

From Pretrained Model to Deployed Assistant

Alignment, Decoding, and Distillation

Robert Minneker

2026-02-12

Sources

Content derived from: J&M Ch. 8 (Transformers), Ch. 7 (Large Language Models)

Administrivia

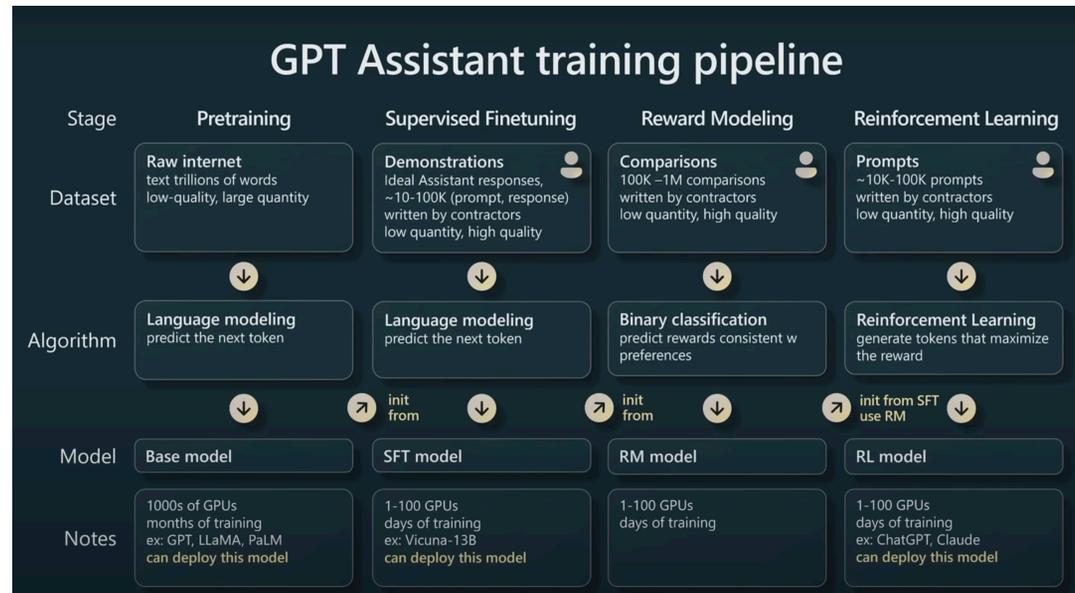
- Project checkpoints due Today!

Learning Objectives

By the end of this lecture, you will be able to:

1. Explain the difference between training-time, inference-time, and deployment-time control in LLMs
2. Implement and compare decoding algorithms
3. Describe how alignment (SFT, RLHF, DPO) modifies model behavior
4. Explain how distillation enables efficient deployment

Today's roadmap



Karpathy (2023): GPT Assistant Training Pipeline

Last time: **Pretraining** (Stage 1). Today: everything after.

Part 5: Instruction Tuning

Base models complete text — they don't follow instructions

Base Model

Prompt: "Summarize this article: The economy grew..."

"...by 3.2% in the third quarter, beating analyst expectations. Experts say this trend is likely to continue..."

Continues the article instead of summarizing

Instruction-Tuned Model

Prompt: "Summarize this article: The economy grew..."

"The article reports that the economy expanded by 3.2% in Q3, exceeding expectations due to strong consumer spending."

Actually summarizes as instructed

Supervised Fine-Tuning (SFT) on instruction-response pairs

Training data format:

Instruction: Translate "Hello, how are you?" to French.

Response: "Bonjour, comment allez-vous?"

