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. Zero-shot prompting baseline for the MATH dataset of competition math problems Hendrycks et al. (2021).
2. Group-Relative Policy Optimization (GRPO) for improving reasoning performance with verified rewards.

1.2 What you will run

1. Measure Qwen 2.5 Math 1.5B zero-shot prompting performance (our baseline).
2. Implement and Run GRPO on Qwen 2.5 Math 1.5B with verified rewards.

1.3 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_alignment/prompts/*: For your convenience, we've provided text files with prompts to minimize possible errors caused by copying-and-pasting prompts from the PDF to your code.

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.

What you can use. We expect you to build most of the RL related components from scratch. You may use tools like vLLM to generate text from language models (§3.1). In addition, you may use HuggingFace Transformers to load the Qwen 2.5 Math 1.5 B model and tokenizer and run forward passes, but you may not use any of the training utilities (e.g., the Trainer class).

1.4 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.

1.5 Grading

- **Test Cases (56% credits)**: There are 14 unit tests for `test_grpo`, each worth 4 points.
- **Analytical Questions (44% credits)**: There are 16 problems in total. Problems 2–7 will be graded via above `pytest`. For the remaining 10 problems, each is worth 4 points, except Problem 16, which is worth 8 points.

2 Background

2.1 Motivation

One of the remarkable use cases of language models is in building generalist systems that can handle a wide range of natural language processing tasks. In this assignment, we will focus on a developing use case for language models: mathematical reasoning. It will serve as a testbed for us to set up evaluations and experiment with teaching LMs to reason using reinforcement learning (RL).

There are going to be two differences from the way we've done our past assignments.

- First, we are not going to be using our language model codebase and models from earlier. We would ideally like to use base language models trained from previous assignments, but finetuning those models will not give us a satisfying result—these models are far too weak to display non-trivial mathematical reasoning capabilities. Because of this, we are going to switch to a modern, high-performance language model that we can access (Qwen 2.5 Math 1.5B Base) and do most of our work on top of that model.
- Second, we are going to introduce a new benchmark with which to evaluate our language models. Up until this point, we have embraced the view that cross-entropy is a good surrogate for many downstream

tasks. However, the point of this assignment will be to bridge the gap between base models and downstream tasks and so we will have to use evaluations that are separate from cross-entropy. We will use the MATH 12K dataset from Hendrycks et al. (2021), which consists of challenging high-school competition mathematics problems. We will evaluate language model outputs by comparing them against a reference answer.

2.2 Chain-of-Thought Reasoning and Reasoning RL

An exciting recent trend in language models is the use of chain-of-thought reasoning to improve performance across a variety of tasks. Chain-of-thought refers to the process of reasoning through a problem step-by-step, generating intermediate reasoning steps before arriving at a final answer.

Chain-of-thought reasoning with LLMs. Early chain-of-thought approaches finetuned language models to solve simple mathematical tasks like arithmetic by using a "scratchpad" to break the problem into intermediate steps Nye et al. (2021). Other work prompts a strong model to "think step by step" before answering, finding that this significantly improves performance on mathematical reasoning tasks like grade-school math questions Wei et al. (2022).

Reasoning RL with verified rewards, o1, and R1. Recent work has explored using more powerful reinforcement learning algorithms with verified rewards to improve reasoning performance. OpenAI's o1 (and subsequent o3/o4) Jaech et al. (2024), DeepSeek's R1 Guo et al. (2025), and Moonshot's kimi k1.5 Team et al. (2025) use policy gradient methods Sutton et al. (1999) to train on math and code tasks where string matching or unit tests verify correctness, demonstrating remarkable improvements in competition math and coding performance. Later works such as Open-R1 Face, SimpleRL-Zoo Zeng et al. (2025), and TinyZero Pan et al. (2025) confirm that pure reinforcement learning with verified rewards even on models as small as 1.5B parameters-can improve reasoning performance.

Our setup: model and dataset. In the following sections, we will consider progressively more complex approaches to train a base language model to reason step-by-step in order to solve math problems. For this assignment, we will be using the Qwen 2.5 Math 1.5B Base model, which was continually pretrained from the Qwen 2.5 1.5B model on high-quality synthetic math pretraining data Yang et al. (2024).

Datasets

We can use the following mathematical reasoning dataset:

- MATH Hendrycks et al. (2021), available here (`https://huggingface.co/datasets/qwedsacf/competition_math`). For the MATH dataset, please use it only for this assignment. Do not share or use it for other purposes due to licensing restrictions.

- GSM8K Cobbe et al. (2021), available here (`https://huggingface.co/datasets/openai/gsm8k`): grade-school math problems, which are easier than MATH but should allow you to debug correctness and get familiar with the reasoning RL pipeline.

To obtain short ground-truth labels (e.g., 1/2) if they are not provided directly, you can process the ground-truth column with a math answer parser such as Math-Verify.

3 Measuring Zero-Shot MATH Performance

We'll start by measuring the performance of our base language model on the 5K example test set of MATH. Establishing this baseline is useful for understanding how each of the later approaches affects model behavior.

