

Assignment 4: LLM Reinforcement Learning

Due April 1, 11:59 PM

1 Introduction

The goal of this assignment is to learn how modern LLM reinforcement-learning post-training works in practice. You will implement the key algorithmic pieces needed to fine-tune a base language model with outcome rewards, and then run a small set of controlled experiments comparing *GR-REINFORCE* and *GRPO*.

The starter code for the assignment is in:

https://github.com/berkeleydeeprlcourse/homework_spring2026/tree/main/hw4

All experiments in this assignment are configured to run on Modal using an H100 GPU. Your Modal class credits should comfortably suffice for the homework. With a correct implementation, performance should improve stably enough that you do *not* need to run multiple random seeds. We recommend that you spend less than a third of your Modal credits (amounting to roughly 40 H100 hours) on this assignment. The assignment can in fact be completed with about 8 H100 hours. We recommend you start early on this assignment.

What you will do. You will:

- implement log-probability, minibatching, advantage-normalization, GR-REINFORCE, and GRPO code,
- run four required training jobs,
- analyze WandB metrics and model generations,
- perform a small GRPO hyperparameter study on the fast `format_copy` task,
- submit a short report and your run artifacts.

Contents

1	Introduction	1
2	Theory	2
2.1	Setup and Notation	3
2.2	Group-Relative Advantages	3
2.3	Approximate KL Regularization	3
2.4	GR-REINFORCE Objective	4
2.5	GRPO Objective	5

2.6	On-Policy GR-REINFORCE vs. Sample Reuse in GRPO	5
3	Modal Setup	6
3.1	One-Time Setup	6
3.2	How Detached Modal Runs Work	6
4	Codebase Overview	6
4.1	Recommended Implementation Order	7
5	Tasks and Rewards	7
5.1	Task 1: <code>format_copy</code>	7
5.2	Task 2: <code>math_hard</code>	7
6	Implementation Checklist	7
7	Required Runs	8
7.1	Required Run 1: Format Copy + GRPO	8
7.2	Required Run 2: Format Copy + GR-REINFORCE	9
7.3	Required Run 3: Math Hard + GR-REINFORCE	9
7.4	Required Run 4: Math Hard + GRPO	10
7.5	Why the Math Commands Are Configured This Way	10
8	Expected Behavior and WandB Diagnostics	11
8.1	The Main Metrics to Watch	11
8.2	What Good Runs Look Like	11
9	GRPO Hyperparameter Study on <code>format_copy</code>	12
10	What to Put in Your Report	13
11	What to Submit	13
11.1	Code/Run Submission	13
11.2	Report Submission	15
2	Theory	

2.1 Setup and Notation

Let x denote a prompt and let $y = (y_1, \dots, y_T)$ denote a sampled completion. The trainable policy is π_θ , the frozen reference policy is π_{ref} , and the scalar reward for a sampled completion is $r(x, y)$. In this assignment, rewards are *verifiable*: they are computed automatically from the final completion text.

For each prompt x_i , we sample a *group* of G completions $y_{i,1}, \dots, y_{i,G}$. Group-based normalization is important in both algorithms in this homework, including the GR-REINFORCE baseline used here.

Naming note. Throughout this document we call the provided `reinforce` implementation *GR-REINFORCE* to emphasize that it uses group-relative advantages and KL regularization. The code and CLI flag are still named `reinforce`; only the writeup uses the more explicit name.

2.2 Group-Relative Advantages

For one prompt x_i , let the rewards of its G sampled completions be $r_{i,1}, \dots, r_{i,G}$. GRPO-style group normalization defines

$$A_{i,j} = \frac{r_{i,j} - \mu_i}{\sigma_i + \varepsilon}, \quad \mu_i = \frac{1}{G} \sum_{j=1}^G r_{i,j}, \quad \sigma_i = \sqrt{\frac{1}{G} \sum_{j=1}^G (r_{i,j} - \mu_i)^2}.$$

This produces one scalar advantage per sampled completion. The advantage is positive when a completion outperforms the other samples in its group and negative when it underperforms.

