Building LLM Reasoners Assignment 3: Reasoning RL

Spring 2026

This assignment is a modified version of Assignment 5 from Stanford CS336,[1] used with permission. All credit for the original development and the original text of this document goes to the Stanford course staff. **If you have questions about the assignment, do NOT contact the Stanford creators; only contact the NYU course staff.**

1 Assignment Overview

In this assignment, you will gain some hands-on experience with training language models to reason when solving math problems.

What you will implement.

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

Note: the Stanford assignment has an **entirely optional** part of the assignment on aligning language models to human preferences. We have not replicated this here, but you can follow it on the original Stanford site if you'd like.

What you will run.

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

What the code looks like. All the assignment code as well as this writeup are available on GitHub at:

`https://github.com/gregdurrett/nyu-llm-reasoners-a3`

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

1. `student/*`: This is where you'll write your code for assignment 5. 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.

[1] `https://stanford-cs336.github.io/spring2025/`

2. `student/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.

Data you will need is available at

`https://drive.google.com/file/d/1nwDi8mw72Wpt7Uz2MFJA9k7zYmdWCAFe/view?usp=sharing`

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 1.5B model and tokenizer and run forward passes (§4.1), but you may not use any of the training utilities (e.g., the `Trainer` class).

Dependencies. Currently vllm does not work on Macs with Apple Silicon. However, you can do much of part 4 (SFT) and part 7 (GRPO) without actually doing inference. We have included a version of `pyproject.toml` and `uv.lock` for Macs to help you out here if you want to develop locally on a Mac.

How to submit. You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- Code uploaded on Gradescope

2 Reasoning with Language Models

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, perform supervised finetuning, 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 new benchmarks 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. Furthermore, we will use the Countdown dataset, described later when we get to GRPO.

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., 2023].

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) [OpenAI et al., 2024], DeepSeek's R1 [DeepSeek-AI 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, 2025], 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 the SFT portion of the 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]. The MATH dataset is available on Huggingface at `https://huggingface.co/datasets/hiyouga/math12k`.

For the GRPO portion, we will use the Qwen 2.5 Math 1.5B Instruct model, which has undergone additional SFT already.

3 Measuring Zero-Shot MATH Performance

We'll start by measuring the performance of our base language model on the 500 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 a dataset released by Prime Intellect:

```
Solve the following math problem efficiently and clearly. Think carefully and step by
↪  step about your response and reason before providing a final response. Conclude your
↪  response with:

Therefore, the final answer is: $\boxed{answer}$. I hope it is correct.

Where [answer] is just the final number or expression that solves the problem. If the
↪  question is a multiple choice question, [answer] should be the letter indicating your
↪  correct response (e.g. \text{A} or \text{B}).
```

This prompt is loacted in `student/prompts/intellect.prompt`. The question is then to be appended to this prompt.

The purpose of having the model generate the `\boxed` indicator is so that we can easily parse the model's output and compare it against a ground truth answer. This fortunately aligns with the format that the Qwen 2.5 Math 1.5B model

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], etc.).

Please see `student/evaluate.py` for an example of using vLLM for evaluation. In the example above, the `LLM` is 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. The recommended model for these experiments is "Qwen/Qwen2.5-Math-1.5B".

3.2 Zero-shot MATH Baseline

Prompting setup. To evaluate zero-shot performance on the MATH test set, we can use the `evaluate.py` script directly.

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, as it can also answer 1/2. 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 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 `student.drgrpo_grader.question_only_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 0.0 and max generation length 2048. Note that for GRPO, we will need to use randomness, but it could make performance worse!

Problem (`math_baseline`): 4 points

(a) Familiarize yourself with `evaluate.py` and run it with Qwen 2.5 Math 1.5B. You will want to print or log the outputs (not supported in the script currently). For the MATH dataset, 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.

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

Deliverable: 1-2 sentences with evaluation metrics.

