HW3: Post-train

1. Assignment Overview

In this assignment, you will gain some hands-on experience with training language models to reason and play agentic tasks.

What you will implement.

1. Zero-shot prompting baseline for the GSM8K dataset. Apply Best-of-N to see the strength of test-time scaling.
2. Group-Relative Policy Optimization (GRPO) for improving reasoning performance with verified rewards.
3. Try your best to use language models for scientific discovery.

2. Measuring Zero-shot GSM8K Performance

We'll start by measuring the performance of our base language model on the test set of GSM8K. Establishing this baseline is useful for understanding how training will affect model behavior.

Unless otherwise specified, for experiments on GSM8K we will use the following prompt from the DeepSeekR1-Zero model. We will refer to this as the `r1_zero` prompt:

```
A conversation between User and Assistant. The User takes a question, and the Assistant
solves it. The Assistant first thinks about the reasoning process in the mind and them
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 `NLPDL-2025Fall/hw3_posttrain/data/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 t he model plays the role of the assistant, and starts generating the thinking process (since we have already included a left think tag `<think>`), 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 Qwen3-0.6B model was pretrained. Liu et al. 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 Qwen3-0.6B was already pretrained on such question-answer pairs.

However, 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.

2.1 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, etc.). To use vLLM to generate continuations for a list of prompts:

```
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 text 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. Since downloads can take a long time (especially for larger models, e.g., 70B parameters), we recommend downloading them in advance to a path on your cluster.

3.2 Zero-shot GSM8K Baseline

prompting setup. To evaluate zero-shot performance, you just 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 GSM8K.

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> Whe 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 GSM8K experiments, we will use a fast and fairly accurate answer parser used in recent work on reasoning RL. This reward function is implemented at `data.drgrpo_grader.r1_zero_reward_fn`, and you should use it to evaluate performance on GSM8K 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:

```
sampling_params.stop = ["</answer>"]
sampling_params.include_stop_str_in_output = True
```

> Problem 1:
>
> (a) Write a script (`zero_shot.py`) to evaluate Qwen3-0.6B zero-shot performance on GSM8K. This script should (1) load the GSM8K test examples from `data/gsm8k/test.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.
>
> **Deliverable**: A script to evaluate baseline zero-shot GSM8K performance
>
> (b) Run your evaluation script on Qwen3-0.6B. 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 Qwen3-0.6B zero-shot baseline perform on GSM8K?
>
> **Deliverable**: 1-2 sentences with evaluation metrics

3.3 Best-of-N Test-Time Scaling

The zero-shot baseline from the previous section measures `pass@1` accuracy: we generate one response and check if it's correct. A simple way to improve performance without any training is to use **Best-of-N (Best@N)** sampling. The idea is to generate N candidate responses from the same prompt using a high temperature (like $T = 1.0$) to get a diverse set of solutions. Then, we use a selection mechanism to choose the "best" response from the N candidates.

In this assignment, we will investigate two related metrics:

1. `pass@N`: This metric measures the model's *capability* to solve the problem. We generate N samples and check if *at least one* of them is correct according to our `r1_zero_reward_fn`. If one or more are correct, the problem is considered "passed." This gives an upper-bound on performance for a given N, assuming a perfect selection method. Note that `pass@1` is exactly the zero-shot baseline you just calculated.
2. `Best@N`: This metric measures the model's ability to *recognize* the correct answer from its own N samples. As per your request, we will *not* use the ground-truth `r1_zero_reward_fn` for selection. Instead, we'll use the model's own "score." The standard choice for this score is the **normalized log-likelihood** (i.e., the average log-probability per token) of the generated sequence.
    - The procedure is: (1) Generate N samples. (2) For each sample, calculate its average log-probability. (3) Select the single sample with the *highest* average log-probability. (4) Evaluate *only this single selected sample* using the `r1_zero_reward_fn`. The `Best@N` accuracy is the percentage of prompts where this highest-scoring sample was correct.