2.3 Approximate KL Regularization

Both algorithms regularize the trainable policy toward a frozen reference policy. At a fixed decoding context, let a be a sampled next token and define

$$\Delta(a) = \log \pi_{\text{ref}}(a \mid s) - \log \pi_\theta(a \mid s).$$

The exact KL divergence at that context is

$$\text{KL}(\pi_\theta(\cdot \mid s) \parallel \pi_{\text{ref}}(\cdot \mid s)) = \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s)} [\log \pi_\theta(a \mid s) - \log \pi_{\text{ref}}(a \mid s)].$$

In code we do not compute this exact full-vocabulary KL. Instead, we use the sampled-token estimator

$$\hat{k}(a) = e^{\Delta(a)} - \Delta(a) - 1.$$

Why does this estimate the KL? Because when $a \sim \pi_\theta(\cdot \mid s)$,

$$\mathbb{E} \left[e^{\Delta(a)} \right] = \sum_a \pi_\theta(a \mid s) \frac{\pi_{\text{ref}}(a \mid s)}{\pi_\theta(a \mid s)} = \sum_a \pi_{\text{ref}}(a \mid s) = 1.$$

Therefore,

$$\begin{aligned}
\mathbb{E}[\widehat{k}(a)] &= \mathbb{E}[e^{\Delta(a)} - \Delta(a) - 1] \\
&= -\mathbb{E}[\Delta(a)] \\
&= \mathbb{E}[\log \pi_{\theta}(a \mid s) - \log \pi_{\text{ref}}(a \mid s)] \\
&= \text{KL}(\pi_{\theta}(\cdot \mid s) \parallel \pi_{\text{ref}}(\cdot \mid s)).
\end{aligned}$$

So $\widehat{k}(a)$ is an unbiased sampled-token estimator of the KL.

Why use this estimator rather than something simpler such as just $-\Delta(a)$? Indeed, $-\Delta(a)$ is also unbiased. But a single sample of $-\Delta(a)$ can be negative even though KL itself is always nonnegative. For example, if the current policy already assigns *less* probability than the reference to the sampled token, then $\Delta(a) > 0$ and therefore $-\Delta(a) < 0$. So if we used $-\Delta(a)$ directly as a per-token penalty, some sampled tokens would contribute a negative term that behaves more like a reward than a penalty. That is not a very natural local surrogate for a divergence, since the role of the KL term is to discourage deviations from the reference in *either* direction.

The estimator $\widehat{k}(a) = e^{\Delta(a)} - \Delta(a) - 1$ fixes this while keeping the same expectation. Let

$$x = e^{\Delta(a)} = \frac{\pi_{\text{ref}}(a \mid s)}{\pi_{\theta}(a \mid s)}.$$

Then

$$\widehat{k}(a) = x - \log x - 1.$$

For every $x > 0$,

$$x - \log x - 1 \geq 0,$$

with equality only at $x = 1$. So each sampled token contributes a nonnegative quantity that is exactly zero when the policy and reference agree on that token's probability ratio, and becomes larger as that ratio moves away from 1. In other words, $\widehat{k}(a)$ behaves much more like a true per-token divergence signal, while $-\Delta(a)$ is only a signed unbiased sample whose individual values can easily flip negative.

In the assignment, this estimator is averaged across sampled completion tokens. So it is a sampled, masked approximation to the token-level KL, not an exact full-vocabulary computation.

