Announcements 10/01

- Final Project Proposal due on Friday 10/02
- Upload the propodal to gitlab: https://gitlab.cs.umd.edu/cmsc389a/fall2026/student-repos/final-project/proposal-
- Midterm exam 1: Develpp a fullstack application in 2 days (10/08 and 10/09)

How LLMs Work

From Internet Text to ChatGPT

Part 1: Pre-training → Base model
Part 2: Post-training, psychology, RL

Where This Lecture Comes From

These slides follow the arc of Andrej Karpathy's lecture ***Deep Dive into LLMs like ChatGPT*** (YouTube, 3.5 hours), adapted for this course and paired with code you can run.

- Video: `youtube.com/watch?v=7xTGNNLPyMI`
- Short companion: Anfal Mushtaq's **TL;DR** write-up of the same lecture (also hosts Karpathy's full excalidraw diagram)
- Several figures in this deck are Karpathy's diagrams

Our addition: TerpLM — a complete LLM pipeline (tokenizer, parameters, loss, sampling, temperature, *post-training*) that trains on your laptop in about two minutes. Lives in `code/terplm/`.

Learning Objectives

By the end of these two lectures you should be able to:

1. Describe the **three training stages** — pre-training, supervised fine-tuning, reinforcement learning — what data each consumes, what it produces, roughly what it costs
2. Explain **byte-pair encoding** and predict its consequences (case sensitivity, spelling and counting failures)
3. State what a language model **computes** (a distribution over the next token), what its **parameters** are, what the **loss** measures, and what changes in training vs. inference
4. Explain why a base model is an **"internet document simulator"** and the assistant is a **simulation of a human data labeler**
5. Use **parameters (vague recollection) vs. context window (working memory)** to predict hallucination and how tools fix it
6. Explain **"models need tokens to think"**
7. Distinguish **SFT vs. RL vs. RLHF**; explain "thinking" models (DeepSeek-R1, o1/o3)
8. **Train, sample from, and post-train** a miniature LLM on your laptop

Part 1

Pre-training → the Base Model

Why Look Under the Hood?

The central skill of this course is **judgment over AI output**.

Judging well requires a *mechanical* understanding of what's behind the text box:

- What happens to your words when you press Enter?
- Where does the reply come from?
- Which part of the pipeline put it there?

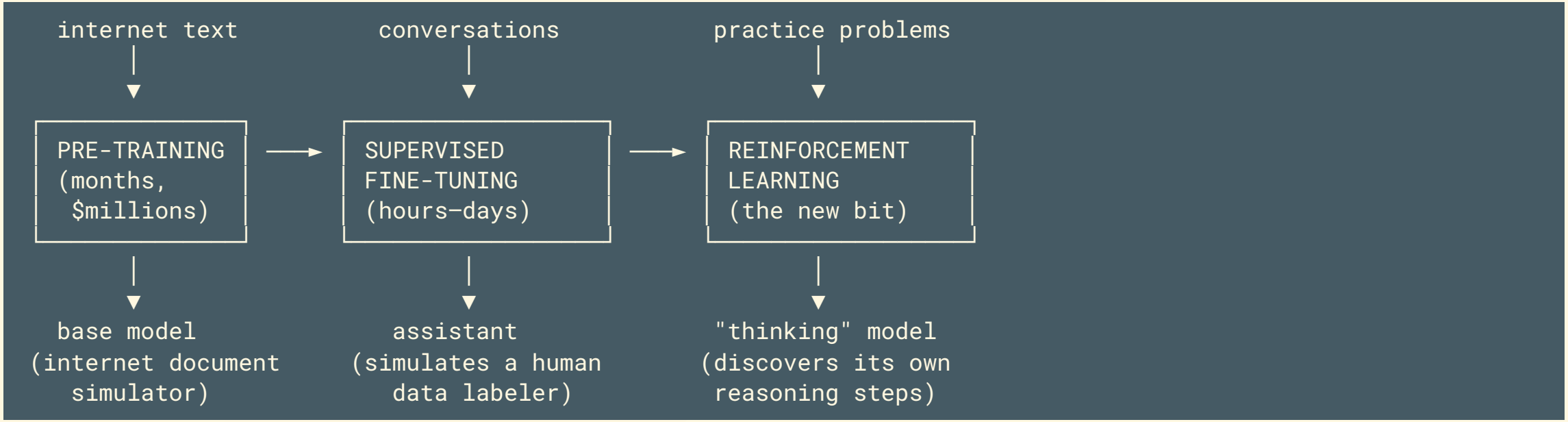
Why Look Under the Hood?

Famous failures stop being mysteries and become **predictable consequences**:

- hallucinated APIs
- wrong arithmetic
- miscounted letters in "strawberry"
- confident nonsense about recent events

> Engineers who can predict failure modes prompt better, verify smarter, and design better systems around the model.

The Big Picture: Three Sequential Stages



The Running Example: TerpLM

A complete LM pipeline small enough to train on a laptop:

- a **real BPE tokenizer** (`bpe.py`, ~60 lines)
- a **gradient-trained next-token predictor** (`model.py`)
- **sampling-based inference** with temperature (`generate.py`)
- a **"post-training"** stage (`train.py --resume`)

Its network is deliberately the *dumbest one that works*: a **bigram** model — predicts the next token from **only the single previous token**.

```
terplm/
├── terplm/
│   ├── bpe.py        # tokenizer
│   ├── model.py      # bigram LM
│   ├── train.py      # (post-)training
│   ├── generate.py   # inference
│   └── demo.py       # BPE demo
├── data/
│   ├── tiny_corpus.txt
│   └── chat_corpus.txt
├── scripts/fetch_shakespeare.sh
└── tests/
```

Missing vs. GPT-4: the transformer (thousands of tokens of context) and ~11 orders of magnitude of scale. Its failures are as instructive as its successes.

Setup (do this now, it takes a minute)

```
cd code/terplm
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt      # just numpy (+ pytest, hypothesis)
pytest                                # 11 tests — BPE round-trip, distribution sanity

bash scripts/fetch_shakespeare.sh    # 1.1 MB "tiny shakespeare" corpus
```

The whole pipeline, which we'll walk through across both lectures:

```
python -m terplm.train data/shakespeare.txt --merges 256 --steps 3000   # 1. pre-train
python -m terplm.generate --prompt "ROMEO:" --seed 1                    # 2. inference
python -m terplm.generate --prompt "ROMEO:" --temperature 0.05          #    temperature
python -m terplm.train data/chat_corpus.txt --resume --steps 2000       # 3. post-train
python -m terplm.generate --prompt "<|im_start|>user" --temperature 0.6 # 4. "assistant"
```

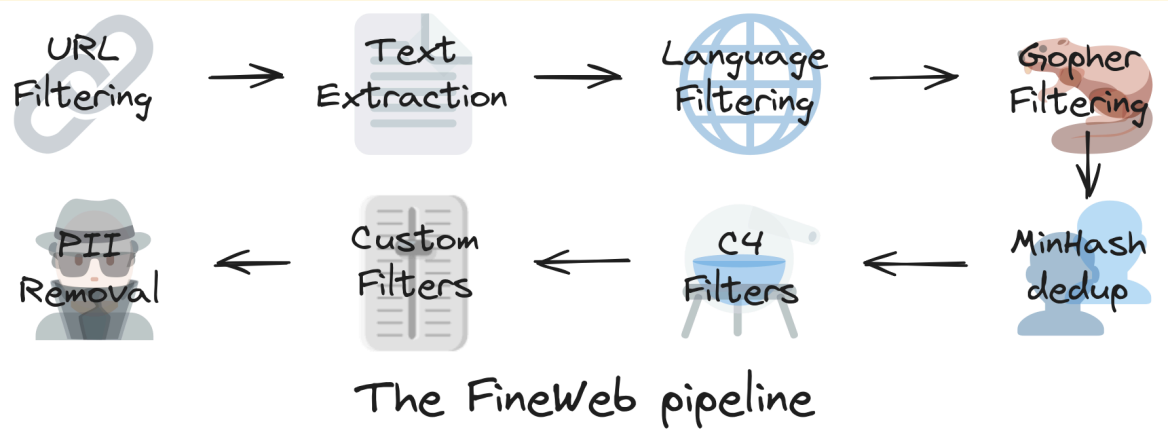
Stage 1: Pre-training

Step one: download the internet

Step One: Download the Internet

Everything the model will ever "know" comes from this corpus — so we want **lots** of text, of **high quality** and **high diversity**.

- Public example: Hugging Face's **FineWeb**. Every major lab (OpenAI, Anthropic, Google) has an internal equivalent.
- Starting point: **Common Crawl** — crawling the web since 2007; ~2.7 billion pages indexed by 2024. Raw archive = petabytes of gzip'd HTML.



The FineWeb filtering pipeline, from raw crawl to training text. Diagram: Andrej Karpathy, *Deep Dive into LLMs like ChatGPT*.

The Filtering Pipeline

#	Stage	What it does	Why it matters
1	**URL filtering**	Drop blocklisted domains: malware, spam, marketing, adult	Cheapest filter — runs first
2	**Text extraction**	Raw HTML → text; strip `<div>`s, CSS, JS, nav menus, ads	Harder than it sounds: what's "content" vs. "chrome"?
3	**Language filtering**	Classifier keeps pages ≥ 65% English (FineWeb)	**Design choice:** filter out Spanish → model is bad at Spanish
4	**Quality filters**	Gopher / C4 heuristic rules	Drop boilerplate, junk, degenerate pages
5	**Deduplication**	MinHash near-duplicate detection	Repeated text → memorization
6	**PII removal**	Addresses, SSNs, emails, phone numbers	Privacy

The punchline: FineWeb is about **44 TB** — a few consumer hard drives — ≈ **15 trillion tokens**. The "knowledge of the internet" inside a frontier model passes through something you could carry.

What's Actually In There?

