140.800: How to AI (for Public Health)

Week 3: (Large) Language Models

Yiqun T. Chen
Email: yiqunc@jhu.edu
Schedule office hours via email

Departments of Biostatistics and Computer Science
Data Science & AI Initiative and Malone Center for Engineering in Health

What are Large Language Models?

Definition:

- Neural networks trained on massive text corpora
- Transformer architecture with billions of parameters
- Learn patterns in language through self-supervised learning
- Can generate human-like text and perform various language tasks

Key Capabilities:

- Text generation: Create coherent, contextual text
- Question answering: Respond to complex queries
- Summarization: Distill key information from documents
- Translation: Convert between languages and domains

The Scale Revolution

Model Size Evolution:

- BERT (2018): 110M – 340M parameters
- GPT-2 (2019): 1.5B parameters
- GPT-3 (2020): 175B parameters
- PaLM (2022): 540B parameters
- GPT-4 (2023): Estimated 1+ trillion parameters

Emergent Abilities:

- Few-shot learning: Learn new tasks from examples
- Chain-of-thought reasoning: Step-by-step problem solving
- In-context learning: Adapt behavior within a conversation
- Task generalization: Apply knowledge across domains

Recap: From Sequence Modeling to Self-Supervision

Traditional Sequence Modeling:

- Traditional word representations are very corpus-limited
- Limited context window and parallel processing (e.g., RNNs, LSTMs)

The Transformer Revolution (2017):

- "Attention is All You Need": Self-attention mechanism
- Parallel processing: All positions processed simultaneously
- Scalability: Efficient training on large datasets

Self-Supervised Learning: The Foundation

What is Self-Supervised Learning?

- Learn from unlabeled data by creating labels from the data itself
- No human annotation required
- Massive scale: unlabeled means we could train on much larger text corpora

Two Main Approaches:

- Masked Language Modeling (BERT-style): "The [MASK] sat on the mat"
- Causal Language Modeling (GPT-style): "The cat sat on the ___" $\rightarrow$ predict next token

Masked Language Model Pretraining

Setup: Given a tokenized sentence $x = (x_1, \ldots, x_T)$, randomly choose a set of masked positions $\mathcal{M}$.

- For $i \in \mathcal{M}$: replace with `[MASK]` token (80%), random token (10%), or keep unchanged (10%).
- Only masked positions contribute to the loss.

Loss:

Masked Language Model Pretraining

Setup: Given a tokenized sentence $x = (x_1, \ldots, x_T)$, randomly choose a set of masked positions $\mathcal{M}$.

- For $i \in \mathcal{M}$: replace with `[MASK]` token (80%), random token (10%), or keep unchanged (10%).
- Only masked positions contribute to the loss.

Loss:

$$\mathcal{L}_{\text{MLM}} = -\sum_{i \in \mathcal{M}} \log p_\theta\big(x_i \,\big|\, x_{\setminus \mathcal{M}}\big).$$

Interpretation: Predict the original words at the masked positions, given the rest of the sentence.

Masked Language Model Pretraining

Toy Example: single mask

Input sequence:

`the cat sat on the [MASK]`

Gold token:

`mat`

Model predictions (top-5):

- `mat`: 0.70
- `floor`: 0.15
- `chair`: 0.10
- `sofa`: 0.03
- `ground`: 0.02

Loss contribution:

$$-\log 0.70 \approx 0.357$$

Masked Language Model Pretraining

How cross-entropy loss works:

- We always look at the probability assigned to the **true token**.
- If $p(\text{gold})$ is high $\Rightarrow$ loss is small.
- If $p(\text{gold})$ is low $\Rightarrow$ loss is large.
- Correct predictions still contribute non-zero loss unless $p(\text{gold}) = 1$.

Example:

$$\ell = -\log p(\text{``}\texttt{mat}\text{''})$$

Model prob.	Loss
$p = 0.85$	$-\log 0.85 \approx 0.16$ (small)
$p = 0.70$	$-\log 0.70 \approx 0.36$
$p = 0.05$	$-\log 0.05 \approx 2.99$ (large)

Key point: Training nudges the model to shift more probability mass to the correct token.