Unless otherwise specified, for experiments on MATH we will use the following prompt from the DeepSeek R1-Zero model Guo et al. (2025). We will refer to this as the r1_zero prompt:

```
A conversation between User and Assistant. The User asks a question, and the Assistant
solves it. The Assistant first thinks about the reasoning process in the mind and
then provides the User with the answer. The reasoning process is enclosed within
<think> </think> and answer is enclosed within <answer> </answer> tags, respectively,
i.e., <think> reasoning process here </think> <answer> answer here </answer>.
User: {question}
Assistant: <think>
```

The r1_zero prompt is located in the text file `cse599o_alignment/prompts/r1_zero.prompt`. In the prompt, question refers to some question that we insert (e.g., Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in April and May?). The expectation is that the model plays the role of the assistant, and starts generating the thinking process (since we have already included a left <think> tag), closes the thinking process with </think> and then generates a final symbolic answer within the answer tags, like <answer>$4x + 10$</answer> . The purpose of having the model generate tags like <answer> </answer> is so that we can easily parse the model's output and compare it against a ground truth answer, and so that we can stop response generation when we see the right answer tag </answer>.

Note on prompt choice. It turns out that the r1_zero prompt is not the best choice for maximizing downstream performance after RL, because of a mismatch between the prompt and how the Qwen 2.5 Math 1.5B model was pretrained. Liu et al. (2025) finds that simply prompting the model with the question (and nothing else) starts with a very high accuracy, e.g., matching the r1_zero prompt after 100+ steps of RL. Their findings suggest that Qwen 2.5 Math 1.5B was already pretrained on such question-answer pairs.

Nonetheless, we choose the r1_zero prompt for this assignment because RL with it shows clear accuracy improvements in a short number of steps, allowing us to walk through the mechanics of RL and sanity check correctness quickly, even if we don't manage the best final performance. As a reality check, you will compare directly to the question_only prompt later in the assignment.

3.1 Using vLLM for offline language model inference

To evaluate our language models, we're going to have to generate continuations (responses) for a variety of prompts. While one could certainly implement their own functions for generation (e.g., as you did in assignment 1), efficient implementation of RL requires high-performance inference techniques, and implementing these inference techniques are beyond the scope of this assignment. Therefore, in this assignment we will recommend using vLLM for offline batched inference. vLLM is a high-throughput and memory-efficient inference engine for language models that incorporates a variety of useful efficiency techniques (e.g., optimized CUDA kernels, PagedAttention for efficient attention KV caching Kwon et al. (2023)). To use vLLM to generate continuations for a list of prompts [1]:

```python
from vllm import LLM, SamplingParams
# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]

# Create a sampling params object, stopping generation on newline.
sampling_params = SamplingParams(
    temperature=1.0, top_p=1.0, max_tokens=1024, stop=["\n"]
)
# Create an LLM.
llm = LLM(model=<path to model>)
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

In the example above, the LLM can be initialized with the name of a HuggingFace model (which will be automatically downloaded and cached if it isn't found locally), or a path to a HuggingFace model. In this assignment, we will use the **Qwen/Qwen2.5-Math-1.5B** model, which can be downloaded from Hugging Face (`https://huggingface.co/Qwen/Qwen2.5-Math-1.5B`).

3.2 Using HuggingFace Models

Loading a HuggingFace model and tokenizer. To load a HuggingFace model and tokenizer from a local dir (in float16), you can use the following starter code.

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained(
    "/models/Qwen2.5-Math-1.5B",
    torch_dtype=torch.float16
)
tokenizer = AutoTokenizer.from_pretrained("/models/Qwen2.5-Math-1.5B")
```

Forward pass. After we've loaded the model, we can run a forward pass on a batch of input IDs and get the logits (with the .logits) attribute of the output. Then, we can compute the loss between the model's predicted logits and the actual labels:

[1] Example taken from https://docs.vllm.ai/en/v0.10.2/serving/offline_inference.html.

```
input_ids = train_batch["input_ids"].to(device)
labels = train_batch["labels"].to(device)
logits = model(input_ids).logits
loss = F.cross_entropy( ... ,  ... )
```

Gradient accumulation. Despite loading the model in float16, even an 80 GB GPU does not have enough memory to support reasonable batch sizes. To use larger batch sizes, we can use a technique called gradient accumulation. The basic idea behind gradient accumulation is that rather than updating our model weights (i.e., taking an optimizer step) after every batch, we'll accumulate the gradients over several batches before taking a gradient step. Intuitively, if we had a larger GPU, we should get the same results from computing the gradient on a batch of 32 examples all at once, vs. splitting them up into 16 batches of 2 examples each and then averaging at the end.