To implement this, you will need to configure vLLM to generate N samples per prompt and to return the log probabilities for each generated token.

> Problem 2:
>
> (a) Write a script (`best_of_n.py`) that implements and evaluates `pass@N` on the GSM8K test set using Qwen3-0.6B. Use the generations from your zero-shot script for $N = 1$.
>
> **Deliverable**: Report `pass@N` accuracy for $N \in \{1, 4, 8, 16\}$. You can sanity check the implementation by aligning your pass@1 result with Problem 1.
>
> (b) Using the *same* generated samples from (a), implement and evaluate `Best@N` using the normalized log-likelihood (average log-probability) as the scoring function to select the best sample.
>
> **Deliverable**: Report `Best@N` accuracy for $N \in \{1, 4, 8, 16\}$. (Note: `Best@1` should equal `pass@1`).
>
> (c) Plot the `pass@N` and `Best@N` accuracies as a function of N. How do they compare? What does the gap between `pass@N` and `Best@N` tell you about the model's ability to solve math problems versus its ability to *recognize* its own correct solutions using likelihood alone?
>
> **Deliverable**: A plot of `pass@N` and `Best@N` vs. N, and 1-3 sentences analyzing the gap.

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. The strongest such open reasoning models, such as DeepSeek R1 and Kimi k2, 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. You can use these materials to recap the key concepts of RL.

4.1 Language Models as Policies

A causal language model (LM) with parameters θ defines a probability distribution oveer the next token $a_t \in \mathcal{V}$ given the current 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

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

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 $\log \pi_\theta(a_t|s_t)$.

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:

$$\tau = (s_0, a_0, s_1, a_1, \cdots, s_T, a_T),$$

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 $s_{t+1} \sim P(\cdot|s_t, a_t)$. 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

$$R_T = R(S_T, a_T) := 1[\text{if the trajectory } s_T||a_T \text{ matches the ground-truth according to our reward function}]$$

1[.] is an indicator function. The return $R(\tau)$ aggregates rewards along the trajectory. Two common choices are finite-horizon undiscounted returns

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

and infinite-horizon discounted returns,

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

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

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

leading to the optimization problem

$$\theta^* = \arg\max_\theta J(\theta).$$

4.4 Vanilla Policy Gradient

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

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

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

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

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

$$P(\tau|\theta) = \rho_0(s_0)\Pi_{t=0}^{T}P(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t).$$

Therefore, the log-probability of a trajectory is:

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

2. The log-derivative trick:

$$\nabla_\theta P = P\nabla_\theta \log P.$$

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

$$\nabla_\theta\rho_0 = \nabla_\theta P = \nabla_\theta R(\tau) = 0.$$

Applying the facts above:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \sum_\tau P(\tau|\theta)R(\tau) \\
&= \sum_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\
&= \sum_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)]
\end{aligned}$$

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:

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

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 roullouts $=\{\tau^{(i)}\}_{i=1}^{N}$ 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

$$\hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) R(\tau^{(i)}).$$

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

4.3 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*: 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:

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

As an example, a reasonable baseline is the on-policy value function $V^\pi(s) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)|s_t = s]$, 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

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

Focusing on the baseline term, we see that

$$\mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = \sum_{t=0}^{T} \mathbb{E}_{s_t}[b(s_t) \mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t)].$$

In general, the expectation of the score function is zero: $\mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0$. Therefore, $B = \nabla_\theta J(\pi_\theta)$,

so we conclude that the baselined policy gradient is unbiased.

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. 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 $\pi_{\theta_{old}}$ to optimize the current policy π_θ. The off-policy policy gradient estimate is

$$\hat{g}_{off-policy} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\frac{\pi_\theta(a_t^{(i)}|s_t^{(i)})}{\pi_{\theta_{old}}(a_t^{(i)}|s_t^{(i)})}\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau^{(i)})$$

This looks like an importance sampled version of the vanilla policy gradient.

