'll see alignment + RLHF in the next lecture, then peek at reasoning in L16's final slide.

Summary · Lecture 15 — summary

- **Chinchilla** · D/N ≈ 20 tokens per parameter is compute-optimal. Modern Llama-style models intentionally overtrain for inference gains.
- **RoPE** · rotate Q and K by position-dependent angles; relative positions baked in; extrapolates. Default in 2026 LLMs.
- **GQA** · grouped-query attention shrinks KV-cache ~4× with near-zero quality loss. Default in Llama 2 70B+.
- **Distributed training** · DP + TP + PP + ZeRO. 70B from scratch is a ~$100M engineering feat.
- **Emergence** · few-shot learning, CoT reasoning, tool use — all appear at scale, not specifically trained.

Read before Lecture 16

HF PEFT docs; Ouyang 2022 (InstructGPT); Rafailov 2023 (DPO).

Next lecture

Alignment & Fine-tuning — SFT, LoRA, QLoRA, RLHF, DPO, one slide on reasoning.

> NOTEBOOK
>
> **Notebook 15** · `15-rope.ipynb` — implement RoPE from scratch; compare to sinusoidal PE on extrapolation.