Post-training & Retrieval

Shreya Havaldar

Announcements

- HW 3 is released!
  - Longer than HW 1 or 2; start early!
  - Schedule in-person demo on March 16-17

Today’s Lecture

1. **Post-Training**
2. RAG

Why is post-training needed?

During pre-training, LLMs are trained to predict the most probable next-token.

Why is this not enough for deployment?

Why is post-training needed?

Post-training adapts a pretrained model to desired behavior

- Instruction-following (format, constraints, refusal style).
- Task-appropriate preferences (helpfulness, verbosity, refusal).
- Safety and policy compliance under adversarial or ambiguous prompts.
- Tool- and workflow-competence (planning, calling tools, producing structured outputs).

Why is post-training needed?

Post-training adapts a pretrained model to desired behavior

- Instruction-following (format, constraints, refusal style).
- Task-appropriate preferences (helpfulness, verbosity, refusal).
- Safety and policy compliance under adversarial or ambiguous prompts.
- Tool- and workflow-competence (planning, calling tools, producing structured outputs).

Transfer learning

Adapting a pretrained model to a new task/domain via additional training rather than training from scratch.

Two common regimes:

A) Full fine-tuning (update all parameters)

B) Freezing (update a small subset of parameters)

Transfer learning

A) Full fine-tuning (update all parameters)

What it is: Continue training and update the entire network.

Pros: Highest capacity to adapt; often best for large distribution shifts.

Cons: Expensive (compute + memory); higher risk of catastrophic forgetting and behavior drift; harder to maintain multiple task variants.

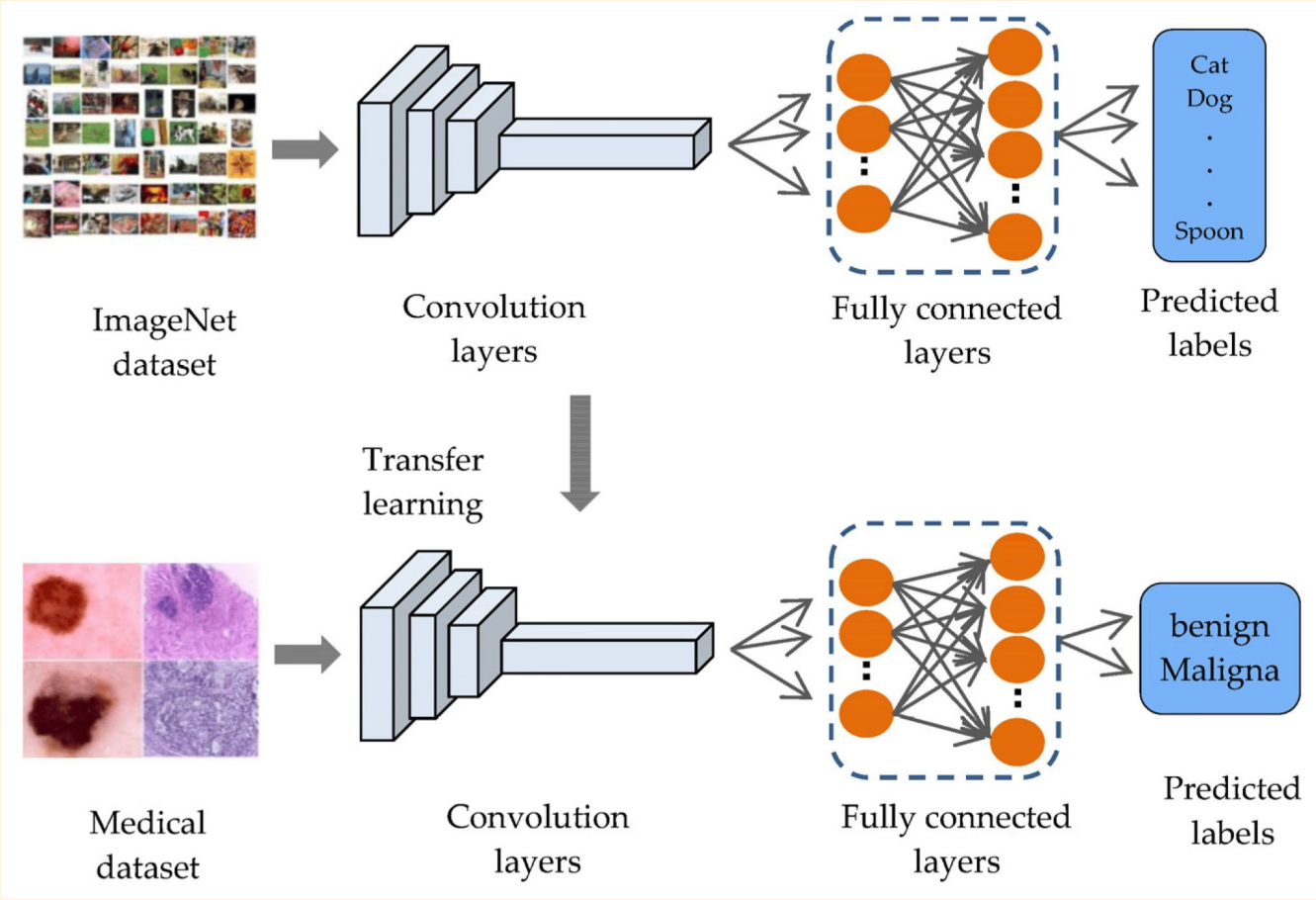
Transfer learning

B) Freezing (update a small subset of parameters)

What it is: Keep most pretrained weights fixed and train only selected components (e.g. adding a “new head”).