4 Supervised Finetuning for MATH

Algorithm 1 Supervised Finetuning (SFT)

```
Input initial policy model $\pi_{\theta_{\text{init}}}$; SFT dataset $\mathcal{D}$
1: policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
2: for step = 1, ..., n_sft_steps do
3:     Sample a batch of question-response pairs $\mathcal{D}_b$ from $\mathcal{D}$
4:     Compute the cross-entropy loss of the responses given the questions using the model $\pi_\theta$
5:     Update the model parameters $\theta$ by taking a gradient step with respect to the cross-entropy loss
6: end for
Output $\pi_\theta$
```

Supervised finetuning for reasoning. In this section, we will finetune our base model on the MATH dataset (Algorithm 1). As our goal is to improve the model's reasoning ability, rather than finetune it to directly predict correct answers, we will finetune it to first generate a chain-of-thought reasoning trace followed by an answer. To this end, we have made available a dataset of such reasoning traces, obtained from Prime Intellect available in the dataset download.

When training a reasoning model in practice, SFT is often used as a warm-start for a second RL finetuning step. There are two main reasons for this. First, SFT requires high-quality annotated data (i.e., with pre-existing reasoning traces), whereas RL requires only the correct answer for feedback. Second, even in settings where annotated data is plentiful, RL can still unlock performance gains by finding better policies than the SFT data. Unfortunately, this will not work in the context of this assignment, so we will not chain these two stages together.

4.1 Using HuggingFace Models

Loading a HuggingFace model and tokenizer. To load a HuggingFace model and tokenizer from a local dir (in `bfloat16` and with FlashAttention-2 to save memory), you can use the following starter code:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-Math-1.5B",
    torch_dtype=torch.bfloat16,
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/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:

```
input_ids = train_batch["input_ids"].to(device)
labels = train_batch["labels"].to(device)

logits = model(input_ids).logits
loss = F.cross_entropy(..., ...)
```

Saving a trained model. To save the model to a directory after training is finished, you can use the `.save_pretrained()` function, passing in the path to the desired output directory. Make sure to save in `/scratch/yourusername` since they can be quite large. We recommend also saving the tokenizer as well (even if you didn't modify it), just so the model and tokenizer are self-contained and loadable from a single directory.

```
# Save the model weights
model.save_pretrained(save_directory=output_dir)
tokenizer.save_pretrained(save_directory=output_dir)
```

Gradient accumulation. Despite loading the model in `bfloat16`, our GPUs do 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:

```
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()
```

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 gradient 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.

4.2 SFT Helper Methods

Next, we will implement some helper methods that you will use during SFT and in the later RL experiments. As a quick note on nomenclature: in the following sections, we will interchangeably refer to a model's completion given a prompt as an "output", "completion", or "response".

Tokenizing prompts and outputs. For each pair of question and target output (q, o), we will tokenize the question and output separately and concatenate them. Then, we can score the log-probabilities of the output with our SFT model (or in later sections, our RL policy). Moreover, we will need to construct a `response_mask`: a boolean mask that is `True` for all tokens in the response, and `False` for all question and padding tokens. We will use this mask in the training loop to ensure that we only compute the loss on the response tokens.

Problem (`tokenize_prompt_and_output`): Prompt and output tokenization (2 points)

Deliverable: Implement a method `tokenize_prompt_and_output` that tokenizes the question and output separately, concatenates them together, and constructs a `response_mask`. The following interface is recommended:

`def tokenize_prompt_and_output(prompt_strs, output_strs, tokenizer):` Tokenize the prompt and output strings, and construct a mask that is 1 for the response tokens and 0 for other tokens (prompt or padding).

Args:

`prompt_strs: list[str]` List of prompt strings.

`output_strs: list[str]` List of output strings.

`tokenizer: PreTrainedTokenizer` Tokenizer to use for tokenization.

Returns:

`dict[str, torch.Tensor]`. Let `prompt_and_output_lens` be a list containing the lengths of the tokenized prompt and output strings. Then the returned dictionary should have the following keys:

- `input_ids` torch.Tensor of shape `(batch_size, max(prompt_and_output_lens) - 1)`: the tokenized prompt and output strings, with the final token sliced off.
- `labels` torch.Tensor of shape `(batch_size, max(prompt_and_output_lens) - 1)`: shifted input ids, i.e., the input ids without the first token.
- `response_mask` torch.Tensor of shape `(batch_size, max(prompt_and_output_lens) - 1)`: a mask on the response tokens in the labels.

To test your code, implement `[adapters.run_tokenize_prompt_and_output]`. Then, run the test with `uv run pytest -k test_tokenize_prompt_and_output` and make sure your implementation passes it.

Logging per-token entropies. When doing RL, it is often useful to keep track of per-token entropies to see if the predictive distribution of the model is becoming (over)confident. We will implement this now and compare how each of our finetuning approaches affects the model's predictive entropy.