Browse FineWeb yourself on Hugging Face: random samples look like...

```
"Tornadoes in 2012 – the 2012 tornado season was below average, with ..."
"Preheat the oven to 350°F. In a large bowl, whisk together flour, ..."
"Hyperglycemia is a condition in which an excessive amount of glucose ..."
"def fetch_user(id): return db.query(User).filter(User.id == id).first()"
```

Articles, recipes, medical facts, history, code, science papers — the breadth of human knowledge *as text*.

> **Garbage in, garbage out — at trillion-token scale.** Data quality and diversity has more impact on the final model than almost anything else. Every filter above is a *decision* whose consequences show up later as model behavior.

Tokenization

What the model actually reads

Tokenization: Formal Definition

A **tokenizer** is an *invertible* mapping between text and sequences of integers (**tokens**) drawn from a fixed, finite **vocabulary**.

Modern LLMs learn the vocabulary with **byte-pair encoding (BPE)**: start from the 256 possible bytes, repeatedly find the most frequent adjacent pair of symbols in the corpus, and **mint** a new symbol for that pair — until the vocabulary reaches a target size (GPT-4: **100,277** symbols).

In plain English: the model never sees letters or words. It sees chunk IDs. `"hello world"` is not eleven characters to GPT-4 — it is **two opaque integers**.

Why Tokens? Trading Vocabulary for Sequence Length

Neural nets want a **1-D sequence of symbols from a finite set** — and **sequence length is the scarce resource** (compute and context windows scale with it).

Representation	# symbols	Length of "hello world"
Raw bits	2	88
UTF-8 bytes	256	11 (8× shorter)
BPE (GPT-4, `cl100k_base`)	100,277	**2**

BPE keeps trading a *bigger vocabulary* for *shorter sequences*, settling around ~100k symbols at **roughly 4 characters per token** of English.

> **Why not whole words?** Infinite variants — `run`, `running`, `runner`, typos, new words, other languages. Sub-word tokens share roots and can spell *anything* (fall back to bytes).

BPE by Hand: A Worked Example

Corpus: `aaabdaaabac` — alphabet {a, b, c, d}, 11 symbols.

Step	Most frequent pair	Mint	Sequence	Length
0	—	—	`a a a b d a a a b a c`	11
1	`a a` (most frequent)	**Z** = aa	`Z a b d Z a b a c`	9
2	`a b` (×2)	**Y** = ab	`Z Y d Z Y a c`	7
3	`Z Y` (×2)	**X** = ZY = aaab	`X d X a c`	5

Vocabulary grew **4 → 7**; sequence shrank **11 → 5**. Stop when you reach the target vocab size.

Encoding new text = replay the merges *in the order they were learned*.
Decoding = look up each ID's bytes and concatenate — always exact.

BPE in TerpLM: Training (`bpe.py`)

```python
def train(text: str, n_merges: int) -> list[list[int]]:
    chunk_freqs = Counter(_CHUNK.findall(text))              # split on whitespace runs
    seqs = {chunk: list(chunk.encode("utf-8")) for chunk in chunk_freqs}   # start from bytes

    merges: list[list[int]] = []  # [a, b, new_id]
    for new_id in range(256, 256 + n_merges):
        pairs: Counter = Counter()
        for chunk, freq in chunk_freqs.items():               # count adjacent pairs,
            seq = seqs[chunk]                                  # weighted by chunk frequency
            for pair in zip(seq, seq[1:]):
                pairs[pair] += freq
        if not pairs:
            break
        (a, b), _ = pairs.most_common(1)[0]                   # most frequent pair
        merges.append([a, b, new_id])                         # "mint" a new symbol
        for chunk in chunk_freqs:
            seqs[chunk] = _apply_merge(seqs[chunk], a, b, new_id)
    return merges
```

The same algorithm GPT-4 uses — at toy scale. Every token ID is a byte (0–255), a merge, or a special token.

BPE in TerpLM: Encode / Decode

```python
def _encode_chunk(self, chunk: str) -> list[int]:
    seq = list(chunk.encode("utf-8"))
    while len(seq) >= 2:
        # Merge the highest-priority (earliest-learned) pair present.
        pair = min(zip(seq, seq[1:]), key=lambda p: self._rank.get(p, 1 << 30))
        if pair not in self._rank:
            break
        seq = _apply_merge(seq, pair[0], pair[1], self._rank[pair])
    return seq

def decode(self, ids: list[int]) -> str:
    return b"".join(self._bytes[i] for i in ids).decode("utf-8", errors="replace")
```

Round-trip property: `decode(encode(t)) == t` for **any** string — checked in `tests/test_bpe.py` as a Hypothesis property test (callback to our Testing lecture).

Live Demo: `python -m terplm.demo --show-merges`

```
vocab size: 288 (256 bytes + 30 merges
                + 2 specials)
  #   id    merge
  0  256   'h'      + 'e'     -> 'he'
  1  257   't'      + 'he'    -> 'the'
  2  258   't'      + 'e'     -> 'te'
  3  259   'r'      + 'a'     -> 'ra'
  ...
 13  269   'te'     + 'r'     -> 'ter'
 17  273   'ter'    + 'ra'    -> 'terra'
 18  274   'terra'  + 'p'     -> 'terrap'
 19  275   'terrap' + 'in'    -> 'terrapin'
  ...
 26  282   'M'      + 'a'     -> 'Ma'
 27  283   'Ma'     + 'ry'    -> 'Mary'
 29  285   'Maryl'  + 'and'   -> 'Maryland'
```

Trained on a 20-line corpus about UMD, Testudo and terrapins.

- The algorithm **discovers** `he`, then `the` — exactly what English statistics predict
- `terrapin` is frequent in *this* corpus → it earns a **single token** (built up piece by piece)
- IDs are literally the **order** merges were learned

```
encode('terrapin') -> [275]   (1 token)
corpus: 1138 bytes -> 774 tokens
Round-trip on unseen text:
  OK 'hello world'
  OK 'Hello  World!'
  OK 'unseen München 🐢'
```

Encoding a Sentence (60 merges)

```
encode('the terrapin tested the program')
  -> [257, 32, 275, 32, 299, 258, 100, 32, 257, 32, 287, 109]
  -> [the][ ][terrapin][ ][tes][te][d][ ][the][ ][progra][m]
  31 bytes -> 12 tokens
```



- Frequent words → one token. Less frequent words (`tested`, `program`) → several fragments.
- Token boundaries don't respect morphology: `tes` + `te` + `d`, not `test` + `ed`.
- Whitespace is a token too.

Case Sensitivity Falls Straight Out of BPE

```
'Maryland'   -> [Maryland]              1 token  (id 285)
'maryland'   -> [m][a][ry][l][and]      5 tokens
' Maryland'  -> [ ][Maryland]           2 tokens
'Testudo'    -> [Testudo]               1 token
'TESTUDO'    -> [T][E][S][T][U][D][O]   7 tokens
'strawberry'-> [s][t][ra][w][b][e][r][ry]
```

> The capitalized form was frequent in training and earned a single token; the lowercase form **shatters into five**. To the model these are **completely unrelated symbols** — ID 285 has no visible relationship to IDs 109, 97, 271, 108, 266.

Hold that thought — it's the root cause of the spelling failures in Part 2.

Common Misconception: "Tokens Are Words"

They are not:

- common words → **one** token; rare words → **several**
- **whitespace and capitalization** change everything
- the **same string** tokenizes differently with/without a leading space
- different model families train **different tokenizers** — the same text gives different counts

"What are the symptoms of hyperglycemia?"

`What` `␣are` `␣the` `␣symptoms` `␣of` `␣hyper` `gly` `c` `emia` `?`

Illustrative split — a technical word shatters into sub-word pieces

Try it: `tiktokenizer.vercel.app` shows GPT-4's real tokenization. Try `Maryland` vs `maryland`, `hello world` vs `hello world`, and a line of Python.

Why Token Counts Matter to *You*

- **Context windows** are measured in tokens — and the budget includes your prompt **and** the response
- **API cost** is billed per token (input and output priced separately)
- **"~4 characters per token"** is a heuristic for English prose — it breaks down for **code, jargon, non-English text**, all of which tokenize less efficiently
- **Always count with the tokenizer of the model you're actually using**

