```python
import os
import sys
from typing import Callable
import math
from dataclasses import dataclass
import torch
import torch.nn as nn
from torch.nn import functional as F
from torch.nn.functional import softmax
from einops import einsum, rearrange, repeat
from execute_util import text, link, image
from lecture_util import named_link
from references import ppo2017, grpo, qwen3, llama3
import matplotlib.pyplot as plt
from tqdm import tqdm

def main():
```

Last lecture: overview of RL from verifiable rewards (policy gradient)
This lecture: deep dive into the mechanics of policy gradient (e.g., GRPO)

```python
    rl_setup_for_language_models()
    policy_gradient()
    training_walkthrough()
```

Summary

- Reinforcement learning is the key to surpassing human abilities
- **If** you can measure it, you can optimize it
- Policy gradient framework is conceptually clear, just need baselines to reduce variance
- RL systems is much more complex than pretraining (inference workloads, manage multiple models)

Final two lectures:

- Junyang Lin (Qwen) [Yang+ 2025]
- Mike Lewis (Llama) [Grattafiori+ 2024]

```python
def rl_setup_for_language_models():
```

State s: prompt + generated response so far
Action a: generate next token

Rewards R: how good the response is; we'll focus on:

- Outcome rewards, which depend on the entire response
- Verifiable rewards, whose computation is deterministic
- Notions of discounting and bootstrapping are less applicable

Example: "... Therefore, the answer is 3 miles."

Transition probabilities T(s' | s, a): deterministic s' = s + a

- Can do planning / test-time compute (unlike in robotics)
- States are really made up (different from robotics), so a lot of flexibility

Policy π(a | s): just a language model (fine-tuned)

Rollout/episode/trajectory: s → a → ... → a → a → R
Objective: maximize expected reward E[R]
(where the expectation is taken over prompts s and response tokens a)

```python
def policy_gradient():
```

For notational simplicity, let *a* denote the entire response.

We want to maximize expected reward with respect to the policy π:

$E[R] = \int p(s)\, \pi(a \mid s)\, R(s, a)$

Obvious thing to do is to take the gradient:

$\nabla E[R] = \int p(s)\, \nabla \pi(a \mid s)\, R(s, a)$

$\nabla E[R] = \int p(s)\, \pi(a \mid s)\, \nabla \log \pi(a \mid s)\, R(s, a)$

$\nabla E[R] = E[\nabla \log \pi(a \mid s)\, R(s, a)]$

Naive policy gradient:

- Sample prompt s, sample response $a \sim \pi(a \mid s)$
- Update parameters based on $\nabla \log \pi(a \mid s)\, R(s, a)$ (same as SFT, but weighted by R(s, a))

Setting: $R(s, a) \in \{0, 1\}$ = whether response is correct or not

- Naive policy gradient only updates on correct responses
- Like SFT, but dataset changing over time as policy changes

Challenge: high noise/variance

In this setting, sparse rewards (few responses get reward 1, most get 0)

In contrast: in RLHF, reward models (learned from pairwise preferences) are more continuous

Baselines

Recall $\nabla E[R] = E[\nabla \log \pi(a \mid s)\, R(s, a)]$

$\nabla \log \pi(a \mid s)\, R(s, a)$ is an unbiased estimate of $\nabla E[R]$, but maybe there are others with lower variance...

Example: two states

- s1: a1 → reward 11, a2 → reward 9
- s2: a1 → reward 0, a2 → reward 2

Don't want s1 → a2 (reward 9) because a1 is better, want s2 → a2 (reward 2), but 9 > 2

Idea: maximize the baselined reward: $E[R - b(s)]$

This is just E[R] shifted by a constant E[b(s)] that doesn't depend on the policy π

We update based on $\nabla \log \pi(a \mid s)\, (R(s, a) - b(s))$

What b(s) should we use?