The entropy of a discrete distribution $p(x)$ with support $\mathcal{X}$ is defined as

$$H(p) = -\sum_{x \in \mathcal{X}} p(x) \log p(x). \tag{1}$$

Given our SFT or RL model's logits, we will compute the per-token entropy, i.e., the entropy of each next-token prediction.

Problem (compute_entropy): Per-token entropy (1 point)

Deliverable: Implement a method `compute_entropy` that computes the per-token entropy of next-token predictions.
The following interface is recommended:

```
def compute_entropy(logits: torch.Tensor) -> torch.Tensor:
```

Get the entropy of the next-token predictions (i.e., entropy over the vocabulary dimension).

Args:

`logits: torch.Tensor` Tensor of shape `(batch_size, sequence_length, vocab_size)` containing unnormalized logits.

Returns:

`torch.Tensor` Shape `(batch_size, sequence_length)`. The entropy for each next-token prediction.

Note: you should use a numerically stable method (e.g., using `logsumexp`) to avoid overflow.

To test your code, implement [adapters.run_compute_entropy]. Then run `uv run pytest -k test_compute_entropy` and ensure your implementation passes.

Getting log-probabilities from a model. Obtaining log-probabilities from a model is a primitive that we will need in both SFT and RL.

For a prefix x, an LM producing next-token logits $f_\theta(x) \in \mathbb{R}^{|\mathcal{V}|}$, and a label $y \in \mathcal{V}$, the log-probability of y is

$$\log p_\theta(y \mid x) = \log \left[\text{softmax}(f_\theta(x))\right]_y , \tag{2}$$

where the notation $[x]_y$ denotes the y-th element of the vector x.

You will want to use a numerically stable method to compute this, and are free to use methods from `torch.nn.functional`. We also suggest including an argument to optionally compute and return token entropies.

Problem (get_response_log_probs): Response log-probs (and entropy) (2 points)

Deliverable: Implement a method `get_response_log_probs` that gets per-token conditional log-probabilities (given the previous tokens) from a causal language model, and optionally the entropy of the model's next-token distribution.
The following interface is recommended:

```
def get_response_log_probs(
    model: PreTrainedModel,
    input_ids: torch.Tensor,
    labels: torch.Tensor,
    return_token_entropy: bool = False,
) -> dict[str, torch.Tensor]:
```

Args:

`model: PreTrainedModel` HuggingFace model used for scoring (placed on the correct device and in inference mode if gradients should not be computed).

`input_ids: torch.Tensor` shape `(batch_size, sequence_length)`, concatenated prompt + response tokens as produced by your tokenization method.

`labels: torch.Tensor` shape `(batch_size, sequence_length)`, labels as produced by your tokenization method.

`return_token_entropy: bool` If `True`, also return per-token entropy by calling `compute_entropy`.

Returns:

`dict[str, torch.Tensor]`.

`"log_probs"` shape `(batch_size, sequence_length)`, conditional log-probabilities $\log p_\theta(x_t \mid x_{<t})$.

`"token_entropy"` optional, shape `(batch_size, sequence_length)`, per-token entropy for each position (present only if `return_token_entropy=True`).

Implementation tips:

- Obtain logits with `model(input_ids).logits`.

To test your code, implement `[adapters.run_get_response_log_probs]`. Then run `uv run pytest -k test_get_response_log_probs` and ensure the test passes. (Note that this test is not run on the autograder.)

SFT microbatch train step. The loss we minimize in SFT is the negative log-likelihood of the target output given the prompt. To compute this loss, we need to compute the log-probabilities of the target output given the prompt and sum over all tokens in the output, masking the tokens in the prompt and padding tokens.

We will implement a helper function for this, that we will also make use of later during RL.

Problem (`masked_normalize`): Masked normalize (1 point)

Deliverable: Implement a method `masked_normalize` that sums over tensor elements and normalizes by a constant while respecting a boolean mask.
The following interface is recommended:

```
def masked_normalize(
    tensor: torch.Tensor,
    mask: torch.Tensor,
    normalize_constant: float,
    dim: int | None = None,
) -> torch.Tensor:
```

Sum over a dimension and normalize by a constant, considering only those elements where `mask == 1`.

Args:

`tensor: torch.Tensor` The tensor to sum and normalize.