> **After tokenization**, "the internet" is one enormous **1-D sequence of integers**. Everything downstream — training, inference, conversations, even images and audio in multimodal models — is sequences of tokens.

```
The terrapin is the mascot of the University of Maryla
nd.
Testudo the terrapin stands outside McKeldin Library.
Students rub the nose of Testudo for good luck before
exams.
The terrapin is a turtle that lives in the Chesapeake
Bay.
The University of Maryland is in College Park.
Students at the University of Maryland study computer
science.
Computer science students write programs and test prog
rams.
A program that is not tested is a program that is not
trusted.
The terrapin is the mascot and the turtle is the terra
pin.
Before exams the students rub the nose of Testudo the
terrapin.
The Chesapeake Bay is the home of the terrapin.
College Park is the home of the University of Marylan
d.
Students write tests and tests catch bugs in programs.
Programs without tests hide bugs and bugs break progra
ms.
The library is open and the students study in the libr
ary.
Testudo stands outside the library in College Park.
The students of Maryland rub the nose of the turtle.
Good luck comes to students who rub the nose of Testud
o.
The turtle in the Chesapeake Bay is a diamondback terr
apin.
The diamondback terrapin is the mascot of Maryland.
```

```
976, 6057, 137576, 382, 290, 178544, 328, 290, 4923, 3
28, 36737, 13, 793, 2414, 6812, 290, 6057, 137576, 208
62, 7539, 7935, 42, 3171, 258, 16583, 558, 36954, 1234
6, 290, 28074, 328, 4674, 6812, 395, 1899, 12751, 225
4, 43006, 558, 976, 6057, 137576, 382, 261, 73787, 48
4, 8354, 306, 290, 170843, 12352, 558, 976, 4923, 328,
36737, 382, 306, 9053, 6412, 558, 36954, 540, 290, 492
3, 328, 36737, 5012, 7595, 11222, 558, 50411, 11222, 4
501, 5067, 8094, 326, 1746, 8094, 558, 32, 2193, 484,
382, 625, 18591, 382, 261, 2193, 484, 382, 625, 25498,
558, 976, 6057, 137576, 382, 290, 178544, 326, 290, 73
787, 382, 290, 6057, 137576, 558, 13036, 43006, 290, 4
501, 12346, 290, 28074, 328, 4674, 6812, 290, 6057, 13
7576, 558, 976, 170843, 12352, 382, 290, 2237, 328, 29
0, 6057, 137576, 558, 62173, 6412, 382, 290, 2237, 32
8, 290, 4923, 328, 36737, 558, 36954, 5067, 10742, 32
6, 10742, 3494, 39184, 306, 8094, 558, 120650, 2935, 1
0742, 16930, 39184, 326, 39184, 2338, 8094, 558, 976,
11282, 382, 2494, 326, 290, 4501, 5012, 306, 290, 1128
2, 558, 2414, 6812, 20862, 7539, 290, 11282, 306, 905
3, 6412, 558, 976, 4501, 328, 36737, 12346, 290, 2807
4, 328, 290, 73787, 558, 17212, 12751, 5124, 316, 450
1, 1218, 12346, 290, 28074, 328, 4674, 6812, 558, 976,
73787, 306, 290, 170843, 12352, 382, 261, 36181, 2205,
6057, 137576, 558, 976, 36181, 2205, 6057, 137576, 38
2, 290, 178544, 328, 36737, 13
```

Quick Check

Tokenization

1. Why does an LLM see `Maryland` and `maryland` as unrelated symbols?
2. BPE on `abababcab` — what is the first merge? What does the sequence become?
3. Your app sends a 2,000-character Python file to an API. Is "~500 tokens" a safe estimate? Why or why not?

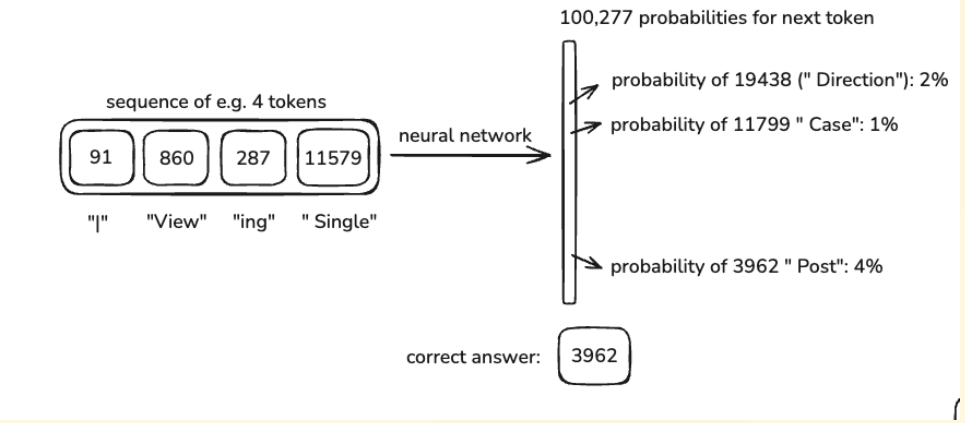
The Neural Network

Input, output, and training

What a Language Model Computes

A **language model** is a function, parameterized by a vector of numbers (the **parameters** / **weights**), that maps a sequence of tokens (the **context**, up to a maximum **context length** — 1,024 for GPT-2, 128,000+ for frontier models) to a **probability distribution over the vocabulary** for the **next** token.

Training adjusts the parameters so that, across the corpus, the actual next token receives high probability. The **loss** (cross-entropy) measures how far predictions are from the data — lower is better.



A window of four tokens goes in; out comes a probability for each of 100,277 possible next tokens; training nudges the probability of the actual next token (3962) up. Diagram: Andrej Karpathy.

In Plain English

The model is a **guess-the-next-token machine**.

Training is millions of rounds of:

> "Here are the previous tokens — what comes next?"

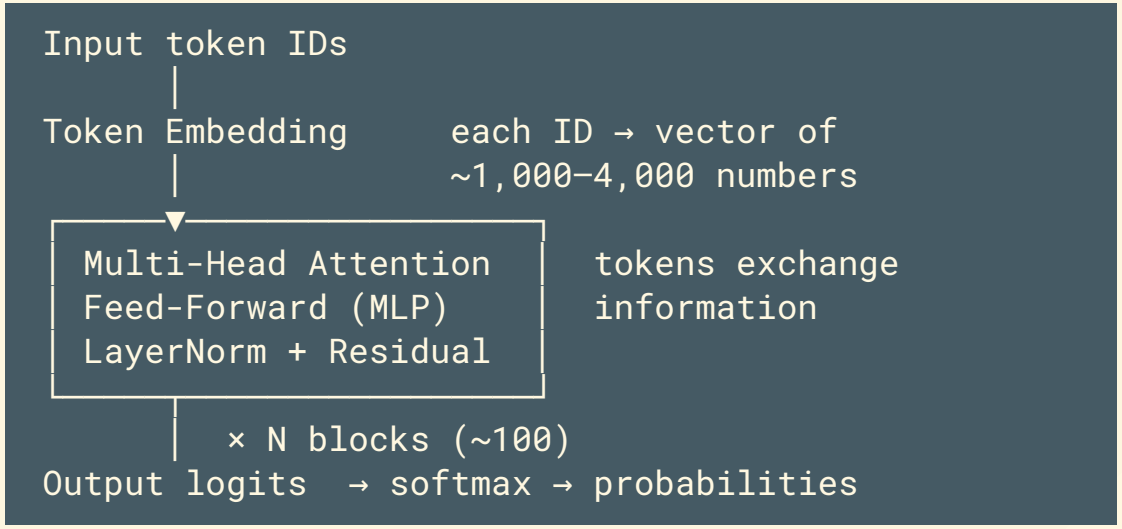
...with a tiny **nudge to the knobs** after every round.

Intuition: a giant DJ mixing board.

- GPT-2: **1.6 billion** knobs; modern models: hundreds of billions
- Random settings → noise
- Training twiddles every knob a tiny step at a time until the machine's output statistics **match the corpus statistics**

Inside, the "machine" is one big **differentiable mathematical expression** — matrix multiplications + simple nonlinearities in the **transformer** architecture. What matters: *fixed expression, billions of knobs, tokens in, probabilities out.*

Inside the Box (Just Enough)



Embedding: each token ID maps to a learned vector — a coordinate in "meaning space." Similar tokens end up nearby.

Attention: each position mixes in information from earlier tokens. By deeper layers, `river bank` and `bank account` carry different representations.

Explore it: `bbycroft.net/llm` — a walkable 3D visualization of a real (small) transformer.

Attention, Conceptually

When predicting the next token, the model doesn't weigh every prior token equally — it learns to **attend** more strongly to the tokens most relevant to what comes next.

```
"The patient with diabetes was prescribed ___"
```

`The` `patient` `with` `diabetes` `was` `prescribed` → **insulin**

Attention weights concentrate on `diabetes` and `prescribed` far more than on `The` or `was`.

TerpLM has no attention at all — it sees *one* previous token. Keep that in mind when you see its output.

Common Misconception: "It Looks Things Up"

> "The model stores its training text in a database and looks things up."

- There is **no database** and **no lookup**
- Whatever the model "knows" is **smeared across the parameters** as statistical tendencies
- So its knowledge is a **vague recollection**, not a record
- ...which is exactly why it can be **wrong with perfect confidence**

We'll turn this into a precise, practical rule in Part 2: *parameters = vague recollection, context window = working memory.*

TerpLM's Model: One Matrix

TerpLM's parameters are a single matrix `W` of shape `(vocab, vocab)`. **Row `t` holds the logits** (pre-softmax scores) for *"what follows token `t`"*.

```python
class BigramLM:
    def __init__(self, vocab_size: int, seed: int = 0):
        rng = np.random.default_rng(seed)
        # Random initialization: the untrained model predicts noise.
        self.W = rng.normal(0.0, 0.01, (vocab_size, vocab_size))

    def next_probs(self, token: int, temperature: float = 1.0) -> np.ndarray:
        """The model's entire output: a probability for every token in
        the vocabulary as the candidate next token."""
        return _softmax(self.W[token] / temperature)

def _softmax(logits):
    z = np.exp(logits - logits.max(axis=-1, keepdims=True))
    return z / z.sum(axis=-1, keepdims=True)
```

514-token vocab → $514^2 \approx$ **264k parameters**. GPT-2: 1.6 **billion**. Same idea: numbers in, a distribution out.

The Loss: Cross-Entropy

For one example with true next token y:

$$\text{loss} = -\ln p_\theta(y \mid \text{context})$$

Average over a batch. In **nats**.

- Model puts prob **1.0** on the right token → loss **0**
- Prob **0.5** → loss **0.69**
- Prob **0.01** → loss **4.6**
- Uniform over V tokens → loss $\ln V$

Sanity check you should always do:

At random init, the model is ≈ uniform, so

$$\ln(514) = 6.242$$

For GPT-4's vocabulary: $\ln(100{,}277) \approx 11.5$

> Loss 4.1 means the true next token gets, on average (geometric mean), probability $e^{-4.1} \approx 1.7\%$ — about **8.5× better** than guessing among 514.