Masked Language Model Pretraining

General-domain pretraining (BERT, RoBERTa, etc.):

- Wikipedia + BookCorpus (original BERT)
- Common Crawl (CC-News, OpenWebText, RoBERTa)
- Large web-scale datasets (C4 for T5, The Pile, etc.)

Domain-specific adaptations:

- **BioBERT**: continues BERT pretraining on PubMed abstracts and PMC full-text articles.
- **SciBERT**: trained from scratch on scientific papers (Semantic Scholar corpus).
- **ClinicalBERT**: fine-tuned on clinical notes (MIMIC-III EHR dataset).
- **FinBERT**: financial text (analyst reports, SEC filings, news).

Key idea: Take a general BERT model and *further pretrain* (domain-adaptive pretraining) or train from scratch on domain corpora $\rightarrow$ embeddings become specialized to that field's vocabulary and style.

Masked Language Model Pretraining (Drawing)

Causal (Autoregressive) Language Modeling

Idea: Predict the next token given all previous ones.

- Unlike BERT (masked LM), no bidirectional context.
- At position t, model only sees $x_{<t} = (x_1, \ldots, x_{t-1})$.

Objective:

Interpretation:

Causal (Autoregressive) Language Modeling

Idea: Predict the next token given all previous ones.

- Unlike BERT (masked LM), no bidirectional context.
- At position t, model only sees $x_{<t} = (x_1, \ldots, x_{t-1})$.

Objective:

$$\mathcal{L}_{\mathsf{CLM}} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

Interpretation: Train the model to generate text one token at a time.

Causal Language Modeling

Cross-entropy loss at each step:

$$\ell_t = -\log p_\theta(x_t \mid x_{<t})$$

- Compares the model's predicted distribution with the true token.
- Model assigns a high probability to the correct token $\rightarrow$ small loss.
- Low probability on correct token $\rightarrow$ large loss.

Example: Input prefix = "`the cat sat on the`"

Gold next token	`mat`
Model $p(\texttt{mat})$	0.75
Loss contribution	$-\log 0.75 \approx 0.29$

Toy example — step by step generation

Prefix: “`the cat`”

- Step 1: predict next token
  - $p(\texttt{sat}) = 0.6$, $p(\texttt{runs}) = 0.2$, $p(\texttt{eats}) = 0.2$
  - Choose `sat`
- Step 2: prefix is now “`the cat sat`”
  - Predicts next token $p(\texttt{on}) = 0.7, p(\texttt{under}) = 0.2, \ldots$
  - Choose `on`

Generated sequence: `the cat sat on __`

Causal Language Modeling

General-domain pretraining corpora:

- GPT-2/3: WebText (scraped from outbound Reddit links).
- GPT-4/5 style: massive curated web + books + code + academic papers.
- The Pile, C4, Common Crawl.

Domain-specialized variants:

- Code models (Codex, CodeGen, StarCoder) $\rightarrow$ source code corpora.
- BioGPT $\rightarrow$ biomedical papers (PubMed).
- LegalGPT, FinGPT $\rightarrow$ legal and financial corpora.

Key point: Same objective, but data domain defines specialization.

Causal Language Modeling (Drawing)

Causal LM Training Loss Across Sentences

How is the loss aggregated?

- GPT treats training text as one long token stream (after tokenization).
- Breakpoints are inserted at **document boundaries** (e.g., end-of-text tokens).
- Within each segment (context window), the loss is computed at **every step**:

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

- Loss is summed (or averaged) across all tokens in the batch.
- No "next sentence prediction" like BERT — continuity is handled by concatenation.

Key Point: GPT learns to model long sequences of text seamlessly, not sentence-by-sentence.

MLM vs. CLM — Which for Which?

Masked LM (BERT-style):

-
-
-

Causal LM (GPT-style):

-
-
-

Summary:

-
-

MLM vs. CLM — Which for Which?