SFT objective:

$$\mathcal{L}_{\text{SFT}} = - \sum_{i=1}^N \log P_{\theta}(y_i | x_i)$$

- x_i : instruction (input)
- y_i : desired response (output)
- Same cross-entropy loss, but on curated instruction data

Instruction tuning: Implementation details

Formatting data for instruction-tuned models:

```
1 # For summarization with instruction-tuned models
2 def prepare_for_summarization(article, tokenizer):
3     prompt = f"Summarize the following article.\n\n{article}\n\nTL;DR:"
4     return tokenizer(prompt, return_tensors="pt")
5
6 # System prompts set the model's "persona"
7 system_prompt = "You are a helpful assistant and an expert at summarizing articles."
```

Fine-tuning with label masking (for training summarization models):

```
1 # We only want to train on generating the SUMMARY, not the article
2 labels = input_ids.clone()
3 labels[:prompt_length] = -100 # -100 is ignored by CrossEntropyLoss
4
5 # The model learns: given article, predict summary tokens
```

Large models are expensive to deploy

70B+

parameters in frontier
models

\$\$\$

GPU costs for serving

High latency

slow responses at scale

We want similar quality at lower cost.

 Preview: Knowledge Distillation (Part 8)

We can train a smaller “student” model to mimic the large model’s behavior — getting most of the quality at a fraction of the cost.

Before turning to efficiency, we’ll address a deeper limitation of SFT: It only learns from **one correct answer** per instruction. Can we do better?

Part 6: Preference Alignment with RLHF

Why RLHF? SFT isn't enough

SFT limitations:

- One "correct" answer per instruction
- Can't encode preferences between valid responses
- Expensive to write high-quality responses
- May learn to imitate surface patterns

RLHF advantages:

- Captures human preferences between options
- Easier to judge than to write
- Learns subtle quality distinctions
- Optimizes for what humans actually want

Key insight: It's easier to say "A is better than B" than to write the perfect A.

RLHF: The three-step process

1 Collect Preference Data

For each prompt, generate multiple responses. Humans rank: $A > B$ or $B > A$



2 Train Reward Model

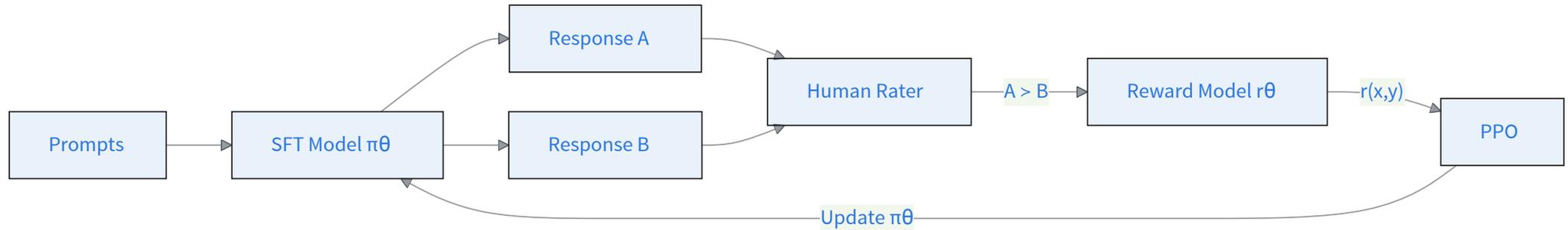
Learn to predict human preferences: $r(x, y) \rightarrow$ scalar score



3 Optimize LLM with RL

Use PPO to maximize expected reward while staying close to SFT model

RLHF data flow



The reward model is a **learned proxy** for human judgment — it scores responses so we don't need a human in the loop at training time.

Reward model intuition: Learning from comparisons

Prompt: "Explain gravity to a 5-year-old"

Response A ✓ preferred

"Gravity is like an invisible magnet in the Earth that pulls everything down — that's why when you jump, you come back!"

Response B

"Gravity is a fundamental force described by general relativity as the curvature of spacetime caused by mass-energy..."

Reward model assigns scores: $r(\mathbf{A}) = 2.4$, $r(\mathbf{B}) = 0.8$

$$P(\mathbf{A} > \mathbf{B}) = \sigma(2.4 - 0.8) = \sigma(1.6) \approx 0.83$$

This intuition leads to the **Bradley-Terry model** →

Training the reward model

Bradley-Terry preference model:

$$P(A \succ B) = \sigma(r_\theta(x, y_A) - r_\theta(x, y_B))$$

Reward model loss:

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(x, y_w, y_l)} [\log \sigma(r_\theta(x, y_w) - r_\theta(x, y_l))]$$

- y_w : preferred (winning) response
- y_l : dispreferred (losing) response
- σ : sigmoid function

The reward model learns to assign higher scores to preferred responses.

