CSE599-O Assignment 3: Post-Training via RL

Version 1.0

CSE 599-O Staff

Fall 2025

Acknowledgment: This is partially adapted from the Assignment 5 of Stanford CS336 (Spring 2025).

1 Assignment Overview

In this assignment, you will gain hands-on experience with post-training language models using reinforcement learning algorithms such as GRPO.

1.1 What you will implement

1. Implement the building blocks of GRPO.
2. Training loop profiling for your GRPO implementation.
3. KL divergence monitoring to track policy drift during GRPO training.
4. Colocated synchronous vs. asynchronous training strategies comparison.
5. Asynchronous training with version control using Ray actors.
6. Replay buffer implementation with trajectory aging strategies.
7. Distributed training with Ray tensor transport mechanisms (object_store, nixl).

1.2 What the code looks like

All the assignment code as well as this writeup are available on GitHub at:

`https://github.com/uw-syfi/assignment3-rl`

Please git clone the repository. If there are any updates, we will notify you and you can git pull to get the latest.

1. cse599o_alignment/*: This is where you'll write your code for this assignment. Note that there's no code in here (aside from a little starter code), so you should be able to do whatever you want from scratch.
2. cse599o_basics/*: This folder should contain your model implementation from Assignments 1 and 2.

3. tests/*.py: This contains all the tests that you must pass. These tests invoke the hooks defined in tests/adapters.py. You'll implement the adapters to connect your code to the tests. Writing more tests and/or modifying the test code can be helpful for debugging your code, but your implementation is expected to pass the original provided test suite.

4. README.md: This file contains some basic instructions on setting up your environment.

1.3 How to submit.

You will submit the following files to Gradescope:

- writeup.pdf: Answer all the written questions. Please typeset your responses.
- code.zip: Contains all the code you've written.

1.4 Grading

- **Test Cases (42% credits)**: There are 14 unit tests for `test_grpo`, each worth 3 points.
- **Analytical Questions (58% credits)**: There are 12 problems in total. Problems 1–6 will be graded via above `pytest`. For the remaining 6 problems, each is worth 10 points, except Problem 12, which is worth 8 points.

2 GRPO Building Blocks

Next, we will describe Group Relative Policy Optimization (GRPO), the variant of policy gradient that you will implement and experiment with for post-training.

2.1 GRPO Algorithm

Advantage estimation. The core idea of GRPO is to sample many outputs for each question from the policy π_θ and use them to compute a baseline. This is convenient because we avoid the need to learn a neural value function $V_\phi(s)$, which can be hard to train and is cumbersome from the systems perspective. For a question q and group outputs $\left\{o^{(i)}\right\}_{i=1}^{G} \sim \pi_\theta(\cdot \mid q)$, let $r^{(i)} = R\left(q.o^{(i)}\right)$ be the reward for the i-th output. DeepSeekMath Shao et al. (2024) and DeepSeek R1 Guo et al. (2025) compute the group-normalized reward for the i-th output as

$$A^{(i)} = \frac{r^{(i)} - \text{mean}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right)}{\text{std}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right) + \text{ advantage\_eps}} \tag{1}$$

where advantage_eps is a small constant to prevent division by zero. Note that this advantage $A^{(i)}$ is the same for each token in the response, i.e., $A_t^{(i)} = A^{(i)}, \forall t \in 1, \ldots, \left|o^{(i)}\right|$, so we drop the t subscript in the following.

High-level algorithm. Before we dive into the GRPO objective, let us first get an idea of the train loop by writing out the algorithm from Shao et al. (2024) in Algorithm 1. [1]

[1] This is a special case of DeepSeekMath's GRPO with a verified reward function, no KL term, and no iterative update of the reference and reward model.

