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. Post-training of your HW1 model, using GRPO and a simple deterministic reward function.
2. Exploration of colocated vs. disaggregated and synchronous vs. asynchronous training strategies.
3. Distributed RL using Ray.

In this assignment, after implementing the core GRPO components, we will reuse your language model implementation from HW1/HW2 and apply GRPO to a simple Keyword Inclusion task, designed to evaluate whether the model can learn to generate responses that include specified keywords.

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/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 (40% credits)**: There are 10 unit tests for `test_grpo`, each worth 4 points.
- **Analytical Questions (60% credits)**: There are 10 problems in total. Problems 1–4 will be graded via above `pytest`. For the remaining 6 problems, each is worth 10 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.

Note: If you are not familiar with the basic concepts of policy gradient methods, we recommend reading the primer in Appendix A. In addition to the theoretical discussion, the following blog provides an excellent end-to-end example of reinforcement learning in practice:

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

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 $V_\phi(s)$ used in algorithms such as PPO (Schulman et al. (2017)), 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.

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

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]

GRPO objective. The GRPO objective combines three ideas:

1. 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

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

[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.

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 of Building Blocks

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.

Note on the reward function: For this test, we use the reward function `r1_zero_reward_fn` defined in `cse599o_alignment/drgrpo_grader.py`. This function computes rewards based on both the response format (e.g., the inclusion of `"</think> <answer>"`) and answer correctness. For this test, the responses and ground-truth answers are generated by the testing framework —you do **not** need to handle them. The correctness evaluation logic is also implemented. You just need to examine `cse599o_alignment/drgrpo_grader.py` and `adapter.py` to understand the input and output parameter types, as these files provide a detailed explanation for the reward function' s expected inputs.

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

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

GRPO-Clip loss. Next, we will implement the more interesting GRPO-Clip loss.

The per-token GRPO-Clip loss is

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

Problem-2 (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
```

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.

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-3 (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`):

```
def masked_mean
```

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-4 (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. Your task is to implement the following function in `cse599o_alignment/grpo.py`:

```
def grpo_microbatch_train_step
```

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. The documentation for Ray core concepts can be found at

`https://docs.ray.io/en/latest/ray-core/walkthrough.html#core-walkthrough`

Moreover, here is an example of using Ray for reinforcement learning:

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

Two skeleton scripts are provided to help you get started, and you are free to customize them as needed:

Colocated and synchronized (for Problems 7/8/9): The training and inference components share the same GPU resources and operate in a tightly coupled manner. Each training iteration waits for inference to finish generating rollouts, and inference waits until the model weights are fully updated before starting the next round. The RL algorithm can be on-policy (one training step per inference round) or off-policy (multiple training steps per inference round).

`cse599o_alignment/train_grpo_ray_colocated.py`

Disaggregated and asynchronous (for Problems 10/11/12): The training and inference components are decoupled and run on separate GPUs, enabling overlap between rollout generation and policy updates. Typically, this setup is only used for off-policy algorithms, allowing training and inference to execute asynchronously and in parallel.

`cse599o_alignment/train_grpo_ray_disaggregated.py`

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 story that includes the word: apple",
"answer": "The red apple fell from the tree."}
```

Reward Function. We need a new reward function `keyword_inclusion_reward_fn(response, keywords)`. You may refer to `r1_zero_reward_fn` (in drgrpo_grader.py) for reference, but the grading

logic here can be much simpler. This function should return:

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

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.

Samples for Training Prompts: To generate prompts for training, use the file `prompts/keywords.txt`, which contains nouns extracted from the TinyStoriesV2 dataset. These keywords can be inserted into prompt templates such as "Generate a story that includes KEYWORD" to create diverse training examples.

3.1.1 Implementation Hints for Ray

We will be using a single controller design, where a driver implements the main training loop by orchestrating method calls to one or more actors. The actor definitions have been provided; your task is to implement the empty methods and the driver code using the GRPO building blocks. You should not need to add any other methods to the actor definitions (but you're welcome to add helper methods).

For the colocated setups, we will create one Ray actor (a ColocatedWorker) that acts as both a Generator and a Learner. Here are some implementation hints for each class:

Generator: The Generator class acts as the text generation engine for GRPO training. Technically, the Generator can share weights directly with the Learner, but we will create a second copy of the model just for generation to more closely match how existing LLM systems are built.