2.4 GR-REINFORCE Objective

The GR-REINFORCE baseline in this assignment uses the group-normalized advantage A_i defined above. For one sampled completion $y_i = (y_{i,1}, \dots, y_{i,T_i})$, define the sequence-average completion log-probability

$$\bar{\ell}_i(\theta) = \frac{1}{T_i} \sum_{t=1}^{T_i} \log \pi_{\theta}(y_{i,t} \mid x_i, y_{i,<t}).$$

The policy-gradient term is then

$$L_{\text{pg}}^{\text{GR-REINFORCE}}(\theta) = -\frac{1}{N} \sum_{i=1}^N A_i \bar{\ell}_i(\theta).$$

This objective is dubbed REINFORCE because it uses the on-policy policy gradient (not any off-policy variant). The full loss adds KL regularization:

$$L^{\text{GR-REINFORCE}}(\theta) = L_{\text{pg}}^{\text{GR-REINFORCE}}(\theta) + \beta \widehat{\text{KL}}(\pi_{\theta} \| \pi_{\text{ref}}).$$

Averaging the token log-probabilities within each completion avoids giving longer completions more weight purely because they contain more tokens.

2.5 GRPO Objective

GRPO reuses a sampled rollout for multiple policy updates with PPO-style clipping. Let π_{old} denote the policy that generated the rollout. For token t of completion i , define the importance ratio

$$\rho_{i,t}(\theta) = \frac{\pi_{\theta}(y_{i,t} \mid x_i, y_{i,<t})}{\pi_{\text{old}}(y_{i,t} \mid x_i, y_{i,<t})}.$$

The clipped token objective is

$$\min\left(\rho_{i,t}(\theta)A_i, \text{clip}(\rho_{i,t}(\theta), 1 - \epsilon, 1 + \epsilon)A_i\right).$$

In this homework, the per-token surrogate is averaged across completion tokens for each sampled completion and then averaged across the minibatch:

$$L_{\text{pg}}^{\text{grpo}}(\theta) = -\frac{1}{N} \sum_{i=1}^N \frac{1}{T_i} \sum_{t=1}^{T_i} \min\left(\rho_{i,t}(\theta)A_i, \text{clip}(\rho_{i,t}(\theta), 1 - \epsilon, 1 + \epsilon)A_i\right).$$

The full GRPO loss is

$$L^{\text{grpo}}(\theta) = L_{\text{pg}}^{\text{grpo}}(\theta) + \beta \widehat{\text{KL}}(\pi_{\theta} \| \pi_{\text{ref}}).$$

2.6 On-Policy GR-REINFORCE vs. Sample Reuse in GRPO

A key theme of the assignment is comparing strictly on-policy updates with controlled sample reuse. In the provided GR-REINFORCE runs, each rollout batch is used for exactly one optimizer step. In the provided GRPO runs, the same rollout batch is reused for two PPO epochs. This means that, for the same number of sampled prompts, GRPO can make more optimization progress per collected rollout batch. That is the main reason you should expect GRPO to improve more sample-efficiently than GR-REINFORCE in this codebase.

Recommended reading. GRPO was introduced in *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. If you want to read beyond the assignment, good follow-ups are recent work on *DrGRPO*, *DAPO*, and *GSPO*. In particular:

- DeepSeekMath (GRPO): <https://arxiv.org/abs/2402.03300>
- DrGRPO: <https://arxiv.org/abs/2503.20783>
- DAPO: <https://arxiv.org/abs/2503.14476>
- GSPO: <https://arxiv.org/abs/2507.18071>

These are optional; you do not need them to complete the homework.

3 Modal Setup

We strongly recommend you run experiments on Modal. The wrapper in `scripts/modal_train.py` defaults to an H100 GPU and persists runs under the mounted Modal volume `/vol`.

3.1 One-Time Setup

From the repository root, run:

```
uv sync
uv run modal token new
uvx wandb login
```

The first command installs dependencies locally. The second authenticates Modal. The third authenticates WandB so your runs can log remotely and write local run artifacts.

3.2 How Detached Modal Runs Work

All training commands in this assignment use `--detach`. This means:

- the job is launched on Modal infrastructure,
- launch and status messages still appear in your local terminal,
- you may close the terminal after launch,
- the run continues remotely,
- outputs are written to `/vol/runs/...`, and
- you can monitor the run in the Modal web UI and in WandB.

4 Codebase Overview

Before implementing anything, read the code in the following order. This order mirrors the data flow through one training iteration.