`mask: torch.Tensor` Same shape as `tensor`; positions with `1` are included in the sum.

`normalize_constant: float` the constant to divide by for normalization.

`dim: int | None` the dimension to sum along before normalization. If None, sum over all dimensions.

Returns:

`torch.Tensor` the normalized sum, where masked elements (`mask == 0`) don't contribute to the sum.

To test your code, implement [adapters.run_masked_normalize]. Then run `uv run pytest -k test_masked_normalize` and ensure it passes.

SFT microbatch train step. We are now ready to implement a single microbatch train step for SFT (recall that for a train minibatch, we iterate over many microbatches if `gradient_accumulation_steps > 1`).

Problem (`sft_microbatch_train_step`): Microbatch train step (3 points)

Deliverable: Implement a single micro-batch update for SFT, including cross-entropy loss, summing with a mask, and gradient scaling.
The following interface is recommended:

```
def sft_microbatch_train_step(
    policy_log_probs: torch.Tensor,
    response_mask: torch.Tensor,
    gradient_accumulation_steps: int,
    normalize_constant: float = 1.0,
) -> 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 SFT 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.

`normalize_constant` The constant by which to divide the sum. It is fine to leave this as 1.0.

Returns:

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

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.

Implementation tips:

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

To test your code, implement [adapters.run_sft_microbatch_train_step]. Then run `uv run pytest -k test_sft_microbatch_train_step` and confirm it passes.

4.3 SFT Experiment

Using the pieces above, you will now implement the full SFT procedure (Algorithm 1) to finetune the Qwen 2.5 Math 1.5B Base model on the MATH dataset. Each example in `/data/a5-alignment/MATH/sft.jsonl` consists of a formatted prompt and a target response, where the target response includes a chain-of-thought reasoning trace and the final answer. In particular, each example is a JSON element of type `{"prompt": str, "response": str}`.

In order to track the progress of your model over the course of training, you should periodically evaluate it on the MATH validation set. You can run your script with 2 GPUs, using one GPU for the policy model and the other for the vLLM instance to evaluate the policy. To get this to work, here is some starter code to initialize vLLM and to load the policy weights into the vLLM instance before every rollout phase:

```python
from vllm.model_executor import set_random_seed as vllm_set_random_seed

def init_vllm(model_id: str, device: str, seed: int, gpu_memory_utilization: float = 0.85):
    """
    Start the inference process, here we use vLLM to hold a model on
    a GPU separate from the policy.
    """
    vllm_set_random_seed(seed)

    # Monkeypatch from TRL:
    # https://github.com/huggingface/trl/blob/
    #   22759c820867c8659d00082ba8cf004e963873c1/trl/trainer/grpo_trainer.py
    # Patch vLLM to make sure we can
    # (1) place the vLLM model on the desired device (world_size_patch) and
    # (2) avoid a test that is not designed for our setting (profiling_patch).
    world_size_patch = patch("torch.distributed.get_world_size", return_value=1)
    profiling_patch = patch(
        "vllm.worker.worker.Worker._assert_memory_footprint_increased_during_profiling",
        return_value=None
    )
    with world_size_patch, profiling_patch:
        return LLM(
            model=model_id,
            device=device,
            dtype=torch.bfloat16,
            enable_prefix_caching=True,
            gpu_memory_utilization=gpu_memory_utilization,
        )

def load_policy_into_vllm_instance(policy: PreTrainedModel, llm: LLM):
    """
    Copied from https://github.com/huggingface/trl/blob/
        22759c820867c8659d00082ba8cf004e963873c1/trl/trainer/grpo_trainer.py#L670.
    """
    state_dict = policy.state_dict()
    llm_model = llm.llm_engine.model_executor.driver_worker.model_runner.model
    llm_model.load_weights(state_dict.items())
```

You may find it helpful to log metrics with respect to both the train and validation steps (this will also be useful in later RL experiments). To do this in `wandb`, you can use the following code:

```python
# Setup wandb metrics
wandb.define_metric("train_step")  # the x-axis for training
wandb.define_metric("eval_step")  # the x-axis for evaluation