Masked LM (BERT-style):

- Strength: bidirectional context $\rightarrow$ strong encoder representations.
- Limitation: not directly generative (needs extra heads).
- Best for: classification, retrieval, embeddings, understanding tasks (e.g., sentiment analysis, named entity recognition, QA retrieval).

Causal LM (GPT-style):

- Strength: autoregressive generation $\rightarrow$ fluent text continuation.
- Best for: text generation, dialogue, summarization, code completion.
- Limitation: no direct bidirectional encoding (left-to-right only).

Summary:

- *BERT/MLM* = "read and understand."
- *GPT/CLM* = "predict and generate."

The Attention Idea

Core motivation: When reading, we don't treat every word equally. Some words are more relevant than others for understanding the current word.

Toy example: Sentence: "The cat sat on the mat."

- To interpret "sat," we care most about "cat" (subject) and "mat" (object).
- Attention is a mechanism to *learn these relevance weights automatically*.
- Each token builds its new representation by looking at others, weighted by importance.

Key idea: Attention lets every token see (and borrow information from) all other tokens.

Recap: Token Embeddings

From words to vectors:

- Words/tokens are mapped to fixed-length vectors (e.g. 300-d in Word2Vec, 768-d in BERT).
- Embeddings capture meaning: similar words $\rightarrow$ nearby vectors.
- In Transformers, we start with a learned embedding lookup table.

Toy example (2D illustration):

Token	Embedding (2D toy)
"cat"	(0.9, 0.8)
"dog"	(0.8, 0.7)
"mat"	(0.1, 0.9)
"sat"	(0.5, 0.3)

Key point: These initial embeddings are the "raw ingredients." Attention will transform them into *contextual embeddings* that depend on surrounding words.

Introducing Q, K, V

How can we compute "relevance" between tokens? We project each token embedding into three spaces:

- **Query (Q):** What am I looking for? (e.g., "sat" asking for subject/object)
- **Key (K):** What do I contain? (e.g., "cat" contains subject info)
- **Value (V):** What information can I provide if I am selected?

Toy analogy:

- "sat" sends out a query vector.
- It matches strongly with the key of "cat," somewhat with "mat," weakly with others.
- Weighted sum of corresponding values = enriched representation of "sat."

Result: Each word representation becomes context-aware.

Introducing Q, K, V

The formula:

$$\text{Attention}(Q, K, V) \;=\; \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

What it means:

1. Compute similarity: $QK^\top$ (dot products between queries and keys).
2. Scale by $\sqrt{d_k}$ to control variance (d_k is the number of rows of K).
3. Apply softmax to get attention weights (probabilities).
4. Multiply weights with V to get a weighted combination of values.

Intuition: Each token asks (Q) "Who is relevant?" and collects info (V) from others according to the match (K).

From Formula to PyTorch

The formula again:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Implementation in PyTorch:

```
import torch
import torch.nn.functional as F

def scaled_dot_product_attention(Q, K, V):
    d_k = Q.size(-1)   # embedding dimension
    # 1. Similarity scores
    scores = torch.matmul(Q, K.transpose(-2, -1))
    # 2. Scale
    scores = scores / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
    # 3. Softmax normalization
    weights = F.softmax(scores, dim=-1)
    # 4. Weighted sum of values
    output = torch.matmul(weights, V)
    return output, weights
```

Note: This is the core step inside every Transformer attention head.

Toy Example: Q, K, V

Sentence: "The cat sat" (focus on "sat")

Step 1. Embeddings (toy 2D)

Token	Embedding
cat	(1, 0)
sat	(0, 1)

Step 2. Linear projections → Q, K, V

- Query ("sat") = (0.2, 0.8)
- Key ("cat") = (0.9, 0.1), Value = (1.0, 0.0)
- Key ("sat") = (0.3, 0.7), Value = (0.0, 1.0)

Step 3. Compute attention scores (dot products)

$$\text{score}(\text{sat} \to \text{cat}) = 0.2 \cdot 0.9 + 0.8 \cdot 0.1 = 0.26$$

$$\text{score}(\text{sat} \to \text{sat}) = 0.2 \cdot 0.3 + 0.8 \cdot 0.7 = 0.62$$