Example: two states

Assuming uniform distribution over (s, a) and $|\nabla \pi(a \mid s)| = 1$

```python
naive_variance = torch.std(torch.tensor([11., 9, 0, 2]))  # @inspect naive_variance
```

Define baseline b(s1) = 10, b(s2) = 1

```python
baseline_variance = torch.std(torch.tensor([11. - 10, 9 - 10, 0 - 1, 2 - 1]))  # @inspect baseline_variance
```

Variance reduced from 5.323 to 1.155

Optimal $b^*(s) = E[(\nabla \pi(a \mid s))^2 R \mid s] / E[(\nabla \pi(a \mid s))^2 \mid s]$ (for one-parameter models)

This is difficult to compute...

...so heuristic is to use the mean reward:

$b(s) = E[R \mid s]$

This is still hard to compute and must be estimated.

Advantage functions

This choice of b(s) has connections to advantage functions.

- V(s) = E[R | s] = expected reward from state s
- Q(s, a) = E[R | s, a] = expected reward from state s taking action a

(Note: Q and R are the same here, because we're assuming *a* has all actions and we have outcome rewards.)

Definition (advantage): A(s, a) = Q(s, a) - V(s)
Intuition: how much better is action a than expected from state s

If b(s) = E[R | s], then the baselined reward is identical to the advantage!
E[R - b(s)] = A(s, a)

In general:

- Ideal: E[∇ log π(a | s) R(s, a)]
- Estimate: ∇ log π(a | s) δ

There are multiple choices of δ, as we'll see later.

[CS224R lecture notes]

```
def training_walkthrough():
```

Group Relative Policy Optimization (GRPO) [Shao+ 2024]

- Simplification to PPO that removes the critic (value function)
- Leverages the group structure in the LM setting (multiple responses per prompt), which provides a natural baseline b(s).

Algorithm 1 Iterative Group Relative Policy Optimization

Input initial policy model $\pi_{\theta_{\text{init}}}$; reward models r_φ; task prompts $\mathcal{D}$; hyperparameters ε, β, μ

1: policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
2: **for** iteration = $1, \ldots,$ I **do**
3: reference model $\pi_{ref} \leftarrow \pi_\theta$
4: **for** step = $1, \ldots,$ M **do**
5: Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$
6: Update the old policy model $\pi_{\theta_{old}} \leftarrow \pi_\theta$
7: Sample G outputs $\{o_i\}_{i=1}^{G} \sim \pi_{\theta_{old}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
8: Compute rewards $\{r_i\}_{i=1}^{G}$ for each sampled output o_i by running r_φ
9: Compute $\hat{A}_{i,t}$ for the t-th token of o_i through group relative advantage estimation.
10: **for** GRPO iteration = $1, \ldots, \mu$ **do**
11: Update the policy model π_θ by maximizing the GRPO objective (Equation 21)
12: Update r_φ through continuous training using a replay mechanism.

Output π_θ

```
simple_task()        # Define a simple task
simple_model()       # Define a simple model
```

Let's now define the GRPO algorithm.

```
run_policy_gradient(num_epochs=1, num_steps_per_epoch=1)
```

Let's actually train some models.

```
experiments()
```

```
def simple_task():
```

Task: sorting n numbers

Prompt: n numbers

```
prompt = [1, 0, 2]
```

Response: n numbers

```
response = [0, 1, 2]
```

Reward should capture how close to sorted the response is.

Define a reward that returns the number of positions where the response matches the ground truth.

```
    reward = sort_distance_reward([3, 1, 0, 2], [0, 1, 2, 3])  # @inspect reward
    reward = sort_distance_reward([3, 1, 0, 2], [7, 2, 2, 5])  # @inspect reward  @stepover
    reward = sort_distance_reward([3, 1, 0, 2], [0, 3, 1, 2])  # @inspect reward  @stepover
```

Define an alternative reward that gives more partial credit.

```
    reward = sort_inclusion_ordering_reward([3, 1, 0, 2], [0, 1, 2, 3])  # @inspect reward
    reward = sort_inclusion_ordering_reward([3, 1, 0, 2], [7, 2, 2, 5])  # @inspect reward  @stepover
    reward = sort_inclusion_ordering_reward([3, 1, 0, 2], [0, 3, 1, 2])  # @inspect reward  @stepover
```

Note that the second reward function provides more credit to the 3rd response than the first reward function.

```
def simple_model():
```