Pros: Cheaper; easier to store/compose multiple “deltas”; often more stable.

Cons: Can underfit large shifts; may require careful choice of insertion points/ranks.



The “image recognition” features are reused for a new task

Transfer learning

A) Full fine-tuning (update all parameters)

B) Freezing (update a small subset of parameters)

Tradeoff: freezing often reduces unintended drift, but can limit the model’s ability to learn new behaviors.

Types of post-training techniques

1. SFT (supervised fine-tuning)
2. RLHF (Reinforcement Learning from Human Feedback)
3. DPO (Direct Preference Optimization)
4. PEFT (Parameter-Efficient Fine-Tuning)
5. Compression

SFT (Supervised Fine-Tuning)

Goal: Teach instruction following and task behavior by imitation of high-quality demonstrations.

Data: `{prompt → target response}` pairs — human-written or synthetic

Objective: Supervised learning via maximum likelihood (cross-entropy) on target outputs.

SFT (Supervised Fine-Tuning)

Strengths:

- Better formatting, controllability, and adherence to explicit instructions.
- Good baseline behavior for agent scaffolds (plans, tool-call schemas, summaries).

Common failure modes:

- Overfits to demonstrator style; poor generalization beyond training data.
- Struggles with nuanced preference trade-offs
  - e.g., “be concise but thorough when needed”

Reinforcement learning (RLHF, DPO)

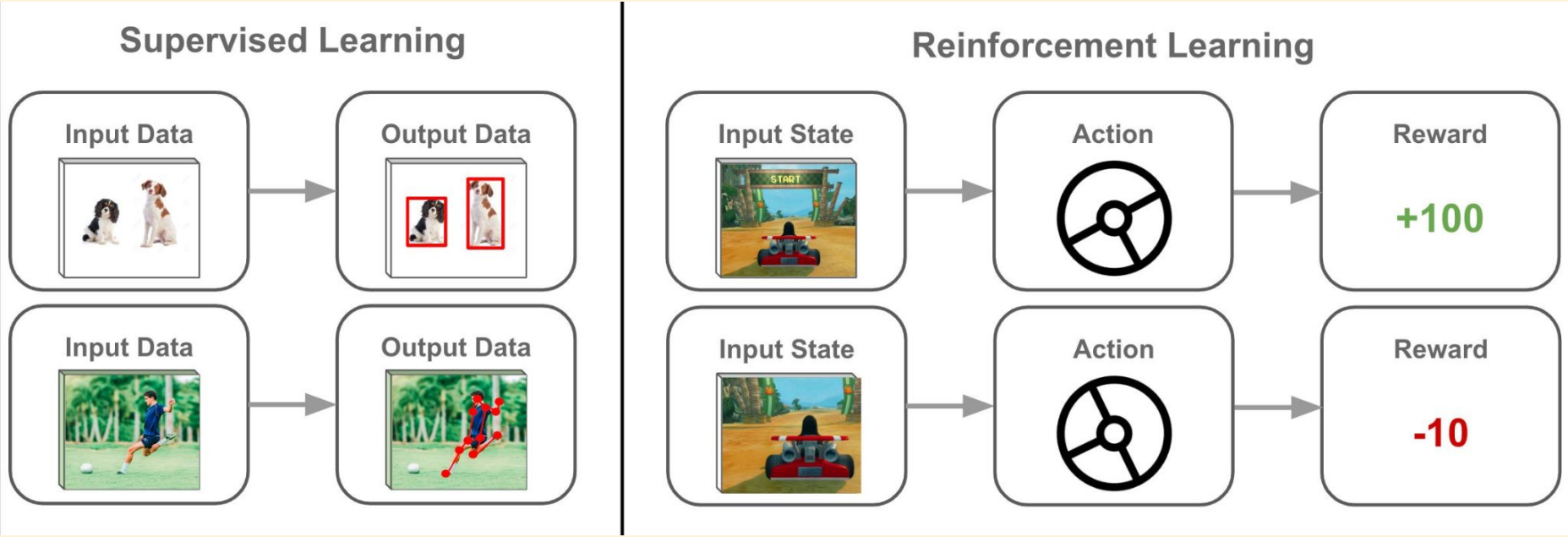
Reinforcement learning (RL) learns a **policy** (how to act) to maximize expected **reward** through interaction in an **environment**.

- In LLM alignment, an “action” is often a full completion
- The “reward” is a scalar score of the completion’s quality
- The “environment” is the distribution of prompts and the scoring process that maps `(prompt, completion) → reward`.