Toy Example: Q, K, V

Step 4. Normalize with softmax

$$\alpha = \mathrm{softmax}([0.26,\ 0.62]) = [0.41,\ 0.59]$$

Step 5. Weighted sum of values (contextual embedding)

$$\text{Output(sat)} = 0.41 \cdot (1, 0) + 0.59 \cdot (0, 1) = (0.41,\ 0.59)$$

Interpretation:

- "sat" looks partly to itself, partly to "cat".
- The new embedding mixes subject + self-information.
- Attention lets "sat" carry forward contextualized meaning.

Attention in Causal Language Modeling

Recap: Attention output for each token

$$h_t = \text{Attention}(Q_t, K_{\leq t}, V_{\leq t})$$

- For position t, we only attend to tokens $x_{\leq t}$ (causal mask).
- The contextual vector h_t is passed through feed-forward layers.
- Finally, h_t is projected onto the vocabulary to predict x_{t+1}.

Same loss function (Causal LM):

$$\mathcal{L}_{\mathsf{CLM}} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

What is a Multi-Head Attention Head?

So far: One set of Q, K, V projections = one "attention head."

Multi-Head setup:

- Use H different sets of projection matrices.
- Each head attends in a different "representation subspace."
- Outputs from all heads are concatenated for next steps.

$$\mathrm{MHA}(Q, K, V) = [\text{head}_1; \ldots; \text{head}_H] W^O$$

Example intuition:

- Head 1: pronoun resolution ("it" $\rightarrow$ "animal")
- Head 2: subject–verb link ("cat" $\leftrightarrow$ "sat")
- Head 3: object link ("sat" $\rightarrow$ "mat")

Takeaway: Multiple heads let the model capture different types of relations in parallel.

Common Hyperparameters

Key design knobs in a Transformer:

- **Embedding dimension (d_{model})** Size of token vectors (128 $\rightarrow$ 4096). *Larger = richer representation, but quadratic cost in GPU memory.*
- **Number of heads (H)** Splits d_{model} into parallel subspaces. Typical: 4–16. *More heads = more perspectives, but each adds compute.*
- **Layers (N)** Depth of stacked Transformer blocks. *Deeper = stronger modeling, but training is slower.*
- **Feed-forward size (d_{ff})** Inner hidden dimension (often 2–4$\times$ d_{model}). *Controls non-linear capacity; memory-intensive.*
- **Context length (sequence length)** Max tokens per batch (e.g. 512, 2k, 8k+). *Attention cost grows as $O(L^2)$ with sequence length.*

Rule of thumb: Each choice trades off *accuracy* vs *GPU cost*.

Practical Sizes and GPU Cost

How big do models need to be?

- **Small (classroom / toy)** d_{model}=128, H=4, N=2-4, context=128. Fits on laptop CPU or single small GPU. Good for demos.
- **Medium (research / fine-tuning)** d_{model}=512–768, H=8-12, N=6-12, context=512–2k. Needs $\sim$1 modern GPU (12–24GB). BERT-base is here.
- **Large models** d_{model}=2k-4k, H=32–64, N=24–48, context=2k–32k. Needs multiple GPUs (A100/H100, TPU pods). Training cost = millions of GPU hours.

Pretraining Recap: What, Why, What's Learned

Data:

- Massive diverse corpora: web pages, books, code, articles, research.
- Trillions of tokens—self-supervised learning via language patterns.
- Cleaning is essential: removing duplicates, noisy or personally identifiable data. :contentReference[oaicite:1]index=1

Objective: Causal LM training:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(x_t \mid x_{<t})$$

E.g., `"The patient showed symptoms of"` $\rightarrow$ `"fever"`

What emerges:

- Syntax, semantics, world knowledge, reasoning.
- Predicting the next token drives internal understanding of language.