# everything that starts with train/ is tied to train_step
wandb.define_metric("train/*", step_metric="train_step")

# everything that starts with eval/ is tied to eval_step
wandb.define_metric("eval/*", step_metric="eval_step")
```

Lastly, we suggest that you use gradient clipping with clip value 1.0.

Problem (`sft_experiment`): Run SFT on the MATH dataset (2 points)

1. Run SFT on the reasoning SFT examples training data in the Prime Intellect dataset using the Qwen 2.5 Math 1.5B base model, varying the number of unique examples for SFT in the range $\{128, 256, 512, 1024\}$, along with using the full dataset. Tune the learning rate and batch size until you see the loss decrease substantially on the training set (around a 40% decrease in loss during training).

   Deliverable: Validation accuracy curves associated with different dataset sizes.

2. What results does your best model achieve on the test sets of (1) Prime Intellect data; (2) MATH data? Note that we are not placing a performance requirement here, but want to see what you've found from your model that has (at least somewhat) fit your training data.

   Deliverable: Describe your findings.

5 Countdown

For the rest of this assignment, we will be using a dataset called Countdown. This dataset enables testing RL methods and seeing gains more easily than the MATH dataset.

The Countdown prompt we provide is as follows, which shows you the type of problem you're dealing with here:

```
Answer the following problem. Explain your reasoning step by step. When you are finished,
give your answer in this format: <answer>(your answer)</answer>.
Problem
Using the numbers in the list [96, 97, 68], create an equation that equals 125.
You can use basic arithmetic operations (+, -, *, /) and each number can only be used once.
Your solution should include a series of steps "Step X:" where each step is a
mathematical operation and the final step ultimately leads to the target number or it should
be a single equation that results in the target.
Give your answer in the following format:
<answer>
(your answer)
</answer>
Where "(your answer)" is the list of steps to reach the target number or
it should be a single equation that results in the target.
```

```
For example:
If the list of numbers was [1, 2, 3] and the target was 1, you could write:
<answer>
Step 1: 1 + 2 = 3
Step 2: 3 / 3 = 1
</answer>
or
<answer>
(1 + 2) / 3
</answer>
Let's think step by step.
```

We distribute a subset of this dataset that you can use, including a training dataset of 10k instances and dev and test sets of 1024 instances.

6 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 [OpenAI et al., 2024, DeepSeek-AI 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, 2018a] and Nathan Lambert's Reinforcement Learning from Human Feedback (RLHF) Book [Lambert, 2024].

6.1 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*

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

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.

6.2 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.

6.3 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

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

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

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

leading to the optimization problem

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

6.4 Vanilla Policy Gradient

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

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

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

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

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

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

Therefore, the log-probability of a trajectory is:

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

2. The log-derivative trick:
$$\nabla_\theta P = P \nabla_\theta \log P. \tag{13}$$

3. The environment terms are constant in θ. ρ_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:

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

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:

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

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

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

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

6.5 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:

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

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

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

Focusing on the baseline term, we see that

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

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

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

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

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

`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.

7 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.

7.1 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 [Shao et al., 2024] and DeepSeek R1 [DeepSeek-AI et al., 2025] compute the group-normalized reward for the i-th output as

$$A^{(i)} = \frac{r^{(i)} - \text{mean}(r^{(1)}, r^{(2)}, \ldots, r^{(G)})}{\text{std}(r^{(1)}, r^{(2)}, \ldots, r^{(G)}) + \texttt{advantage\_eps}}, \tag{27}$$

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., $A_t^{(i)} = A^{(i)}, \forall t \in 1, \ldots, |o^{(i)}|$, 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 2.[2]

Algorithm 2 Group Relative Policy Optimization (GRPO)