Algorithm 1 Group Relative Policy Optimization (GRPO)

Require: initial policy model $\pi_{\theta_{\text{init}}}$; reward function R; task questions $\mathcal{D}$

policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
for step $= 1, \ldots, n_{\text{grpo\_steps}}$ **do**
  Sample a batch of questions $\mathcal{D}_b$ from $\mathcal{D}$
  Set the old policy model $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
  Sample G outputs $\{o_j^{(i)}\}_{j=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
  Compute rewards $\{r_j^{(i)}\}_{j=1}^{G}$ for each sampled output $o_j^{(i)}$ by running reward function $R(q, o^{(i)})$
  Compute $\mathcal{A}^{(i)}$ with group normalization (Eq.1)
  for train step $= 1, \ldots, n_{\text{train\_steps\_per\_rollout\_batch}}$ **do**
    Update the policy model π_θ by maximizing the GRPO-Clip objective (to be discussed, Eq.2)
  end for
end for
Output: π_θ

GRPO objective. The GRPO objective combines three ideas:

1. Off-policy policy gradient.
2. Computing advantages $A^{(i)}$ with group normalization, as in Eq. 1.
3. A clipping mechanism, as in Proximal Policy Optimization (PPO, Schulman et al. (2017)).

The purpose of clipping is to maintain stability when taking many gradient steps on a single batch of rollouts. It works by keeping the policy π_θ from straying too far from the old policy.

Let us first write out the full GRPO-Clip objective, and then we can build some intuition on what the clipping does:

$$J_{\text{GRPO-Clip}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\left\{o^{(i)}\right\}_{i=1}^{G}\sim\pi_\theta(\cdot|q)} \left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{\left|o^{(i)}\right|}\sum_{t=1}^{\left|o^{(i)}\right|} \underbrace{\min\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}A^{(i)}, \text{clip}\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}, 1-\epsilon, 1+\epsilon\right)A^{(i)}\right)}_{\text{per-token objective}}\right]. \tag{2}$$

The hyperparameter $\epsilon > 0$ controls how much the policy can change. To see this, we can rewrite the per-token objective in a more intuitive way following Achiam (2018a,b). Define the function

$$g\left(\epsilon, A^{(i)}\right) = \begin{cases} (1+\epsilon)A^{(i)} & \text{if } A^{(i)} \geq 0 \\ (1-\epsilon)A^{(i)} & \text{if } A^{(i)} < 0 \end{cases} \tag{3}$$

We can rewrite the per-token objective as

$$\text{per-token objective } = \min\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}A^{(i)}, g\left(\epsilon, A^{(i)}\right)\right)$$

We can now reason by cases. When the advantage $A^{(i)}$ is positive, the per-token objective simplifies to

$$\text{per-token objective } = \min\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}, 1+\epsilon\right) A^{(i)}$$

Since $A^{(i)} > 0$, the objective goes up if the action $o_t^{(i)}$ becomes more likely under π_θ, i.e., if $\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)$ increases. The clipping with min limits how much the objective can increase: once $\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right) > (1+\epsilon)\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)$, this per-token objective hits its maximum value of $(1+\epsilon)A^{(i)}$. So, the policy π_θ is not incentivized to go very far from the old policy $\pi_{\theta_{\text{old}}}$.

Analogously, when the advantage $A^{(i)}$ is negative, the model tries to drive down $\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)$, but is not incentivized to decrease it below $(1-\epsilon)\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)$ (refer to Achiam (2018a) for the full argument).

2.2 Implementation

Now that we have a high-level understanding of the GRPO training loop and objective, we will start implementing pieces of it.

Computing advantages (group-normalized rewards). First, we will implement the logic to compute advantages for each example in a rollout batch, i.e., the group-normalized rewards. We will consider two possible ways to obtain group-normalized rewards: the approach presented above in Eq. 1, and a recent simplified approach.

Dr. GRPO Liu et al. (2025) highlights that normalizing by std $\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right)$ rewards questions in a batch with low variation in answer correctness, which may not be desirable. They propose simply removing the normalization step, computing

$$A^{(i)} = r^{(i)} - \text{mean}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right). \tag{4}$$

Problem-1 (compute_group_normalized_rewards): Group normalization

Deliverable: Implement a method `compute_group_normalized_rewards` that calculates raw rewards for each rollout response, normalizes them within their groups, and returns both the normalized and raw rewards along with any metadata you think is useful.

The following interface is recommended (in `cse599o_alignment/grpo.py`):

```
def compute_group_normalized_rewards
```

To test your code, implement `[adapters.run_compute_group_normalized_rewards]`. Then run:

```
uv run pytest -k test_compute_group_normalized_rewards
```

and ensure your implementation passes.