Optimizing the LLM with reinforcement learning

Objective:

$$\max_{\theta} \mathbb{E}_{y \sim \pi_{\theta}(y|x)} [r(x, y)] - \beta \cdot \text{KL}[\pi_{\theta} || \pi_{\text{ref}}]$$

Reward Maximization

$$\mathbb{E}[r(x, y)]$$

Generate responses that score high

KL Penalty

$$-\beta \cdot \text{KL}[\pi || \pi_{\text{ref}}]$$

Don't deviate too far from SFT model

- PPO (Proximal Policy Optimization) is commonly used
- KL penalty prevents “reward hacking”

From RLHF to DPO: What changes?

RLHF

- Train a separate reward model
- Run PPO optimization loop
- Complex, unstable training
- Three separate stages



DPO

- No reward model needed
- No RL loop required
- Directly optimize preference pairs
- Standard supervised learning

Key insight: Same preference data, dramatically simpler training.

Alternatives to RLHF: Direct Preference Optimization

Key insight: The optimal RLHF policy has a closed-form relationship with the reward model. DPO exploits this to optimize *directly* on preference pairs — no reward model or RL needed.

DPO loss (Rafailov et al., 2023):

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E} \left[\log \sigma \left(\beta \log \frac{\pi_{\theta}(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_{\theta}(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right) \right]$$

Reading the formula: increase the probability of the preferred response y_w (relative to the reference model) and decrease the probability of the dispreferred response y_l .

RLHF

Preferences → Reward Model → RL

Complex, unstable, expensive

DPO

Preferences → Direct optimization

Simpler, stable, supervised learning

Alignment goals: Helpful, Harmless, Honest

Helpful

Answers questions effectively
Completes requested tasks
Provides useful information

Harmless

Refuses dangerous requests
Avoids harmful content
Doesn't assist malicious use

Honest

Acknowledges uncertainty
Doesn't make things up
Corrects mistakes

Tension

These goals can conflict: A helpful answer to a harmful request is itself harmful.

Concept check

Quick questions (think for 30 seconds):

1. In the RL objective, what happens if we set $\beta = 0$?
2. An RLHF-trained chatbot starts adding “As an AI language model...” to every response. What went wrong?

RL Objective reminder:

$$\max_{\theta} \mathbb{E}_{y \sim \pi_{\theta}(y|x)} [r(x, y)] - \beta \cdot \text{KL}[\pi_{\theta} || \pi_{\text{ref}}]$$

From training to generation

We've covered how LLMs are **trained** to be helpful:

1. **Pretraining** — learn language from raw text
2. **SFT** — learn to follow instructions (Part 5)
3. **RLHF/DPO** — align with human preferences (Part 6)

Now: how do they actually **generate text**?

Given a trained model and a prompt, how do we produce an output — one token at a time?

But note: **what you put in matters as much as how tokens come out**. We'll cover the *output* side today (decoding), and the *input* side next time (prompting, in-context learning, reasoning).

Three control knobs in LLM systems

Training-time

(change weights)

- SFT
- RLHF
- DPO

Inference-time

(change decoding)

- Greedy
- Temperature
- Top-k / Top-p

Deployment-time

(change size)

- Knowledge distillation

These mechanisms operate on different axes but interact in final behavior.