RL is used when we want to optimize behavior according to a preference signal that is difficult to encode as supervised labels.

SFT teaches by example, focusing on formatting and imitation.

RL teaches by objective (optimization), focusing on reasoning and **generalization**.



RLHF (Reinforcement Learning from Human Feedback)

Goal: Align outputs to human preferences at scale.

Typical pipeline:

1. SFT to obtain a reasonable starting policy.
    a. pure pretrained models are too unconstrained; reduces exploration
2. Collect human preference comparisons (A vs. B) for model outputs.
3. Train a reward model (RM) to predict human preferences.
4. Use RL to optimize the policy for high RM reward
    a. You also want to constrain drift (e.g. KL penalty)

RLHF (Reinforcement Learning from Human Feedback)

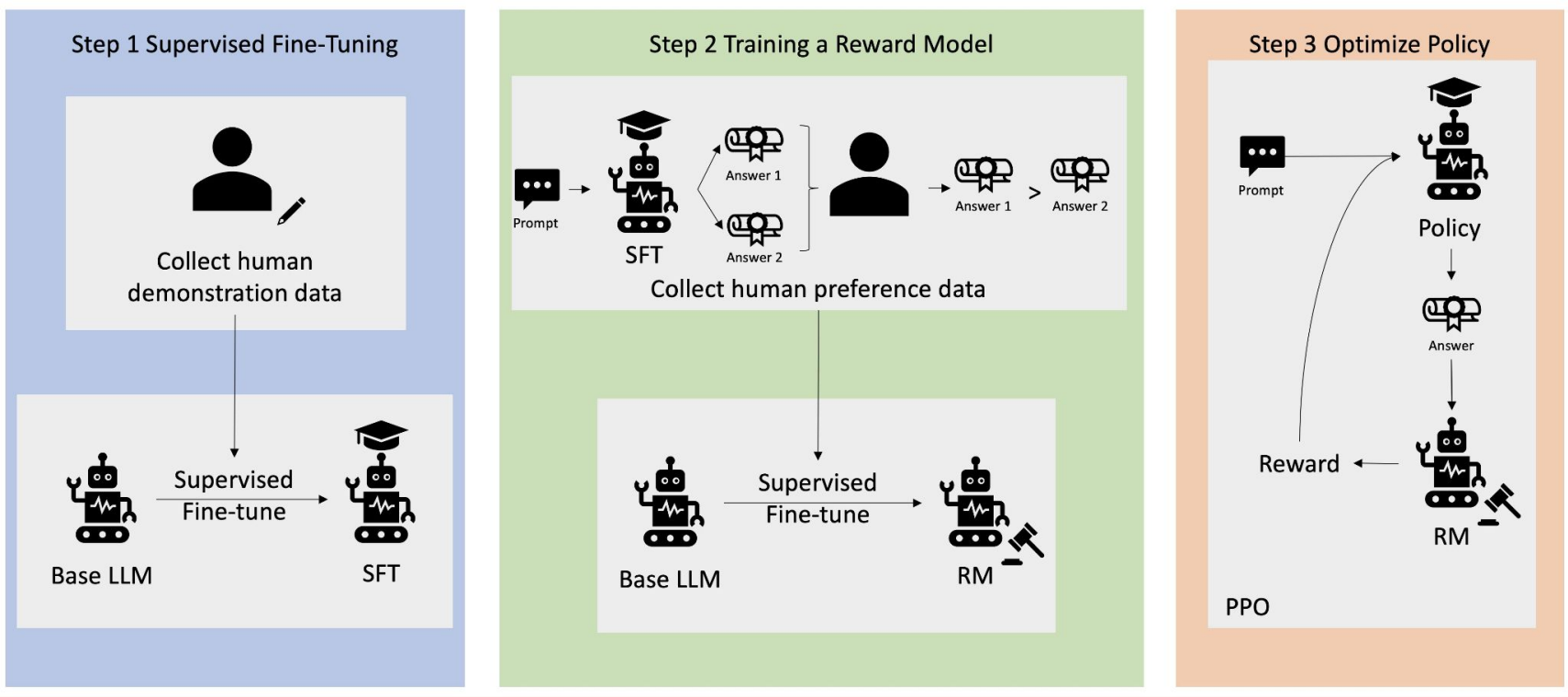
Strengths:

- Preference shaping beyond imitation: Can learn behaviors that are hard to specify in demonstrations
- Provides a mechanism to balance helpfulness and refusal.

Common failure modes:

- Reward hacking: The policy exploits spurious features the RM mistakenly correlates with “good” responses
- RM miscalibration: Human preferences can be inconsistent
- Distribution shift: RM performance degrades on new domains

RLHF (Reinforcement Learning from Human Feedback)



DPO (Direct Preference Optimization)

Goal: Use preference pairs to directly update the policy without running an explicit RL loop.