Define a simple model that maps prompts to responses

- Assume fixed prompt and response length
- Captures positional information with separate per-position parameters
- Decode each position in the response independently (not autoregressive)

```
    model = Model(vocab_size=3, embedding_dim=10, prompt_length=3, response_length=3)
```

Start with a prompt s

```
    prompts = torch.tensor([[1, 0, 2]])  # [batch pos]
```

Generate responses a

```
    torch.manual_seed(10)
    responses = generate_responses(prompts=prompts, model=model, num_responses=5)  # [batch trial pos]  @inspect responses
```

Compute rewards R of these responses:

```
    rewards = compute_reward(prompts=prompts, responses=responses, reward_fn=sort_inclusion_ordering_reward)  # [batch trial]  @inspect rewards
```

Compute deltas δ given the rewards R (for performing the updates)

```
    deltas = compute_deltas(rewards=rewards, mode="rewards")  # [batch trial]  @inspect deltas
    deltas = compute_deltas(rewards=rewards, mode="centered_rewards")  # [batch trial]  @inspect deltas
    deltas = compute_deltas(rewards=rewards, mode="normalized_rewards")  # [batch trial]  @inspect deltas
    deltas = compute_deltas(rewards=rewards, mode="max_rewards")  # [batch trial]  @inspect deltas
```

Compute log probabilities of these responses:

```
    log_probs = compute_log_probs(prompts=prompts, responses=responses, model=model)  # [batch trial]  @inspect log_probs
```

Compute loss so that we can use to update the model parameters

```
    loss = compute_loss(log_probs=log_probs, deltas=deltas, mode="naive")  # @inspect loss

    freezing_parameters()

    old_model = Model(vocab_size=3, embedding_dim=10, prompt_length=3, response_length=3)  # Pretend this is an old checkpoint @stepover
    old_log_probs = compute_log_probs(prompts=prompts, responses=responses, model=old_model)  # @stepover
    loss = compute_loss(log_probs=log_probs, deltas=deltas, mode="unclipped", old_log_probs=old_log_probs)  # @inspect loss
    loss = compute_loss(log_probs=log_probs, deltas=deltas, mode="clipped", old_log_probs=old_log_probs)  # @inspect loss
```

Sometimes, we can use an explicit KL penalty to regularize the model.

This can be useful if you want RL a new capability into a model, but you don't want it to forget its original capabilities.

KL(p || q) = E_{x ~ p}[log(p(x)/q(x))]

KL(p || q) = E_{x ~ p}[-log(q(x)/p(x))]

KL(p || q) = E_{x ~ p}[q(x)/p(x) - log(q(x)/p(x)) - 1] because E_{x ~ p}[q(x)/p(x)] = 1

```
    kl_penalty = compute_kl_penalty(log_probs=log_probs, ref_log_probs=old_log_probs)  # @inspect kl_penalty
```

Summary:

- Generate responses
- Compute rewards R and δ (rewards, centered rewards, normalized rewards, max rewards)
- Compute log probs of responses
- Compute loss from log probs and δ (naive, unclipped, clipped)

```
def freezing_parameters():
```

Motivation: in GRPO you'll see ratios: p(a | s) / p_old(a | s)

When you're optimizing, it is important to freeze and not differentiate through p_old

```
    w = torch.tensor(2., requires_grad=True)
    p = torch.nn.Sigmoid()(w)
    p_old = torch.nn.Sigmoid()(w)
    ratio = p / p_old
    ratio.backward()
    grad = w.grad  # @inspect grad
```