Training vs. inference: Different levers

Training (shapes probabilities)

- Alignment modifies the token distribution
- Changes relative likelihoods

Inference (selects from distribution)

- Decoding determines which tokens get chosen

These interact in practice:

- Greedy decoding can amplify repetitive artifacts from training
- High temperature increases hallucination risk regardless of alignment
- Beam search often reduces diversity even in well-aligned models

Part 7: Text Generation and Decoding

The autoregressive generation loop

```
1 def generate(model, prompt_ids, max_new_tokens=50):
2     input_ids = prompt_ids
3
4     for _ in range(max_new_tokens):
5         # Forward pass: get logits for all positions
6         logits = model(input_ids) # (B, seq_len, vocab_size)
7
8         # Extract logits for the LAST position only
9         next_token_logits = logits[:, -1, :] # (B, vocab_size)
10
11        # DECODE: Select next token (this is where methods differ!)
12        next_token = decode(next_token_logits)
13
14        # Append and continue
15        input_ids = torch.cat([input_ids, next_token.unsqueeze(1)], dim=1)
16
17    return input_ids
```

The `decode()` function is where greedy, sampling, top-k, and top-p differ!

Decoding as search in sequence space

Greedy Local argmax — pick the best token at each step

Beam search Approximate global argmax — track top-B sequences

Sampling Stochastic search — randomness enables exploration

Top-p Entropy-aware truncation — adapt randomness to confidence

Generating text is approximate search over an exponential space.

Why does the decoding method matter?

Prompt: "The city of Paris is"

Greedy decoding:

"The city of Paris is the capital of France. The capital of France is the capital of France. The capital of France is the capital of..."

→ Repetitive loops

Pure sampling:

"The city of Paris is accordion Tuesday gravel. Momentum faintly bicycle the whiskers of launching..."

→ Incoherent (low-probability tokens sampled)

Nucleus sampling ($p=0.9$):

"The city of Paris is known for its stunning architecture, world-class museums, and vibrant café culture that draws millions of visitors..."

→ Fluent and diverse

The challenge: find the sweet spot between **coherence** and **diversity**.

Greedy decoding: Always pick the most likely token

$$\hat{w}_t = \arg \max_w P(w \mid w_1, \dots, w_{t-1})$$

```
1 def greedy(logits):  
2     """Select the token with the highest logit"""  
3     return torch.argmax(logits, dim=-1)
```

Advantages:

- Deterministic (reproducible)
- Fast (no sampling)
- Often coherent

Limitations:

- Repetitive outputs
- No diversity
- Can get stuck in loops

Vanilla sampling: Sample from the distribution

$$w_t \sim P(w \mid w_1, \dots, w_{t-1})$$

```
1 def sample(logits):
2     """Sample from the softmax distribution"""
3     probs = F.softmax(logits, dim=-1)
4     return torch.multinomial(probs, num_samples=1).squeeze(-1)
```

- `softmax` converts logits to probabilities
- `multinomial` samples from the categorical distribution
- Each run produces different output

Problem with Pure Sampling

Low-probability tokens can still be selected, leading to incoherent text:

“The capital of France is **accordion...**”

Temperature sampling: Control randomness

$$P_T(w) = \frac{\exp(z(w)/T)}{\sum_{w'} \exp(z(w')/T)}$$

```
1 def temperature(logits, t=1.0):
2     """Apply temperature scaling before sampling"""
3     scaled_logits = logits / t
4     probs = F.softmax(scaled_logits, dim=-1)
5     return torch.multinomial(probs, num_samples=1).squeeze(-1)
```

Temperature	Effect	Distribution
$T \rightarrow 0$	More deterministic	Approaches greedy
$T = 1$	Original distribution	Standard sampling
$T > 1$	More random	Flatter distribution

Temperature effects visualized

Original logits: [4.0, 2.0, 1.0, 0.5]

$T = 0.5$ (sharper)

probs \approx [0.88, 0.09, 0.02, 0.01]



$T = 1.0$ (original)

probs \approx [0.64, 0.24, 0.09, 0.05]



$T = 2.0$ (flatter)

probs \approx [0.42, 0.26, 0.18, 0.14]



Top-k sampling: Only consider the k best tokens

```
1 def topk(logits, k=50):
2     """Sample from only the top k tokens"""
3     # Get top k logits and their indices
4     top_logits, top_indices = torch.topk(logits, k, dim=-1)
5
6     # Sample from the top k
7     probs = F.softmax(top_logits, dim=-1)
8     sampled_idx = torch.multinomial(probs, num_samples=1)
9
10    # Map back to vocabulary indices
11    return torch.gather(top_indices, -1, sampled_idx).squeeze(-1)
```

Idea: Truncate the long tail of unlikely tokens, then sample from the rest.