1. `hw4/train.py`
2. `hw4/models/logprobs.py`
3. `hw4/rollout/rollout_buffer.py`
4. `hw4/rl/reinforce.py`
5. `hw4/rl/grpo.py`
6. `hw4/tasks/format_copy.py`
7. `hw4/tasks/math_hard.py`
8. `hw4/rollout/hf_sampler.py`

The task files are worth reading even though you do not need to remove or rewrite anything there. They tell you exactly how rewards are computed and exactly which evaluation metrics are logged.

4.1 Recommended Implementation Order

Implement the TODOs in this order:

1. `compute_per_token_logprobs`, `build_completion_mask`, and `approx_kl_from_logprobs`
2. `iter_minibatches`
3. `compute_group_advantages` and `maybe_normalize_advantages`
4. `Reinforce.update`
5. `GRPO.update`

This order matters. For example, even the GR-REINFORCE baseline in this codebase still depends on group-relative advantages, so you should implement `compute_group_advantages` before you try to debug it.

5 Tasks and Rewards

5.1 Task 1: `format_copy`

This is a tiny debugging task designed to let you validate the full RL pipeline quickly. The model is prompted to copy an integer exactly and emit it in XML. The reward has three additive components:

- +1.0 if the predicted integer is exactly correct,
- +0.2 if the completion contains an `<answer>` tag, and
- +0.1 if the completion is strict XML only.

This task is intentionally easy. Its purpose is to make implementation bugs obvious quickly.

5.2 Task 2: `math_hard`

This is the real task in the homework. The training data is a filtered level-5 subset of the MATH dataset, and the default evaluation uses a deterministic 512-example held-out subset. The system prompt asks the model to return only a final boxed answer of the form `\boxed{NUMBER}`.

The reward components making up the total reward for `math_hard` are:

- +1.0 if the boxed-answer parser extracts the correct number,
- +0.1 if the completion contains the literal substring `\boxed{`, and
- +0.1 if the relaxed last-number parser is correct but the boxed parser is not.

Note: reward design is very important in LLM RL, and indeed for this assignment the reward components beyond final ground truth answer are useful for facilitating exploration.

6 Implementation Checklist

As you work through the code, keep the following checklist in mind.

- **Token log-probs.** Align logits at position $t - 1$ with target token x_t .
- **Completion masking.** Only score generated completion tokens; exclude prompt tokens and padding.
- **Approximate KL.** Compute the masked sampled-token estimator described in Section 2.
- **Minibatching.** Slice every tensor field in a `RolloutBatch` consistently, and keep debug lists aligned.
- **Advantages.** Reshape rewards to `[num_groups, group_size]` before group normalization.
- **GR-REINFORCE.** Use one on-policy pass over the rollout; average token log-probs within each completion.
- **GRPO.** Use PPO clipping with the cached rollout log-probs and return the required training metrics.

Implementation deliverable. By the end of this phase, your code should be able to run all four commands in Section 7 without crashing, and WandB should show nontrivial reward improvement on `format_copy`.

7 Required Runs

You must complete the following four runs. Do not change the commands in any way; these are the runs the Gradescope autograder expects. **Important:** run the `format_copy` commands first to debug your implementation, as they will run quickly, costing little time/compute.

7.1 Required Run 1: Format Copy + GRPO

```
uv run modal run --detach scripts/modal_train.py -- \
  --task format_copy \
  --algo grpo \
  --output_dir /vol/runs/modal_format_copy_grpo \
  --steps 51 \
  --batch_size 8 \
  --group_size 6 \
  --min_new_tokens 1 \
  --max_new_tokens 24 \
  --lr 3e-5 \
  --ppo_epochs 2 \
  --minibatch_size 8 \
  --grad_accum_steps 6 \
  --clip_eps 0.2 \
  --kl_coef 0.05 \
  --max_grad_norm 0.5 \
  --wandb_enabled \
  --wandb_project llm-rl-hw4 \
  --wandb_name format_copy_grpo \
  --sample_markdown_log_interval 1 \
  --sample_log_interval 10 \
  --sample_log_n 6 \
```

```
--eval_interval 50 \  
--save_interval 50 \  
--warmup_steps 10
```