Pretraining in Practice: Challenges, Infrastructure & Cost

Challenges:

- **Compute:** Requires thousands of GPUs for weeks.
- **Stability:** Models can diverge $\rightarrow$ need LR warmup, clipping, normalization.
- **Data:** Web text is noisy; filtering & deduplication are critical.

Infrastructure & Cost:

- GPT-3 scale: $\sim$10k GPUs, cost $\sim$tens of millions.
- **Scaling:** Distributed training (data/tensor/pipeline) keeps GPUs busy.
- **Efficiency:** Mixed precision (FP16/BF16, now 4-bit) cuts memory & boosts speed.

Takeaway: Simple next-token loss, but enormous compute + careful engineering required.

So… Does This Mean We Can't Do LLMs with \$1M?

Obviously not! Training GPT-4 scale from scratch costs hundreds of millions, but we don't need to start from zero.

Solution: Use pretrained models:

- Hugging Face hosts thousands of ready-to-use models (BERT, GPT-2/3 variants, LLaMA, Mistral, Falcon, etc.).
- You can adapt them to your domain for **a tiny fraction of the cost**.

Example: Hugging Face Model Hub

Not From Scratch: Model APIs & Hosting

What is an API?

- API = **Application Programming Interface**
- A standardized way for software to communicate (send a request, get a response).
- For LLMs: you send text input $\rightarrow$ provider's server runs the model $\rightarrow$ you get back text output.

Why it matters for LLMs:

- No need to train or even host large models yourself.
- Provider handles GPUs, scaling, and updates.
- You focus on your application logic.

Common API providers: OpenAI (GPT-4/4o), Anthropic (Claude), Hugging Face Inference API.

How an API Call Works

Steps to use a hosted model:

1. Get an **API key** from the provider.
2. Install their Python client or use HTTP requests.
3. Send text input $\rightarrow$ receive model output.

Example (OpenAI, text completion):

```
from openai import OpenAI
client = OpenAI(api_key="YOUR_KEY")

resp = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role":"user",
               "content":"Explain photosynthesis in one sentence."}]
)

print(resp.choices[0].message.content)
# -> "Plants make food from sunlight, water, and CO2."
```

Takeaway: 3–5 lines of code = LLM in your app.

Zero-Shot Learning

What is it? Model solves tasks without any task-specific training, just by following instructions.

Example (Hugging Face):

```
from transformers import pipeline

clf = pipeline("zero-shot-classification",
               model="facebook/bart-large-mnli")

text = "This patient shows signs of high fever and cough."
labels = ["sports", "finance", "medical"]

result = clf(text, candidate_labels=labels)
print(result["labels"])
# -> ['medical']
```

Zero-Shot Learning

Beyond classification: APIs also let you generate text completions.

Example (OpenAI, completion):

```
from openai import OpenAI
client = OpenAI(api_key="YOUR_KEY")

resp = client.completions.create(
    model="gpt-4o",
    prompt="The cat sat on the",
    max_tokens=10
)

print(resp.choices[0].text)
# -> "mat and purred softly."
```

Key idea: One API, many tasks (Q&A, dialogue, code, completion).

Beyond Zero-Shot: Supervised Fine Tuning

Setup: Map input (e.g., customer feedback) x to a label token $y \in \{\texttt{NEG}, \texttt{NEU}, \texttt{POS}\}$ (e.g., sentiment classfication).

Example:

Input	`"Service was quick and friendly."`
Target label token	`POS`

Loss (token-level Cross-Entropy):

$$\mathcal{L}_{\mathsf{SFT}}(\theta) = -\log p_\theta(y \mid x) \overset{\text{Use one-layer NN}}{=} -\log \operatorname{softmax}(W h(x))_y$$

where $h(x)$ is the model representation used for classification (e.g., sentence embedding).

After training, you will have a classifier on top of the original model.

Beyond categorical labels: Open-Ended Responses

Discussion: When multiple answers can be valid, how should we evaluate the quality of different responses?