Do it properly:

```
    w = torch.tensor(2., requires_grad=True)
    p = torch.nn.Sigmoid()(w)
    with torch.no_grad():  # Important: treat p_old as a constant!
        p_old = torch.nn.Sigmoid()(w)
    ratio = p / p_old
    ratio.backward()
    grad = w.grad  # @inspect grad


def compute_reward(prompts: torch.Tensor, responses: torch.Tensor, reward_fn: Callable[[list[int], list[int]], float]) -> torch.Tensor:
    """
    Args:
        prompts (int[batch pos])
        responses (int[batch trial pos])
    Returns:
        rewards (float[batch trial])
    """
    batch_size, num_responses, _ = responses.shape
    rewards = torch.empty(batch_size, num_responses, dtype=torch.float32)
    for i in range(batch_size):
        for j in range(num_responses):
            rewards[i, j] = reward_fn(prompts[i, :], responses[i, j, :])
    return rewards


def sort_distance_reward(prompt: list[int], response: list[int]) -> float:  # @inspect prompt, @inspect response
    """
    Return how close response is to ground_truth = sorted(prompt).
    In particular, compute number of positions where the response matches the ground truth.
    """
    assert len(prompt) == len(response)
    ground_truth = sorted(prompt)
    return sum(1 for x, y in zip(response, ground_truth) if x == y)
```

```python
def sort_inclusion_ordering_reward(prompt: list[int], response: list[int]) -> float:  # @inspect prompt, @inspect response
    """
    Return how close response is to ground_truth = sorted(prompt).
    """
    assert len(prompt) == len(response)

    # Give one point for each token in the prompt that shows up in the response
    inclusion_reward = sum(1 for x in prompt if x in response)  # @inspect inclusion_reward

    # Give one point for each adjacent pair in response that's sorted
    ordering_reward = sum(1 for x, y in zip(response, response[1:]) if x <= y)  # @inspect ordering_reward

    return inclusion_reward + ordering_reward


class Model(nn.Module):
    def __init__(self, vocab_size: int, embedding_dim: int, prompt_length: int, response_length: int):
        super().__init__()
        self.embedding_dim = embedding_dim
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # For each position, we have a matrix for encoding and a matrix for decoding
        self.encode_weights = nn.Parameter(torch.randn(prompt_length, embedding_dim, embedding_dim) / math.sqrt(embedding_dim))
        self.decode_weights = nn.Parameter(torch.randn(response_length, embedding_dim, embedding_dim) / math.sqrt(embedding_dim))

    def forward(self, prompts: torch.Tensor) -> torch.Tensor:
        """
        Args:
            prompts: int[batch pos]
        Returns:
            logits: float[batch pos vocab]
        """
        # Embed the prompts
        embeddings = self.embedding(prompts)   # [batch pos dim]

        # Transform using per prompt position matrix, collapse into one vector
        encoded = einsum(embeddings, self.encode_weights, "batch pos dim1, pos dim1 dim2 -> batch dim2")

        # Turn into one vector per response position
        decoded = einsum(encoded, self.decode_weights, "batch dim2, pos dim2 dim1 -> batch pos dim1")

        # Convert to logits (input and output share embeddings)
        logits = einsum(decoded, self.embedding.weight, "batch pos dim1, vocab dim1 -> batch pos vocab")

        return logits


def generate_responses(prompts: torch.Tensor, model: Model, num_responses: int) -> torch.Tensor:
    """
    Args:
        prompts (int[batch pos])
    Returns:
        generated responses: int[batch trial pos]

    Example (batch_size = 3, prompt_length = 3, num_responses = 2, response_length = 4)
    p1 p1 p1 r1 r1 r1 r1
```

```
             r2 r2 r2 r2
    p2 p2 p2 r3 r3 r3 r3
             r4 r4 r4 r4
    p3 p3 p3 r5 r5 r5 r5
             r6 r6 r6 r6
    """
    logits = model(prompts)  # [batch pos vocab]
    batch_size = prompts.shape[0]

    # Sample num_responses (independently) for each [batch pos]
    flattened_logits = rearrange(logits, "batch pos vocab -> (batch pos) vocab")
    flattened_responses = torch.multinomial(softmax(flattened_logits, dim=-1), num_samples=num_responses, replacement=True)  # [batch pos trial]
    responses = rearrange(flattened_responses, "(batch pos) trial -> batch trial pos", batch=batch_size)
    return responses


def compute_log_probs(prompts: torch.Tensor, responses: torch.Tensor, model: Model) -> torch.Tensor:
    """
    Args:
        prompts (int[batch pos])
        responses (int[batch trial pos])
    Returns:
        log_probs (float[batch trial pos]) under the model
    """
    # Compute log prob of responses under model
    logits = model(prompts)  # [batch pos vocab]
    log_probs = F.log_softmax(logits, dim=-1)  # [batch pos vocab]

    # Replicate to align with responses
    num_responses = responses.shape[1]
    log_probs = repeat(log_probs, "batch pos vocab -> batch trial pos vocab", trial=num_responses)  # [batch trial pos vocab]

    # Index into log_probs using responses
    log_probs = log_probs.gather(dim=-1, index=responses.unsqueeze(-1)).squeeze(-1)  # [batch trial pos]

    return log_probs


def compute_deltas(rewards: torch.Tensor, mode: str) -> torch.Tensor:  # @inspect rewards
    """
    Args:
        rewards (float[batch trial])
    Returns:
        deltas (float[batch trial]) which are advantage-like quantities for updating
    """
    if mode == "rewards":
        return rewards

    if mode == "centered_rewards":
        # Compute mean over all the responses (trial) for each prompt (batch)
        mean_rewards = rewards.mean(dim=-1, keepdim=True)  # @inspect mean_rewards
        centered_rewards = rewards - mean_rewards  # @inspect centered_rewards
        return centered_rewards

    if mode == "normalized_rewards":
        mean_rewards = rewards.mean(dim=-1, keepdim=True)  # @inspect mean_rewards
```