5. GRPO

Next, we will describte Group Relative Policy Opimitzation (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 $\{o^{(i)}\}_{i=1}^{G} \sim \pi_\theta(\cdot|q)$, let $r^{(i)} = R(q, o^{(i)})$ be the reward for the i-th output. DeepSeekMath and DeepSeek R1 compute the group-normalized reward for the i-th output as

$$A^{(i)} = \frac{r^{(i)} - mean(r^{(1)}, r^{(2)}, \cdots, r^{(G)})}{std(r^{(1)}, r^{(2)}, \cdots, r^{(G)}) + advantage\_eps} \quad (Eq.28)$$

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., $A_t^{(i)} = A^{(i)}, \forall t \in 1, \cdots, |o^{(i)}|$, so we drop the t subscript in the following.

GRPO objective. The GRPO objective combines three ideas:

1. Off-policy policy gradient;
2. Computing advantage $A^{(i)}$ with group normalization;
3. A clipping mechanism, as in PPO.

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.

The GRPO-Clip objective uses a min function to clip the probability ratio, preventing the policy from deviating too far from the old policy during training.

Let us first write out the full GRPO-Clip objective, and then we can build some intuition on what the clipping does (Eq.29):

$$J_{GRPO-Clip}(\theta) = E_{q\sim\mathcal{D},\{o^{(i)}\}_{i=1}^{G}\sim\pi_\theta(\cdot|q)}$$
$$[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o^{(i)}|}\sum_{t=1}^{|o^{(i)}|}\min(\frac{\pi_\theta(o_t^{(i)}|q,o_{<t}^{(i)})}{\pi_{\theta_{old}}(o_t^{(i)}|q,o_{<t}^{(i)})}A^{(i)}, clip(\frac{\pi_\theta(o_t^{(i)}|q,o_{<t}^{(i)})}{\pi_{\theta_{old}}(o_t^{(i)}|q,o_{<t}^{(i)})}, 1-\epsilon, 1+\epsilon)A^{(i)})]$$

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. Define the function

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

We can rewrite the per-token objective as

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

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

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

Since $A^{(i)} > 0$, the objective goes up if the action $o_t^{(i)}$ becomes more likely under π_θ, i.e., if $\pi_\theta(o_t^{(i)}|q,o_{<t}^{(i)})$ increases. The clipping with min limits how much the objective can increase. So the policy π_θ is not incentivized to go very far from the old policy $\pi_{\theta_{old}}$.

Analogously, when the advantage is negative, the model tries to drive down $\pi_\theta(o_t^{(i)}|q,o_{<t}^{(i)})$, but is not incentivized to decrease it below $(1-\epsilon)\pi_{\theta_{old}}(o_t^{(i)}|q,o_{<t}^{(i)})$.

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.

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 in Eq. 28, and a recent simplified approach.

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

$$A^{(i)} = r^{(i)} - \text{mean}(r^{(1)}, r^{(2)}, \cdots, r^{(G)}).$$

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

> Problem 3: Group normalization
>
> **Deliverable**: Implement a method `compute_group_normalized_rewards` that calculates raw rewards for each rollout response, normalizes them within their groups, and return both the normalized and raw rewards along with any metadata you think is useful.

> **Testing**: Run `pytest tests/test_grpo.py::test_compute_group_normalized_rewards_normalize_by_std` and `pytest tests/test_grpo.py::test_compute_group_normalized_rewards_no_normalize_by_std` to verify your implementation.
>
> The following interface is recommended:

```
def compute_group_normalized_rewards(
reward_fn,
 rollout_responses,
 repeated_ground_truths,
 group_size,
 advantage_eps,
 normalized_by_std,
):
 """
   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_response: list[str] Rollouts from the policy. The length of this list is
rollout_batch_size = n_prompts_per_rollout_batch * group_size.
     repeated_groud_truth: list[str] The ground truths for the examples. The length of
this list is rollout_batch_size, because the ground truth for each example is repletaed
grou_size times.
     group_size: int Number of responses per question (group).
     advantage_eps: float. Small constant to avoid division by zero in 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
resposne.
       `metadata` your choice of other statistics to log (e.g., mean, std, max/min of
rewards)
 """
```

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.