Gradient accumulation is straightforward to implement in PyTorch. Recall that each weight tensor has an attribute .grad that stores its gradient. Before we call loss.backward(), the .grad attribute is None. After we call loss.backward(), the .grad attribute contains the gradient. Normally, we'd take an optimizer step, and then zero the gradients with optimizer.zero_grad(), which resets the .grad field of the weight tensors:

```
# Standard training loop
for inputs, labels in data_loader:
    # Forward pass
    logits = model(inputs)
    loss = loss_fn(logits, labels)

    # Backward pass
    loss.backward()

    # Update weights
    optimizer.step()
    # Zero gradients in preparation for next iteration
    optimizer.zero_grad()

# ---------------------------------------------
# Gradient accumulation version
# ---------------------------------------------
# To implement gradient accumulation, we' ll just call the optimizer.step()
# and optimizer.zero_grad() every k steps, where k is the number of
# gradient accumulation steps. We divide the loss by
# gradient_accumulation_steps before calling loss.backward() so that
# the gradients are averaged across the accumulation steps.

gradient_accumulation_steps = 4
for idx, (inputs, labels) in enumerate(data_loader):
    # Forward pass
    logits = model(inputs)
    loss = loss_fn(logits, labels) / gradient_accumulation_steps
    # Backward pass
    loss.backward()

    if (idx + 1) % gradient_accumulation_steps == 0:
        # Update weights every `gradient_accumulation_steps` batches
        optimizer.step()
        # Zero gradients every `gradient_accumulation_steps` batches
        optimizer.zero_grad()
```

As a result, our effective batch size when training is multiplied by k, the number of gradient accumulation steps.

3.3 Zero-shot MATH Baseline

Prompting setup. To evaluate zero-shot performance on the MATH test set, we'll simply load the examples and prompt the language model to answer the question using the r1_zero prompt from above.

Evaluation metric. When we evaluate a multiple-choice or binary response task, the evaluation metric is clear-we test whether the model outputs exactly the correct answer.

In math problems we assume that there is a known ground truth (e.g. 0.5) but we cannot simply test whether the model outputs exactly `0.5`—it can also answer `<answer> 1/2 </answer>`. Because of this, we must address the tricky problem of matching for semantically equivalent responses from the LM when we evaluate MATH.

To this end, we want to come up with some answer parsing function that takes as input the model's output and a known ground-truth, and returns a boolean indicating whether the model's output is correct. For example, a reward function could receive the model's string output ending in `<answer> She sold 15 clips. </answer>` and the gold answer `72`, and return `True` if the model's output is correct and `False` otherwise (in this case, it should return `False`).

For our MATH experiments, we will use a fast and fairly accurate answer parser used in recent work on reasoning RL Liu et al. (2025). This reward function is implemented at cse599o_alignment.drgrpo_grader.r1_zero_reward_fn, and you should use it to evaluate performance on MATH unless otherwise specified.

Generation hyperparameters. When generating responses, we'll sample with temperature 1.0, top-p 1.0, max generation length 1024. The prompt asks the model to end its answer with the string </answer>, and therefore we can direct vLLM to stop when the model outputs this string:

```
# Based on Dr. GRPO: stop when the model completes its answer
# https://github.com/sail-sg/understand-r1-zero/blob/
# c18804602b85da9e88b4aeeb6c43e2f08c594fbc/train_zero_math.py#L167
sampling_params.stop = ["</answer>"]
sampling_params.include_stop_str_in_output = True
```

Problem-1 (math_baseline)

(a) Write a script to evaluate Qwen 2.5 Math 1.5 B zero-shot performance on MATH. This script should (1) load the MATH validation examples from /data/a5-alignment/MATH/validation.jsonl, (2) format them as string prompts to the language model using the r1_zero prompt, and (3) generate outputs for each example. This script should also (4) calculate evaluation metrics and (5) serialize the examples, model generations, and corresponding evaluation scores to disk for analysis in subsequent problems. It might be helpful for your implementation to include a method evaluate_vllm with arguments similar to the following, as you will be able to reuse it later:

```python
def evaluate_vllm(
    vllm_model: LLM,
    reward_fn: Callable[[str, str], dict[str, float]],
    prompts: List[str],
    eval_sampling_params: SamplingParams
) -> None:
    """
    Evaluate a language model on a list of prompts,
    compute evaluation metrics, and serialize results to disk.
    """
```

Deliverable: A script to evaluate baseline zero-shot MATH performance.

(b) Run your evaluation script on Qwen 2.5 Math 1.5B. How many model generations fall into each of the following categories: (1) correct with both format and answer reward 1, (2) format reward 1 and answer reward 0, (3) format reward 0 and answer reward 0 ? Observing at least 10 cases where format reward is 0, do you think the issue is with the base model's output, or the parser? Why? What about in (at least 10) cases where format reward is 1 but answer reward is 0?

Deliverable: Commentary on the model and reward function performance, including examples of each category.

(c) How well does the Qwen 2.5 Math 1.5 B zero-shot baseline perform on MATH?

Deliverable: 1-2 sentences with evaluation metrics.

4 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).

4.1 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

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

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.

4.2 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.

4.3 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

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

4.4 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 $\theta . \rho_0, P(\cdot \mid \cdot)$ 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 $\theta \leftarrow \theta + \alpha \widehat{g}$.

4.5 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

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

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.

4.6 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 $\frac{\pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right)}{\pi_{\theta \text{ old}}\left(a_t^{(i)} \mid s_t^{(i)}\right)}$. 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 **?** for more on this.

5 Group Relative Policy Optimization

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

5.1 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

$$A^{(i)} = \frac{r^{(i)} - \operatorname{mean}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right)}{\operatorname{std}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right) + \text{ advantage\_eps}} \tag{28}$$

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. [2]

GRPO objective. The GRPO objective combines three ideas:

1. Off-policy policy gradient, as in Eq. 27.
2. Computing advantages $A^{(i)}$ with group normalization, as in Eq. 28.
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.