- Initialization: Load a pre-trained TransformerLM model from a checkpoint.
- Generation: In generate_responses(), produce G responses per prompt via autoregressive decoding—tokenize the prompt and generate tokens step by step based on model outputs.
- Sampling: Apply sampling and record log probabilities for each generated token.
- Weights update: Update the model weights.

Learner: The Learner holds the training copy of the weights and handles GRPO policy optimization. Key steps:

- Initialize TransformerLM and load model/optimizer checkpoints;
- In update_policy(), tokenize each prompt-response pair into; input/label tensors for next-token prediction;
- Compute policy log probabilities and use response_mask to train only on generated tokens;

- Call grpo_microbatch_train_step() with advantages to get clipped policy gradient loss, then apply gradient clipping and optimizer step.

Driver: Implement the driver loop in `run_training`. For the colocated, on-policy variant, the driver should:

- Call one round of generation on the Generator.
- Pass the outputs of generation to the Learner and perform one training step.
- Synchronize the updated Learner weights back to the Generator.

GPU Usage: Ray code should use "@ray.remote(num_gpus=1)" for actor definitions. Ray will handle scheduling across the GPUs visible to the driver through CUDA_VISIBLE_DEVICES. Then just calling `.to( "cuda" )` from the actor code should place tensors on the device assigned by Ray.

3.1.2 Hyperparameters

We start with the following hyperparameters. You are encouraged to further tune these hyperparameters and include additional fine-grained control parameters if needed.

```
n_grpo_steps: int = 100
learning_rate: float = 5e-4
group_size: int = 4
sampling_temperature: float = 0.8
sampling_max_tokens: int = 60
advantage_eps: float = 1e-8
loss_type: str = "grpo_clip"
use_std_normalization: bool = True
```

Problem-5 (grpo_keyword_inclusion): GRPO Training on Your Own Model

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. We utilize prompts that include a single keyword.

Profiling: Instrument your GRPO training code to measure: Time spent in rollout generation, policy optimization steps, and weight synchronization.

Deliverables.

1. Profiling Measurement Results: covering key stages of generation, learning, and weight synchronization. Explain which stage takes the longest time and why that is expected.
2. Several example rollouts during training, illustrating whether or not the model improves in including the required keywords.

Hints: Implement the TODOs in train_grpo_ray_colocated.py to complete the actor definitions using the GRPO building blocks from earlier problems. Use a synchronous, on-policy training loop: for each inference batch (one round of rollouts + scoring), wait for completion, then run exactly one optimizer step on that batch before starting the next round.

Note: For the training-loop 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.

Unit Tests: Starting from this problem, grading will be based on your PDF writeup submitted to Gradescope, including the required deliverables, results, and descriptions —instead of unit tests. However, we still strongly recommend adding your own unit tests to aid development and debugging. For this problem, we also provide a few example unit tests that you can customize.

```
uv run pytest -k test_train_grpo_colocated
```

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-6 (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. Extend `ColocatedWorker` in `train_grpo_ray_colocated.py` so that the Generator, Learner, and ReferenceModel all reside within a single Ray actor on the same GPU.
- 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:

- Plots showing KL divergence over training steps

- 1–2 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.

3.3 Off-Policy Training Strategies

A 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). Allowing these to diverge can often improve system efficiency but can also lead to training instability. We explore several strategies that trade off between these.

Problem-7 (grpo_colocated_sync): Strategy 1 - Colocated On-policy vs Off-policy

Description: Compare on-policy GRPO (one gradient step per rollout batch) against taking multiple training steps per rollout batch. "Colocated" here specifically refers to running all actors on a single GPU.

Implement both variants:

Variant A —Fully Synchronous (Implemented in Problem 7):

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

Variant B —Multiple Steps per Rollout (New in this problem):

- Generate k rollout batches 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.

Your implementation should only need to modify the driver loop.

Deliverables:

- Analyze the **system efficiency**:
  - **Colocated Synchronous Profiling**: Measure wall-clock runtime and profile key stages—rollout generation, policy optimization, and weight synchronization overhead. Visualize the results as a time-series plot
  - **Asynchronous Throughput Comparison**: Using the same number of processing samples, measure whether asynchronous execution can improve throughput compared to synchronous training. Calculate:
    - Samples processed per second for both modes
    - Speedup ratio (sync_time / async_time). Explain your result.