Naive policy gradient loss. Next, we will implement the methods for computing "losses". As a reminder/disclaimer, these are not really losses in the canonical sense and should not be reported as

evaluation metrics. When it comes to RL, you should instead track the train and validation returns, among other metrics.

We will start with a naive policy gradient loss, which simply multiplies the advantage by the logprobability of actions (and negates). With question q, response o, and response token o_t, the naive per-token policy gradient loss is

$$-A_t \cdot \log p_\theta \left(o_t \mid q, o_{<t}\right) \tag{5}$$

Problem-2 (compute_naive_policy_gradient_loss): Naive policy gradient

Deliverable: Implement a method `compute_naive_policy_gradient_loss` that computes the per-token policy-gradient loss using raw rewards or pre-computed advantages.
The following interface is recommended (in `cse599o_alignment/grpo.py`):

```
def compute_naive_policy_gradient_loss(
    raw_rewards_or_advantages: torch.Tensor,
    policy_log_probs: torch.Tensor,
) -> torch.Tensor:
```

Implementation tips:

- Broadcast `raw_rewards_or_advantages` over the `sequence_length` dimension.

To test your code, implement `[adapters.run_compute_naive_policy_gradient_loss]`. Then run:

```
uv run pytest -k test_compute_naive_policy_gradient_loss
```

and ensure the test passes.

GRPO-Clip loss. Next, we will implement the more interesting GRPO-Clip loss.
The per-token GRPO-Clip loss is

$$-\min\left(\frac{\pi_\theta\left(o_t \mid q, o_{<t}\right)}{\pi_{\theta_{\text{old}}}\left(o_t \mid q, o_{<t}\right)} A_t, \text{clip}\left(\frac{\pi_\theta\left(o_t \mid q, o_{<t}\right)}{\pi_{\theta_{\text{old}}}\left(o_t \mid q, o_{<t}\right)}, 1-\epsilon, 1+\epsilon\right) A_t\right). \tag{6}$$

Problem-3 (compute_grpo_clip_loss): GRPO-Clip loss

Deliverable: Implement a method `compute_grpo_clip_loss` that computes the per-token GRPO-Clip loss.
The following interface is recommended (in `cse599o_alignment/grpo.py`):

```
def compute_grpo_clip_loss(
    advantages: torch.Tensor,
    policy_log_probs: torch.Tensor,
    old_log_probs: torch.Tensor,
    cliprange: float,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
```

Implementation tips:

- Broadcast `advantages` over the `sequence_length` dimension.

To test your code, implement `[adapters.run_compute_grpo_clip_loss]`. Then run:

```
uv run pytest -k test_compute_grpo_clip_loss
```

and ensure the test passes.

Policy gradient loss wrapper. We will be running ablations comparing three different versions of policy gradient:

1. **no_baseline:** Naive policy gradient loss without a baseline, i.e., advantage is just the raw rewards $A = R(q, o)$.
2. **reinforce_with_baseline:** Naive policy gradient loss but using our group-normalized rewards as the advantage. If $\bar{r}$ are the group-normalized rewards from `compute_group_normalized_rewards` (which may or may not be normalized by the group standard deviation), then $A = \bar{r}$.
3. **grpo_clip**: GRPO-Clip loss.

For convenience, we will implement a wrapper that lets us easily swap between these three policy gradient losses.

Problem-4 (compute_policy_gradient_loss): Policy-gradient wrapper

Deliverable: Implement `compute_policy_gradient_loss`, a convenience wrapper that dispatches to the correct loss routine (`no_baseline`, `reinforce_with_baseline`, or `grpo_clip`) and returns both the per-token loss and any auxiliary statistics.
The following interface is recommended (in `cse599o_alignment/grpo.py`):

```python
def compute_policy_gradient_loss(
    policy_log_probs: torch.Tensor,
    loss_type: Literal["no_baseline", "reinforce_with_baseline", "grpo_clip"],
    raw_rewards: torch.Tensor | None = None,
    advantages: torch.Tensor | None = None,
    old_log_probs: torch.Tensor | None = None,
    % cliprange: float | None = None,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
```

Implementation tips:

- Delegate to `compute_naive_policy_gradient_loss` or `compute_grpo_clip_loss`.
- Perform argument checks (see assertion pattern above).
- Aggregate any returned metadata into a single dict.