Leveraging Human Preference: RLHF

Intuition (pairwise preference):

- For a prompt x, humans compare two model responses (y_w, y_l) and mark the *preferred* one (y_w).
- Train a *reward model* $r_\phi(x, y)$ to predict these human preferences.
- Optimize the policy π_θ to *increase reward* while staying close to an SFT reference policy.

RLHF: Intuition Behind the Math

Objective:

$$\max_{\theta} \; \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}[\, r_\phi(x, y) \,] \; - \; \beta \, D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathsf{SFT}})$$

Breakdown:

- **First term:** maximize reward $r_\phi(x, y)$ (model should generate responses humans like).
- **Second term:** penalize KL divergence from π_{SFT} $\rightarrow$ keep the fine-tuned model close to the supervised baseline.
- β: tradeoff between learning new behavior and staying safe/stable.

Intuition: Think of it as: "learn from preferences, but don't drift too far from what we know works."

RLHF: Toy Example

Prompt x: `"Write a polite email declining a job offer."`
Candidate responses:

- y_w: `"Thank you for the offer. After careful thought I will not be accepting, but I truly appreciate the opportunity."` (preferred)
- y_l: `"I don't want this job."` (less preferred)

Baseline SFT policy π_{SFT}:

- Trained on generic instruction data.
- Knows how to decline but doesn't reliably choose polite over blunt style.
- Might assign: $\pi_{\text{SFT}}(y_w|x) = 0.45,\ \pi_{\text{SFT}}(y_l|x) = 0.40.$

RLHF update:

- Reward model gives higher score to y_w.
- New policy π_θ shifts probability mass:
  $\pi_\theta(y_w|x) = 0.70,\ \pi_\theta(y_l|x) = 0.15.$

Takeaway: RLHF amplifies preferences while keeping π_θ close to π_{SFT}.

RLHF: Why the KL Term Matters

Example 1: Creative Writing Request Prompt: `"Write a short story about a detective solving a mystery."` **Without KL penalty:**

- Reward model learns users rate "surprising" and "unique" content highly.
- Output: `"The detective was actually the criminal's pet goldfish who gained consciousness through quantum mechanics and solved the case by swimming through interdimensional portals."`
- Problem: Technically "surprising," but nonsensical $\rightarrow$ reward hacking.

With KL penalty:

- Model stays anchored to coherent storytelling patterns from SFT.
- Output: `The muddy prints led to the garden shed, where the detective discovered the missing antique vase.`

Limitations of RLHF: Why Look Beyond It?

RLHF has been very successful, but it comes with challenges:

- **Expensive and slow:** Requires collecting many human preference labels, plus training a separate reward model and doing RL (e.g., PPO).
- **Instability:** Reward model can be gamed $\rightarrow$ risk of reward hacking if KL term is not tuned carefully.
- **Engineering overhead:** Complex pipeline (SFT $\rightarrow$ reward model $\rightarrow$ RLHF). Harder to reproduce and scale compared to simple finetuning.
- **Opaque behavior:** Reward models may encode hidden biases; alignment is indirect.

Motivation: Simpler approaches like Direct Preference Optimization (DPO) aim to keep the benefits of preference learning but *avoid extra reward models and RL machinery*.

Direct Preference Optimization (DPO): Overview

Idea: Align to human preferences *without* training a reward model or running RL.

- Given prompt x and two responses (y_w, y_l) with $y_w \succ y_l$ (human prefers y_w).
- Push policy π_θ to prefer y_w over y_l, *relative* to a reference policy π_{ref} (usually SFT).

Objective:

$$\mathcal{L}_{\text{DPO}} = -\log \sigma\Big(\beta\,(\Delta\!\log \pi_\theta - \Delta\!\log \pi_{\text{ref}})\Big)$$

where $\Delta\!\log \pi_\star = \log \pi_\star(y_w \mid x) - \log \pi_\star(y_l \mid x)$ and σ is logistic.

Takeaway: Increase the *margin* favoring y_w beyond what the reference (SFT) already does.