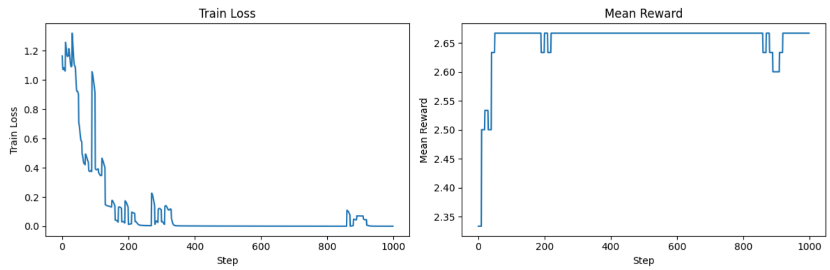
```python
        std_rewards = rewards.std(dim=-1, keepdim=True)  # @inspect std_rewards
        centered_rewards = rewards - mean_rewards  # @inspect centered_rewards
        normalized_rewards = centered_rewards / (std_rewards + 1e-5)  # @inspect normalized_rewards
        return normalized_rewards

    if mode == "max_rewards":
        # Zero out any reward that isn't the maximum for each batch
        max_rewards = rewards.max(dim=-1, keepdim=True)[0]
        max_rewards = torch.where(rewards == max_rewards, rewards, torch.zeros_like(rewards))
        return max_rewards

    raise ValueError(f"Unknown mode: {mode}")


def compute_loss(log_probs: torch.Tensor, deltas: torch.Tensor, mode: str, old_log_probs: torch.Tensor | None = None) ->
torch.Tensor:
    if mode == "naive":
        return -einsum(log_probs, deltas, "batch trial pos, batch trial -> batch trial pos").mean()

    if mode == "unclipped":
        ratios = log_probs / old_log_probs  # [batch trial]
        return -einsum(ratios, deltas, "batch trial pos, batch trial -> batch trial pos").mean()

    if mode == "clipped":
        epsilon = 0.01
        unclipped_ratios = log_probs / old_log_probs  # [batch trial]
        unclipped = einsum(unclipped_ratios, deltas, "batch trial pos, batch trial -> batch trial pos")

        clipped_ratios = torch.clamp(unclipped_ratios, min=1 - epsilon, max=1 + epsilon)
        clipped = einsum(clipped_ratios, deltas, "batch trial pos, batch trial -> batch trial pos")
        return -torch.minimum(unclipped, clipped).mean()

    raise ValueError(f"Unknown mode: {mode}")

def compute_kl_penalty(log_probs: torch.Tensor, ref_log_probs: torch.Tensor) -> torch.Tensor:
    """
    Compute an estimate of KL(model | ref_model), where the models are given by:
        log_probs [batch trial pos vocab]
        ref_log_probs [batch trial pos vocab]
    Use the estimate:
        KL(p || q) = E_p[q/p - log(q/p) - 1]
    """
    return (torch.exp(ref_log_probs - log_probs) - (ref_log_probs - log_probs) - 1).sum(dim=-1).mean()


def run_policy_gradient(num_epochs: int = 100,
                        num_steps_per_epoch: int = 10,
                        compute_ref_model_period: int = 10,
                        num_responses: int = 10,
                        deltas_mode: str = "rewards",
                        loss_mode: str = "naive",
                        kl_penalty: float = 0.0,
                        reward_fn: Callable[[list[int], list[int]], float] = sort_inclusion_ordering_reward,
                        use_cache: bool = False) -> tuple[str, str]:
    """Train a model using policy gradient.
    Return:
    - Path to the image of the learning curve.
    - Path to the log file
```

```python
    """
    torch.manual_seed(5)

    image_path = f"var/policy_gradient_{deltas_mode}_{loss_mode}.png"
    log_path = f"var/policy_gradient_{deltas_mode}_{loss_mode}.txt"

    # Already ran, just cache it
    if use_cache and os.path.exists(image_path) and os.path.exists(log_path):
        return image_path, log_path

    # Define the data
    prompts = torch.tensor([[1, 0, 2], [3, 2, 4], [1, 2, 3]])
    vocab_size = prompts.max() + 1
    prompt_length = response_length = prompts.shape[1]

    model = Model(vocab_size=vocab_size, embedding_dim=10, prompt_length=prompt_length, response_length=response_length)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    records = []
    ref_log_probs = None
    ref_model = None
    old_log_probs = None

    if use_cache:
        out = open(log_path, "w")
    else:
        out = sys.stdout

    for epoch in tqdm(range(num_epochs), desc="epoch"):