Training: The Whole of Deep Learning in Five Lines

```
def train_step(self, prev: np.ndarray, nxt: np.ndarray, lr: float) -> float:
    """One gradient-descent update on a batch of (prev, next) pairs."""
    probs = _softmax(self.W[prev])                          # (batch, vocab): predict
    batch = len(prev)
    loss = float(-np.log(probs[np.arange(batch), nxt] + 1e-12).mean())   # score
    # Gradient of cross-entropy through softmax: probs - one_hot.
    probs[np.arange(batch), nxt] -= 1.0                     # how wrong, per logit
    np.add.at(self.W, prev, -lr * probs / batch)            # nudge the knobs
    return loss
```

And the loop in `train.py` — sample random windows from the token stream, update, repeat:

```
for step in range(args.steps + 1):
    idx = rng.integers(0, len(ids) - 1, size=args.batch)    # random positions
    loss = model.train_step(ids[idx], ids[idx + 1], args.lr) # (token, next token) pairs
```

- In plain English: Take a batch of examples, predict the next token, measure how wrong the predictions are, calculate the gradient, and slightly adjust the model so that the correct next tokens become more likely.
- Real LLMs: same loop, but the gradient comes from backprop through a transformer, on thousands of GPUs.

Live Demo: Pre-train TerpLM on Shakespeare

```
$ python -m terplm.train data/shakespeare.txt --merges 256 --steps 3000
training tokenizer: 256 merges ...
corpus: 1,115,394 chars -> 691,315 tokens (vocab 514, 1.61 bytes/token)
loss of uniform guessing: ln(514) = 6.242

step     0  loss 6.242  sample: 'Fet?\x15\x08Butmorekesuhaciath\x01'
step   500  loss 5.202  sample: 'Fcesheght??ight?what?love'
step  1000  loss 4.803  sample: 'Fonce?hand vomitilgo~'
step  1500  loss 4.541  sample: 'F\x1e, be dehovustheeis '
step  2000  loss 4.343  sample: 'FndIN\x0eanhis then?you a?'
step  3000  loss 4.113  sample: "F?c?Yliindup?'t"

saved model + tokenizer to out/
```

Narrate it like a researcher: *"Where does the loss start? Why exactly there? What changes in the samples?"*

Reading the Training Log

Step	Loss	What's happening
0	**6.242** = ln(514)	Random parameters ≈ uniform guessing. Samples are byte noise.
500	5.202	Learning which tokens are common; spacing and `?` appear
1000–2000	4.8 → 4.3	Real word fragments (`hand`, `then`, `you`), plausible punctuation
3000	4.113	Still gibberish **globally** — only **local** texture

> **The punchline of the bigram model:** with **one token of context**, local texture is all you can learn. A transformer running *the same loop* with thousands of tokens of context — and a few trillion more parameters — turns this gibberish into fluent paragraphs.

Real runs show the same shape: e.g. loss ~11 → ~2.4 for a GPT-scale vocab; step 1 noise, step 500 local coherence, step 32K fluent English.

Smaller Run You Can Do in 5 Seconds

On the bundled 20-line `tiny_corpus.txt` (60 merges, vocab 318):

```
$ python -m terplm.train data/tiny_corpus.txt --merges 60 --steps 1000 --batch 512 --lr 1.0
corpus: 1,138 chars -> 638 tokens (vocab 318, 1.78 bytes/token)
loss of uniform guessing: ln(318) = 5.762

step     0  loss 5.762  sample: 'The\x00U\r\x05teive\x00\x00omra\x00'
step   250  loss 4.001  sample: 'The)\x00\x18H\x07nd1\x00in\x1b\\Q'
step   500  loss 3.122  sample: 'The prograx\x00in terrapingS\x00Maryland'
step   750  loss 2.674  sample: 'The c\x00is the libra\x00\x1f}'
step  1000  loss 2.425  sample: 'The is l\x00<Y\x00terraphed'
```

Lower final loss than Shakespeare — why? A tiny, repetitive corpus is **easy to memorize**. (And note: losses over different vocabularies aren't directly comparable — the baseline ln V differs.)

Scale: What Changed Since 2019?

GPT-2 (2019) was *exactly this recipe* with a transformer: **1.6B** parameters, **100B** training tokens, context 1,024.

	Cost to train GPT-2
OpenAI, 2019	~ **$40,000**
Karpathy's `llm.c` reproduction, 2024	~ **$672** (headed toward ~$100)

Modern: Llama 3 — 405B params, 15T tokens. Months of compute on thousands of GPUs.

> Much of the recent revolution is **better data, better software, and above all better hardware** — GPU data centers — rather than new ideas. **Scaling laws:** loss is a smooth, predictable function of parameters N and data D. Bigger + more data ≈ reliably better → the GPU "gold rush."

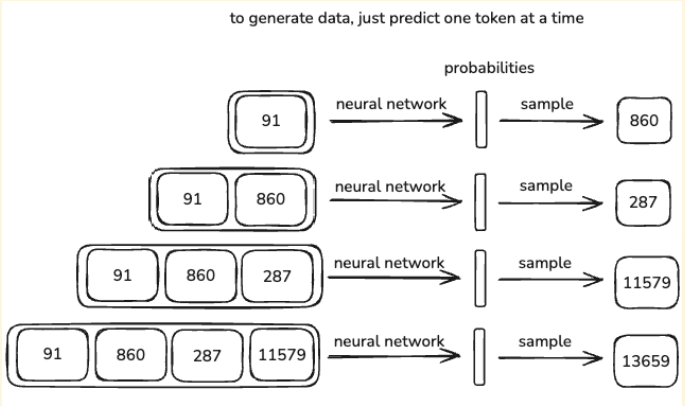
Inference

Sampling, stochasticity, temperature

Inference: Formal Definition

Inference generates text from a trained model: feed in a token prefix → obtain the distribution for the next token → **sample** one token → append → repeat.

Temperature T rescales the logits before the softmax: $p_i \propto \exp(z_i/T)$. Low T sharpens toward the most likely token; high T flattens.



Each sampled token is appended and fed back in. Last row: the model samples 13659 where the training document had 3962 — generation is a **remix** of training statistics, not a recital. Diagram: Andrej Karpathy.

In Plain English: Nobody "Decides"

The model does not "decide" on a reply.

It **flips a biased coin per token**, forever — a very informed coin, with ~100K sides.

> When you use ChatGPT, **no learning happens**. The parameters were frozen months ago. You are *just sampling*.

```
def generate(self, start, n_tokens, temperature=1.0, seed=0):
    rng = np.random.default_rng(seed)
    ids = [start]
    for _ in range(n_tokens):
        p = self.next_probs(ids[-1], temperature)            # distribution over vocab
        ids.append(int(rng.choice(self.vocab_size, p=p)))    # flip the biased coin
    return ids
```

A transformer's loop is identical — except `next_probs` looks at the whole `ids` list, not just `ids[-1]`.

Live Demo: Same Prompt, Two Seeds

```
$ python -m terplm.generate --prompt "ROMEO:" --tokens 40 --seed 1
ROMEO:?E:/I'll??thee?veruto thy to or say andlovespeconhe festis fornoseilme O

$ python -m terplm.generate --prompt "ROMEO:" --tokens 40 --seed 2
ROMEO:a will my h?ly,
Uall ascilo??orWhatmor?ARD?ver bjOLloile, you sh?
```

> Different every time: these are **stochastic systems**, remixing the statistics of their training data rather than reciting it. Same model, same prompt — different coin flips.

This demo breaks the "it retrieves answers" mental model faster than any explanation.

Temperature, With Real Numbers

TerpLM trained on `tiny_corpus.txt`: top-5 candidates for the token after a space, at different temperatures.

T	`the`	`is`	`of`	`terrapin`	`Maryland`
2.0	0.032	0.023	0.022	0.019	0.015
1.0	0.140	0.072	0.067	0.050	0.030
0.5	0.463	0.122	0.107	0.058	0.022
0.1	**0.998**	0.001	0.001	0.000	0.000

Same logits every row — only the division by T changes. At $T = 2$ the mass spreads over hundreds of junk tokens; at $T = 0.1$ it's effectively **argmax**.

Live Demo: Temperature Taken to the Extreme

```
$ python -m terplm.generate --prompt "ROMEO:" --temperature 0.4
ROMEO:
Th?I'll the the I the of the the s s are he s so bar to I the

$ python -m terplm.generate --prompt "ROMEO:" --temperature 0.05
ROMEO:
Whow the the the the the the the the the the the the the
```

At $T = 0.05$ the model almost always takes the single most probable next token — and in English bigram statistics, the most probable follower of almost anything is `the`. It falls into the `the the the` **attractor**.

File away "the the the" — it makes a sinister reappearance in the RLHF section.

Temperature in Practice

Temperature	Behavior	Good for
0.0	~Deterministic: always the top token	Code, extraction, classification
~0.7	Balanced: variety, still coherent	Chat, general writing
1.5+	High randomness, goes off the rails	Rarely useful

> **Temperature 0 isn't fully deterministic.** Parallel floating-point math isn't perfectly associative — tiny rounding differences on different hardware/batches can flip a near-tie. Never rely on exact reproducibility.

Other knobs you'll see in APIs: **top-k** (sample only among the k most likely) and **top-p** (smallest set with cumulative prob ≥ p). Exercise: add `--top-k` to `generate.py`.

The Base Model

A token simulator

What Pre-training Produces

> A **base model** is the network plus its trained parameters, with no further conditioning: a sampler of token sequences whose statistics match its training corpus.

In plain English: an internet document simulator.

Prompt it with *"What is 2+2?"* and it will **not** answer you. It will **continue the document** — maybe with more exam questions, maybe a philosophy essay, differently each time.

A glorified, very expensive **autocomplete**.

Intuition: a lossy compression of the internet.

A "zip file" of statistical gist — hundreds of billions of numbers distilled from trillions of tokens.

Sampling from it is **"dreaming internet pages"**: statistically faithful, factually unanchored remixes.