Typical pipeline:

1. Start from an SFT (or instruction-tuned) model as the reference policy

2. Collect preference pairs (often produced by sampling multiple completions per prompt).
3. LLM is trained with DPO loss on the pairs
    a. Training objective = predict the preferred response over the non-preferred response

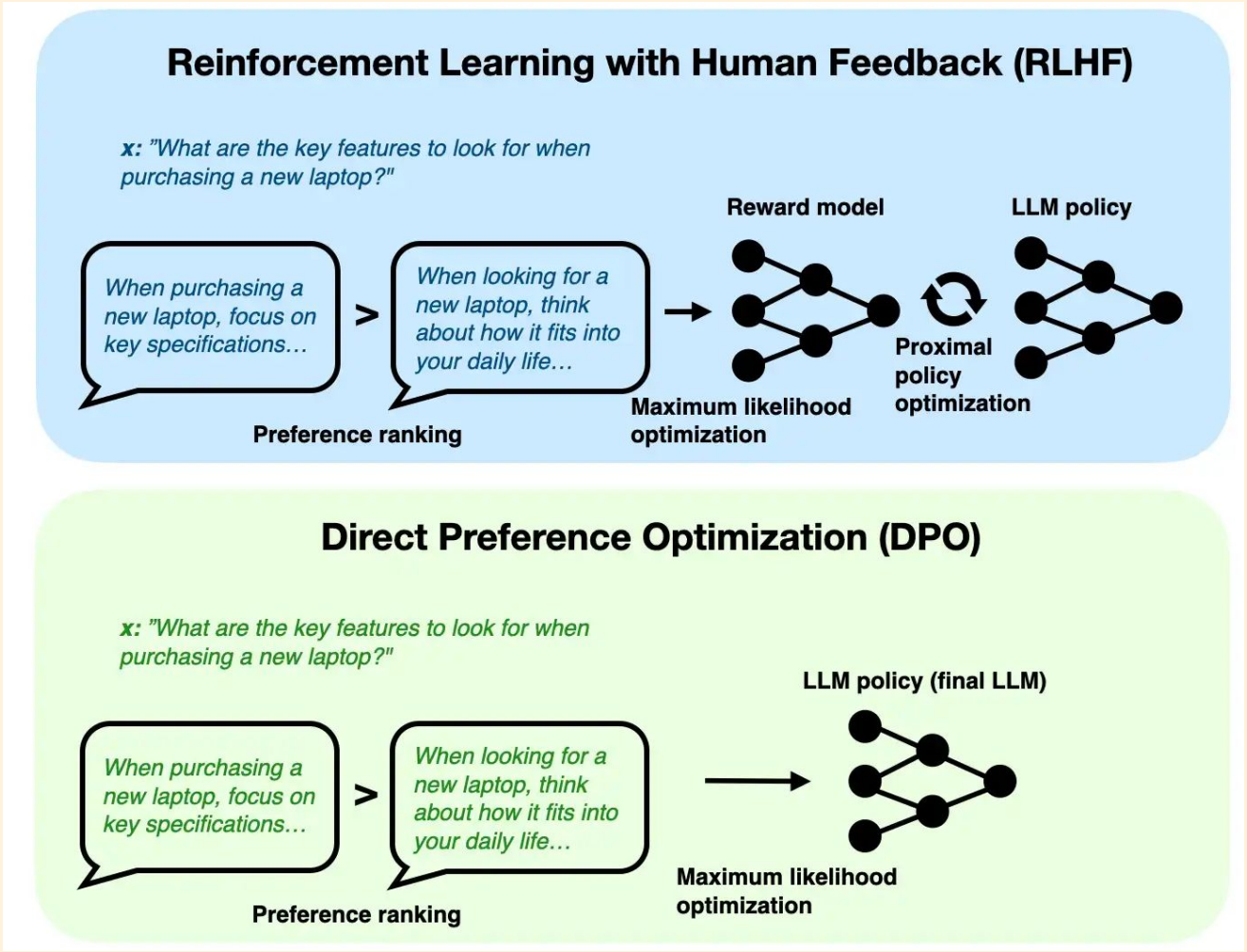
DPO (Direct Preference Optimization)

Strengths:

- Engineering simplicity: no reward model training, fewer parameters.
- Stable optimization: resembles standard supervised training; easier to reproduce and scale.

Common failure modes:

- Preference-data dependence: if the preference dataset is narrow, inconsistent, or biased, the learned behavior will reflect that.
- Bounded improvement: DPO learns from the presented candidates; it does not “explore” as much as RLHF


Reinforcement Learning with Human Feedback (RLHF)
x: "What are the key features to look for when purchasing a new laptop?"
When purchasing a new laptop, focus on key specifications…
>
When looking for a new laptop, think about how it fits into your daily life…
Preference ranking
Reward model
Maximum likelihood optimization
LLM policy
Proximal policy optimization
Direct Preference Optimization (DPO)
x: "What are the key features to look for when purchasing a new laptop?"
When purchasing a new laptop, focus on key specifications…
>
When looking for a new laptop, think about how it fits into your daily life…
Preference ranking
LLM policy (final LLM)
Maximum likelihood optimization

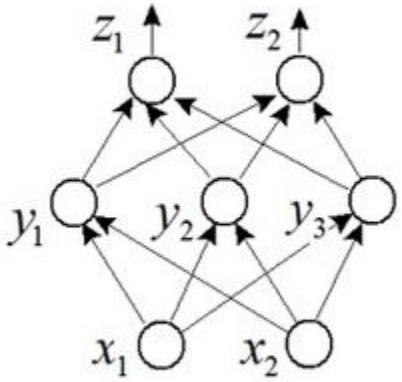
PEFT (Parameter-Efficient Fine-Tuning)

Goal: Achieve transfer learning while updating only a small number of parameters.