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

$$-A_t \cdot \log p_\theta(o_t|q, o_{<t}).$$

> Problem 4: 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.

Testing: Run `pytest tests/test_grpo.py::test_compute_naive_policy_gradient_loss` to verify your implementation.

The following interface is recommended:

```
def compute_naive_policy_gradient_loss(
 raw_rewards_or_advantages: torch.Tensor,
 policy_log_probs: torch.Tensor,
) -> torch.Tensor:
 """
 Compute the per-token policy-gradient loss, using either raw rewards or
 pre-computed (already-normalized) advantages.

 Args:
     raw_rewards_or_advantages (torch.Tensor):
         Tensor of shape (batch_size, 1). Scalar reward/advantage for each
         rollout response.
     policy_log_probs (torch.Tensor):
         Tensor of shape (batch_size, sequence_length). Log probabilities
         for each generated token.

 Returns:
     torch.Tensor:
         Tensor of shape (batch_size, sequence_length) containing the
         per-token policy-gradient loss, which can be aggregated across
         batch and sequence dimensions during training.

 Notes:
     Broadcast `raw_rewards_or_advantages` across the sequence_length
     dimension before applying the loss.
 """
```

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

The per-token GRPO-Clip loss is

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

Problem 5: GRPO-Clip loss

Deliverable: Implement a method `compute_grpo_clip_loss` that computes the per-token GRPO-Clip loss.

Testing: Run `pytest tests/test_grpo.py::test_compute_grpo_clip_loss_large_cliprange` and `pytest tests/test_grpo.py::test_compute_grpo_clip_loss_small_cliprange` to verify your implementation.

The following interface is recommended:

```
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]]:
    """
    Compute the per-token GRPO-Clip loss.

    Args:
        advantages (torch.Tensor):
            Shape (batch_size, 1). Per-example advantages A.
        policy_log_probs (torch.Tensor):
            Shape (batch_size, sequence_length). Per-token log probs from the
            policy being trained.
        old_log_probs (torch.Tensor):
            Shape (batch_size, sequence_length). Per-token log probs from the
            old policy.
        cliprange (float):
            Clip parameter ϵ (e.g., 0.2).

    Returns:
        tuple[torch.Tensor, dict[str, torch.Tensor]]:
            loss:
                Tensor of shape (batch_size, sequence_length), the per-token
                clipped loss.
            metadata:
                Dict containing whatever you want to log. We suggest logging
                whether each token was clipped or not—i.e., whether the
                clipped policy-gradient term on the RHS of the min was lower
                than the LHS.

    Notes:
        • Broadcast `advantages` over the `sequence_length` dimension.
    """
```

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

(a) `no_baseline`: Naive policy gradient loss without a baseline, i.e., advantage is just the raw rewards $A = R(q, o)$.

(b) `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 $A = \bar{r}$.

(c) `grpo_clip`: GRPO-Clip loss.

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

> Problem 6: 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.

Testing: Run `pytest tests/test_grpo.py::test_compute_policy_gradient_loss_no_baseline`, `pytest tests/test_grpo.py::test_compute_policy_gradient_loss_reinforce_with_baseline`, and `pytest tests/test_grpo.py::test_compute_policy_gradient_loss_grpo_clip` to verify your implementation.

