

LLM from scratch

Aayush Bajaj

July 10, 2026

Contents

1	the objective	1
2	scaling laws	2
3	the training pipeline	2
4	inference	3
4.1	sampling	3
4.2	making it fast	4
5	emergent abilities, honestly	4
6	prompting and evaluation	4
7	see also	5

a large language model is a decoder-only **transformer** trained on one absurdly simple objective — predict the next token — scaled until the emergent behaviour stops looking simple. this page is the map from that objective to a deployed assistant: the loss, the scaling laws that size the model, the pretrain-align pipeline, and the inference tricks. the hands-on build is **nanogpt**.

everything downstream — chat, code, reasoning — is a fine-tune or a prompt on top of this one objective

1 the objective

given a token sequence $x_1, \dots, x_T$, the model factorises the joint probability autoregressively and is trained to maximise its log-likelihood — equivalently, to minimise the average next-token **cross-entropy**:

$$\mathcal{L}(\theta) = -\frac{1}{T} \sum_{t=1}^T \log p_{\theta}(x_t \mid x_{<t}) . \quad (1)$$

that is the entire pretraining loss. three things worth stating plainly:

- the label for position t is just x_{t+1} — the data *is* its own supervision, which is why the internet’s worth of raw text suffices (self-supervised, no annotation).
- the **causal mask** makes all T predictions computable in one forward pass, so a length- T sequence yields T training signals at once.
- $\exp(\mathcal{L})$ is **perplexity**, the effective branching factor — “how many equally-likely

tokens is the model choosing among”. a perplexity of 10 means the model is as uncertain as a fair 10-sided die at each step.

2 scaling laws

the empirical discovery that made the field industrial: test loss falls as a smooth **power law** in model parameters N , dataset tokens D , and compute C , over many orders of magnitude (kaplan et al. 2020, *scaling laws for neural language models*):

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad (2)$$

with small exponents ($\alpha_N \approx 0.076$ in the original fit). the practical question is how to spend a fixed compute budget $C \approx 6ND$ ¹ between a bigger model and more data.

kaplan et al. concluded compute should go mostly to model size. the **chinchilla** correction (hoffmann et al. 2022, *training compute-optimal large language models*) re-ran the sweep more carefully and found N and D should scale in **equal proportion**: roughly **20 training tokens per parameter** at the compute-optimal point. honest caveats: the 20:1 ratio is compute-optimal for *training*, not for *deployment* — if you will serve a model billions of times, it is rational to “over-train” a smaller model far past 20:1 to save inference cost forever, which is exactly what the llama models do. and the constants are dataset- and architecture-dependent; 20:1 is a landmark, not a law of nature.

gpt-3 at 175B params trained on 300B tokens was badly under-trained by this rule; chinchilla at 70B on 1.4T beat it

3 the training pipeline

a shipped assistant is three stages stacked on the pretrained base.

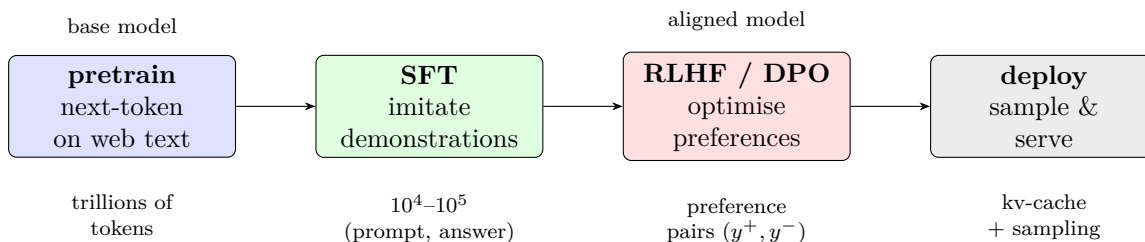


Figure 1: the modern llm pipeline: self-supervised pretraining produces a base model; supervised fine-tuning teaches instruction-following; preference optimisation (rlhf or dpo) aligns it to human judgements.

- **pretraining.** the cross-entropy objective above, on trillions of tokens. produces a **base model** — a superb next-token predictor that will happily continue a prompt but has no notion of “being helpful”.

¹the factor 6 counts multiply-accumulates per parameter per token: roughly 2 for the forward pass and 4 for the backward pass, so total training compute is $C \approx 6ND$ flops.

references

Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia (2017). *Attention is All You Need*.

- **supervised fine-tuning (sft).** continue the same next-token training on a curated set of (instruction, ideal-response) pairs. cheap, and it teaches the format of following instructions. detail lives at [fine tuning](#).
- **preference optimisation.** sft imitates demonstrations; the last stage optimises against *comparisons*, which are easier for humans to give (“A is better than B”) than perfect answers.
 - **rlhf** (ouyang et al. 2022, *training language models to follow instructions*): train a reward model on human preference pairs, then optimise the policy against it with ppo, penalised by a kl term to stay near the sft model.
 - **dpo** (rafailov et al. 2023, *direct preference optimization*): skip the reward model and rl entirely. dpo shows the rlhf objective has a closed-form optimum, giving a simple classification loss directly on preference pairs (y^+ preferred over y^-):

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x, y^+, y^-)} \left[\log \sigma \left(\beta \log \frac{\pi_{\theta}(y^+ | x)}{\pi_{\text{ref}}(y^+ | x)} - \beta \log \frac{\pi_{\theta}(y^- | x)}{\pi_{\text{ref}}(y^- | x)} \right) \right], \quad (3)$$

where π_{ref} is the frozen sft model and β controls how far the policy may drift. same goal as rlhf, no reward model, no rollout instability — which is why it took over.