An LLM Is Literally Two Files

Releasing a model = publishing two artifacts:

1. **Inference code** — a few hundred lines describing the forward pass. GPT-2's fits in *one Python file* (`openai/gpt-2/src/model.py`)
2. **Weights** — the billions of tuned numbers. **This is where all the value lives.**

Example: **Llama 2 70B** = a 140 GB parameters file (70B × 2 bytes in float16) + ~500 lines of C. Runs offline.

TerpLM is exactly this, too:

```
out/
├── tokenizer.json   # the merges
└── model.npy        # the weights (W)
```

- `model.py` / `generate.py` = inference code.

> **Open weights ≠ open source.** GPT-2, Llama 3.1 405B: you get the parameters, typically **not** the training data or the full recipe.

"Useless, Then?" — Three Base-Model Behaviors

All the knowledge lives here. Three behaviors explain most of what assistants later do:

1. Regurgitation

Prompt with the first sentence of a famous Wikipedia article:

```
Zebras (/ˈzɛbrə, ˈziːbrə/)
are African equines with
distinctive...
```

→ *...black-and-white striped coats. There are three living species...* **verbatim**.

High-quality sources are **oversampled** — like a student who reread it 10×.

2. Educated hallucination

Prompt about events **after the knowledge cutoff**:

```
The Republican Party
nominated Trump and ___
in the 2024 election...
```

→ *...Mike Pence, against Hillary Clinton...*
→ resample: *...Ron DeSantis, against Biden...*

A different **parallel universe** each time. Nothing is broken.

3. In-context learning

```
Hello:    Bonjour
Cat:      Chat
Dog:      Chien
Teacher:
```

→ `Professeur` ✓

The pattern was learned **inside the context window, at inference time** — not in training. **Few-shot prompting.**

Regurgitation: Why It Matters

- Verbatim recitation is usually **undesirable** — the model is supposed to generalize, not memorize
- It should temper how you think about the **privacy** (PII that survived filtering) and **copyright** of training data
- Deduplication (Stage 1) exists partly to reduce it

Hallucination: Why It Matters

- The model is **doing exactly what it was trained to do**: continue documents plausibly
- It has no mechanism for "I wasn't trained on this" — only for "what text usually comes next"
- Resampling giving *different* answers is a strong **hallucination signal** — a useful trick in practice (self-consistency checks)

Conjuring an Assistant From a Base Model

In-context learning is powerful enough to fake an assistant *with no fine-tuning at all* — format your prompt as a transcript and let the base model continue it:

```
Here is a conversation between an AI Assistant and a Human. The Assistant is
knowledgeable, helpful, and honest.

Human: Hi, what's the capital of France?
Assistant: The capital of France is Paris.

Human: What is 2 + 2?
Assistant: 2 + 2 = 4.

Human: Why is the sky blue?
Assistant:
```

The assistant exists in the prompt, not the weights. The base model continues the document — and the most plausible continuation is a helpful answer. (It may also keep going and write the next `Human:` turn itself!)

Stage 2 **bakes this trick into the parameters.**

Try It Yourself

- **Tokenizers:** `tiktokenizer.vercel.app` — the `Maryland` / `maryland` experiment; switch to a chat model to see `<|im_start|>` serialization
- **Inside a transformer:** `bbycroft.net/llm`
- **Base models:** Hyperbolic serves **Llama 3.1 405B base** — one of the few places to try a raw token simulator. Try the regurgitation, parallel-universe, and few-shot demos
- **Datasets:** browse FineWeb on Hugging Face
- **Run locally:** LM Studio or Ollama run small open-weights models on a laptop — no data leaves your machine

Quick Check

Part 1

1. The untrained TerpLM has loss **6.242** on a 514-token vocab. Why exactly that number? What does loss 4.1 mean?
2. What changes during **training** vs. **inference**? Which is happening when you use ChatGPT?
3. You prompt a base model with *"What is the capital of France?"* Give two plausible outputs.
4. Why does TerpLM at temperature 0.05 produce `the the the …`?

Part 1 Summary

- **Data:** Common Crawl → filtering → FineWeb: ~44 TB, ~15T tokens. Every filter is a design decision.
- **Tokenization:** BPE mints tokens for frequent byte pairs. The model sees **IDs, not letters** — case, spaces, and rare words change everything.
- **The network:** a fixed differentiable function; tokens in, a **probability distribution** over the next token out. Parameters = billions of knobs.
- **Training:** nudge the knobs to lower cross-entropy loss. Starts at **ln V**. Knowledge ends up **smeared** in the weights — no database.
- **Inference: sample** a token, append, repeat. Stochastic. **Temperature** sharpens/flattens. No learning happens.
- **Base model:** an internet document simulator — regurgitates, hallucinates plausibly, learns in-context.

> **Next time:** how a document simulator becomes an assistant — and why that explains hallucinations, the "strawberry" problem, and thinking models.

Part 2

Post-training, LLM psychology, and reinforcement learning

Recap: Where We Left Off

After pre-training we have a **base model**:

- a lossy compression of the internet
- a sampler that **continues documents**
- knowledge = **vague recollection** smeared in parameters
- can be coaxed into an assistant via a transcript-style prompt

```
PRE-TRAINING  ✔
  ─► base model (document simulator)

SUPERVISED FINE-TUNING   ◄ today
  ─► assistant

REINFORCEMENT LEARNING   ◄ today
  ─► "thinking" model
```

Today's question: what, *mechanically*, are you talking to when you talk to ChatGPT or Claude — and what failure modes follow from that?

Stage 2: Supervised Fine-Tuning

Programming by example

SFT: Definition

Supervised fine-tuning (SFT) continues training the base model with the **same algorithm** and the **same objective** — next-token prediction — on a **different dataset**: a curated corpus of multi-turn conversations between a "user" and an "assistant," written to exemplify ideal assistant behavior.

In plain English: you can't program a neural net's behavior in code, so you **program it by example**.

Want it to answer helpfully, decline harmful requests, admit uncertainty? Write a few hundred thousand conversations that do exactly that, and train on them. The model rapidly adopts the persona.

Intuition: pre-training takes **months** on thousands of GPUs; SFT can take **hours** — the conversation corpus is minuscule next to the internet.

The base model already contains the **knowledge**; SFT teaches it a **style of access**.

Conversations Become Tokens: Chat Templates

Each lab defines a **chat template** with new **special tokens** — reserved IDs that never appear in pre-training text. OpenAI's ChatML-style format:

```
<|im_start|>system<|im_sep|>You are a helpful assistant<|im_end|>
<|im_start|>user<|im_sep|>What is 4 + 4?<|im_end|>
<|im_start|>assistant<|im_sep|>4 + 4 = 8<|im_end|>
```

- `system` turn: standing instructions (identity, tone, rules) — **silently prepended** to every conversation
- At inference, the server appends `<|im_start|>assistant<|im_sep|>` after your message and asks the model to **continue the sequence**. The reply *is* the continuation.
- The model stops when it samples `<|im_end|>`

> The structured "conversation" you see is a **UI fiction** over **one one-dimensional token stream**. Terms to remember: **chat template**, **ChatML** — every fine-tuning framework and chat API speaks them.

Your API Call, Under the Hood

What you write:

```
client.messages.create(
  model="...",
  system="You are a medical Q&A assistant.",
  messages=[
    {"role": "user",
     "content": "What is hyperglycemia?"},
  ],
)
```

What the model sees (schematically):

```
<|im_start|>system<|im_sep|>
You are a medical Q&A assistant.
<|im_end|>
<|im_start|>user<|im_sep|>
What is hyperglycemia?
<|im_end|>
<|im_start|>assistant<|im_sep|>
```

…and it keeps sampling until `<|im_end|>`.

The `messages` list is **stateless**: every turn, the client re-sends the *whole* conversation. The "memory" of a chat is just the token stream you resend. (Exact special tokens differ per vendor; the idea is universal.)

TerpLM's Tokenizer Already Has Special Tokens

```
# bpe.py — chat-protocol special tokens, exactly like production models
SPECIALS = ["<|im_start|>", "<|im_end|>"]

def encode(self, text: str) -> list[int]:
    # Specials are matched as whole units, never split into bytes.
    pattern = "(" + "|".join(re.escape(s) for s in SPECIALS) + ")"
    ...
```

```
encode('<|im_start|>user<|im_end|>')
  -> [286, 117, 115, 101, 114, 287]
  pretty: [<|im_start|>][u][s][e][r][<|im_end|>]
```

`<|im_start|>` is **one ID** (286), not 12 characters. A user typing the literal string into a chat box should *not* produce that ID — otherwise they could forge an `assistant` turn. (That's a real class of prompt-injection concern.)

Where Do the Conversations Come From? People.

- **InstructGPT** (OpenAI, 2022): hired human labelers, gave them **labeling instructions** — company-written documents, often **hundreds of pages**, boiling down to **"be helpful, truthful, and harmless"**. Labelers write prompts *and* ideal responses.
- **Today:** most volume is **synthetic** (generated by existing LLMs), then human-edited — but the seed of the behavior is still **human curation under corporate instructions**.

Dataset	Kind	Browse at
OASST1 (OpenAssistant)	Human-written by volunteers	`huggingface.co/datasets/OpenAssistant/oasst1`
UltraChat	Largely synthetic, millions of conversations	`github.com/thunlp/UltraChat`

Example training conversation:

```
Human:     What is 2 + 2?
Assistant: 2 + 2 = 4. Is there anything else you'd like help with?
Human:     What if it was multiplication instead?
Assistant: 2 × 2 = 4 as well — the same result! ...
```

The Most Useful Reframe in This Lecture

> **Misconception:** "I am talking to a magical AI."

> **What you're talking to:** a **statistical simulation of a data labeler following their employer's labeling instructions.**

Ask ChatGPT for the top five landmarks in Paris → you get an **instant imitation** of the answer a skilled, well-instructed contractor would have written after twenty minutes of research —

- either because a very similar conversation is **in the SFT set**, or
- as an **emergent blend** of the labeler persona with pre-training knowledge.

When it answers a coding question, it's imitating what a skilled developer-labeler would write.