[2]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 $\mathcal{D}$
 1: policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
 2: for step $= 1, \ldots, n_{\text{grpo\_steps}}$ do
 3:     Sample a batch of questions $\mathcal{D}_b$ from $\mathcal{D}$
 4:     Set the old policy model $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
 5:     Sample $G$ outputs $\{o_j^{(i)}\}_{j=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
 6:     Compute rewards $\{r_j^{(i)}\}_{j=1}^G$ for each sampled output $o_j^{(i)}$ by running reward function $R(q, o^{(i)})$
 7:     Compute $\mathcal{A}^{(i)}$ with group normalization (Eq. 28)
 8:     for train step $= 1, \ldots, n_{\text{train\_steps\_per\_rollout\_batch}}$ do
 9:         Update the policy model $\pi_\theta$ by maximizing the GRPO-Clip objective (to be discussed, Eq. 29)
10:     end for
11: end for
12: Output: $\pi_\theta$
```

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

$$J_{\text{GRPO-Clip}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\left\{o^{(i)}\right\}_{i=1}^{G}\sim\pi_\theta(\cdot|q)} \left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{\left|o^{(i)}\right|}\sum_{t=1}^{\left|o^{(i)}\right|}\underbrace{\min\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}A^{(i)}, \operatorname{clip}\left(\frac{\pi_\theta\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}{\pi_{\theta_{\text{old}}}\left(o_t^{(i)} \mid q, o_{<t}^{(i)}\right)}, 1-\epsilon, 1+\epsilon\right)A^{(i)}\right)}_{\text{per-token objective}}\right]. \tag{29}$$

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 (2018b,a). Define the function

$$g\left(\epsilon, A^{(i)}\right) = \begin{cases} (1+\epsilon)A^{(i)} & \text{if } A^{(i)} \geq 0 \\ (1-\epsilon)A^{(i)} & \text{if } A^{(i)} < 0 \end{cases} \tag{30}$$

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).

5.2 Implementation

Now that we have a high-level understanding of the GRPO training loop and objective, we will start implementing pieces of it. Many of the pieces implemented in the SFT and EI sections will also be reused for GRPO.

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. 28, 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

$$A^{(i)} = r^{(i)} - \text{mean}\left(r^{(1)}, r^{(2)}, \ldots, r^{(G)}\right). \tag{31}$$

We will implement both variants and compare their performance later in the assignment.

Problem-2 (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:

```
def compute_group_normalized_rewards(
    reward_fn,
    rollout_responses,
    repeated_ground_truths,
    group_size,
    advantage_eps,
    normalize_by_std,
):
```

Compute rewards for each group of rollout responses, normalized by the group size.
Args:

`reward_fn`

`Callable[[str, str], dict[str, float]]` —Scores the rollout responses against the ground truths, producing a dict with keys `"reward"`, `"format_reward"`, and `"answer_reward"`.

rollout_responses

list[str] —Rollouts from the policy. The length of this list is rollout_batch_size = n_prompts_per_rollout_batch * group_size.

repeated_ground_truths

list[str] —Ground truths for the examples. The length of this list is rollout_batch_size, because the ground truth for each example is repeated group_size times.

group_size

int —Number of responses per question (group).

advantage_eps

float —Small constant to avoid division by zero during normalization.

normalize_by_std

bool —If **True**, divide by the per-group standard deviation; otherwise subtract only the group mean.

Returns:

tuple[torch.Tensor, torch.Tensor, dict[str, float]]

advantages: shape (rollout_batch_size,), group-normalized rewards for each rollout response.

raw_rewards: shape (rollout_batch_size,), unnormalized rewards for each rollout response.

metadata: any additional statistics to log (e.g., mean, std, max/min of 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 some 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 (cf. Section 4.6 for discussion).

We will start with the 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

$$-A_t \cdot \log p_\theta \left(o_t \mid q, o_{<t}\right) \tag{32}$$

Problem-3 (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:

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

Compute the policy-gradient loss at every token, where `raw_rewards_or_advantages` is either the raw reward or an already-normalized advantage.
Args:

`raw_rewards_or_advantages`
`torch.Tensor` —shape `(batch_size, 1)`, scalar reward or advantage for each rollout response.

`policy_log_probs`
`torch.Tensor` —shape `(batch_size, sequence_length)`, log-probabilities for each token.

Returns:

`torch.Tensor`
—shape `(batch_size, sequence_length)`, the per-token policy-gradient loss (to be aggregated across the batch and sequence dimensions in the training loop).

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

$$-\min\left(\frac{\pi_\theta\left(o_t \mid q, o_{<t}\right)}{\pi_{\theta_{\text{old}}}\left(o_t \mid q, o_{<t}\right)} A_t, \operatorname{clip}\left(\frac{\pi_\theta\left(o_t \mid q, o_{<t}\right)}{\pi_{\theta_{\text{old}}}\left(o_t \mid q, o_{<t}\right)}, 1-\epsilon, 1+\epsilon\right) A_t\right). \tag{33}$$

Problem-4 (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:

```python
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]]:
```

Args:

`advantages`
: `torch.Tensor` —shape `(batch_size, 1)`, per-example advantages $\mathcal{A}$.

`policy_log_probs`
: `torch.Tensor` —shape `(batch_size, sequence_length)`, per-token log-probabilities from the policy being trained.

`old_log_probs`
: `torch.Tensor` —shape `(batch_size, sequence_length)`, per-token log-probabilities from the old policy.

`cliprange`
: `float` —clipping parameter ϵ (e.g., 0.2).

Returns:

`tuple[torch.Tensor, dict[str, torch.Tensor]]`
: `loss`: shape `(batch_size, sequence_length)`, per-token clipped loss.
: `metadata`: `dict`, containing optional logs (e.g., fraction of tokens that were clipped, or where RHS of the min was chosen).

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-5 (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:

```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]]:
```

Select and compute the desired policy-gradient loss.
Args:

`policy_log_probs`
(`batch_size, sequence_length`), per-token log-probabilities from the policy being trained.

`loss_type`
One of `"no_baseline"`, `"reinforce_with_baseline"`, or `"grpo_clip"`.

`raw_rewards`
Required if `loss_type == "no_baseline"`; shape `(batch_size, 1)`.

`advantages`
Required for `"reinforce_with_baseline"` and `"grpo_clip"`; shape `(batch_size, 1)`.

`old_log_probs`
Required for `"grpo_clip"`; shape `(batch_size, sequence_length)`.

cliprange

Required for "grpo_clip"; scalar ϵ used for clipping.

Returns:

tuple[torch.Tensor, dict[str, torch.Tensor]]

loss (batch_size, sequence_length), per-token loss.

metadata dict, statistics from the underlying routine (e.g., clip fraction for GRPO-Clip).

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, and helpful statistics like per-token entropies and clip fractions. 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 $[i, j] == 1$).

Normalizing by the sequence length has been canonical in most codebases for doing RL with LLMs, but it is not obvious that this is the right thing to do - you may notice, looking at our statement of the policy gradient estimate in (21), that there is no normalization factor $\frac{1}{T^{(i)}}$. We will start with this standard primitive, often referred to as a masked_mean, but will later test out using the masked_normalize method that we implemented during SFT.

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-6 (masked_mean): Masked mean

Deliverable: Implement a method masked_mean that averages tensor elements while respecting a boolean mask.

The following interface is recommended:

```
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`.

Args:

`tensor`	`torch.Tensor` —the data to be averaged.
`mask`	`torch.Tensor` —same shape as `tensor`; positions with `1` are included in the mean.
`dim`	`int	None` —dimension over which to average. If `None`, compute the mean over all masked elements.

Returns:

`torch.Tensor`

—the masked mean; shape matches `tensor.mean(dim)` semantics.

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-7 (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:

```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.

Args:

`policy_log_probs`
: `torch.Tensor` —shape `(batch_size, sequence_length)`, per-token log-probabilities from the policy being trained.

`response_mask`
: `torch.Tensor` —shape `(batch_size, sequence_length)`, with `1` for response tokens and `0` for prompt or padding tokens.

`gradient_accumulation_steps`
: `int` —number of microbatches per optimizer step.

`loss_type`
: `Literal["no_baseline", "reinforce_with_baseline", "grpo_clip"]` —specifies which loss variant to compute.

`raw_rewards`
: `torch.Tensor | None` —required when `loss_type` == `"no_baseline"`; shape `(batch_size, 1)`.

`advantages`
: `torch.Tensor | None` —required when `loss_type` ≠ `"no_baseline"`; shape `(batch_size, 1)`.

`old_log_probs`
: `torch.Tensor | None` —required for GRPO-Clip; shape `(batch_size, sequence_length)`.

`cliprange`
: `float | None` —clip parameter ϵ for GRPO-Clip.

Returns:

`tuple[torch.Tensor, dict[str, torch.Tensor]]`
: `loss`: scalar tensor —the microbatch loss adjusted for gradient accumulation (returned for logging).
: `metadata`: dictionary containing metadata from the underlying loss computation and any other useful statistics.

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.

Putting it all together: GRPO train loop. Now we will put together a complete train loop for GRPO. You should refer to the algorithm in Section 5.1 for the overall structure, using the methods we've implemented where appropriate.

Below we provide some starter hyperparameters. If you have a correct implementation, you should see reasonable results with these.

```
n_grpo_steps: int = 200
learning_rate: float = 1e-5
advantage_eps: float = 1e-6
rollout_batch_size: int = 256
group_size: int = 8
sampling_temperature: float = 1.0
sampling_min_tokens: int = 4 # As in Expiter, disallow empty string responses
sampling_max_tokens: int = 1024
epochs_per_rollout_batch: int = 1 # On-policy
train_batch_size: int = 256 # On-policy
gradient_accumulation_steps: int = 128 # microbatch size is 2, will fit on H100
gpu_memory_utilization: float = 0.85
loss_type: Literal[
    "no_baseline",
    "reinforce_with_baseline",
    "grpo_clip",
] = "reinforce_with_baseline"
use_std_normalization: bool = True
optimizer = torch.optim.AdamW(
    policy.parameters(),
    lr=learning_rate,
    weight_decay=0.0,
    betas=(0.9, 0.95),
)
```

Note that the above parameters are tuned for H100 GPUs. If you are using other GPUs such as Quadro RTX 6000 on Tomago or Tempura, you can reduce parameters such as rollout_batch_size and train_batch_size to avoid out-of-memory issues.

These default hyperparameters will start you in the on-policy setting-for each rollout batch, we take a single gradient step. In terms of hyperparameters, this means that train_batch_size is equal to rollout_batch_size, and epochs_per_rollout_batch is equal to 1 .

Here are some sanity check asserts and constants that should remove some edge cases and point you in the right direction:

```
assert train_batch_size % gradient_accumulation_steps == 0, (
    "train_batch_size must be divisible by gradient_accumulation_steps"
)
micro_train_batch_size = train_batch_size // gradient_accumulation_steps
assert rollout_batch_size % group_size == 0, (
    "rollout_batch_size must be divisible by group_size"
)
n_prompts_per_rollout_batch = rollout_batch_size // group_size
assert train_batch_size >= group_size, (
    "train_batch_size must be greater than or equal to group_size"
)
n_microbatches_per_rollout_batch = rollout_batch_size // micro_train_batch_size
```

And here are a few additional tips:

- Remember to use the r1_zero prompt, and direct vLLM to stop generation at the second answer tag </answer>, as in the previous experiments.
- We suggest using typer for argument parsing.
- Use gradient clipping with clip value 1.0.
- You should routinely log validation rewards (e.g., every 5 or 10 steps). You should evaluate on at least 1024 validation examples to compare hyperparameters, as CoT/RL evaluations can be noisy.
- With our implementation of the losses, GRPO-Clip should only be used when off-policy (since it requires the old log-probabilities).
- In the off-policy setting with multiple epochs of gradient updates per rollout batch, it would be wasteful to recompute the old log-probabilities for each epoch. Instead, we can compute the old log-probabilities once and reuse them for each epoch.
- You should not differentiate with respect to the old log-probabilities.
- You should log some or all of the following for each optimizer update:
  - The loss.
  - Gradient norm.
  - Token entropy.
  - Clip fraction, if off-policy.
  - Train rewards (total, format, and answer).
  - Anything else you think could be useful for debugging.

Problem-8 (grpo_train_loop): GRPO train loop

Deliverable: Implement a complete training loop for GRPO. Begin training a policy on the MATH dataset and confirm that validation rewards improve over time, alongside sensible rollouts.
Provide:

- A plot showing validation rewards versus training steps.

- A few example rollouts illustrating qualitative improvement over time.

We highly encourage you to explore implementing 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. Ray provides a unified programming model and a suite of specialized libraries for distributed data processing and model training, making it easier to build scalable and performant applications. A skeleton script is provided to help you get started (`cse599o_alignment/train_grpo_ray.py`). Ray documentation can be found here: `https://docs.ray.io/en/latest/index.html`.

6 GRPO Experiments

Now we can start experimenting with our GRPO train loop, trying out different hyperparameters and algorithm tweaks. Each experiment will take 2 GPUs, one for the vLLM instance and one for the policy.

Note on stopping runs early. if you see significant differences between hyperparameters before 200 GRPO steps (e.g., a config diverges or is clearly suboptimal), you should of course feel free to stop the experiment early, saving time and compute for later runs. The GPU hours mentioned below are a rough estimate.

Problem-9 (grpo_learning_rate): Tune the learning rate

Description: Starting from the suggested hyperparameters above, perform a sweep over multiple learning rates and report the final validation answer rewards (or note divergence if the optimizer diverges).

Deliverables:

- Validation reward curves for multiple learning rates.
- A brief (2-sentence) discussion describing any other trends you observe in the logged metrics.

For the rest of the experiments, you can use the learning rate that performed best in your sweep above. Effect of baselines. Continuing on with the hyperparameters above (except with your tuned learning rate), we will now investigate the effect of baselining. We are in the on-policy setting, so we will compare the loss types:

- no_baseline
- reinforce_with_baseline

Note that use_std_normalization is True in the default hyperparameters.

Problem-10 (grpo_baselines): Effect of baselining

Description: Train a policy using both `reinforce_with_baseline` and `no_baseline`. Compare their learning behavior and performance.

Deliverables:

- Validation reward curves for each loss type.
- A brief (2-sentence) discussion summarizing any trends you observe in other logged metrics.

For the next few experiments, you should use the best loss type found in the above experiment.

Length normalization. As hinted at when we were implementing masked_mean, it is not necessary or even correct to average losses over the sequence length. The choice of how to sum over the loss is an important hyperparameter which results in different types of credit attribution to policy actions.

Let us walk through an example from Lambert (2025) to illustrate this. Inspecting the GRPO train step, we start out by obtaining per-token policy gradient losses (ignoring clipping for a moment):

```
advantages # (batch_size, 1)
per_token_probability_ratios # (batch_size, sequence_length)
per_token_loss = -advantages * per_token_probability_ratios
```

where we have broadcasted the advantages over the sequence length. Let's compare two approaches to aggregating these per-token losses:

- The masked_mean we implemented, which averages over the unmasked tokens in each sequence.
- Summing over the unmasked tokens in each sequence, and dividing by a constant scalar (which our masked_normalize method supports with constant_normalizer $! = 1.0$) Liu et al. (2025).
  We will consider an example where we have a batch size of 2 , the first response has 4 tokens, and the second response has 7 tokens. Then, we can see how these normalization approaches affect the gradient.