7.2 Required Run 2: Format Copy + GR-REINFORCE

```
uv run modal run --detach scripts/modal_train.py -- \  
  --task format_copy \  
  --algo reinforce \  
  --output_dir /vol/runs/modal_format_copy_reinforce \  
  --steps 51 \  
  --batch_size 8 \  
  --group_size 6 \  
  --min_new_tokens 1 \  
  --max_new_tokens 24 \  
  --lr 3e-5 \  
  --minibatch_size 8 \  
  --grad_accum_steps 6 \  
  --kl_coef 0.05 \  
  --max_grad_norm 0.5 \  
  --wandb_enabled \  
  --wandb_project llm-rl-hw4 \  
  --wandb_name format_copy_reinforce \  
  --sample_markdown_log_interval 1 \  
  --sample_log_interval 10 \  
  --sample_log_n 6 \  
  --eval_interval 50 \  
  --save_interval 50 \  
  --warmup_steps 10
```

7.3 Required Run 3: Math Hard + GR-REINFORCE

```
uv run modal run --detach scripts/modal_train.py -- \  
  --task math_hard \  
  --algo reinforce \  
  --output_dir /vol/runs/modal_math_hard_reinforce \  
  --steps 201 \  
  --batch_size 8 \  
  --group_size 8 \  
  --min_new_tokens 8 \  
  --max_new_tokens 512 \  
  --max_prompt_tokens 512 \  
  --temperature 0.8 \  
  --top_p 0.95 \  
  --lr 3e-5 \  
  --minibatch_size 8 \  
  --grad_accum_steps 8 \  
  --max_grad_norm 0.5 \  
  --kl_coef 0.05 \  
  --wandb_enabled \  
  --wandb_project llm-rl-hw4 \  
  --wandb_name math_hard_reinforce
```

```
--sample_markdown_log_interval 1 \  
--sample_log_interval 10 \  
--sample_log_n 8 \  
--cuda_empty_cache_interval 50 \  
--eval_interval 100 \  
--save_interval 100
```

7.4 Required Run 4: Math Hard + GRPO

```
uv run modal run --detach scripts/modal_train.py -- \  
  --task math_hard \  
  --algo grpo \  
  --output_dir /vol/runs/modal_math_hard_grpo \  
  --steps 501 \  
  --batch_size 8 \  
  --group_size 8 \  
  --min_new_tokens 8 \  
  --max_new_tokens 512 \  
  --max_prompt_tokens 512 \  
  --temperature 0.8 \  
  --top_p 0.95 \  
  --lr 3e-5 \  
  --ppo_epochs 2 \  
  --minibatch_size 8 \  
  --grad_accum_steps 8 \  
  --clip_eps 0.2 \  
  --max_grad_norm 0.5 \  
  --kl_coef 0.05 \  
  --wandb_enabled \  
  --wandb_project llm-rl-hw4 \  
  --wandb_name math_hard_grpo \  
  --sample_markdown_log_interval 1 \  
  --sample_log_interval 10 \  
  --sample_log_n 8 \  
  --cuda_empty_cache_interval 50 \  
  --eval_interval 100 \  
  --save_interval 100
```

7.5 Why the Math Commands Are Configured This Way

The two math commands are intentionally similar. They use the same rollout batch size, the same minibatch size, the same gradient-accumulation setting, and the same learning rate. The main algorithmic difference is that GRPO uses `--ppo_epochs 2`, whereas GR-REINFORCE is single-pass on-policy.

This is deliberate. It lets you compare the algorithms in a controlled way. Over the first 200 training iterations, GRPO should typically learn more sample-efficiently because it performs two clipped optimization passes over each collected rollout batch, while GR-REINFORCE uses that batch exactly once.