To test your code, implement `[adapters.run_compute_policy_gradient_loss]`. Then run:

```
uv run pytest -k test_compute_policy_gradient_loss
```

and verify it passes.

Masked mean. Up to this point, we have the logic needed to compute advantages, log probabilities, pertoken losses. To reduce our per-token loss tensors of shape (batch_size, sequence_length) to a vector of losses (one scalar for each example), we will compute the mean of the loss over the sequence dimension, but only over the indices corresponding to the response (i.e., the token positions for which mask $[i, j] == 1$).

We will allow specification of the dimension over which we compute the mean, and if dim is None, we will compute the mean over all masked elements. This may be useful to obtain average per-token entropies on the response tokens, clip fractions, etc.

Problem-5 (masked_mean): Masked mean

Deliverable: Implement a method `masked_mean` that averages tensor elements while respecting a boolean mask.
The following interface is recommended (in `cse599o_alignment/grpo.py`):

```python
def masked_mean(
    tensor: torch.Tensor,
    mask: torch.Tensor,
    dim: int | None = None,
) -> torch.Tensor:
```

Compute the mean of `tensor` along a given dimension, considering only those elements where `mask == 1`.
To test your code, implement `[adapters.run_masked_mean]`. Then run:

```
uv run pytest -k test_masked_mean
```

and ensure it passes.

GRPO microbatch train step. Now we are ready to implement a single microbatch train step for GRPO (recall that for a train minibatch, we iterate over many microbatches if gradient_accumulation_steps > 1). Specifically, given the raw rewards or advantages and log probs, we will compute the per-token loss, use masked_mean to aggregate to a scalar loss per example, average over the batch dimension, adjust for gradient accumulation, and backpropagate.

Problem-6 (grpo_microbatch_train_step): Microbatch train step

Deliverable: Implement a single micro-batch update for GRPO, including policy-gradient loss, averaging with a mask, and gradient scaling.
The following interface is recommended (in `cse599o_alignment/grpo.py`):

```python
def grpo_microbatch_train_step(
    policy_log_probs: torch.Tensor,
```

```
    response_mask: torch.Tensor,
    gradient_accumulation_steps: int,
    loss_type: Literal["no_baseline", "reinforce_with_baseline", "grpo_clip"],
    raw_rewards: torch.Tensor | None = None,
    advantages: torch.Tensor | None = None,
    old_log_probs: torch.Tensor | None = None,
    cliprange: float | None = None,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
```

Execute a forward-and-backward pass on a microbatch.

Implementation tips:

- Call `loss.backward()` inside this function. Make sure to divide the loss appropriately by `gradient_accumulation_steps`.

To test your code, implement `[adapters.run_grpo_microbatch_train_step]`. Then run:

```
uv run pytest -k test_grpo_microbatch_train_step
```

and confirm it passes.

3 GRPO Training Loop with Ray

Now that we have implemented the core GRPO algorithm, we turn our attention to the systems-level challenges that arise when scaling GRPO to distributed environments. This section explores training loop bottlenecks, policy drift monitoring, and various strategies for efficient off-policy training.

3.1 Training Loop

GRPO train loop. Now we will put together a complete train loop for GRPO. You can refer to the algorithm in Section 2.1 for the overall structure, using the methods we've implemented where appropriate.

On Utilizing Ray as the Framework. We will implement the training loop using **Ray** Moritz et al. (2018). Ray is an open-source, industry-standard framework for scaling AI and Python applications, including reinforcement learning workloads. It offers a unified programming model and a suite of specialized libraries for distributed data processing and model training, making it easier to build scalable and high-performance applications. A skeleton script is provided to help you get started, and you are free to customize it:

`cse599o_alignment/train_grpo_ray.py`

Comprehensive documentation for Ray can be found at `https://docs.ray.io/en/latest/index.html`. Moreover, here is an example of using Ray for reinforcement learning: `https://www.anyscale.com/blog/ray-direct-transport-rdma-support-in-ray-core`.

On reusing model from HW1/HW2. In this problem, we will reuse your language model implementation from HW1/HW2 to train with GRPO on a simpler text generation task. Instead of mathematical

reasoning, we will use a lightweight Keyword Inclusion task, which tests whether the model can learn to generate responses that include certain required keywords.