```
Input 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, ..., n_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 $\{o^{(i)}\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
 6:     Compute rewards $\{r^{(i)}\}_{i=1}^G$ for each sampled output $o^{(i)}$ by running reward function $R(q, o^{(i)})$
 7:     Compute $A^{(i)}$ with group normalization (Eq. 27)
 8:     for train step = 1, ..., n_train_steps_per_rollout_batch do
 9:         Update the policy model $\pi_\theta$ by maximizing the GRPO-Clip objective (to be discussed, Eq. 28)
10:     end for
11: end for
Output $\pi_\theta$
```

GRPO objective. The GRPO objective combines three ideas:

1. Off-policy policy gradient, as in Eq. 33.
2. Computing advantages $A^{(i)}$ with group normalization, as in Eq. 27.
3. A clipping mechanism, as in Proximal Policy Optimization (PPO, Schulman et al. [2017]).

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

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

$$J_{\text{GRPO-Clip}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\ \{o^{(i)}\}_{i=1}^G \sim \pi_\theta(\cdot|q)} \left[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o^{(i)}|} \sum_{t=1}^{|o^{(i)}|} \underbrace{\min\left(\frac{\pi_\theta(o_t^{(i)} \mid q, o_{<t}^{(i)})}{\pi_{\theta_{\text{old}}}(o_t^{(i)} \mid q, o_{<t}^{(i)})} A^{(i)}, \operatorname{clip}\left(\frac{\pi_\theta(o_t^{(i)} \mid q, o_{<t}^{(i)})}{\pi_{\theta_{\text{old}}}(o_t^{(i)} \mid q, o_{<t}^{(i)})}, 1-\epsilon, 1+\epsilon \right) A^{(i)} \right)}_{\text{per-token objective}} \right]. \tag{28}$$

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

$$g(\epsilon, A^{(i)}) = \begin{cases} (1+\epsilon)A^{(i)} & \text{if } A^{(i)} \geq 0 \\ (1-\epsilon)A^{(i)} & \text{if } A^{(i)} < 0. \end{cases} \tag{29}$$

[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.

We can rewrite the per-token objective as

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

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

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

Since $A^{(i)} > 0$, the objective goes up if the action $o_t^{(i)}$ becomes more likely under π_θ, i.e., if $\pi_\theta(o_t^{(i)} \mid q, o_{<t}^{(i)})$ increases. The clipping with min limits how much the objective can increase: once $\pi_\theta(o_t^{(i)} \mid q, o_{<t}^{(i)}) > (1+\epsilon)\pi_{\theta_{\text{old}}}(o_t^{(i)} \mid q, o_{<t}^{(i)})$, 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 $\pi_\theta(o_t^{(i)} \mid q, o_{<t}^{(i)})$, but is not incentivized to decrease it below $(1-\epsilon)\pi_{\theta_{\text{old}}}(o_t^{(i)} \mid q, o_{<t}^{(i)})$ (refer to Achiam [2018b] for the full argument).

7.2 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 section 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. 27, and a recent simplified approach.

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

$$A^{(i)} = r^{(i)} - \text{mean}(r^{(1)}, r^{(2)}, \ldots, r^{(G)}). \tag{30}$$

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

Problem (`compute_group_normalized_rewards`): Group normalization (2 points)

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]` The 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 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 response.
- `metadata` your choice of other statistics to log (e.g. mean, std, max/min of rewards).

To test your code, implement [adapters.run_compute_group_normalized_rewards]. Then, run the test with `uv run pytest -k test_compute_group_normalized_rewards` and make sure your implementation passes it.

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 6.5 for discussion).

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

$$-A_t \cdot \log p_\theta(o_t | q, o_{<t}). \tag{31}$$

Problem (compute_naive_policy_gradient_loss): Naive policy gradient (1 point)

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:

```
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/advantage for each rollout response.

`policy_log_probs: torch.Tensor` Shape `(batch_size, sequence_length)`, logprobs 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 the `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

$$-\min\left(\frac{\pi_\theta(o_t|q,o_{<t})}{\pi_{\theta_{\text{old}}}(o_t|q,o_{<t})}A_t, \text{clip}\left(\frac{\pi_\theta(o_t|q,o_{<t})}{\pi_{\theta_{\text{old}}}(o_t|q,o_{<t})}, 1-\epsilon, 1+\epsilon\right)A_t\right). \tag{32}$$

Problem (`compute_grpo_clip_loss`): GRPO-Clip loss (2 points)

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 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 `torch.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 loss on the RHS of the min was lower than the LHS.

Implementation tips:

- Broadcast `advantages` over `sequence_length`.

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:

(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 these three policy gradient losses.

Problem (compute_policy_gradient_loss): Policy-gradient wrapper (1 point)

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:

```
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, 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 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 (masked_mean): Masked mean (1 point)

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 (grpo_microbatch_train_step): Microbatch train step (3 points)

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:

```
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 ϵ for GRPO-Clip.

Returns:

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

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.

Implementation tips:

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

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 7.1 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 = 16
group_size: int = 8
sampling_temperature: float = 0.7
sampling_min_tokens: int = 4
sampling_max_tokens: int = 1024
epochs_per_rollout_batch: int = 1  # On-policy
train_batch_size: int = 64  # On-policy
gradient_accumulation_steps: int = 128
gpu_memory_utilization: float = 0.8
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. Our reference implementation is able to achieve around 37% validation accuracy on Countdown with this configuration.