Also note why the GR-REINFORCE math run is *strictly on-policy*. With `batch_size = 8` and

`group_size = 8`, each rollout contains $8 \times 8 = 64$ sampled completions. With `minibatch_size = 8`, that rollout is split into 8 minibatches. Setting `grad_accum_steps = 8` means gradients are accumulated across all 8 minibatches before taking one optimizer step. So each rollout batch contributes to exactly one optimizer update.

8 Expected Behavior and WandB Diagnostics

8.1 The Main Metrics to Watch

The two most important evaluation metrics are:

- `eval/format_copy_fraction_predicted_number_matches_target_integer_exactly`
- `eval/math_hard_test_subset_split_fraction_exact_match_using_boxed_answer_parser`

The main training-time reward metric is:

- `rollout/mean_total_reward_across_all_completions_in_batch_and_groups`

Look primarily at these three plots to understand how your training is going. If you're interested, also take a look at:

- `train/approximate_kl_divergence_policy_vs_reference_mean_over_minibatches`
- `train/fraction_of_completion_tokens_where_ppo_ratio_was_clipped_mean_over_minibatches` for GRPO
- `train/gradient_global_norm_after_clipping_mean_over_optimizer_steps`

Also open the WandB panel that shows sampled generations. It shows prompts, completions, parsed answers, rewards, and advantages, and looking at this is often the fastest way to understand whether your RL updates are doing something sensible.

8.2 What Good Runs Look Like

Successful runs should follow this overall trend:

Run	Expected trend	Rough lower bound for debugging	Approx. runtime
Format Copy + GRPO	reward starts near 0, often jumps sharply by about step 10, and usually reaches about 1.3 by the end; eval exact match usually approaches 1.0	if final eval exact match is materially below 0.8, debug	about 5 minutes
Format Copy + GR-REINFORCE	reward starts near 0, often jumps sharply by about step 10, and usually reaches about 1.3 by the end; eval exact match usually approaches 1.0	if final eval exact match is materially below 0.8, debug	about 5 minutes
Math Hard baseline	before RL, eval exact match is about 0.23 and train reward is about 0.26	if you are far below this before training, debug generation / parsing	included in each run
Math Hard + GR-REINFORCE (201 steps)	final train reward about 0.32 and final eval about 0.31 (could be as low as 0.29)	if final eval is below about 0.27, debug	about 2 hours
Math Hard + GRPO (501 steps)	final train reward above 0.55 and final eval above 0.37	if final eval is below about 0.35, debug	about 5 hours

Important interpretation note. The rollout reward plot is expected to fluctuate a lot from one training iteration to the next. That is normal. The evaluation curves should be noticeably smoother and are the right curves to use when comparing algorithms.

9 GRPO Hyperparameter Study on `format_copy`

In addition to the four required runs, you will perform a small ablation study using GRPO on `format_copy`. This task is cheap enough that you can quickly try several variants.

You should vary at least the following hyperparameters:

- `ppo_epochs`
- `minibatch_size` in tandem with `grad_accum_steps`
- `kl_coef`
- `clip_eps`

A good strategy is to start from the default `format_copy + grpo` command and then try at least five additional runs. At minimum, your set of extra runs should include:

- one run with a different `ppo_epochs`,
- one run with a smaller KL coefficient,
- one run with a larger KL coefficient,
- one run with a different clipping threshold, and

- one run with `grad_accum_steps` set to 1 (so we do `ppo_epochs * (batch_size * group_size / minibatch_size)` optimizer updates).

Try to play around with the hyperparameters such that you learn about the sweet spot of stable-run-inducing hyperparam configurations (i.e., push hyperparam values until performance starts to degrade), and if possible, find one setting that is competitive with or slightly better than the default command. Use WandB to support your conclusions.

Ablation deliverable. Your report must include a concise description of every extra `format_copy + grpo` run you tried, what hyperparameter you changed, what happened to reward/eval performance, and what the other important wandb metrics, such as KL and clip-fraction, suggest are happening.

10 What to Put in Your Report

Submit a short PDF report to the report Gradescope assignment. You may paste plots and example generations directly from WandB instead of re-plotting them yourself (though feel free to plot yourself if you want nicer looking plots).