        # If using KL penalty, need to get the reference model (freeze it every few epochs)
        if kl_penalty != 0:
            if epoch % compute_ref_model_period == 0:
                ref_model = model.clone()

        # Sample responses and evaluate their rewards
        responses = generate_responses(prompts=prompts, model=model, num_responses=num_responses)  # [batch trial pos]
        rewards = compute_reward(prompts=prompts, responses=responses, reward_fn=reward_fn)  # [batch trial]
        deltas = compute_deltas(rewards=rewards, mode=deltas_mode)  # [batch trial]

        if kl_penalty != 0:  # Compute under the reference model
            with torch.no_grad():
                ref_log_probs = compute_log_probs(prompts=prompts, responses=responses, model=ref_model)  # [batch trial]

        if loss_mode != "naive":  # Compute under the current model (but freeze while we do the inner steps)
            with torch.no_grad():
                old_log_probs = compute_log_probs(prompts=prompts, responses=responses, model=model)  # [batch trial]

        # Take a number of steps given the responses
        for step in range(num_steps_per_epoch):
            log_probs = compute_log_probs(prompts=prompts, responses=responses, model=model)  # [batch trial]
            loss = compute_loss(log_probs=log_probs, deltas=deltas, mode=loss_mode, old_log_probs=old_log_probs)  # @inspect loss
            if kl_penalty != 0:
                loss += kl_penalty * compute_kl_penalty(log_probs=log_probs, ref_log_probs=ref_log_probs)

            # Print information
            print_information(epoch=epoch, step=step, loss=loss, prompts=prompts, rewards=rewards, responses=responses, log_probs=log_probs, deltas=deltas, out=out)
```

```
            global_step = epoch * num_steps_per_epoch + step
            records.append({"epoch": epoch, "step": global_step, "loss": loss.item(), "mean_reward": rewards.mean().item()})

            # Backprop and update parameters
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    if use_cache:
        out.close()

    if use_cache:
        # Plot step versus loss and reward in two subplots
        steps = [r["step"] for r in records]
        losses = [r["loss"] for r in records]
        rewards = [r["mean_reward"] for r in records]

        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

        # Loss subplot
        ax1.plot(steps, losses)
        ax1.set_xlabel("Step")
        ax1.set_ylabel("Train Loss")
        ax1.set_title("Train Loss")

        # Reward subplot
        ax2.plot(steps, rewards)
        ax2.set_xlabel("Step")
        ax2.set_ylabel("Mean Reward")
        ax2.set_title("Mean Reward")

        plt.tight_layout()
        plt.savefig(image_path)
        plt.close()

    return image_path, log_path


def print_information(epoch: int, step: int, loss: torch.Tensor, prompts: torch.Tensor, rewards: torch.Tensor, responses: torch.Tensor, log_probs: torch.Tensor, deltas: torch.Tensor, out):
    print(f"epoch = {epoch}, step = {step}, loss = {loss:.3f}, reward = {rewards.mean():.3f}", file=out)
    if epoch % 1 == 0 and step % 5 == 0:
        for batch in range(prompts.shape[0]):
            print(f"  prompt = {prompts[batch, :]}", file=out)
            for trial in range(responses.shape[1]):
                print(f"    response = {responses[batch, trial, :]}, log_probs = {tstr(log_probs[batch, trial])}, reward = {rewards[batch, trial]}, delta = {deltas[batch, trial]:.3f}", file=out)


def tstr(x: torch.Tensor) -> str:
    return "[" + ", ".join(f"{x[i]:.3f}" for i in range(x.shape[0])) + "]"


def experiments():
```