The following interface is recommended:

```
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 (torch.Tensor):
         Shape (batch_size, sequence_length). Per-token log-probabilities
         from the policy being trained.
     loss_type (Literal):
         One of "no_baseline", "reinforce_with_baseline", or "grpo_clip".
     raw_rewards (torch.Tensor | None):
         Required if loss_type == "no_baseline"; shape (batch_size, 1).
     advantages (torch.Tensor | None):
         Required for "reinforce_with_baseline" and "grpo_clip";
         shape (batch_size, 1).
     old_log_probs (torch.Tensor | None):
         Required for "grpo_clip"; shape (batch_size, sequence_length).
     cliprange (float | None):
         Required for "grpo_clip"; scalar ϵ used for clipping.

 Returns:
     tuple[torch.Tensor, dict[str, torch.Tensor]]:
         loss:
             Tensor of shape (batch_size, sequence_length), per-token loss.
         metadata:
             Dict of statistics from the underlying routine (e.g., clip
             fraction for GRPO-Clip).

 Notes:
     • Delegate to `compute_naive_policy_gradient_loss` or
       `compute_grpo_clip_loss` as appropriate.
     • Perform argument checks (assert required tensors/values are present).
     • Aggregate any returned metadata into a single dict.
 """
```

Masked mean. Up to this point, we have the logic needed to compute advantages, log probabilities, per-token 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 sequence length is common in RL for LLMs, but it's not always the right choice. We'll start with a standard primitive—often called a **masked_mean**—and later evaluate alternatives. We'll allow specifying the dimension over which to compute the mean; if `dim is None`, we compute the mean over **all** masked elements. This is handy for averaging per-token entropies, clip fractions, etc.

> Problem 7: Masked mean
>
> **Deliverable**: Implement a method `masked_mean` that averages tensor elements while respecting a boolean mask.
>
> **Testing**: Run `pytest tests/test_grpo.py -k masked_mean` to verify your implementation (tests for different dimensions).
>
> The following interface is recommended:

```python
def masked_mean(
 tensor: torch.Tensor,
 mask: torch.Tensor,
 dim: int | None = None,
) -> torch.Tensor:
 """
 Compute the mean of `tensor` along a given dimension, considering only
 those elements where `mask == 1`.

 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 follows `tensor.mean(dim)` semantics.
 """
```

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 8: microbatch train step

Deliverable: Implement a single micro-batch update for GRPO, including policy-gradient loss, averaging with a mask, and gradient scaling.

Testing: Run `pytest tests/test_grpo.py::test_grpo_microbatch_train_step_grpo_clip` and `pytest tests/test_grpo.py::test_grpo_microbatch_train_step_grpo_clip_10_steps` to verify your implementation.

```
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 (batch_size, sequence_length):
         Per-token log-probabilities from the policy being trained.
     response_mask (batch_size, sequence_length):
         1 for response tokens, 0 for prompt/padding.
     gradient_accumulation_steps:
         Number of microbatches per optimizer step.
     loss_type:
         One of "no_baseline", "reinforce_with_baseline", "grpo_clip".
     raw_rewards:
         Needed when loss_type == "no_baseline"; shape (batch_size, 1).
     advantages:
         Needed when loss_type != "no_baseline"; shape (batch_size, 1).
     old_log_probs:
         Required for GRPO-Clip; shape (batch_size, sequence_length).
     cliprange:
         Clip parameter c for GRPO-Clip.

 Returns:
     (loss, metadata)
     loss:
         Scalar tensor. The microbatch loss, adjusted for gradient
         accumulation. We return this so we can log it.
     metadata:
         Dict with metadata from the underlying loss call, and any other
         statistics you might want to log.
 """
```

- Call `loss.backward()` in this function. Make sure to adjust for gradient accumulation.

Putting it all together: GRPO train loop. Now we will put together a complete train loop for GRPO. Belo we provide some starter hyperparameters. If you have a correct implementation, you should see reasonable results with these.

```
# If your GPU memory size is small, please lower batch size.
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
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),
)
```

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 contstants that should remove some edge cases and point you in the right direction:

```
assert train_batch_size % gradient_accumulation_steps == 0
micro_train_batch_size = train_batch_size // gradient_accumulation_steps
assert rollout_batch_size % group_size == 0
n_prompts_per_rollout_batch = rollout_batch_size // group_size
assert train_batch_size >= group_size
n_microbatches_per_rollout_batch = rollout_batch_size // micro_train_batch_size
```

Additional tips

- Use the `r1_zero` prompt.
- Configure vLLM to stop generation at the **second** answer tag `</answer>` (as in previous experiments).
- Use **Typer** for argument parsing.
- Enable gradient clipping with **clip value = 1.0**.
- Routinely log **validation rewards** (e.g., every 5–10 steps).
- Evaluate on **≥ 1024 validation examples** when comparing hyperparameters, since CoT/RL evaluations can be noisy.