Note: For the following experiments, although we report model accuracy on the Keyword Inclusion task, our primary focus is on **system-level performance** (e.g., timing measurements) and the correctness of the GRPO workflow. Since the model from HW1/HW2 is relatively small, it is reasonable to expect that the validation accuracy will not be very high.

Problem-7 (grpo_keyword_inclusion): GRPO Training on Your Own Model

Task. For simplicity, we implement the GRPO training loop using a simple Keyword Inclusion task. In this task, the model is prompted to generate a response that includes specific keywords. For example, a prompt–response pair can be:

```
{"prompt": "Write a sentence that includes the words: apple, red.",
"answer": "The red apple fell from the tree."}
```

Use your GRPO training loop with your own LM implementation (from HW1/HW2) as the model, and train it on this Keyword Inclusion task. Use the `keyword_inclusion_reward_fn` to compute rewards for rollouts. By changing the keywords, you can generate additional prompts for training purposes.

Reward Function. Implement a new reward function `keyword_inclusion_reward_fn(response, keywords)` that returns:

$$R = \begin{cases} 1 & \text{if all required keywords appear in the model's response (case-insensitive)} \\ 0 & \text{otherwise.} \end{cases}$$

For the prompt:

```
"Write a sentence that includes the words: apple, red."
```

If the model outputs:

```
"The red apple fell from the tree."
```

then `reward = 1`, since both "apple" and "red" appear.

Profiling: Instrument your GRPO training code to measure:

- Time spent in rollout generation
- Time spent in reward computation
- Time spent in policy optimization steps
- Time spent in weight synchronization

Deliverables.

1. A script that trains your HW1/HW2 model with GRPO on the Keyword Inclusion task.

2. Measurement Results from Profiling.

3. Several example rollouts before and after training, showing improved inclusion of required keywords. And include screenshots of logs that demonstrate the training process.

3.2 KL Divergence and Policy Drift Monitoring

As the policy evolves during training, it gradually diverges from its initial state. Monitoring this divergence through KL divergence provides crucial insights into training stability.

The KL divergence between the current policy and a reference model is:

$$\mathbb{D}_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}}) = \mathbb{E}_{x \sim \pi_\theta} \left[\log \pi_\theta(x) - \log \pi_{\mathrm{ref}}(x) \right] \tag{7}$$

In practice, this is approximated using generated rollouts:

$$\widehat{\mathbb{D}}_{\mathrm{KL}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{|o^{(i)}|} \left[\log \pi_\theta(o_t^{(i)} | q, o_{<t}^{(i)}) - \log \pi_{\mathrm{ref}}(o_t^{(i)} | q, o_{<t}^{(i)}) \right] \tag{8}$$

Problem-8 (grpo_kl_monitoring): KL Divergence Drift Analysis

Description: Implement KL divergence monitoring to track policy drift during GRPO training and analyze its relationship to performance.
Augment your GRPO training loop to compute and log KL divergence between:

- Current policy π_θ and the initial checkpoint (frozen reference model)
- Current policy π_θ and the policy from the previous step $\pi_{\theta_{t-1}}$

Implementation tips:

- Load a frozen reference model at the start of training
- Compute per-token log probabilities for both models during training
- Use `torch.no_grad()` for reference model computations
- Mask out prompt tokens when computing KL over responses only

Deliverables:

- Updated training loop script incorporating the KL term
- Plots showing KL divergence over training steps
- 3–4 Sentence Analysis (including but not limited to): Discuss how the KL divergence evolves during training —does it grow linearly, exponentially, or follow another pattern? Examine whether higher KL divergence correlates with better or worse validation rewards. Identify the point (if any) at which rapid KL divergence growth may indicate potential training instability or degradation.

3.3 Off-Policy Training Strategies

The core challenge in GRPO is managing the distribution mismatch between the policy that generated rollouts ($\pi_{\theta_{\text{old}}}$) and the policy being optimized (π_θ). We explore several strategies for handling this off-policy challenge.