Let's start with updating based on raw rewards.

```
    image_path, log_path = run_policy_gradient(num_epochs=100, num_steps_per_epoch=10, num_responses=10, deltas_mode="rewards", loss_mode="naive", reward_fn=sort_inclusion_ordering_reward, use_cache=True)  # @stepover
```

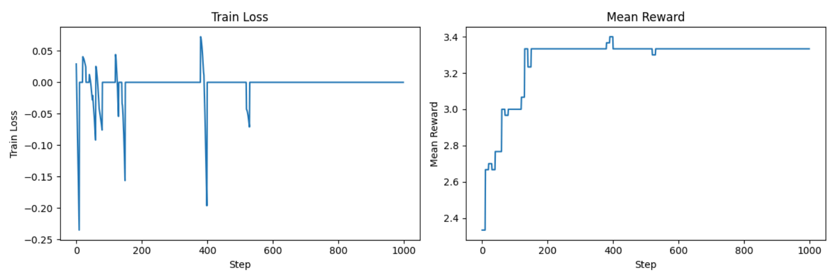


var/policy_gradient_rewards_naive.txt

Looking through the output, you'll see that by the end, we haven't really learned sorting very well (and this is still the training set).

Let's try using centered rewards.

```
image_path, log_path = run_policy_gradient(num_epochs=100, num_steps_per_epoch=10, num_responses=10, deltas_mode="centered_rewards", loss_mode="naive", reward_fn=sort_inclusion_ordering_reward, use_cache=True)  # @stepover
```



var/policy_gradient_centered_rewards_naive.txt

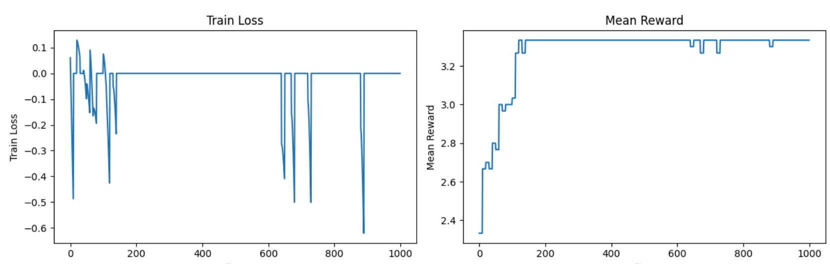
This seems to help, as:

- Suboptimal rewards get a negative gradient update, and
- If all the responses for a given prompt have the same reward, then we don't update.

Overall, this is better, but we're still getting stuck in local optima.

Finally, let's try normalizing by the standard deviation.

```
image_path, log_path = run_policy_gradient(num_epochs=100, num_steps_per_epoch=10, num_responses=10, deltas_mode="normalized_rewards", loss_mode="naive", reward_fn=sort_inclusion_ordering_reward, use_cache=True)  # @stepover
```



var/policy_gradient_normalized_rewards_naive.txt

There is not much difference here, and indeed, variants like Dr. GRPO do not perform this normalization to avoid length bias (not an issue here since all responses have the same length. [Liu+ 2025]

Overall, as you can see, reinforcement learning is not trivial, and you can easily get stuck in suboptimal states.
The hyperparameters could probably be tuned better...

```
if __name__ == "__main__":
    main()
```