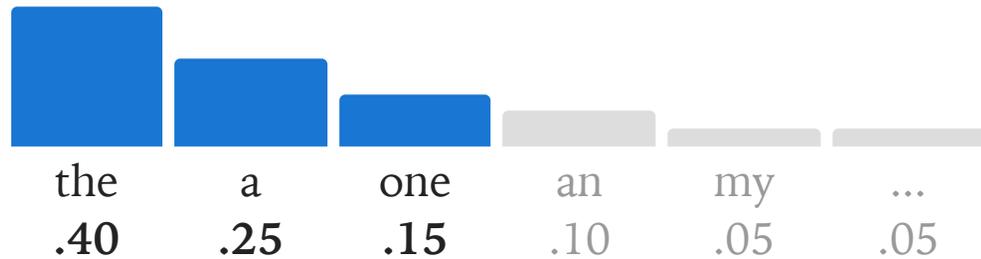
k = 50: Sample from top 50 tokens only

k = 1: Equivalent to greedy decoding

Visualizing truncation: top-k vs top-p

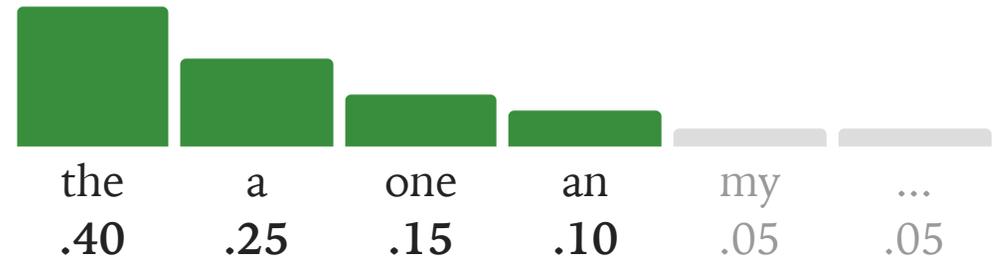
Same distribution, different truncation:

Top-k (k=3): Fixed — always 3 tokens



Covers 80% of probability mass

Top-p (p=0.9): Adaptive — cumsum ≥ 0.9



Covers 90% of probability mass (4 tokens)

Top-k misses “an” (10% probability) because it always takes exactly k.
Top-p includes it because 90% mass hasn’t been reached yet.

Top-p (nucleus) sampling: Adaptive truncation

Idea: Keep the smallest set of tokens whose cumulative probability $\geq p$

Algorithm (Holtzman et al., 2020):

1. Sort tokens by probability (highest first)
2. Compute cumulative probabilities
3. Find the cutoff where cumulative probability exceeds p
4. Zero out all tokens beyond the cutoff
5. Sample from the remaining tokens

Top-p implementation

```
1 def topp(logits, p=0.9):
2     """Nucleus sampling: dynamic truncation based on cumulative probability"""
3     # Step 1: Sort by probability (descending)
4     sorted_logits, sorted_indices = torch.sort(logits, descending=True, dim=-1)
5     sorted_probs = F.softmax(sorted_logits, dim=-1)
6
7     # Step 2: Compute cumulative probabilities
8     cumsum_probs = torch.cumsum(sorted_probs, dim=-1)
9
10    # Step 3-4: Find and apply cutoff
11    sorted_indices_to_remove = cumsum_probs > p
12    # Tricky: shift right so we INCLUDE the token that crosses p
13    sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
14    sorted_indices_to_remove[..., 0] = False # Always keep at least one token
15
16    sorted_logits[sorted_indices_to_remove] = float('-inf')
17
18    # Step 5: Sample from remaining tokens
19    probs = F.softmax(sorted_logits, dim=-1)
20    sampled_idx = torch.multinomial(probs, num_samples=1)
21    return torch.gather(sorted_indices, -1, sampled_idx).squeeze(-1)
```

The right-shift trick (lines 8–9)

Without the shift, we'd *exclude* the token that crosses the threshold. The shift ensures the nucleus always covers at least p probability mass.