4 inference

4.1 sampling

the base model outputs a distribution over the vocabulary; how you draw from it is a decoding choice, not a training one.

- **temperature T :** divide logits by T before softmax. $T < 1$ sharpens toward greedy (safe, repetitive), $T > 1$ flattens (diverse, riskier), $T \rightarrow 0$ is argmax.
- **top-k:** keep only the k most probable tokens, renormalise, sample.
- **top-p (nucleus):** keep the smallest set of tokens whose cumulative probability exceeds p , renormalise, sample — an *adaptive* cutoff that widens on flat distributions and narrows on peaked ones.

```

1 import numpy as np
2
3 vocab = ["the", "a", "cat", "dog", "ran", "sat", "quickly", "blue", "?", "!"]
4 logits = np.array([3.2, 2.9, 1.5, 1.4, 0.8, 0.7, -0.3, -1.0, -2.0, -3.5])
5
6 def softmax(z):
7     z = z - z.max(); e = np.exp(z); return e / e.sum()
8
9 def temperature(logits, T):
10     return softmax(logits / T)
11
12 def top_k(logits, k):
13     idx = np.argsort(logits)[::-1][:k] # keep k largest
14     masked = np.full_like(logits, -np.inf); masked[idx] = logits[idx]
15     return softmax(masked)
16
17 def top_p(logits, p):
18     probs = softmax(logits)
19     order = np.argsort(probs)[::-1]
20     cum = np.cumsum(probs[order])

```

```

21 keep = order[:np.searchsorted(cum, p) + 1] # smallest set with mass >= p
22 masked = np.full_like(logits, -np.inf); masked[keep] = logits[keep]
23 return softmax(masked)
24
25 def show(name, probs):
26     top = np.argsort(probs)[::-1][:5]
27     print(f"{name:<16} " + " ".join(f"{vocab[i]}={probs[i]:.3f}" for i in top
    ↪     if probs[i] > 0))
28
29 show("T=1.0", temperature(logits, 1.0))
30 show("T=0.5 (sharp)", temperature(logits, 0.5))
31 show("T=2.0 (flat)", temperature(logits, 2.0))
32 show("top-k=3", top_k(logits, 3))
33 show("top-p=0.9", top_p(logits, 0.9))

```

T=1.0	the=0.432	a=0.320	cat=0.079	dog=0.071	ran=0.039
T=0.5 (sharp)	the=0.615	a=0.338	cat=0.021	dog=0.017	ran=0.005
T=2.0 (flat)	the=0.271	a=0.233	cat=0.116	dog=0.110	ran=0.082
top-k=3	the=0.520	a=0.385	cat=0.095		
top-p=0.9	the=0.479	a=0.355	cat=0.087	dog=0.079	

lowering T concentrates mass on **the** and **a**; raising it spreads probability toward the tail. top-k=3 hard-truncates to three tokens; top-p=0.9 keeps four here because that is the smallest nucleus reaching 90% mass.

4.2 making it fast

- **kv-cache.** the dominant inference optimisation: cache each layer’s keys and values so generating token $n + 1$ costs $O(nd)$ attention instead of recomputing the prefix at $O(n^2d)$. its memory footprint is the main limit on context length — see [transformers](#).
- **speculative decoding.** run a small cheap “draft” model to propose several tokens, then verify them all in *one* forward pass of the big model, accepting each draft token with probability $\min(1, p_{\text{big}}/p_{\text{draft}})$ and resampling from the corrected residual distribution on the first rejection — which provably preserves the big model’s output distribution (greedy decoding degenerates to “longest agreeing prefix”). because verification is parallel while generation is sequential, this gives a 2–3x wall-clock speedup with **no change to the output distribution** — the big model’s samples are provably unchanged.

5 emergent abilities, honestly

some capabilities (multi-step arithmetic, instruction following, in-context learning) appear to switch on abruptly past a scale threshold rather than improving smoothly (wei et al. 2022, *emergent abilities of large language models*). the honest caveat: a well-cited rebuttal (schaeffer et al. 2023, *are emergent abilities a mirage?*) argues much of the apparent discontinuity is an artefact of **discontinuous metrics** — exact-match accuracy jumps, but the underlying per-token log-likelihood improves smoothly. the phenomenon is real and useful to plan around; the “phase transition” framing is contested.

6 prompting and evaluation

- **few-shot / in-context learning.** put k worked examples in the prompt and the model generalises the pattern with no weight update — a behaviour that itself

emerged with scale (brown et al. 2020, *language models are few-shot learners*).

- **chain-of-thought.** prompting the model to “think step by step” before answering elicits intermediate reasoning and sharply improves multi-step tasks — the model conditions its final answer on its own generated scratchpad (wei et al. 2022, *chain-of-thought prompting*).
- **evaluation.** benchmarks (mmlu for knowledge, gsm8k for maths, humaneval for code) give comparable numbers but leak into training sets over time; pairwise human or llm-judge comparisons (arena-style elo) capture preference but are noisy and gameable. no single number suffices — every eval measures a proxy.

(Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia, 2017) is the architecture underneath all of it; the **tokeniser** fixes what counts as a token in the loss above.

7 see also

- **nanogpt** — build the decoder that this page scales up
- **running llms locally** — inference on consumer hardware
- **fine tuning llms** — the sft and adapter stage in detail
- **retrieval-augmented generation** — grounding the model in external documents
- **tokenisers** — how the input sequence is built
- **transformers** — the block that is stacked and scaled