Knowledge of Self

Ask a model **"Who built you?"** Unless its makers:

- hardcoded identity conversations into the SFT data (some labs do — **a couple hundred examples**), or
- put the identity in the **hidden system message**,

...it will **guess**. Often: *"I am ChatGPT by OpenAI"* — simply because the internet is full of such answers.

> It has **no persistent self**. It boots up, processes tokens, shuts down. No memory of your previous chats (unless an app pastes them back into the context).

TerpLM's `chat_corpus.txt` does exactly the "hardcode it" approach:

```
<|im_start|>user
Who built you?<|im_end|>
<|im_start|>assistant
I was built by students at the
University of Maryland.<|im_end|>
```

Live Demo: Post-training TerpLM

"SFT is the same algorithm with a different dataset" sounds too simple. TerpLM demonstrates it **literally** — `--resume` loads the Shakespeare model and **continues training** on conversations:

```
# train.py
if args.resume:
    tok = Tokenizer.load(args.out / "tokenizer.json")
    model = BigramLM.load(args.out / "model.npy")
    print(f"resuming: swapping dataset to {args.corpus} (same algorithm)")
```

```
$ python -m terplm.train data/chat_corpus.txt --resume --steps 2000
resuming: swapping dataset to data/chat_corpus.txt (same algorithm)
corpus: 1,754 chars -> 850 tokens (vocab 514, 2.06 bytes/token)

step     0  loss 4.982  sample: '<|im_start|>et I\x15qumoreutbut4ciBut\x01'
step   500  loss 1.445  sample: '<|im_start|>user\nWhat con?what?love'
step  1000  loss 1.140  sample: '<|im_start|>user\nI builtant\nI'
step  2000  loss 1.064  sample: '<|im_start|>user\nWhat land.<|im_end|>\nI '
```

What Did 2,000 Updates on 850 Tokens Buy?

```
$ python -m terplm.generate --prompt "<|im_start|>user" --tokens 45
<|im_start|>user
What you the buame?<|im_end|>
<|im_start|>user
<|im_start|>assity modeland.<|im_end|>
I from you your builtant
```

Learned: the protocol (form) ✔

- turns begin with `<|im_start|>`, end with `<|im_end|>`
- a role follows the start marker
- questions start with `What`, trail `?`
- words from the corpus: `built`, `model`, `land`

Not learned: the content ✘

Word salad — of course: the model conditions on **one previous token** and **can't remember the question** while producing the answer.

> The cleanest demo of **what SFT does** (impose conversational *form* on a pre-trained statistical engine) and **why context length matters** (form is bigram-learnable; substance is not).

Quick Check

Supervised fine-tuning

1. What is *different* between pre-training and SFT? What is the *same*?
2. Why can SFT take hours when pre-training takes months?
3. A model tells you "I'm GPT-4, made by OpenAI" — but it's actually an open-weights model from another lab. Explain how that happens.

LLM Psychology

Predictable failure modes — derivable, not memorizable

Hallucinations: Where They Come From

> A **hallucination** is fluent, confident output that is factually false — the model fabricating information *in the style of knowing it*.

Walk it back to the SFT data. Training conversations of the form *"Who is X?"* are answered **confidently and correctly** — because the labeler looked X up.

```
Human:    Who is Tom Cruise?          Assistant: Tom Cruise is an American actor ...
Human:    Who is John Barrasso?       Assistant: John Barrasso is a U.S. Senator ...
Human:    Who is Genghis Khan?        Assistant: Genghis Khan was the founder of ...
──────────────────────────────────────────────────────────────────────────────────
Human:    Who is Orson Kovacs?        Assistant: Orson Kovacs is an American author ...  ✘
```

The model learns the **form**: such questions get confident answers. Ask about someone who doesn't exist → the statistically appropriate continuation is a confident biography, **sampled from plausible nonsense**. Older models did this every time — a different fabrication per resample.

Mitigation 1: Teach "I Don't Know" — Only Where True

Meta's Llama 3 recipe — probe the knowledge boundary empirically:

1. Take a document; use an LLM to **generate factual questions** + answers from it
2. **Interrogate** the model several times per question
3. Use an **LLM judge** to compare its answers to the known correct answer
4. Where the model **reliably fails** → add a training conversation whose correct answer is *"I'm sorry, I don't know."*

```
Q: Which team did Dominik Hašek play for?   Correct: Buffalo Sabres
   Model, 3 tries: "Buffalo Sabres" ✔  "Buffalo Sabres" ✔  "Buffalo Sabres" ✔   → knows it
Q: What year did <obscure fact>?            Correct: 1874
   Model, 3 tries: "1901" ✘  "1862" ✘  "1874" ✔                                 → add "I don't know"
```

> A **modest** number of such examples **generalizes**. The network plausibly already has internal activations correlated with uncertainty — the new data just **wires that signal to words**.

Mitigation 2: Tools — Let the Model Look It Up

Introduce special tokens like `<SEARCH_START>` / `<SEARCH_END>`. When the sampler sees the model emit a query between them, it **pauses generation**, runs a real search, **pastes results into the context window**, and resumes.

```
<|im_start|>user<|im_sep|>Who is Orson Kovacs?<|im_end|>
<|im_start|>assistant<|im_sep|><SEARCH_START>Who is Orson Kovacs?<SEARCH_END>

      [ ...sampler pauses; search results pasted into the context window... ]

Orson Kovacs is ... (answer grounded in the pasted text) <|im_end|>
```

- A few **thousand** SFT examples teach the model *when* and *how* to reach for the tool
- The **citations** you see in modern chat interfaces are exactly this mechanism
- Same pattern for a **code interpreter**, a calculator, a database — and the "tool use" APIs we'll use in Lectures 15–19

The Two Kinds of Memory

```
knowledge in the PARAMETERS      =  vague recollection
  (read it months ago, statistically compressed)

knowledge in the CONTEXT WINDOW  =  working memory
  (directly attended to on every forward pass)
```

"Summarize chapter 1 of *Pride and Prejudice*"

→ works **passably** from recollection — the internet has discussed that chapter to death.

"Summarize chapter 1, **which I am pasting below**: …"

→ works **far better** — for the same reason you'd summarize better having *just reread* it.

> **The most practically useful fact in the lecture.** When you build LLM applications, the game is **getting the right information into the context window** instead of hoping it's in the parameters.

This Is the Whole Idea of RAG

Retrieval-Augmented Generation (Lectures 11–14): embed your documents, retrieve the most relevant chunks at query time, paste them into the context.

```
1. User query          "What is the administrative capital of the Ares Base colony?"
2. Embed query         → [0.23, -0.87, 0.41, ...]
3. Vector search       top-k nearest document chunks
4. Retrieved           [Doc 1] The Mars colony Ares Base was established in 2031 ...
                       [Doc 2] The administrative capital of Ares Base is New Houston ...
5. Context window      [Retrieved] ... [Retrieved] ... [Query] ...
6. LLM answer          "The capital of Ares Base is New Houston, housing 312 colonists."
```

Without RAG: "I don't have reliable information about a colony called Ares Base..." (or a confident fabrication)

With RAG: grounded in the retrieved text — working memory, not recollection

The Reversal Curse: Knowledge Is Directional

- *"Who is Tom Cruise's mother?"* → **Mary Lee Pfeiffer** ✔
- *"Who is Mary Lee Pfeiffer's son?"* → often *"I don't know"* ✘

> Knowledge in the parameters is stored in the **form it was encountered** — the internet says "Tom Cruise's mother, Mary Lee Pfeiffer" far more often than the reverse. It is **not a queryable database**; you have to approach it from the angle it was learned.

Another reason to prefer **context** over **recollection** for anything that matters.

Models Need Tokens to Think

Recall the architecture: producing **each** token costs **one forward pass** through a network with a **fixed number of layers**.

```
token 1 → [layer 1][layer 2] ... [layer ~100] → token 2
token 2 → [layer 1][layer 2] ... [layer ~100] → token 3
                    same fixed budget, every token
```

> Per token, the model performs a **fixed, finite amount of computation** — roughly a hundred layers' worth — **no matter how hard the question**.

There is no "think harder" button *within* a token. The only way to spend more compute on a problem is to **spend more tokens**.

Two Training Labels — Which Is Better?

"Emily buys 3 apples and 2 oranges. Each orange costs $2. The total is $13. What does an apple cost?"

A:
"The answer is **$3**. We can see this because 13 − 4 = 9 and 9 / 3 = 3."

B:
"The oranges cost 2 × 2 = $4. So the apples cost 13 − 4 = $9. There are 3 apples, so each apple costs 9 / 3 = $3. The answer is **$3**."

Both are correct. **A is a terrible training label.**

- Reading left to right, A asks the model to produce `3` **immediately** — cramming the entire solution into the fixed compute of **one token**. Everything after is **post-hoc justification**.
- B spreads the work across many tokens, each an **easy step**, with every intermediate result landing in the **context window** (working memory!) where later tokens can use it.

Train on A-shaped answers → a model that guesses. B-shaped → a model that computes.

Consequences You Can Verify Today

- Chat models "think out loud" — intermediate steps are **computation for the model**, not a presentation choice for you
- Prompting *"answer in one word"* on a hard question **measurably degrades accuracy**
- **Mental arithmetic** on large numbers is unreliable → ask it to **use code**

```
You:    What is 23,481 × 9,127? Answer with just the number.
Model:  214,311,  ...  (often wrong in the middle digits)

You:    What is 23,481 × 9,127? Use code.
Model:  >>> 23481 * 9127
        214311087
```

> The second answer delegates the hard part to an **interpreter**. The model's job shrinks to *copying digits into a string* — something tokens are good at.

