2. Pretraining

High-level concepts for interviews and real-world LLM work

Table of contents

1 Overview

Most ML engineers will **not** pretrain an LLM from scratch at work—but interviews still test whether you understand the *physics* and *failure modes* of pretraining, because they show up everywhere (mid-training, data pipelines, serving, eval, safety).

> Note
>
> **ELI5:** *Pretraining is teaching a model to predict the next word from lots of text so it learns "how language works" and picks up general knowledge.*

1.1 What you should know (even if you never pretrain from scratch)

- How data quality and mixture impact downstream behavior (and safety).
- Why compute/memory constraints force tradeoffs (context length, batch size, architecture).
- How to monitor training and debug regressions (loss spikes, contamination, instability).
- How pretraining choices constrain mid-training, SFT, and RL.

```
flowchart LR
  A[Data lake] --> B[Filtering & dedup]
  B --> C[Tokenizer]
  C --> D[Pretraining run]
  D --> E[Checkpoints & eval]
  E --> F[Base model  0]
```

```
F --> G[Mid-training / CPT]
F --> H[SFT / DPO / RL]
```

---

2 Learning goals

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

- Explain what pretraining optimizes (and what it does **not** guarantee).
- Describe the end-to-end data pipeline at a high level (sources → cleaning → mixture → tokens).
- Estimate the biggest drivers of cost (parameters, context, batch size, precision, parallelism).
- Name the top failure modes (contamination, dedup errors, instability, safety regressions) and how to detect them.
- Map pretraining decisions to later stages (CPT, alignment, tool use, reasoning, inference constraints).

> Tip
>
> **ELI5:** *Pretraining builds the "engine." Later stages teach the "driver" and add "features."*

---

3 What pretraining is (and isn't)

3.1 The objective

Standard LLM pretraining is **next-token prediction** over large corpora.

- It learns broad linguistic/statistical structure and absorbs patterns in the data.
- It does not inherently learn *truth*—it learns what text usually looks like.

> Note
>
> **ELI5:** *It's like learning by reading a huge library: you get fluent, but you can also pick up mistakes from bad books.*

3.2 "Capability" vs "behavior"

- **Capabilities** (language fluency, general knowledge, pattern recognition) mostly come from pretraining + mid-training.

- **Behavior** (helpfulness, refusal style, tool schemas, safety policy) mostly comes from post-training.

---

4 Data (the main lever)

If you only remember one thing: **data dominates**.

> Note
>
> **ELI5:** *The model becomes what it eats—if the data is noisy, the model is noisy.*

4.1 Where data comes from (high-level)

- **Web** (broad coverage, high noise)
- **Books / papers** (higher quality, licensing constraints)
- **Code repositories** (tool use + reasoning patterns, but licensing and leakage risk)
- **Domain corpora** (company docs, medical, legal, finance)
- **Synthetic / curated mixes** (lower noise, risk of bias and mode collapse)

4.2 Data governance (practical reality)

- licensing/terms-of-use, privacy and PII policies, internal security requirements
- audit trails: what was trained on, when, and how it was filtered

> Warning
>
> **ELI5:** *You can't "untrain" sensitive data easily—avoid ingesting it in the first place.*

---

5 Data processing (what interviews ask about)

5.1 Filtering

Typical filters (conceptual categories): - **URL/domain filtering:** remove spam farms, low-quality domains, unsafe sources - **content filtering:** boilerplate, templates, link farms, gibberish - **language filtering:** keep target languages, remove mixed/noise - **safety filtering:** PII, explicit content, disallowed categories

Tip

ELI5: *Filtering is throwing away the trash so the model doesn't learn garbage.*

5.2 Deduplication and contamination

Two classic interview topics:

- **Dedup (train-train):** removes repeats that cause memorization and overfitting to boilerplate.
- **Contamination (train-test):** prevents "cheating" on benchmarks (model saw test items during training).

Practical strategies: - exact match hashing + near-duplicate detection (shingles/minhash/embeddings) - benchmark holdout dedup (remove overlap with eval sets) - keep metadata for audit and reproducibility

Note

ELI5: *Dedup is not reading the same page 1,000 times; contamination is not reading the exam answers before the test.*

---

6 Data mixture and distribution

Pretraining is rarely "one dataset." It's a **mixture**.

6.1 Why mixture matters

- more code → better coding/tool patterns, sometimes worse chat style
- more math/derivations → better symbolic reasoning, sometimes more verbosity
- more domain text → better domain recall, risk of forgetting general skills if overdone

Note

ELI5: *Mixing data is like planning a diet: too much of one food can cause deficiencies elsewhere.*

6.2 Practical mixture design (high-level)

A common pattern: - majority **general** text for broad capabilities - a meaningful slice of **high-quality** data for grounding and style - targeted **specialty** data (code/math/domain) based on product goals

Rule of thumb (for interviews): Start with a conservative domain fraction, then increase only if you can show measurable gains without broad regressions.