```
from your_utils import masked_mean, masked_normalize
ratio = torch.tensor([
    [1, 1, 1, 1, 1, 1, 1,],
    [1, 1, 1, 1, 1, 1, 1,],
], requires_grad=True)
advs = torch.tensor([
    [2, 2, 2, 2, 2, 2, 2,],
    [2, 2, 2, 2, 2, 2, 2,],
])
masks = torch.tensor([
    # generation 1: 4 tokens
    [1, 1, 1, 1, 0, 0, 0,],
    # generation 2: 7 tokens
    [1, 1, 1, 1, 1, 1, 1,],
])
# Normalize with each approach
max_gen_len = 7
masked_mean_result = masked_mean(ratio * advs, masks, dim=1)
masked_normalize_result = masked_normalize(
    ratio * advs, masks, dim=1, constant_normalizer=max_gen_len)
```

```
print("masked_mean", masked_mean_result)
print("masked_normalize", masked_normalize_result)
# masked_mean tensor([2., 2.], grad_fn=<DivBackward0>)
# masked_normalize tensor([1.1429, 2.0000], grad_fn=<DivBackward0>)
masked_mean_result.mean().backward()
print("ratio.grad", ratio.grad)
# ratio.grad:
# tensor([[0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000, 0.0000],
# [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429]])
ratio.grad.zero_()
masked_normalize_result.mean().backward()
print("ratio.grad", ratio.grad)
# ratio.grad:
# tensor([[0.1429, 0.1429, 0.1429, 0.1429, 0.0000, 0.0000, 0.0000],
# [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429]])
```

Now, let's compare masked_mean with masked_normalize empirically.

Problem-11 (grpo_length_normalization): Effect of length normalization

Deliverable: Compare normalization with `masked_mean` and `masked_normalize` with an end-to-end GRPO training run. Report the validation answer reward curves. Comment on the findings, including any other metrics that have a noticeable trend.

Hint: Consider metrics related to stability, such as the gradient norm.

Fix to the better performing length normalization approach for the following experiments.

Normalization with group standard deviation. Recall our standard implementation of compute_group_normalized_rewards (based on Shao et al. (2024), Guo et al. (2025)), where we normalized by the group standard deviation. Liu et al. (2025) notes that dividing by the group standard deviation could introduce unwanted biases to the training procedure: questions with lower standard deviations (e.g., too easy or too hard questions with all rewards almost all 1 or all 0) would receive higher weights during training.

Liu et al. (2025) propose removing the normalization by the standard deviation, which we have already implemented in compute_group_normalized_rewards and will now test.

Problem-12 (grpo_group_standard_deviation): Effect of standard deviation normalization

Deliverable: Compare the performance of `use_std_normalization == True` and `use_std_normalization == False`. Report the validation answer reward curves. Comment on the findings, including any other metrics that have a noticeable trend.

Hint: Consider metrics related to stability, such as the gradient norm.

Fix to the better performing group normalization approach for the following experiments.

Off-policy versus on-policy. The hyperparameters we have experimented with so far are all on-policy: we take only a single gradient step per rollout batch, and therefore we are almost exactly using the "principled" approximation $\widehat{g}$ to the policy gradient (besides the length and advantage normalization choices mentioned above).

While this approach is theoretically justified and stable, it is inefficient. Rollouts require slow generation from the policy and therefore are the dominating cost of GRPO; it seems wasteful to only take a single gradient step per rollout batch, which may be insufficient to meaningfully change the policy's behavior.

We will now experiment with off-policy training, where we take multiple gradient steps (and even multiple epochs) per rollout batch.

Problem-13 (grpo_colocate): Colocated and Off-Policy GRPO

Implement off-policy GRPO training with colocated training and inference. In this setting, the same model instance performs both rollout generation and policy optimization. You will compare two synchronization strategies that differ in how often the model parameters are updated relative to rollout generation.

Synchronization Strategies.

- **(A) Fully Synchronous:** Perform one round of rollout generation, then exactly one round of gradient updates. The rollout generator and optimizer remain perfectly synchronized.
- **(B) Multiple-Steps-per-Rollout:** Generate one rollout batch, then perform multiple epochs of gradient updates on that same batch before generating new rollouts. This simulates a mild off-policy regime that increases sample reuse and may speed up convergence.

Depending on your implementation of the full GRPO train loop above, you may already have the infrastructure to do this. If not, you need to implement the following:

- You should be able to take multiple epochs of gradient steps per rollout batch, where the number of epochs and optimizer updates per rollout batch are controlled by `rollout_batch_size`, `epochs_per_rollout_batch`, and `train_batch_size`.
- Edit your main training loop to get response logprobs from the policy after each rollout batch generation phase and before the inner loop of gradient steps —these will be the `old_log_probs`. We suggest using `torch.inference_mode()`.
- You should use the `"GRPO-Clip"` loss type.

Deliverables.

- A plot of validation reward versus training steps for both synchronization variants.
- A brief analysis (3–5 sentences) discussing how taking multiple training steps per rollout affects convergence, stability, and runtime.

Now we can use the number of epochs and optimizer updates per rollout batch to control the extent to which we are off-policy. In next problem, you will explore **asynchronous and off-policy GRPO variants**, where training and inference are disaggregated and may proceed in parallel.

Problem-14 (grpo_async_offpolicy): Asynchronous and Off-Policy GRPO