- With our loss implementations, **GRPO-Clip should only be used off-policy** (it requires old log-probabilities).
- When doing multiple epochs of gradient updates per rollout batch, **compute the old log-probabilities once and reuse them for each epoch** (don't recompute every epoch).
- **Do not differentiate** with respect to the old log-probabilities.
- Track the **clip fraction** (relevant in off-policy training).
- What to log each optimizer update
    - Loss
    - Gradient norm
    - Token entropy
    - Clip fraction (off-policy only)
        - Train rewards
            - Total
            - Format
            - Answer
    - Anything else that could be useful for debugging

> Problem 8: GRPO train loop.
>
> **Deliverable**: Implement a complete train loop for GRPO. Begin training a policy on MATH and confirm that you see validation rewards improving, along with sensible rollouts over time. Provide a plot with the validation rewards with respect to steps, and a few example rollouts over time.

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: Tune the learning rate (2 points) (6 H100 hrs)
>
> - **Deliverable:** Starting with the suggested hyperparameters above, perform a sweep over the learning rates and report the final validation answer rewards (or note divergence if the optimizer diverges).
> - **Deliverable:** A model that achieves validation accuracy of at least 25% on GSM8K.
> - **Deliverable:** A brief 2 sentence discussion on any other trends you notice on other 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: Effect of baselining (2 points) (2 H100 hrs)
>
> - **Deliverable:** Train a policy with `reinforce_with_baseline` and with `no_baseline`.
> - **Deliverable:** Validation reward curves associated with each loss type.
> - **Deliverable:** A brief 2 sentence discussion on any other trends you notice on 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 [2024] 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, Yu 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]])
```

> Problem 11: Think about length normalization (1 point)
>
> - **Deliverable:** Compare the two approaches (without running experiments yet). What are the pros and cons of each approach? Are there any specific settings or examples where one approach seems better?

> Problem 12: Effect of length normalization (2 points) (2 H100 hrs)
>
> - **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], DeepSeek-AI 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...

> Problem 13: Effect of standard deviation normalization (2 points) (2 H100 hrs)
>
> - **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 $\tilde{g}$ 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 14: Implement off-policy GRPO
>
> - **Deliverable**: Implement off-policy GRPO training.
>
> 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.

Now we can use the number of epochs and optimizer updates per rollout batch to control the extent to which we are off-policy.

> Problem 15: Off-policy GRPO hyperparameter sweep (4 points) (12 H100 hrs)
>
> - **Deliverable**: Fixing rollout_batch_size = 256, choose a range over epochs_per_rollout_batch and train_batch_size to sweep over. First do a broad sweep for a limited number of GRPO steps (≤ 50) to get a sense of the performance landscape, and then a more focused sweep for a larger number of GRPO steps (200). Provide a brief experiment log explaining the ranges you chose.
>
> Compare to your on-policy run with epochs_per_rollout_batch = 1 and train_batch_size = 256, reporting plots with respect to number of validation steps as well as with respect to wall-clock time.
>
> Report the validation answer reward curves. Comment on the findings, including any other metrics that have a noticeable trend such as entropy and response length. Compare the entropy of the model's responses over training to what you observed in the EI experiment.
> - **Hint:** you will need to change `gradient_accumulation_steps` to keep memory usage constant.

Ablating clipping in the off-policy setting. Recall that the purpose of clipping in GRPO-Clip is to prevent the policy from moving too far away from the old policy when taking many gradient steps on a single rollout batch. Next, we will ablate clipping in the off-policy setting to test to what extent it is actually necessary. In other words, we will use per-token loss

$$-\frac{\pi_\theta(o_t|q,o_{<t})}{\pi_{\theta_{old}}(o_t|q,o_{<t})}A_t.$$

> **Problem (grpo_off_policy_clip_ablation): Off-policy GRPO-Clip ablation (2 points) (2 H100 hrs)**
>
> - **Deliverable:** Implement the unclipped per-token loss as a new loss type `"GRPO-No-Clip"`. Take your best performing off-policy hyperparameters from the previous problem and run the

> unclipped version of the loss. Report the validation answer reward curves. Comment on the findings compared to your GRPO-Clip run, including any other metrics that have a noticeable trend such as entropy, response length, and gradient norm.

Effect of prompt. As a last ablation, we'll investigate a surprising phenomenon: the prompt used during RL can have a dramatic effect on the performance of the model, depending on how the model was pretrained. Instead of using the `R1-Zero` prompt at `data/r1_zero.prompt`, we will instead use an extremely simple prompt at `data/question_only.prompt`:

```
{question}
```

You will use this prompt for both training and validation, and will change your reward function (used both in training and validation) to the `question_only_reward_fn` located in `drgrpo_grader.py`.

> Problem 16: Prompt ablation (2 points) (2 H100 hrs)
>
> - **Deliverable:** Report the validation answer reward curves for both the R1-Zero prompt and the question-only prompt. How do metrics compare, including any other metrics that have a noticeable trend such as entropy, response length, and gradient norm? Try to explain your findings.

7. Using LLMs for Scientific Discovery: The Kissing Number Challenge

In the previous sections, you trained models on tasks where the "ground truth" is known (i.e., the answer to a math problem). In this final, open-ended section, you will use a language model for a more challenging task: **scientific discovery**. Your goal is to find a new *lower bound* for a famous mathematical problem where the optimal answer is unknown.

7.1 The Problem: Kissing Number in 11 Dimensions

The **kissing number** τ_d in d dimensions is the maximum number of non-overlapping unit spheres that can all simultaneously touch (or "kiss") a central unit sphere of the same size. Finding the exact value of τ_d is a notoriously difficult problem. The values are only known for $d = 1, 2, 3, 4, 8, 24$. For $d = 11$, the exact value is unknown, but the bound is known to be $595 \leq \tau_{11} \leq 870$.

Your task is to use an LLM to find a large set of vectors that proves a strong lower bound for τ_{11}.

7.2 Your Task: A Verifiable Construction

You will construct a set of integer vectors C in $\mathbb{R}^{11}$ (i.e., $C \subset \mathbb{Z}^{11}$) such that $0 \notin C$ and the following condition holds:

$\min_{x \neq y \in C} \|x - y\| \geq \max_{x \in C} \|x\|$

If this condition holds, it can be proven that unit spheres centered at the points $\{2x/\|x\| : x \in C\}$ form a valid kissing configuration. This means you will have established a new lower bound, $\tau_{11} \geq |C|$.

Your score is simply $|C|$, the number of vectors in your set.

7.3 Resources and Evaluation

To help you, we are providing several resources:

1. **Free API Access:** You will be given free API credits for **Qwen3-32B**, a highly capable model. You are encouraged to use this model, but you may also use any other model you have access to. NOTICE: we will not refund any API cost, and we appreciate you using cheap models to achieve good results, so we don't hope you spend a lot of money on this task.
2. **Starter Program:** In `data/kissing_number/program.py`, you will find a (very simple) starter program that outputs a set of vectors. This is the file you will modify.
3. **Verifier:** In `data/kissing_number/evaluator.py`, we provide a verifier script. You can run it from your terminal:

   Bash

   ```
   python data/kissing_number/evaluator.py data/kissing_number/program.py
   ```

   The evaluator will run your `program.py` in a sandboxed environment, extract the set of vectors C, and check the condition. It will output metrics, including `num_points` ($|C|$), `dimension` (must be 11), and `validity` (1.0 if the condition holds, 0.0 otherwise). Your score is `num_points` *if and only if* `validity` is 1.0.

7.4 Suggested Approaches

This is an open-ended problem. You can use any method you like. Here are some suggestions inspired by the techniques in this assignment and recent research.

1. Baseline: Simple Prompting

Start by simply asking the model (e.g., Qwen3-32B) to solve the problem.