---

7 Tokenization (the "hidden" engineering detail)

Tokenization affects: - **compression ratio** (tokens per character/word) - **throughput and cost** (more tokens → more compute) - **domain performance** (jargon splitting hurts) - **multilingual tradeoffs** (a tokenizer is never perfect for every language)

> Note
>
> **ELI5:** *A tokenizer is a way to chop text into LEGO bricks—the wrong bricks make building slow and messy.*

7.1 Key choices (high-level)

- BPE/Unigram variants (implementation detail; interviews rarely need deep internals)
- vocabulary size (tradeoff: fewer tokens vs larger embedding tables)
- normalization rules (case, punctuation, unicode)
- handling numbers, whitespace, code symbols

7.2 When you need tokenizer extension

If important domain terms fragment into many sub-tokens, extension can help—but it creates training/compatibility complexity.

> Tip
>
> **ELI5:** *Tokenizer extension is adding new "words" to the model's dictionary so it stops spelling jargon out letter by letter.*

---

8 Compute and scaling (back-of-the-envelope level)

This chapter stays high-level; you only need enough to reason about tradeoffs.

8.1 What drives cost

- **model size** (parameters)
- **context length**
- **tokens trained** (dataset size × epochs)
- **precision** (bf16/fp16/fp8/int8)
- **parallelism and efficiency** (pipeline/tensor/data parallel, kernel quality)

> Note
>
> **ELI5:** *Training cost is mostly "how many numbers you multiply" times "how many tokens you see."*

8.2 The simplest interview estimation

When asked "what makes training 10× more expensive?": - doubling parameters roughly doubles FLOPs per token - doubling context can increase attention cost and memory - increasing tokens trained increases compute linearly

---

9 Training recipe (conceptual, not cookbook)

You don't need the full optimizer math for most roles, but you should recognize the knobs.

9.1 Common knobs

- learning rate schedule (warmup + decay)
- batch size / gradient accumulation
- regularization (weight decay, dropout)
- checkpointing strategy (frequency, best-of evals)
- precision (bf16/fp16) and stability tradeoffs

> Tip
>
> **ELI5:** *The learning rate is how big each "correction step" is—too big and you wobble, too small and you crawl.*

9.2 Distributed training (names you should know)

- **Data parallel (DP):** split batches across GPUs
- **Tensor parallel (TP):** split layers' matrix multiplies across GPUs
- **Pipeline parallel (PP):** split layers into stages across GPUs

> Note
>
> **ELI5:** *Distributed training is like having multiple cooks: DP splits orders, TP splits chopping, PP splits the recipe into stations.*

---

10 Monitoring and debugging

10.1 What to watch

- **loss curves** (overall + per-domain channels)
- **loss spikes** (do they recover?)
- **perplexity** on a fixed eval set (track trend, not single numbers)
- **throughput** (tokens/sec), GPU utilization, memory, kernel efficiency
- **quality gates** (small benchmark suite)

Practical: keep a fixed "probe set" (e.g., 200 prompts/docs) and track PPL and a few qualitative generations.

> Tip
>
> **ELI5:** *Monitoring is checking the dashboard while driving so you don't discover an engine failure after you crash.*

10.2 Common failure modes (what interviewers love)

- **instability:** exploding loss, NaNs, divergence
- **overfitting/memorization:** too many repeats, insufficient dedup
- **contamination:** suspiciously high benchmark scores with poor generalization
- **distribution shift:** model gets better in one area and worse elsewhere
- **safety regressions:** new unsafe patterns appear due to data changes

---

11 Evaluation (high level)

11.1 Perplexity (PPL)

PPL is a useful sanity check, but: - compare meaningfully only within the same tokenizer + eval setup - lower PPL doesn't always mean better downstream instruction following

> **Note**
>
> **ELI5:** *Perplexity is how "surprised" the model is by the next word—less surprised often means it learned the patterns better.*

11.2 Benchmarks (what to say in interviews)

- use a **small, relevant** benchmark suite as regression gates
- include **domain evals** if you trained on domain data
- include **safety and leakage checks** as part of standard CI for models

---

12 How pretraining choices show up later

12.1 Mid-training (CPT)

- CPT is "more pretraining," but on a narrower distribution.
- If your base pretraining data is weak in a domain, CPT must do more work (and risk more regressions).

12.2 Post-training (SFT/RL)

- if pretraining data includes good tool-use patterns, SFT/RL is easier
- if the base is heavily contaminated or biased, alignment must fight upstream

12.3 Inference

- tokenizer and context-length choices influence KV cache size and serving cost
- architecture choices (e.g., GQA/MQA) affect memory pressure at decode

---

13 Interview drills

1. **Data:** How would you build a data pipeline that avoids benchmark contamination?

2. **Mixture:** You added more code data and coding improved, but chat quality dropped—why?
3. **Tokenization:** Your domain terms split into many pieces; what do you do and what can go wrong?
4. **Monitoring:** Loss spiked 3× mid-run then recovered—how do you triage?
5. **System link:** Why can a longer context window make inference much more expensive?

---

14 Appendix: Minimal pseudocode (conceptual)

```
# Conceptual sketch: pretraining data pipeline
raw = crawl_web() + load_books() + load_code()
filtered = filter_urls(raw)
filtered = filter_content(filtered)
deduped = deduplicate(filtered)
tokens = tokenize(deduped, tokenizer)

# Conceptual sketch: training loop
for batch in batcher(tokens):
    loss = next_token_loss(model(batch), batch.targets)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

ℹ Note

ELI5: *The code is simple on paper—the hard part is data quality, scaling, and not breaking things.*