Problem-8 (grpo_async_version_control): Strategy 2 - Disaggregated, off-policy

Description: Implement disaggregated off-policy GRPO where rollout generation and policy optimization are run in parallel on different GPUs. Rollout generation will execute on an older version of the model. We'll limit staleness of rollouts so that generation is at most one version behind training. Use Ray to implement separate Generator and Learner actors, each using 1 GPU (2 GPUs total). You should be able to directly reuse the Generator and Learner classes implemented in the colocated setup. The changes needed will be in the driver loop (`run\_once`). The driver loop should call Generator and Learner methods in order so that it implements 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
- Analyze the **system efficiency**: Compare wall-clock runtime and total GPU hours (wall-clock time × number of GPUs) between colocated on-policy, colocated off-policy, and disaggregated off-policy and explain the reason behind any differences.

Note: You can start with `train_grpo_ray_disaggregated.py` and reuse the `Learner` and `Generator` definitions from earlier problems, rather than rewriting them from scratch.

3.4 Speed Up Weights Synchronization via Ray Direct Transport

For disaggregated and asynchronous RL, weight synchronization can be of the primary bottlenecks. Next, we investigate methods to accelerate weight transfer. Ray Direct Transport (RDT) enables fast, zero-copy GPU data transfers in Ray through RDMA-backed communication. Refer to the following blog and documentation page to understand how RDT works and to learn how to use this API:

`https://www.anyscale.com/blog/ray-direct-transport-rdma-support-in-ray-core`
`https://docs.ray.io/en/latest/ray-core/direct-transport.html`

Problem-9 (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_disaggregated.py`, implement weights transfer using NCCL and observe the impact on performance.

Deliverables:

- Measure and compare the weight transfer latency with RDT enabled and disabled.

Problem-10 (optimization_idea): Open-Ended Question

Deliverables: Propose and justify one optimization you would add next based on the results you have obtained so far. Describe both the high-level idea and how you would design and implement it in your current system.

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.

T. Degris, M. White, and R. S. Sutton. Off-policy actor-critic, 2013. URL https://arxiv.org/abs/1205.4839.

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.

A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al. Openai o1 system card. arXiv preprint arXiv:2412.16720, 2024.

N. Lambert. Reinforcement learning from human feedback. arXiv preprint arXiv:2504.12501, 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.

S. M. Ross. Simulation. academic press, 2022.

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.

K. Team, A. Du, B. Gao, B. Xing, C. Jiang, C. Chen, C. Li, C. Xiao, C. Du, C. Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. arXiv preprint arXiv:2501.12599, 2025.

A Primer on Policy Gradients

An exciting new finding in language model research is that performing RL against verified rewards with strong base models can lead to significant improvements in their reasoning capabilities and performance Jaech et al. (2024); Guo et al. (2025). The strongest such open reasoning models, such as DeepSeek R1 and Kimi k1.5 Team et al. (2025), were trained using policy gradients, a powerful reinforcement learning algorithm that can optimize arbitrary reward functions.

We provide a brief introduction to policy gradients for RL on language models below. Our presentation is based closely on a couple great resources which walk through these concepts in more depth: OpenAI's Spinning Up in Deep RL Achiam (2018b) and Nathan Lambert's Reinforcement Learning from Human Feedback (RLHF) Book Lambert (2025).

A.1 Language Models as Policies

A causal language model (LM) with parameters θ defines a probability distribution over the next token $a_t \in \mathcal{V}$ given the current text prefix s_t (the state/observation). In the context of RL, we think of the next token a_t as an action and the current text prefix s_t as the state. Hence, the LM is a categorical stochastic policy

$$a_t \sim \pi_\theta(\cdot \mid s_t), \quad \pi_\theta(a_t \mid s_t) = [\text{softmax}(f_\theta(s_t))]_{a_t}. \tag{3}$$

Two primitive operations will be needed in optimizing the policy with policy gradients:

1. Sampling from the policy: drawing an action a_t from the categorical distribution above;
2. Scoring the log-likelihood of an action: evaluating $\log \pi_\theta(a_t \mid s_t)$.

Generally, when doing RL with LLMs, s_t is the partial completion/solution produced so far, and each a_t is the next token of the solution; the episode ends when an end-of-text token is emitted, like <|end_of_text|>, or </answer> in the case of our r1_zero prompt.

A.2 Trajectories

A (finite-horizon) trajectory is the interleaved sequence of states and actions experienced by an agent:

$$\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T) \tag{4}$$

where T is the length of the trajectory, i.e., a_T is an end-of-text token or we have reached a maximum generation budget in tokens.