Top-p example: Adaptive behavior

Confident prediction (peaked)

probs: [0.70, 0.15, 0.08,
0.04, 0.03]

cumsum: [0.70, 0.85, **0.93**,
0.97, 1.0]

$p=0.9 \rightarrow$ keep 3 tokens

Uncertain prediction (flat)

probs: [0.25, 0.20, 0.18,
0.15, 0.12, 0.10]

cumsum: [0.25, 0.45, 0.63,
0.78, **0.90**, 1.0]

$p=0.9 \rightarrow$ keep 5 tokens

Key advantage over top-k: The “effective k” adapts to the distribution!

Concept check

Quick questions (think for 30 seconds):

1. What happens to entropy as temperature increases?
2. When does top-k approximate top-p?
3. Why might greedy decoding increase repetition?

Comparing decoding methods

Method	Diversity	Coherence	Deterministic?	Best for
Greedy	Low	High	Yes	Code, factual QA
Temperature (low)	Low	High	No	Slight variation
Temperature (high)	High	Low	No	Brainstorming
Top-k	Medium	Medium-High	No	General use
Top-p	Medium	High	No	Dialogue, creative

In practice: Top-p ($p=0.9$) with temperature ($t=0.7-0.9$) is a common default.

What about beam search?

Beam search keeps the top-B most likely *sequences* at each step (not just tokens).

Still used for:

- Machine translation
- Summarization
- Tasks with a single "correct" output

Less used for open-ended generation:

- Tends to produce bland, generic text
- Maximizes likelihood \neq maximizes quality
- Sampling methods preferred for dialogue

Modern LLM APIs (ChatGPT, Claude) use sampling-based methods, not beam search.

Generation evaluation metrics

Perplexity (lower = better):

$$\text{PPL} = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(w_i | w_1, \dots, w_{i-1}) \right)$$

Perplexity

Measures naturalness
"How surprised is the model?"

Fluency

Grammatical
acceptability
(e.g., CoLA classifier)

Diversity

Unique n-grams ratio
(higher = more varied)

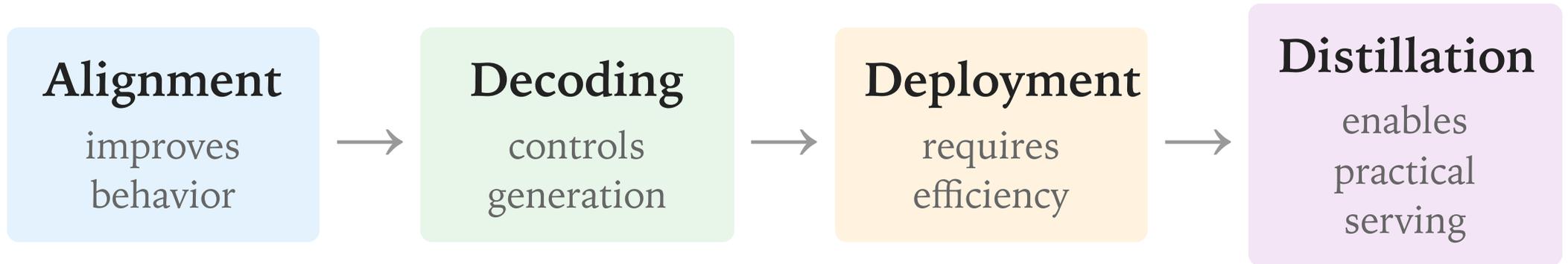
How decoding methods affect metrics

Method	Perplexity	Fluency	Diversity
Greedy	Best	High	None
High temperature	Worse	Lower	High
Top-p (p=0.9)	Good	High	Good

Top-p gives the best balance — good perplexity *and* diversity without sacrificing fluency.