Problem-9 (grpo_colocated_sync): Strategy 1 - Colocated Synchronous vs Asynchronous

Description: Compare fully synchronous GRPO (one gradient step per rollout batch) against taking multiple training steps per rollout batch.
Implement both variants:
Variant A - Fully Synchronous:

- Generate rollout batch with current policy
- Take exactly one gradient step
- Repeat

Variant B - Multiple Steps per Rollout:

- Generate rollout batch with policy version v
- Store old log probabilities from version v
- Take k gradient steps using GRPO-Clip loss
- Repeat

Test with $k \in \{1, 2, 4, 8\}$ gradient steps per rollout batch.
Deliverables:

- Training scripts.
- Analyze the **system efficiency**: for example, compare wall-clock runtime and provide a profiling breakdown of time spent in rollout generation, policy optimization, and weight synchronization.
- Analyze the **training performance**: for example, include a plot of validation reward versus training steps for both synchronization variants.

Problem-10 (grpo_async_version_control): Strategy 2 - Asynchronous with Version Control

Description: Implement asynchronous GRPO where rollout generation and policy optimization run in parallel, with version control to limit staleness.
Version is a sequential counter that tracks how many training steps the policy model has completed - each time the Learner performs a gradient update, it increments its version number, allowing the Generator to know whether it has the latest model weights or is using an outdated policy for rollout generation.

Use Ray to implement separate Generator and Learner actors and implement the "one-version-behind" constraint: ensure the generator is never more than one policy version behind the learner.
Deliverables:

- Scripts for asynchronous GRPO implementation with version control
- Log and print version changes from both the Generator and Learner sides, including screenshots.
- Analyze the **system efficiency**: for example, compare wall-clock runtime between asynchronous and synchronous variants.
- Analyze the **training performance**: for example, include a plot of validation reward versus training steps.

Problem-11 (grpo_replay_buffer): Strategy 3 - Replay Buffer

Description: Implement GRPO with a replay buffer that stores scored trajectories for reuse across multiple training steps. Replay buffer is a storage container that accumulates scored trajectories (rollouts with computed rewards and advantages) so the Learner can sample from multiple past experiences for training, rather than only using the most recent rollout, which improves sample efficiency and training stability. Implement a replay buffer with trajectory aging (e.g., FIFO with fixed capacity) and sampling strategy (e.g., recency-based deterministic sampling).
Deliverables:

- Implement a **replay buffer** with your designed aging and sampling strategies.
- Describe your aging and sampling strategies in your report.
- Analyze the **training performance**: for example, compare the replay buffer variant against the standard GRPO baseline.

3.4 Speed Up Weights Synchronization via Ray Direct Transport

Ray Direct Transport enables fast, zero-copy GPU data transfers in Ray through RDMA-backed communication. Refer to the following documentation to learn how to use this API for building high-performance distributed systems, such as reinforcement learning pipelines for LLMs:

`https://www.anyscale.com/blog/ray-direct-transport-rdma-support-in-ray-core`

Problem-12 (grpo_distributed_rdt): Fast Transfers via Ray Direct Transport

Description: Extend your distributed GRPO implementation to leverage **Ray Direct Transport (RDT)** for efficient weight synchronization between distributed actors. Building on your `train_grpo_ray.py`, experiment with different tensor transport backends and observe their impact on performance.
To enable RDT, add annotations as shown below. Without these annotations, Ray will default to using the `object_store` for tensor transfers.

```python
@ray.remote(num_gpus=1)
class Learner:
    @ray.method(tensor_transport="nixl")
    def get_weights(self):
        return self.model.state_dict()
```

Deliverables:

- Describe your modifications required to enable RDT (e.g., where and how you added the annotations).
- Measure and compare the weight transfer latency between the default `object_store` and RDT.

Note: Please use NIXL as the backend for RDT, since ray.get() currently supports only OBJECT_STORE and NIXL tensor transport.

References

J. Achiam. Simplified ppo-clip objective. URl: https://drive. google. com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/view, 2018b, 22, 2018a.

J. Achiam. Spinning up in deep reinforcement learning.(2018). URL https://github. com/openai/spinningup, 2018b.

D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025.

Z. Liu, C. Chen, W. Li, P. Qi, T. Pang, C. Du, W. S. Lee, and M. Lin. Understanding r1-zero-like training: A critical perspective. arXiv preprint arXiv:2503.20783, 2025.

P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In 13th USENIX symposium on operating systems design and implementation (OSDI 18), pages 561–577, 2018.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024.