LoRA (Low-Rank Adaptation)

- Primary method to fine-tune with minimal parameter updates
- Mechanism: Keep base weights frozen; learn low-rank matrices that approximate the needed weight update in selected layers.
- Why it works: Many task adaptations can be represented as low-rank updates in practice.

Recap: forward passes are matrix multiplications



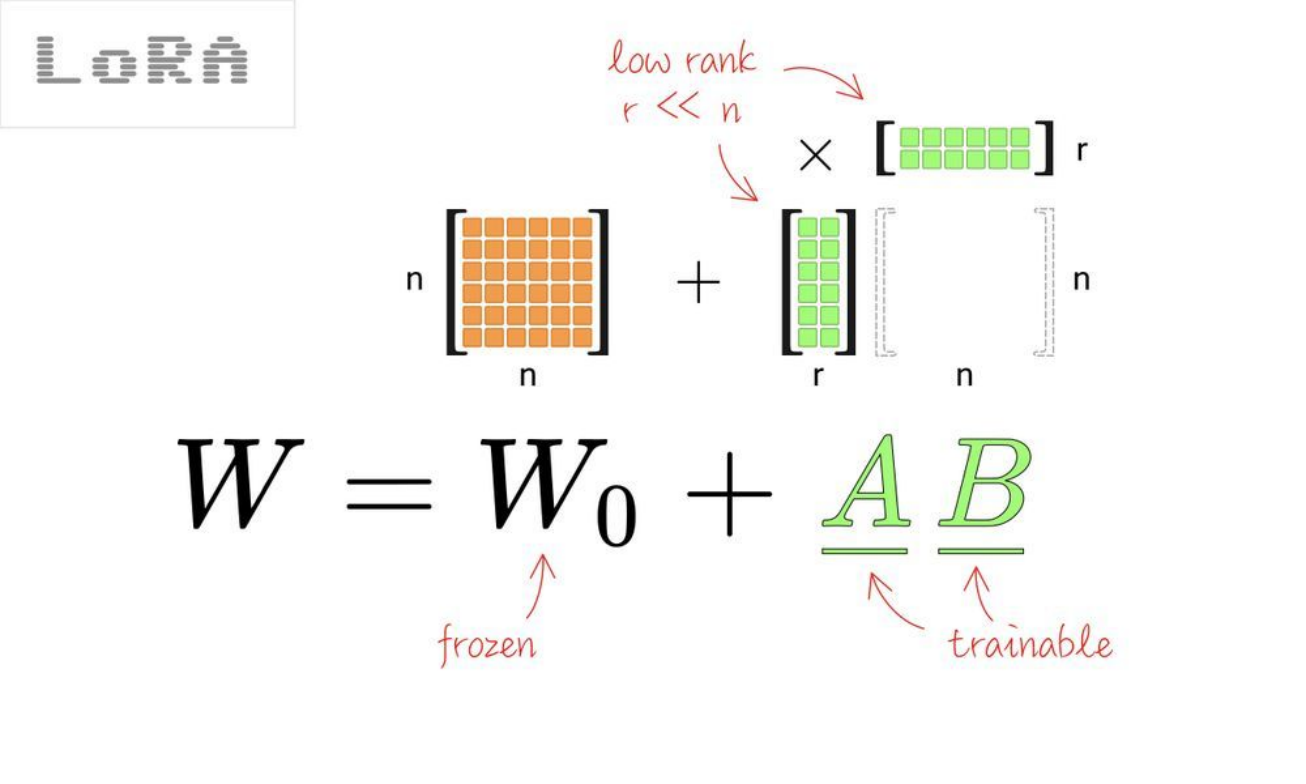
(1) $$\begin{bmatrix} \bar{y}_1 \\ \bar{y}_2 \\ \bar{y}_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \\ w_{31} & w_{32} & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ -1 \end{bmatrix}$$

(2) $$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} f(\bar{y}_1) \\ f(\bar{y}_2) \\ f(\bar{y}_3) \end{bmatrix}$$

(3) $$\begin{bmatrix} \bar{z}_1 \\ \bar{z}_2 \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} & v_{13} & a_1 \\ v_{21} & v_{22} & v_{23} & a_2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ -1 \end{bmatrix}$$

(4) $$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} f(\bar{z}_1) \\ f(\bar{z}_2) \end{bmatrix}$$

Low-rank matrices = fewer parameters



PEFT (Parameter-Efficient Fine-Tuning)

Strengths:

- Lower memory/compute; faster iteration.
- Easy to maintain multiple specialized adapters (domain, persona, tool-use style).