The initial state is drawn from the start distribution, $s_0 \sim \rho_0(s_0)$; in the case of RL with LLMs, $\rho_0(s_0)$ is a distribution over formatted prompts. In general settings, state transitions follow some environment dynamics $s_{t+1} \sim P(\cdot \mid s_t, a_t)$. In RL with LLMs, the environment is deterministic: the next state is the old prefix concatenated with the emitted token, $s_{t+1} = s_t \| a_t$. Trajectories are also called episodes or rollouts; we will use these terms interchangeably.

A.3 Rewards and Return

A scalar reward $r_t = R(s_t, a_t)$ judges the immediate quality of the action taken at state s_t. For RL on verified domains, it is standard to assign zero reward to intermediate steps and a verified reward to the terminal action

$$r_T = R(s_T, a_T) := \begin{cases} 1 & \text{if the trajectory } s_T \| a_T \text{ matches the ground-truth according to our reward function} \\ 0 & \text{otherwise.} \end{cases}$$

The return $R(\tau)$ aggregates rewards along the trajectory. Two common choices are finite-horizon undiscounted returns

$$R(\tau) := \sum_{t=0}^{T} r_t, \tag{5}$$

and infinite-horizon discounted returns

$$R(\tau) := \sum_{t=0}^{\infty} \gamma^t r_t, \quad 0 < \gamma < 1. \tag{6}$$

In our case, we will use the undiscounted formulation since episodes have a natural termination point (end-of-text or max generation length).

The objective of the agent is to maximize the expected return

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{7}$$

leading to the optimization problem

$$\theta^* = \arg\max_\theta J(\theta). \tag{8}$$

A.4 Vanilla Policy Gradient

Next, let us attempt to learn policy parameters θ with gradient ascent on the expected return:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J\left(\theta_k\right). \tag{9}$$

The core identity that we will use to do this is the REINFORCE policy gradient, shown below.

$$\nabla_\theta J\left(\pi_\theta\right) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) R(\tau)\right]. \tag{10}$$

Deriving the policy gradient. How did we get this equation? For completeness, we will give a derivation of this identity below. We will make use of a few identities.

1. The probability of a trajectory is given by

$$P(\tau \mid \theta) = \rho_0\left(s_0\right) \prod_{t=0}^{T} P\left(s_{t+1} \mid s_t, a_t\right) \pi_\theta\left(a_t \mid s_t\right). \tag{11}$$

Therefore, the log-probability of a trajectory is:

$$\log P(\tau \mid \theta) = \log \rho_0\left(s_0\right) + \sum_{t=0}^{T}\left[\log P\left(s_{t+1} \mid s_t, a_t\right) + \log \pi_\theta\left(a_t \mid s_t\right)\right]. \tag{12}$$

2. The log-derivative trick:

$$\nabla_\theta P = P \nabla_\theta \log P. \tag{13}$$

3. The environment terms are constant in θ.$\rho_0, P(\cdot \mid \cdot)$ and $R(\tau)$ do not depend on the policy parameters, so

$$\nabla_\theta \rho_0 = \nabla_\theta P = \nabla_\theta R(\tau) = 0. \tag{14}$$

Applying the facts above:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] && (15)\\
&= \nabla_\theta \sum_\tau P(\tau \mid \theta) R(\tau) && (16)\\
&= \sum_\tau \nabla_\theta P(\tau \mid \theta) R(\tau) && (17)\\
&= \sum_\tau P(\tau \mid \theta) \nabla_\theta \log P(\tau \mid \theta) R(\tau) && (18)\\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log P(\tau \mid \theta) R(\tau)\right], \quad \text{(Log-derivative trick)} && (19)
\end{aligned}$$

and therefore, plugging in the log-probability of a trajectory and using the fact that the environment terms are constant in θ, we get the vanilla or REINFORCE policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) R(\tau)\right]. \tag{20}$$

Intuitively, this gradient will increase the log probability of every action in a trajectory that has high return, and decrease them otherwise.

Sample estimate of the gradient. Given a batch of N rollouts $\mathcal{D} = \left\{\tau^{(i)}\right\}_{i=1}^{N}$ collected by sampling a starting state $s_0^{(i)} \sim \rho_0(s_0)$ and then running the policy π_θ in the environment, we form an unbiased estimator of the gradient as

$$\widehat{g} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) R\left(\tau^{(i)}\right). \tag{21}$$

This vector is used in the gradient-ascent update $\theta \leftarrow \theta + \alpha\widehat{g}$.