DPO: Intuition Behind the Math

Pairwise margin view:

$$\Delta \log \pi_\theta \;=\; \log \pi_\theta(y_w|x) - \log \pi_\theta(y_l|x) \quad \text{vs} \quad \Delta \log \pi_{\text{ref}}$$

- If $\Delta \log \pi_\theta > \Delta \log \pi_{\text{ref}}$, the model prefers y_w more than the reference $\Rightarrow$ low loss.
- If $\Delta \log \pi_\theta \leq \Delta \log \pi_{\text{ref}}$, the model has not improved preference margin $\Rightarrow$ higher loss.
- β scales the strength of the margin push (temperature).

Why this works: No explicit reward model; just compare (win, lose) pairs and teach the model to *separate* them more than the SFT baseline.

DPO: Toy Example (with Reference SFT)

Prompt x: `"Explain Newton's First Law in simple terms."`

Responses:

- y_w (preferred, plain): `"Objects keep moving or stay still unless something pushes or pulls them."`
- y_l (less preferred, jargon): `"A body maintains its velocity vector unless acted on by an external resultant force."`

Reference (SFT) policy:

$$\pi_{\text{ref}}(y_w|x) = 0.42, \quad \pi_{\text{ref}}(y_l|x) = 0.38, \quad \Delta \log \pi_{\text{ref}} \approx \log(0.42) - \log(0.38) = 0.10$$

New policy (after DPO):

$$\pi_\theta(y_w|x) = 0.65, \quad \pi_\theta(y_l|x) = 0.20, \quad \Delta \log \pi_\theta \approx \log(0.65) - \log(0.20) = 1.18$$

Interpretation: Margin improved $0.10 \to 1.18$; the loss drops because π_θ *more strongly* prefers the human-preferred answer than SFT did.

Policy vs. LLM Output: What Gets Updated?

Supervised learning recap:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

In RLHF / DPO:

$$\pi_\theta(y_t \mid x, y_{<t}) = \mathsf{softmax}(W h_\theta(x, y_{<t}))$$

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell_{\mathsf{pref}}(\pi_\theta(x_i),\ y_i^w, y_i^l)$$

- **Policy** π_θ = LLM token distribution.
- **Output text** = sample from π_θ.
- Updating θ = same as ERM, but loss ℓ_{pref} comes from preferences (e.g. reward+KL in RLHF, margin in DPO).

Efficiency: LoRA $\Rightarrow$ only train small low-rank adapters in attention.

Fine-Tuning Helps... But Has Drawbacks

Problems with naive fine-tuning / RLHF:

- **Training instability & reward hacking:** Models may game the reward, producing strange outputs that score well but are unhelpful.
- **Model collapse:** Training on self-generated outputs can degrade diversity and accuracy over time.
- **Cost & scale:** Full fine-tuning of large LLMs requires huge compute + data. Even partial methods (e.g., RLHF with PPO) are still expensive.

So we ask: Can we get aligned behavior *without* retraining the whole model?

In-Context Learning (ICL): Few-Shot Prompting

What is In-Context Learning?

Model learns a task from a few examples (**shots**) provided directly in the prompt. **No fine-tuning or gradient updates are needed!**

$$\text{Prompt} = [\underbrace{\text{ex}_1, \ldots, \text{ex}_k}_{\text{Few-shot examples}}, \underbrace{\text{new input}}_{\text{Query}}] \rightarrow \text{Model Output}$$

Examples in Action

Math: Input: 2 $\rightarrow$ Output: 4

Input: 3 $\rightarrow$ Output: 9

Input: 4 $\rightarrow$?

Model Output: `16`

(Learns the rule $x \mapsto x^2$)

Medical: Patient: cough, fever $\rightarrow$ Flu

Patient: chest pain $\rightarrow$ Heart Issue

Patient: runny nose $\rightarrow$?

Model Output: `Cold`

(Learns symptom mapping)

Chain-of-Thought (CoT) Prompting

Key Idea: Instead of just asking for the answer, you ask for the **process**. This decomposes the problem into simpler, intermediate steps.