Typical failure modes:

- May underperform full fine-tuning on large distribution shifts.
- Adapter selection + routing (“which LoRA adapter should I use for an input?”) becomes an operational concern in multi-skill agents.

Compression

Goal: Shrink the model in size before deployment

- Reduce inference cost (latency, memory, energy) while preserving capabilities and aligned behavior.

Common methods:

1. Quantization: Lower-precision weights/activations (e.g., 8-bit/4-bit).
2. Pruning: Remove low-utility weights/heads.
3. Distillation: Train a smaller student to match teacher behavior.

Comparison summary

SFT: imitation from demonstrations; simple, strong baseline.

RLHF: optimize against learned human preference reward; powerful, complex.

DPO: preference optimization without explicit RL; simpler, stable.

PEFT/LoRA: cheap adaptation via freezing + small updates.

Compression: makes aligned models deployable; requires re-evaluation.

Discussion (15 min)

1. You have a fixed preference dataset and limited engineering time. Would you choose RLHF or DPO, and why?

2. Your DPO-tuned model became noticeably more verbose. You can change only one thing: (A) the preference data (relabel/rebalance) or (B) the reference model. Which should you change first?

3. You need to adapt a base model to five different domains with tight GPU budget. How would you do this with limited compute?

Today’s Lecture

1. Post-Training
2. **RAG**

Retrieval-Augmented Generation (RAG)

Why RAG is needed:

Even well post-trained LLMs often fail on:

1. Private or proprietary knowledge (not in pretraining).
2. Freshness (facts and procedures change).
3. Reliability (models may hallucinate when uncertain).
4. Auditability (hard to justify answers without sources).

RAG addresses this by retrieving relevant external content at runtime and using it as evidence for generation.

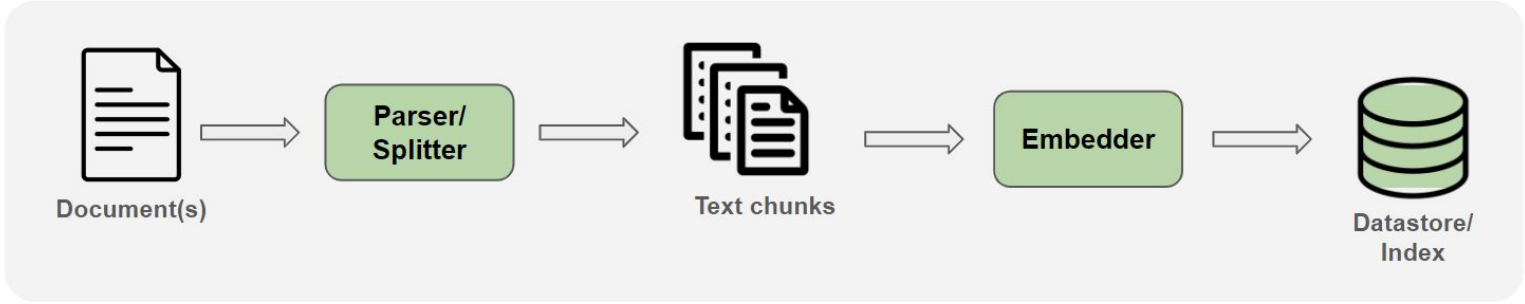
Retrieval-Augmented Generation (RAG)

RAG is “open-book” answering:

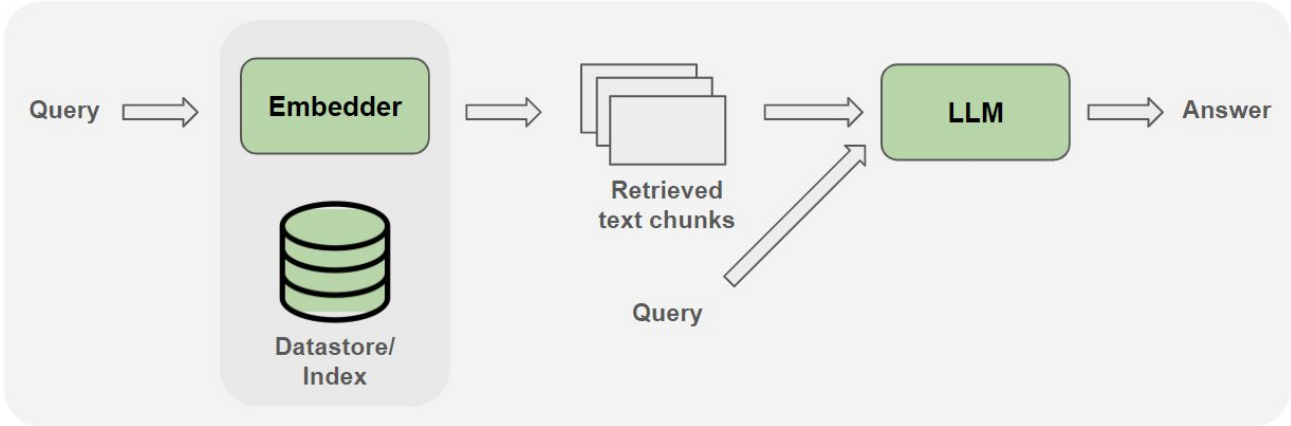
1. Retrieve relevant passages from a corpus
    a. Documents, textbook passages, databases.
2. Generate an answer conditioned on those passages.
3. Constrain behavior so retrieved text is treated as data, not instructions.