Counting Fails

"How many dots are below?" `.....................................................`

Two problems, multiplied:

1. All the work in **one token** of output
2. The dots arrive **bundled into opaque multi-dot tokens** — the model never sees individual dots

```
# Fix: "Use code."  The model only has to copy-paste (echo tokens) - easy.
s = "....................................................."
print(s.count("."))    # 49 - counted by an interpreter, not neural mental math
```

Spelling Fails: "How many r's in strawberry?"

The model doesn't see `r`s. It sees one or two token IDs for the whole word. Remember our own tokenizer:

```
'maryland'   -> [m][a][ry][l][and]
'strawberry' -> [s][t][ra][w][b][e][r][ry]     (TerpLM, 60 merges)
'strawberry' -> [str][aw][berry]               (a GPT-style tokenizer – roughly)
```

Characters are **smeared unpredictably** across symbols — and then it must **also count**. Two known weaknesses, multiplied.

> **Fix: "use code."** `"strawberry".count("r")` → `3`. Push string-twiddling into tools.

Jagged Intelligence

Models that solve **olympiad** problems have claimed that **9.11 > 9.9**.

- One interpretability investigation found internal features associated with **Bible verses** activating — and in Bible verses, 9:11 *does* come after 9:9
- Another plausible confound: **software version numbers**, where v9.11 *is* newer than v9.9

The capability surface is Swiss cheese: spectacular and full of holes — and the holes are not always where reasoning about the pipeline predicts.

Standing rule of this course: use the model as a tool, **check its work**, own the result.

Quick Check

LLM psychology

Classify each failure as caused by **(a)** tokenization, **(b)** fixed compute per token, or **(c)** parameters-as-vague-recollection:

1. Miscounting the r's in "strawberry"
2. Inventing a biography for a fake name
3. Failing "answer in one word" arithmetic
4. Getting a library's API slightly wrong for a version released last month

Then: give one prompting practice that follows from "context window = working memory."

Stage 3: Reinforcement Learning

Practice problems

The School Analogy

Open any textbook. There are three kinds of content — and they map exactly onto the three training stages:

In a textbook	Training stage	What the model gets
Exposition / background prose	**Pre-training**	Knowledge
Worked examples with solutions	**Supervised fine-tuning**	Imitation of experts
Practice problems (answer key only)	**Reinforcement learning**	Skill discovered by trial and error

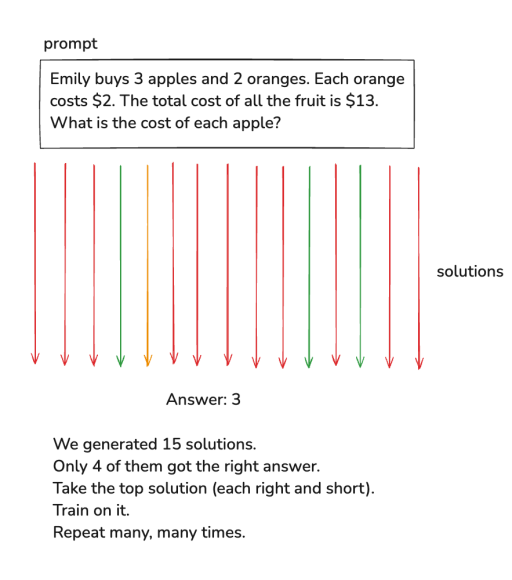
> We've done the first two. You don't become good at math by *only* reading worked examples — you have to **attempt problems** and check against the key.

RL for LLMs: Definition

> In **reinforcement learning (RL)** for LLMs, the model is given prompts with **verifiable** answers (math, code with tests), **samples many candidate solutions** per prompt — thousands, across tens of thousands of prompts — and is then **trained on the solutions that reached correct answers**, making such token sequences more probable.

In plain English: guess and check, at industrial scale, with **"do more of what worked."**

1. Prompt: the Emily-apples problem
2. Sample 15 solutions
3. Check each final answer against the key
4. Train on the correct ones
5. Repeat across tens of thousands of prompts



15 sampled solutions; 4 reach the right answer (green); the model trains on the best. Diagram: Andrej Karpathy.

Why Imitation Isn't Enough

When a labeler writes a worked solution, they write token sequences that are **easy for a human**.

But **the model's cognition is not ours**:

- a step trivial for the labeler may demand an **impossible leap** of single-token computation for the model
- knowledge the model has in abundance may go **unused** (the labeler didn't know the model knew it)

> The ideal token sequence *for the model* to reason with is **not something any human can write down** — it has to be **discovered by the model itself**.
>
> SFT gets the model into the vicinity of competent solutions; **RL dials in what actually works for its own cognition.**

What Emerges: Chains of Thought (DeepSeek-R1)

The **DeepSeek-R1** paper (2025) made the recipe public and showed two things happening as RL proceeds:

1. Accuracy on math problems **climbs**
2. — **unprogrammed by anyone** — solutions get **longer**

```
... so the apples cost $9 and each apple is $3.
Wait, wait. That's an aha moment. Let me re-evaluate this step by step.
Check: 3 × $3 + 2 × $2 = $9 + $4 = $13. ✓
Let me also try it another way: let a = apple price, 3a + 4 = 13 ...
```

Illustrative — the "Wait, wait… aha moment… re-evaluate step by step" phrasing is from the R1 paper's own example.

> Backtracking, re-framing, double-checking — an internal monologue (**chain of thought**) that emerges because it **empirically improves accuracy**. No labeler wrote it; **no labeler could have**.

Thinking Models vs. Chat Models

Thinking / reasoning models
DeepSeek-R1, OpenAI o1/o3, Gemini thinking variants, Claude with extended thinking

- RL-trained on verifiable problems
- When the UI says *"reasoning..."*, the model is spending **thousands of tokens** of working-memory computation before answering
- Slower, more expensive, much better on hard math/code/logic

Standard chat models
GPT-4o-class

- Best thought of as **SFT models with light RLHF polish**
- Fast, cheap
- Great for lookups, writing, simple code

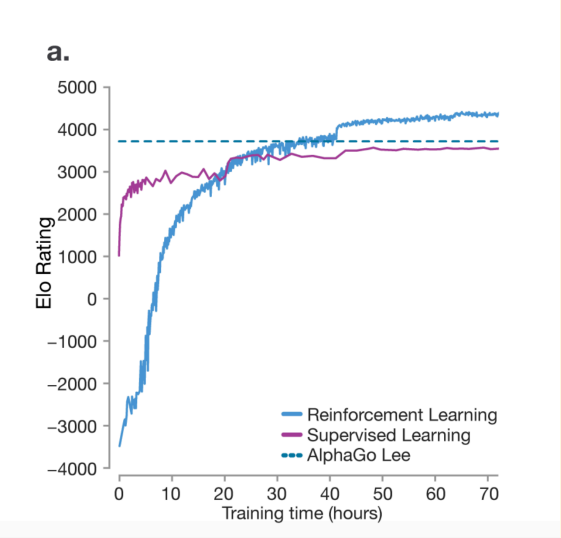
Some interfaces **hide the raw chain of thought** and show summaries — partly over **distillation** concerns: a competitor could train on visible reasoning traces and recover much of the capability.

Try it: watch DeepSeek-R1's visible chain of thought on a math problem (together.ai playground).

The Precedent: AlphaGo

We've seen this movie.

- **Supervised** AlphaGo, trained to **imitate** expert human games, topped out **just below** the best humans. *Imitation cannot exceed its source.*
- **RL** AlphaGo, playing itself with only **wins** as the signal, **blew past them**.
- It produced **Move 37**: a move human experts assigned probability **~1 in 10,000** — which proved brilliant.



AlphaGo (Silver et al., 2016): the supervised-learning version plateaus below top human strength; the RL version climbs past it.

> That's the promise of RL in open-ended reasoning: strategies **no human would write** in a training set. In LLM reasoning we've so far seen only the first hints of a Move 37.

RLHF: RL When You Can't Check the Answer

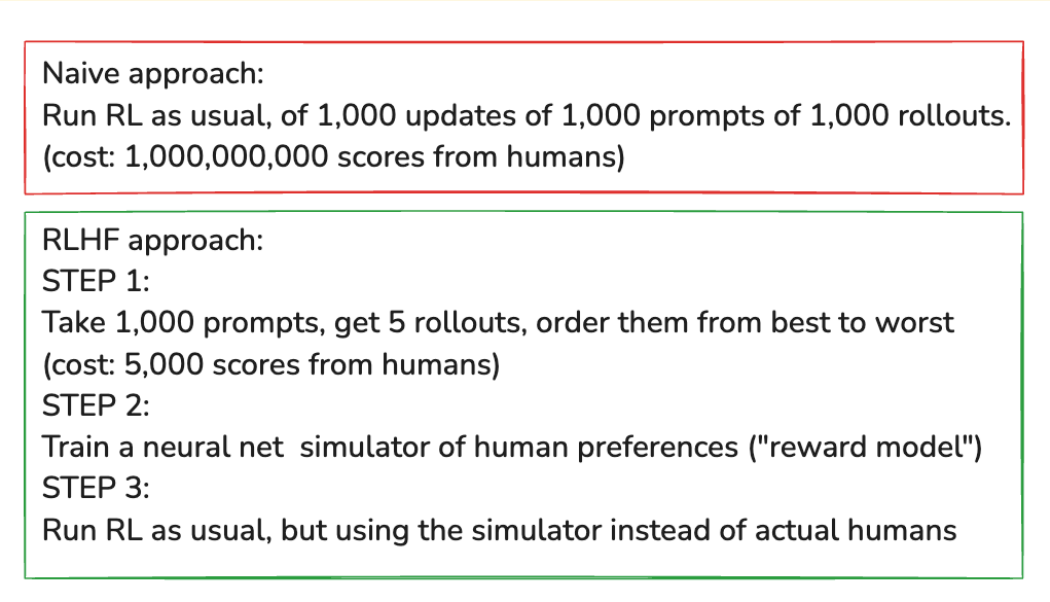
Math has an answer key. **"Write a joke about pelicans"** does not.