1. Implement and compare two asynchronous GRPO variants:
   - **(A) One-version-behind training:** Disaggregate training and inference, but ensure the rollout generator (inference model) is at most one policy version behind the trainer. This mimics a mild off-policy setup with bounded policy lag.
   - **(B) Replay-buffer training:** Maintain a replay buffer of scored trajectories, and sample from it when updating the policy. This allows stale rollouts to be reused across updates and increases off-policyness.
2. Implement lightweight synchronization logic between the trainer and the rollout generator. You may use either a **Ray-based actor design** (e.g., Generator, Scorer, Learner in `cse599o_alignment/train_grpo_ray.py`) or a simpler `asyncio`-based coordination model (`https://docs.python.org/3/library/asyncio.html`).
3. Profile your system to break down time spent in:
   - Rollout generation (inference time),
   - Policy optimization (training time),
   - Model synchronization or weight updates.

   Use simple wall-clock timers to collect these metrics, and report averages per training step.

Deliverables.

- Scripts to implement two variants.
- Plots of:
  - Validation rewards versus training steps for Variant (A) and (B).
  - Time breakdown per GRPO step (training vs inference vs synchronization).
- Discussion (3–5 sentences): How does increasing off-policyness (e.g., replay buffer size or policy lag) affect stability and convergence? What component dominates runtime, and how might you overlap them more effectively?

On KL divergence. We also note that in the above experiments, we did not include a KL divergence term with respect to some reference model (usually this is a frozen SFT or pretrained checkpoint). In our experiments and others from the literature Liu et al. (2025); NTT123 (2025), we found that omitting the KL term had no impact on performance while saving GPU memory (no need to store a reference model). However, many GRPO repos include it by default and you are encouraged to experiment with KL or other forms of regularization. Many GRPO implementations measure the KL divergence between the current policy and a frozen reference model. This value quantifies how far the policy has drifted from the original model during reinforcement learning. Monitoring KL divergence can reveal whether updates are stable or overly aggressive. Next, we have one task to explore the KL divergence term.

Problem-15 (grpo_kl_divergence): Measuring KL Divergence Drift

Augment your GRPO training loop to **measure** the average KL divergence between:

- your current policy π_{new}, and
- a frozen reference model π_{ref} (e.g., the pretrained `Qwen2.5-Math-1.5B` checkpoint).

At every GRPO step, compute:

$$\mathrm{KL}(\pi_{\text{new}} \,\|\, \pi_{\text{ref}}) = \mathbb{E}_{x\sim\pi_{\text{new}}}\big[\log \pi_{\text{new}}(x) - \log \pi_{\text{ref}}(x)\big].$$

You can approximate this using per-token log probabilities from both models.

Implementation Tips.

- Initialize a frozen reference model:
- During training, compute per-token log probabilities for both models and record:
- It is *not* necessary to add the KL term to your loss; this task only monitors policy drift.

Deliverables.

1. A plot of KL divergence vs. training step, together with validation accuracy.
2. A short (2–3 sentence) discussion:
   - How fast does the KL divergence grow during training?
   - Does higher KL correlate with reward improvements or instability?

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.

Problem-16 (grpo_keyword_inclusion): GRPO Training on Your Own Model

Dataset. We need to generate the dataset using an off-the-shelf model (e.g., `Qwen/Qwen2.5-Math-1.5B` or others) via vLLM. The dataset consists of prompt–response pairs, where the response must include specified keywords. Split the collected examples into `train.jsonl` (80%) and `validation.jsonl` (20%). Each entry should have:

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

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.

Task. Use your GRPO training loop with your own LM implementation (from HW1/HW2) as the policy model, and train it on this Keyword Inclusion dataset. Use the new `keyword_inclusion_reward_fn` to compute rewards for rollouts.

Deliverables.

1. A script that trains your student model with GRPO on the Keyword Inclusion dataset.
2. A plot showing validation rewards versus training steps.
3. Several example rollouts before and after training, showing improved inclusion of required keywords.

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.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.

H. Face. Open r1: A fully open reproduction of deepseek-r1, january 2025. URL https://github. com/huggingface/open-r1, page 9.

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.

D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874, 2021.

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.

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th symposium on operating systems principles, pages 611–626, 2023.

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.

NTT123. Grpo-zero. `https://github.com/policy-gradient/GRPO-Zero`, 2025.

M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

J. Pan, J. Zhang, X. Wang, L. Yuan, H. Peng, and A. Suhr. Tinyzero. https://github.com/Jiayi-Pan/TinyZero, 2025. Accessed: 2025-01-24.

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.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems, 12, 1999.

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.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35:24824–24837, 2022.

A. Yang, B. Zhang, B. Hui, B. Gao, B. Yu, C. Li, D. Liu, J. Tu, J. Zhou, J. Lin, et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. arXiv preprint arXiv:2409.12122, 2024.

W. Zeng, Y. Huang, Q. Liu, W. Liu, K. He, Z. Ma, and J. He. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. arXiv preprint arXiv:2503.18892, 2025.