A.5 Policy Gradient Baselines

The main issue with vanilla policy gradient is the high variance of the gradient estimate. A common technique to mitigate this is to subtract from the reward a baseline function b that depends only on the state. This is a type of control variate Ross (2022): the idea is to decrease the variance of the estimator by subtracting a term that is correlated with it, without introducing bias.

Let us define the baselined policy gradient as:

$$B = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\left(R(\tau) - b(s_t)\right)\right]. \tag{22}$$

As an example, a reasonable baseline is the on-policy value function $V^\pi(s) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau) \mid s_t = s]$, i.e., the expected return if we start at $s_t = s$ and follow the policy π_θ from there. Then, the quantity $(R(\tau) - V^\pi(s_t))$ is, intuitively, how much better the realized trajectory is than expected.

As long as the baseline depends only on the state, the baselined policy gradient is unbiased. We can see this by rewriting the baselined policy gradient as

$$B = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) R(\tau) \right] - \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) b \left(s_t \right) \right]. \tag{23}$$

Focusing on the baseline term, we see that

$$\mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) b \left(s_t \right) \right] = \sum_{t=0}^{T} \mathbb{E}_{s_t} \left[b \left(s_t \right) \mathbb{E}_{a_t \sim \pi_\theta(\cdot \mid s_t)} \left[\nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) \right] \right] \tag{24}$$

In general, the expectation of the score function is zero: $\mathbb{E}_{x \sim P_\theta} \left[\nabla_\theta \log P_\theta(x) \right] = 0$. Therefore, the expression in Eq. 24 is zero and

$$B = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) R(\tau) \right] - 0 = \nabla_\theta J \left(\pi_\theta \right) \tag{25}$$

so we conclude that the baselined policy gradient is unbiased. We will later run an experiment to see whether baselining improves downstream performance.

A note on policy gradient "losses." When we implement policy gradient methods in a framework like PyTorch, we will define a so-called policy gradient loss pg_loss such that calling pg_loss.backward() will populate the gradient buffers of our model parameters with our approximate policy gradient $\widehat{g}$. In math, it can be stated as

$$\text{pg\_loss} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) \left(R \left(\tau^{(i)} \right) - b \left(s_t^{(i)} \right) \right). \tag{26}$$

pg_loss is not a loss in the canonical sense-it's not meaningful to report pg_loss on the train or validation set as an evaluation metric, and a good validation pg_loss doesn't indicate that our model is generalizing well. The pg_loss is really just some scalar such that when we call pg_loss.backward(), the gradients we obtain through backprop are the approximate policy gradient $\widehat{g}$.

When doing RL, you should always log and report train and validation rewards. These are the "meaningful" evaluation metrics and what we are attempting to optimize with policy gradient methods.

A.6 Off-Policy Policy Gradient

REINFORCE is an on-policy algorithm: the training data is collected by the same policy that we are optimizing. To see this, let us write out the REINFORCE algorithm:

1. Sample a batch of rollouts $\left\{ \tau^{(i)} \right\}_{i=1}^{N}$ from the current policy π_θ.
2. Approximate the policy gradient as $\nabla_\theta J \left(\pi_\theta \right) \approx \hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) R \left(\tau^{(i)} \right)$.
3. Update the policy parameters using the computed gradient: $\theta \leftarrow \theta + \alpha \widehat{g}$.

We need to do a lot of inference to sample a new batch of rollouts, only to take just one gradient step. The behavior of an LM generally cannot change significantly in a single step, so this on-policy approach is highly inefficient.

Off-policy policy gradient. In off-policy learning, we instead have rollouts sampled from some policy other than the one we are optimizing. Off-policy variants of popular policy gradient algorithms like PPO and GRPO use rollouts from a previous version of the policy $\pi_{\theta_{\text{old}}}$ to optimize the current policy π_θ. The off-policy policy gradient estimate is

$$\widehat{g}_{\text{off-policy}} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\frac{\pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(a_t^{(i)} \mid s_t^{(i)}\right)}\nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) R\left(\tau^{(i)}\right). \tag{27}$$

This looks like an importance sampled version of the vanilla policy gradient, with reweighting terms $\frac{\pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)}{\pi_{\theta\text{ old}}\left(a_t^{(i)}|s_t^{(i)}\right)}$. Indeed, Eq. 27 can be derived by importance sampling and applying an approximation that is reasonable as long as π_θ and $\pi_{\theta_{\text{old}}}$ are not too different: see Degris et al. (2013) for more on this.