Your report should answer the following questions.

1. **Approximate KL.** In a short paragraph, explain why the estimator

$$e^{\Delta} - \Delta - 1$$

is a valid sampled-token estimator for the KL term used in this assignment, and why computing the exact full-vocabulary KL at every token position would be much more expensive in both compute and memory.

2. **Implementation.** Briefly describe the order in which you implemented the TODOs and one bug or confusion point that you had to resolve.
3. **GR-REINFORCE vs. GRPO on math.** Compare the WandB curves for the math runs. How do the two methods differ over the first 200 iterations? Why is that comparison interesting given the way the provided commands were chosen?
4. **GRPO ablations on format_copy.** Summarize the extra GRPO runs you tried. Which hyperparameters mattered most? Which settings made learning worse, and why do you think they did?
5. **Qualitative behavior.** Include one or two model-generation examples from WandB that you found informative or surprising from the math-hard task.

11 What to Submit

11.1 Code/Run Submission

For the autograder submission, you should submit both your full homework codebase and the compact run-artifact bundle built from your required experiments. The bundle script accepts partial submissions: you may submit whatever runs you have completed so far, and Gradescope will score

the runs that are present while treating missing required runs as missing until you add them in a later submission.

Whenever you want to submit, first build the compact submission bundle on Modal. Pass `--run_dir` once for each completed run you want to include. When you have all four required runs, the command is:

```
uv run modal run scripts/modal_train.py::bundle_submission_remote -- \
  --run_dir /vol/runs/modal_format_copy_grpo \
  --run_dir /vol/runs/modal_format_copy_reinforce \
  --run_dir /vol/runs/modal_math_hard_grpo \
  --run_dir /vol/runs/modal_math_hard_reinforce \
  --output_dir /vol/submissions/hw4_gradescope_submission \
  --overwrite
```

Then download the resulting zip locally:

```
uv run modal volume get hw4-llm-rl-volume /submissions/hw4_gradescope_submission.zip .
```

Now package your code and artifacts together as follows:

1. Make a clean copy of your homework codebase locally.
2. Unzip `hw4_gradescope_submission.zip`.
3. Put your code folder `hw4/` and the unzipped folder `hw4_gradescope_submission/` side by side in the same parent directory.
4. Zip those two folders together directly, with no extra outer wrapper directory, and upload that zip to the autograder Gradescope assignment.

A recommended local directory layout *before zipping* is:

```
Desktop/
  hw4/
    README.md
    hw4/
    scripts/
    ...
  hw4_gradescope_submission/
    format_copy_grpo/
    format_copy_reinforce/
    math_hard_grpo/
    math_hard_reinforce/
    submission_manifest.json
```

Then create a zip so that the uploaded file contains these two folders at the top level:

```
hw4_submission.zip
  hw4/
    README.md
    hw4/
    scripts/
    ...
  hw4_gradescope_submission/
```

...

Do not place `hw4/` and `hw4_gradescope_submission/` inside an additional enclosing folder before zipping.

If your current working directory is the parent directory containing both `hw4/` and `hw4_gradescope_submission/`, one convenient command is:

```
zip -r hw4_submission.zip hw4 hw4_gradescope_submission \  
-x "hw4/.venv/*" "hw4/.git/*" "hw4/**/__pycache__/*" "*.pyc" ".DS_Store"
```

Do not include large unnecessary directories such as `.venv/`, `__pycache__`, downloaded model caches, or checkpoint adapters in the uploaded code zip.

11.2 Report Submission

Upload your report PDF to the report Gradescope assignment.

Final deliverables checklist. Before you submit, make sure you have:

- completed the four required runs,
- completed the extra GRPO ablations on `format_copy`,
- written a short report answering the required questions,
- built `hw4_gradescope_submission.zip`,
- packaged that bundle together with your full codebase in one zip for the autograder submission, and
- uploaded the code/artifacts zip and the report PDF to the two Gradescope assignments.