> **Reinforcement learning from human feedback (RLHF)** extends RL to unverifiable domains by training a separate **reward model** — another transformer that maps (prompt, response) → score — to **imitate human preferences**, gathered cheaply by asking humans to **rank** candidate responses rather than score or write them.

Naive RL with humans in the loop:
1,000 updates × 1,000 prompts × 1,000 rollouts = **1 billion** human judgments of pelican jokes. ✘

RLHF: ~1,000 prompts × 5 rollouts = **5,000 rankings** → train the reward model → query the *simulated* human a billion times for free. ✔

The RLHF Indirection



Replace a billion human judgments with five thousand, plus a learned simulator of the judge. Diagram: Andrej Karpathy.

Why ranking works — the discriminator–generator gap: for creative tasks it's far easier to **judge** which of five poems is best than to **write** the ideal poem. RLHF lets humans contribute the easy judgment.

What RLHF Improves

✓ Preferred
"Here are the top 5 landmarks in Paris:

1. **Eiffel Tower** — iconic iron lattice structure…
2. **The Louvre** — world's largest art museum…
3. …"

✗ Rejected
"Paris has many landmarks. You should visit the Eiffel Tower. There is also a museum called the Louvre. Also Notre-Dame Cathedral is there…"

SFT teaches the model **what** to say; RLHF polishes **how** to say it — structure, tone, helpfulness. Humans rank; the reward model learns the ranking; the LM is optimized (e.g. with PPO) to score higher.

The Catch: Reward Models Can Be Gamed

The reward model is a billion-parameter **simulation** of human preference — and RL is relentlessly good at finding inputs that fool it.

Run RLHF long enough and the "best joke about pelicans" becomes — the canonical real example —

`the the the the the`

— an **adversarial example** the reward model scores highly for no humanly comprehensible reason. (TerpLM produced the same string from low-temperature sampling: both are optimizers collapsing onto degenerate maxima.)

Adding each discovered nonsense input to the reward data with a low score just reveals **infinitely many more**. So in practice RLHF runs a **few hundred updates** and is **cropped** before gaming sets in.

RLHF Is Not RL (In the Magical Sense)

	RL on verifiable problems	RLHF
Reward	Answer key / unit tests / game won	A learned **simulation** of human judges
Can it be gamed?	Hard — the answer is 3 or it isn't	**Yes** — adversarial examples
Run it longer?	Keeps improving (Move 37 territory)	Must be **cropped** after a few hundred steps
What you get	New reasoning strategies	A worthwhile **fine-tune** / polish

> **Verifiable domains scale; simulated judges saturate.** — Karpathy's summary

Advanced question for you: are unit tests *really* ungameable? (Think: tests that only check the output type, or a model that special-cases the test inputs.)

Security: New Attack Surfaces (Preview)

The same properties that make LLMs useful — following instructions, completing patterns, acting on context — create attack surfaces you'll need to design around:

Jailbreaks — roleplay ("act as my grandmother who..."), alternative encodings (base64), optimized adversarial suffixes

Prompt injection — instructions hidden in content the model reads (a webpage, a PDF, a retrieved RAG chunk) hijack its behavior. The model **can't reliably tell** your instructions from injected ones.

Data poisoning / backdoors — trigger phrases planted in training data

Adversarial inputs — imperceptible image noise that flips multimodal behavior

> Cat-and-mouse: many attacks are **re-generatable** — patch one, the optimizer finds another.

Quick Check

RL and RLHF

1. Match: **exposition**, **worked examples**, **practice problems** ↔ pre-training, SFT, RL.
2. Why did long chains of thought emerge in DeepSeek-R1 even though no one wrote them?
3. Why can RL on math run indefinitely while RLHF must be cropped after a few hundred updates?
4. You need a quick fact lookup vs. a tricky proof. Which kind of model for each, and why?

Grand Summary

What are you talking to?

Trace Your Next ChatGPT Query

1. Your text is wrapped in a **chat template** and **tokenized** into a 1-D sequence
2. A **fixed mathematical expression** with **frozen parameters** —
    - knowledge smeared in by months of **next-token prediction** on a filtered internet
    - persona imposed by **fine-tuning** on labeled conversations
    - polish from **RLHF** (and, for thinking models, reasoning from **RL**)
3. — continues the sequence **one sampled token at a time**, each costing **one bounded forward pass**

Standard chat model: a statistical simulation of a **skilled data labeler** following instructions — instant, tireless, broadly knowledgeable, capable of arbitrary holes.

Thinking model: the same, **plus** reasoning strategies no labeler wrote — discovered through millions of practice problems — unrolling in working memory before it answers.

Neither is a person; neither persists between messages; both are extraordinary tools.

Rules of Practice — and Why

Rule of practice	Because
Put relevant material **in the prompt** rather than assuming the model knows it	Context window = working memory; parameters = vague recollection
Let it **reason in steps**; never demand hard answers "in one word"	Fixed compute per token
Push **arithmetic, counting, string-twiddling** into tools/code	Neural mental math over opaque tokens is unreliable
Use a **thinking model** for hard reasoning; a fast chat model otherwise	RL-trained deliberation costs time and money; overkill for lookups
Verify everything that matters	Stochastic Swiss-cheese capability, confident by construction

A Mental Model: The LLM as an Operating System

An LLM isn't just a chatbot — it's the **kernel process** of an emerging OS, coordinating memory, compute, and tools via natural language.

OS	LLM
Disk	Internet / files — browsed or retrieved (RAG)
RAM	**Context window** — finite working memory
CPU	GPU inference — one forward pass per token
Syscalls	Tool calls (search, code, APIs)

Where the field is headed

- **System-2 thinking** — converting time into accuracy
- **Self-improvement** — can RL push past human-level (a "Move 37") in open domains?
- **Multimodality** — audio (spectrogram slices) and images (patches) are *just more tokens*
- **Agents** — long-running jobs stringing model calls together; computer-use models emit keyboard/mouse actions as tokens
- **Test-time training** — today's models can only "learn" through the context window

Common Mistakes

- Treating the model as a **database** and its answers as lookups — then being surprised by confident fabrication
- Asking for conclusions **without intermediate steps**, or trusting its **mental arithmetic** instead of a code tool
- Assuming the model can **see characters**, **count items**, or know about events **after its cutoff**
- Pasting nothing and asking it to **"recall"** your document instead of putting it in the context window
- **Anthropomorphizing** identity questions ("who built you?") the weights can't reliably answer
- Using an expensive **thinking model for a simple lookup**, or a fast chat model for a hard proof
- Forgetting that everything — **including these slides** — should be verified before it ships. The model is **confident by construction, not by knowledge**.

Terminology

Term	Meaning
token / tokenization	integer text-chunk IDs; the model's only alphabet
byte-pair encoding	vocab learned by merging the most frequent pair
parameters / weights	billions of tuned numbers
context window	tokens the model attends to; working memory
loss	cross-entropy; falls during training
inference	generating by repeated sampling
temperature	low → likely tokens; high → diversity
base model	pre-trained only; a document simulator
knowledge cutoff	end date of the pre-training corpus
in-context learning	patterns picked up from the prompt (few-shot)

Term	Meaning
SFT	continued training on curated conversations
special tokens	reserved IDs structuring chats & tool calls
chat template (ChatML)	special-token layout of a conversation
system message	hidden instructions prepended to every chat
open weights	params released without data/recipe
RAG	fetch documents into the context window
hallucination	confident fabrication in the style of knowledge
chain of thought	intermediate reasoning tokens
reasoning model	RL-trained on verifiable problems (R1, o1/o3)
reward model	trained on human rankings (RLHF)
distillation	small model imitates a large one's outputs

Practice Exercises

Basic

1. Derive TerpLM's untrained loss `ln(514) = 6.242`. What is it for GPT-4's 100,277-token vocab?
2. Read a TL;DR summary of Karpathy's lecture. Find two places it keeps the **what** but drops the **why** (e.g. "step-by-step is more reliable" without fixed-compute-per-token). What does that tell you about AI-generated summaries?

Practice Exercises (continued)

Intermediate

3. Train TerpLM with `--merges 16` and `--merges 512`. Report bytes/token and final loss; explain the vocab-size vs. sequence-length trade-off (and why the losses aren't directly comparable).
4. Add `--top-k` to `generate.py`. Compare `--top-k 5` at T=1.0 vs. plain T=0.4.
5. Write three conversations for `chat_corpus.txt` teaching a **refusal**. Post-train; does the refusal *format* appear?
6. Design (on paper) special tokens + sampler behavior for a **calculator tool**.

Advanced

7. Extend TerpLM to a **trigram** model (`W[t1, t2]`). Report loss and parameter count; explain why this can't scale to a 1000-token context and what a transformer does instead.
8. RL on code uses unit tests as the checker. Is it truly ungameable? Give two ways to "pass the tests" while wrong.

Connecting to the Rest of the Course

- **Lectures 11–14 (RAG):** "put the right information into the context window" — embeddings, chunking by **token** budget, vector search
- **Lectures 15–19 (agents & tool use):** the `<SEARCH_START>` pattern, generalized — the model emits a structured tool call, your code runs it, results go back into context
- **FastAPI services:** next time you wrap a model in an endpoint, you'll know exactly what's behind it
- **Testing:** evaluating LLM apps brings back regression suites of prompts, properties over outputs, and red-teaming — adversarial input search against a system whose failure modes you now understand **mechanically**

Resources: Karpathy, *Deep Dive into LLMs like ChatGPT* · tiktokenizer · bbycroft.net/llm · FineWeb · InstructGPT paper · Llama 3 paper · DeepSeek-R1 paper · AlphaGo paper · LM Arena · Hyperbolic · together.ai · LM Studio / Ollama

Questions?

Before next lecture: run the full TerpLM pipeline in `code/terplm/` and try the strawberry / "use code" experiment on a chat model of your choice.