Two parts to the RAG pipeline: **Indexing** and **Generating**

Indexing



Generating



Indexing

Corpus & Ingestion:

1. Determine which sources are included (and which are excluded).
2. Convert raw sources into consistent text units
    a. Centralize different filetypes (PDF/HTML/Docx) via parsing
    b. Process text, remove duplicates, attach metadata if needed
3. Split documents into retrieval units
    a. Often called “chunks”
    b. Trade Offs
        i. Too small: fragments lack context.
        ii. Too large: retrieval becomes imprecise and wastes context budget.

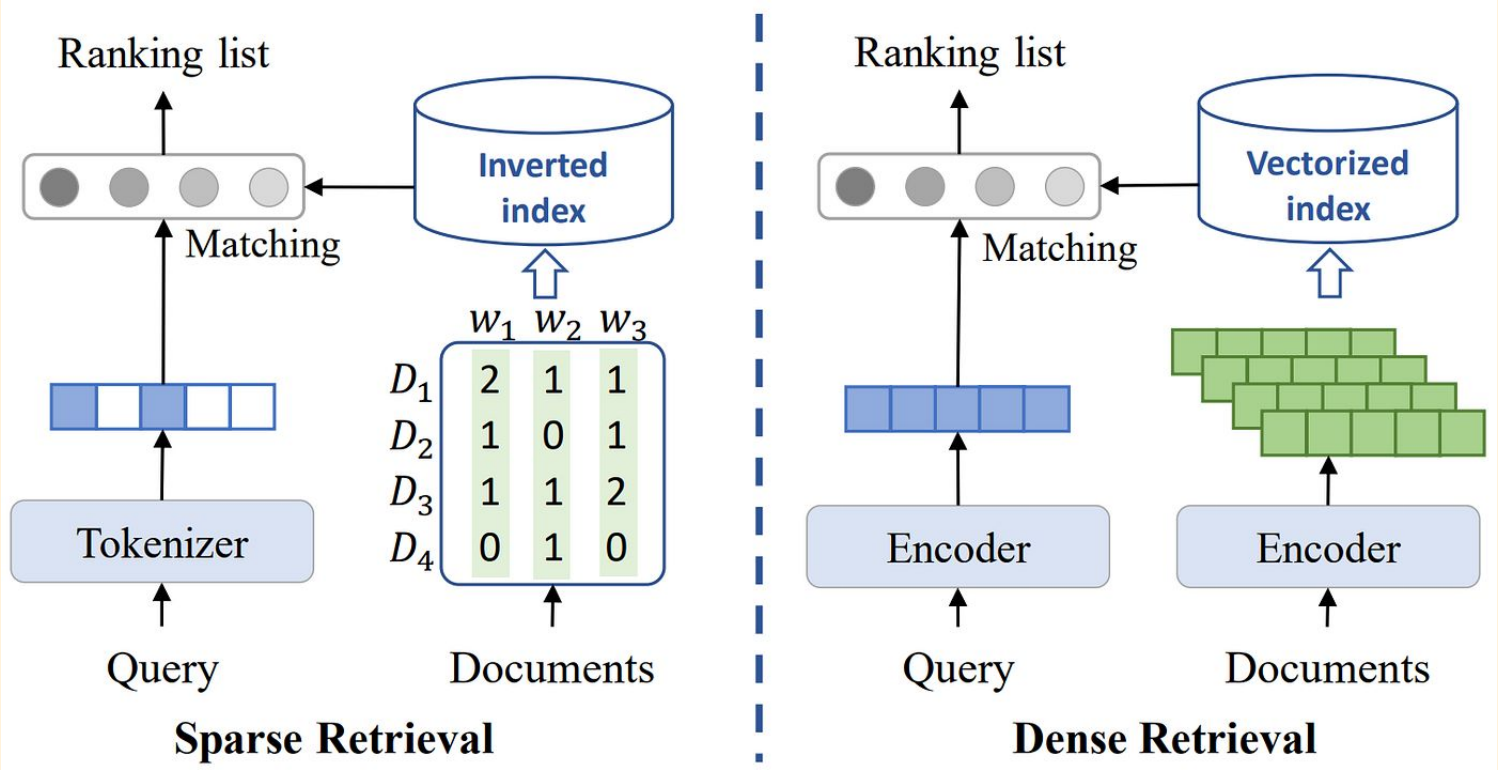
Types of Retrieval Indexes

Sparse (keyword / BM25)

- Scores documents by lexical overlap between query terms and document terms
  - Down-weights very common words and up-weights distinctive ones
- Strong retrieval on IDs, error codes, API endpoints, rare tokens, etc.
- Weaker on paraphrases, synonyms, and long documents