> **Prompt:** "I need to find a set of integer vectors C in 11 dimensions such that 0 is not in C and $\min_{x \neq y \in C} \|x - y\| \geq \max_{x \in C} \|x\|$. My score is the number of vectors in C. Please write a Python program that defines a global variable `points` as a list of such vectors."

Run the output through the evaluator. It will likely fail or give a very small score.

2. Agentic Refinement

This is where the real discovery begins. Treat the evaluator as a tool in an agentic loop (similar to the method in the paper you were given, [arXiv:2507.15855]).

- **Step 1: Generate.** Prompt the LLM to generate a Python script with a candidate set `points`.
- **Step 2: Verify.** Run `evaluator.py` and get the feedback.
- **Step 3: Refine (Loop).**
  - **If Failed (validity: 0.0):** The evaluator will give a specific error, e.g., `"Minimum squared distance 36 < maximum squared norm 40"`. Copy this *exact error* back into your prompt.

    > **Prompt:** "Your previous attempt failed with this error: `Minimum squared distance 36 < maximum squared norm 40`. This means the condition was violated. Please analyze your set of vectors and propose a new, corrected set that fixes this error."

- **If Succeeded (validity: 1.0):** You have a valid lower bound! Now, try to improve it.

  > **Prompt:** "Your previous set of vectors was valid and achieved a score of [Your Score]. This is great! Now, can you try to *add more vectors* to this set while still satisfying the condition? For example, can you find a new vector `z` such that its norm is no more than the current max norm, and its distance to all other vectors is at least that max norm?"

Repeat this loop, using your conversation history to build upon previous successes and learn from failures. This is a form of manual, verifier-guided search.

3. Best-of-N with a Verifier

You can apply the Best@N idea from Section 3.3.

1. Generate N different programs/vector sets from the LLM (e.g., using a high temperature or different "seed" prompts).
2. Run the `evaluator.py` on all N of them.
3. Your "best" sample is the one that has `validity: 1.0` and the highest `num_points`.

This is a powerful test-time method that uses the verifier as your "selection mechanism" instead of log-likelihoods.

4. Hybrid Methods (you can combine Best-of-N with any method)

> Problem 17: Kissing Number Challenge (10 points)
>
> **Deliverables**:
>
> 1. `program.py`: Your final, best-performing Python script located in `data/kissing_number/program.py`. It must be runnable by `evaluator.py` and output your best vector set C.
> 2. `conversation_log.json`: The *entire conversation log* (as a JSON dict) from your interactions with the LLM(s). This is mandatory to show your work and how you used the model for discovery.
> 3. `report.md`: A 1-2 page report detailing:
>   - **Methodology:** What approach(es) did you use? (e.g., "Iterative refinement loop with Qwen3-32B," "Best@N with verifier," etc.).
>   - **Process:** Describe your "journey." What was your initial score? What was your final score? Show 1-2 examples of prompts that led to breakthroughs (or interesting failures).
>   - **Final Result:** State your final score ($|C|$) for $d = 11$.
>   - **Reflection:** What did you learn? What was the most effective part of your strategy?
>
> **Grading**: This problem is open-ended. Your grade will be based on the effort, creativity, and rigor demonstrated in your report and conversation log, as well as on the final score $|C|$ you achieve. A valid submission with a non-trivial score ($|C| > 11$) and a clear report will receive a good grade. Exceptional scores will receive bonus points.