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 `countdown` prompt.
- The Countdown prompt asks the model to end on `</answer>`, so you should direct vLLM to stop generation at this tag.
- Use gradient clipping with clip value 1.0.
- You should routinely log validation rewards (e.g., every 5 or 10 steps). You should ensure you evaluate on enough validation examples to reduce noise in the process.
- With our implementation of the losses, GRPO-Clip should only be used when off-policy (since it requires 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 (`grpo_train_loop`): GRPO train loop (5 points)

Deliverable: Implement a complete train loop for GRPO. Begin training a policy on Countdown 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.

8 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 that the HPC policies are set to kill jobs if GPUs are underutilized. **For this course, we have asked for them to allow 90 minutes before a job is killed.** This gives you some additional buffer to conduct experiments, but you won't burn through hours if you leave an experiment running unintentionally.

Note on stopping runs early. if you see significant differences between hyperparameters early in a run (e.g., a config diverges or is clearly suboptimal), you should of course feel free to stop the experiment early.

Problem (`grpo_learning_rate`): Tune the learning rate (2 points)

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). Try at least 3 learning rates.

Deliverable: Validation reward curves associated with multiple learning rates.
Deliverable: A model that achieves validation accuracy of at least 30% on Countdown.
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 (`grpo_baselines`): Effect of baselining (2 points)

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 (think_about_length_normalization): 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?

Now, let's compare masked_mean with masked_normalize empirically.

Problem (grpo_length_normalization): Effect of length normalization (2 points)

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 implemented in `compute_group_normalized_rewards` and will now test.

Problem (`grpo_group_standard_deviation`): Effect of standard deviation normalization (2 points)

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.

[OPTIONAL] Off-policy versus on-policy. 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 $\{\tau^{(i)}\}_{i=1}^{N}$ from the current policy π_θ.
2. Approximate the policy gradient as $\nabla_\theta J(\pi_\theta) \approx \widehat{g} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau^{(i)})$.
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.

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

$$\widehat{g}_{\text{off-policy}} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\frac{\pi_\theta(a_t^{(i)}|s_t^{(i)})}{\pi_{\theta_{\text{old}}}(a_t^{(i)}|s_t^{(i)})}\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau^{(i)}). \tag{33}$$

This looks like an importance sampled version of the vanilla policy gradient, with reweighting terms $\frac{\pi_\theta(a_t^{(i)}|s_t^{(i)})}{\pi_{\theta_{\text{old}}}(a_t^{(i)}|s_t^{(i)})}$. Indeed, Eq. 33 can be derived by importance sampling and applying an approximation that is reasonable as long as π_θ and $\pi_{\theta_{\text{old}}}$ are not too different: see Degris et al. [2013] for more on this.

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).

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

Problem (`grpo_off_policy`): 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.

References

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. URL `https://arxiv.org/abs/2103.03874`.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. URL `https://arxiv.org/abs/2112.00114`.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL `https://arxiv.org/abs/2201.11903`.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel

Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL `https://arxiv.org/abs/2412.16720`.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, and Zonghan Yang. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL `https://arxiv.org/abs/2501.12599`.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL `https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf`.

Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL `https://github.com/huggingface/open-r1`.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild, 2025. URL `https://arxiv.org/abs/2503.18892`.

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. Tinyzero. https://github.com/Jiayi-Pan/TinyZero, 2025. Accessed: 2025-01-24.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement, 2024. URL `https://arxiv.org/abs/2409.12122`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. arXiv:2309.06180.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025. URL `https://arxiv.org/abs/2503.20783`.

Joshua Achiam. Spinning up in deep reinforcement learning. 2018a.

Nathan Lambert. Reinforcement learning from human feedback, 2024. URL `https://rlhfbook.com`.

Sheldon M Ross. *Simulation*. academic press, 2022.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL `https://arxiv.org/abs/2402.03300`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL `https://arxiv.org/abs/1707.06347`.

Joshua Achiam. Simplified ppo-clip objective, 2018b. URL `https://drive.google.com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/view`.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL `https://arxiv.org/abs/2503.14476`.

Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic, 2013. URL `https://arxiv.org/abs/1205.4839`.