Part 8: Knowledge Distillation

From research model to production system



Distillation is **deployment engineering** — it doesn't change *what* the model knows, but makes it practical to serve.

Knowledge distillation: training smaller models from larger ones

Core idea: A small “student” model learns to mimic a large “teacher” model



Why soft labels? They encode richer information

Hard label (one-hot):

$$P(\text{next token}) = [0, 0, 1, 0, 0, \dots]$$

Soft label (teacher distribution):

$$P(\text{next token}) = [0.01, 0.15, 0.60, 0.12, 0.08, \dots]$$

Example: "The cat sat on the ___"

Hard label:

mat: 1.0

everything else: 0.0

Soft label (from teacher):

mat: 0.45, rug: 0.20

floor: 0.15, couch: 0.08...

Soft labels teach: "rug is similar to mat, floor is plausible too"

The distillation loss combines hard and soft targets

Training objective:

$$\mathcal{L}_{\text{distill}} = \alpha \cdot \mathcal{L}_{\text{soft}} + (1 - \alpha) \cdot \mathcal{L}_{\text{hard}}$$

where:

$$\mathcal{L}_{\text{soft}} = - \sum_i P_T^{(\tau)}(i) \log P_S^{(\tau)}(i)$$

$$P^{(\tau)}(i) = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

- τ = temperature (same concept as Part 7! Higher \rightarrow softer, reveals more structure)
- P_T = teacher probability, P_S = student probability
- α balances learning from teacher vs. ground truth

Distillation training loop

```
1 # Knowledge distillation training loop (simplified)
2 for articles, reference_summaries in dataloader:
3     # Teacher generates soft labels (no gradient needed)
4     with torch.no_grad():
5         teacher_logits = teacher_model(articles, reference_summaries)
6
7     # Student predicts
8     student_logits = student_model(articles, reference_summaries)
9
10    # Soft loss: match teacher's distribution (with temperature)
11    soft_loss = F.kl_div(
12        F.log_softmax(student_logits / tau, dim=-1),
13        F.softmax(teacher_logits / tau, dim=-1),
14        reduction="batchmean"
15    ) * (tau ** 2) # Scale factor corrects for temperature
16
17    # Hard loss: standard cross-entropy on ground truth
18    hard_loss = F.cross_entropy(student_logits, reference_summaries)
19
20    # Combined loss
21    loss = alpha * soft_loss + (1 - alpha) * hard_loss
22    loss.backward()
```

Distillation in practice: notable examples

Student Model	Teacher	Key Benefit
DistilBERT	BERT-base	40% smaller, 60% faster, 97% performance
TinyLlama (1.1B)	LLaMA-7B	Mobile-friendly, strong performance
Gemma 2B	Larger Gemma	Efficient on-device deployment
Phi-3	GPT-4 (synthetic data)	High capability in small package

In Assignment 3: You'll use Qwen-1.5B (teacher) to generate synthetic summaries, then fine-tune GPT-2 124M (student) on them.

Summary: From pretrained model to deployed assistant

We traced the full path from a pretrained LLM to a deployed assistant:

1. **Instruction Tuning (SFT)**: Teach the model to follow instructions using curated data
2. **RLHF / DPO**: Align with human preferences using pairwise comparisons
3. **Decoding**: Generate text with greedy, temperature, top-k, or top-p sampling
4. **Knowledge Distillation**: Compress large models into smaller, deployable students



Tip

These are three independent control axes that together determine real-world model behavior.
(Training, Inference, and Deployment)

Coming up next

We built the model and learned how it generates text. But **how you use it** matters enormously:

- **Prompting strategies:** Zero-shot, few-shot, and chain-of-thought
- **In-context learning:** How LLMs learn from examples in the prompt — without any gradient updates
- **Reasoning:** Can LLMs solve multi-step problems? When and why do they fail?

Today: We control the *output* (decoding algorithms)

Next: We control the *input* (prompting and reasoning)