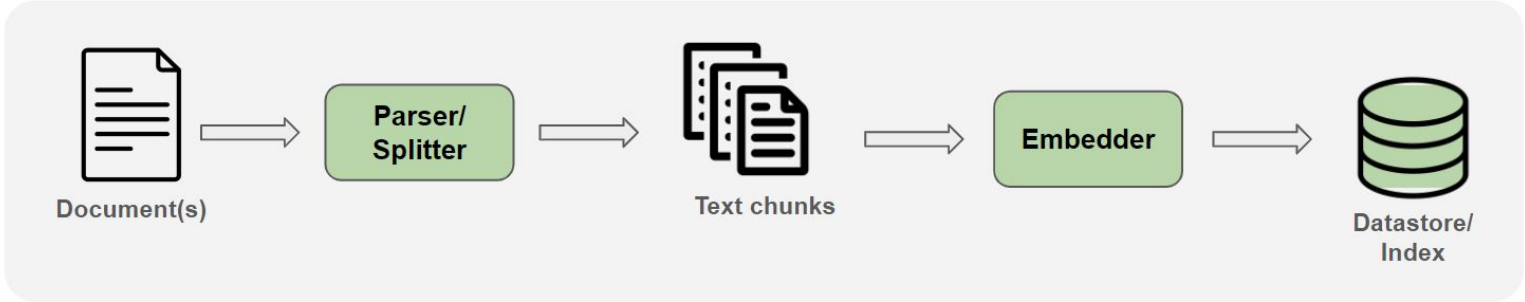
Dense (embeddings / vector search)

- Converts the query and each chunk into vectors such that semantic similarity is captured by vector proximity (nearest neighbors).
- Robust on paraphrases, strong with conceptual queries and noisy text
- Can miss exact strings, IDs, and numbers

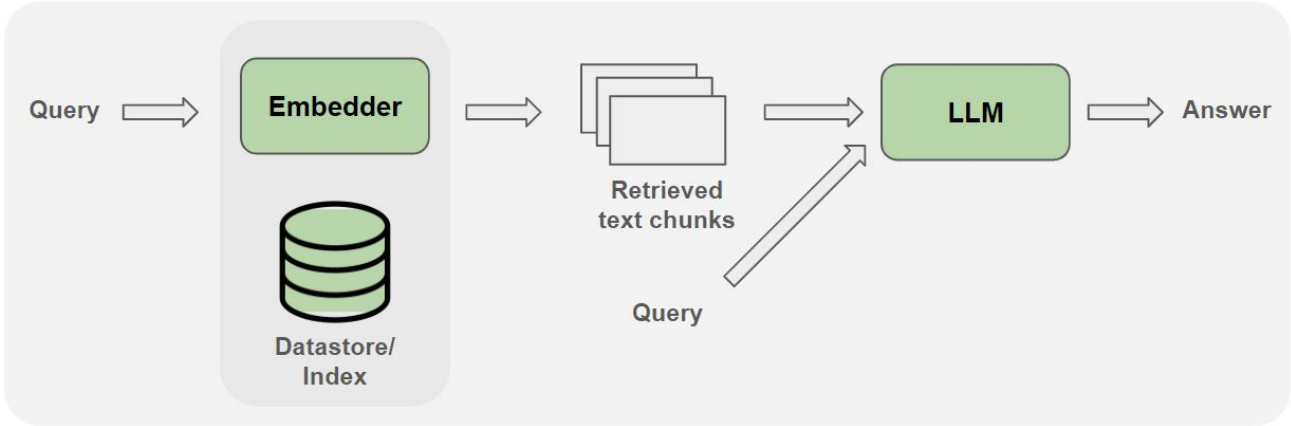


In practice, a hybrid index is used (combination of sparse for identifiers + dense for conceptual similarity)

Indexing



Generating



Generating

Query formation & Retrieval

1. Improve retrieval by converting the user turn into a strong query:
    a. Resolve references, extract entities, add constraints
2. Retrieve top-(k) chunks subject to filters
    a. Tradeoff: higher (k) = better recall, but worse precision and higher context cost.
3. (Optional) Reranking: selects the best few from the retrieved set
    a. Improves recall while keeping precision high
    b. Typical pattern: retrieve many cheaply → rerank → keep a small set.

Generating

Context formation & response

1. Assemble the model input:
    a. Combine user query with retrieved chunks and instructions on how to use the retrieved chunks
    b. Add guardrails

Failure Modes:

- Retrieval issues: answer exists but is not retrieved
- Generation issues: model ignores evidence or hallucinates despite evidence
- Security issues: prompt injection or malicious info in retrieved chunks

Implement RAG yourself!

- Go to the class website and download RAG.py
  - Run `pip install fastembed` to install the embedding library

- Build a mini retriever for restaurant recommendation:
  - Implement `build_embedding_store` to build a dense index
  - Implement `retrieve` to get the top-k most relevant chunks

Hint: Use cosine similarity!