Zero-Shot CoT Example

Standard Prompt:
Q: A jug has 1000ml of water. I pour 250ml into a glass and then use 150ml for cooking. How much is left? ***A:*** 750ml (Incorrect)

CoT Prompt:
Q: A jug has 1000ml of water. I pour 250ml into a glass and then use 150ml for cooking. How much is left? ***Let's think step by step. A:***

1. Start with 1000ml.
2. Pouring 250ml into a glass leaves 1000ml - 250ml = 750ml.
3. Using 150ml for cooking leaves 750ml - 150ml = 600ml.

Final Answer: 600ml (Correct)

Beyond CoT: Advanced Reasoning Techniques

Simple CoT can fail on harder tasks. Advanced methods structure reasoning or connect to external tools:

- **Self-Consistency:**
  - Sample multiple CoT traces with temperature > 0.
  - Aggregate by majority vote on the final answer.
  - Reduces reliance on any single flawed chain.
- **Tree of Thoughts (ToT):**
  - Extends CoT into a **tree of reasoning steps**.
  - At each step, generate several "thoughts," evaluate, and prune.
  - Useful for planning and search-heavy tasks (games, puzzles).
- **ReAct (Reasoning + Acting):**
  - Interleaves **thoughts** with **actions** (e.g., API calls, web searches).
  - Grounds reasoning with external tools, overcoming knowledge cutoffs.
  - Example: `search("current price of NVIDIA stock")`.

Beyond CoT: Advanced Reasoning Techniques

- **Self-Consistency:**
  - Task: "What is 23×47?"
  - Run the same CoT multiple times with randomness.
  - Outputs: [1081, 1081, 981, 1081, 1081].
  - Majority vote $\rightarrow$ 1081 (correct).
- **Tree of Thoughts (ToT):**
  - Task: "Can the 8-puzzle be solved from this start state?"
  - Model explores moves as a tree: Step 1: try sliding left / up / right. Step 2: evaluate partial board states.
  - Prune bad branches $\rightarrow$ find a valid solution path.
- **ReAct (Reasoning + Acting):**
  - Task: "Who won the 2024 NBA finals?"
  - Thought: "Need current info."
  - Action: `search("2024 NBA finals winner")`
  - Observation: "Boston Celtics defeated Dallas Mavericks."
  - Final Answer: "The Celtics won in 2024."

Automating Prompt Engineering

Manual prompt design is brittle, time-consuming, and often fails to generalize — this is **prompt fragility**. New methods treat prompt design as an *optimization problem* rather than manual trial-and-error.

- **Automatic Prompt Engineer (APE):** LLM generates and scores candidate instructions.
- **DSPy:** Prompt-as-programming with modules (`ChainOfThought`, `ReAct`); compiler optimizes prompts and examples.
- **TextGrad:** Views prompts as differentiable "parameters," enabling gradient-style search.
- **Microsoft APO:** Iterative RL-style framework to refine prompts for robust performance.

Key idea: Moving from manual prompt engineering to **automated prompt programming**.

Comparison: Fine-Tuning vs. In-Context Learning

Fine-Tuning (SFT / RLHF / DPO)

Core Idea	Update parameters θ: $\min_\theta \frac{1}{N} \sum \ell(f_\theta(x_i), y_i)$
Infrastructure	Heavy: GPUs/TPUs, training pipelines, monitoring
Performance	Specialized: SOTA in domain tasks; embeds deep knowledge
Challenges	Expensive; catastrophic forgetting; alignment tax; collapse risk
Use When...	Need domain expertise, safety, and long-term consistency

In-Context Learning (ICL)

Core Idea	Keep θ fixed; condition on demos: $\pi(y\|x, \text{demo})$
Infrastructure	Light: API or local inference; no retraining
Performance	Flexible: effective few/zero-shot; adapts quickly across tasks
Challenges	Prompt fragility; context window limits; inference cost/latency
Use When...	Need rapid prototyping, ad-hoc reasoning, or lack labeled data

Takeaway: Fine-tuning $\Rightarrow$ update θ. ICL $\Rightarrow$ reuse θ via conditioning.