CS336 作业 5（对齐）：对齐和推理 RL

版本1.0.2

CS336工作人员

2025 年春季

1 作业概述

在本作业中，您将获得一些训练语言模型在解决数学问题时进行推理的实践经验。

你将实施什么。

1. 竞赛数学问题 MATH 数据集的零样本提示基线 Hendrycks 等人。 [2021]。 2. 监督微调，给出来自更强推理模型的推理轨迹（DeepSeek R1、DeepSeek-AI 等人，2025）。 3. 专家迭代通过经过验证的奖励来提高推理性能。

4. 组相对策略优化（GRPO），用于通过经过验证的奖励来提高推理性能。

对于那些感兴趣的人，我们将在作业中提供一个完全可选的部分，即根据人类偏好调整语言模型，该部分将在未来几天内发布。

你将运行什么。

1. 衡量 Qwen 2.5 Math 1.5B 零样本提示表现（我们的基准）。 2. 使用来自 R1 的推理轨迹在 Qwen 2.5 Math 1.5B 上运行 SFT。 3. 在 Qwen 2.5 Math 1.5B 上运行专家迭代并获得经过验证的奖励。 4. 在 Qwen 2.5 Math 1.5B 上运行 GRPO，并获得经过验证的奖励。

代码是什么样的。所有作业代码以及这篇文章都可以在 GitHub 上找到：

`github.com/stanford-cs336/assignment5-alignment`

请`git clone`存储库。如果有任何更新，我们会通知您，您可以`git pull`获取最新信息。

1. `cs336_alignment/*`：您将在此处编写作业 5 的代码。请注意，这里没有任何代码（除了一些入门代码），因此您应该能够从头开始执行您想做的任何操作。

2. `cs336_alignment/prompts/*`：为了您的方便，我们提供了带有提示的文本文件，以尽量减少将 PDF 中的提示复制并粘贴到代码中可能导致的错误。 3. `tests/*.py`：这包含您必须通过的所有测试。您只需通过 **`tests/test_sft.py`** 和 **`tests/test_grpo.py`** 中的测试 - 其余测试针对作业的非强制性部分。这些测试调用 `tests/adapters.py` 中定义的钩子。您将实现适配器以将代码连接到测试。编写更多测试和/或修改测试代码有助于调试代码，但您的实现预计会通过原始提供的测试套件。 4. `README.md`：此文件包含有关设置环境的一些基本说明。

你能用什么。我们希望您从头开始构建大多数 RL 相关组件。您可以使用 vLLM 等工具从语言模型生成文本（第 3.1 节）。此外，您可以使用 HuggingFace Transformers 加载 Qwen 2.5 Math 1.5B 模型和分词器并运行前向传递（第 4.1 节），但您不得使用任何训练实用程序（例如 `Trainer` 类）。

如何提交。您将向 Gradescope 提交以下文件：

- `writeup.pdf`：回答所有书面问题。请排版您的回复。
- `code.zip`：包含您编写的所有代码。

2 用语言模型进行推理

2.1动机

语言模型的显着用例之一是构建可以处理各种自然语言处理任务的通才系统。在本次作业中，我们将重点关注语言模型的一个开发用例：数学推理。它将作为我们设置评估、执行监督微调以及使用强化学习 (RL) 进行推理教学实验的测试平台。

我们过去完成任务的方式将有两个不同之处。

- 首先，我们不会使用之前的语言模型代码库和模型。理想情况下，我们希望使用从以前的作业中训练出来的基本语言模型，但微调这些模型不会给我们带来令人满意的结果——这些模型太弱，无法显示重要的数学推理能力。因此，我们将切换到我们可以访问的现代高性能语言模型（Qwen 2.5 Math 1.5B Base），并在该模型之上完成大部分工作。

- 其次，我们将引入一个新的基准来评估我们的语言模型。到目前为止，我们已经接受了这样的观点：交叉熵是许多下游任务的良好替代品。然而，这项任务的目的是弥合基础模型和下游任务之间的差距，因此我们必须使用与交叉熵分开的评估。我们将使用 Hendrycks 等人的 MATH 12K 数据集。 [2021]，其中包括具有挑战性的高中数学竞赛问题。我们将通过将语言模型输出与参考答案进行比较来评估它们。

2.2℃ 思路推理和推理 克RL

语言模型最近一个令人兴奋的趋势是使用 *chain-of-thought* 推理来提高各种任务的性能。思维链是指逐步推理问题的过程，在得出最终答案之前产生中间推理步骤。

法学硕士的思维链推理。早期的思想链方法通过使用“便签本”将问题分解为中间步骤来微调语言模型来解决算术等简单的数学任务[Nye et al., 2021]。其他工作促使强大的模型在回答之前“一步一步思考”，发现这可以显着提高小学数学问题等数学推理任务的表现[Wei et al., 2023]。

通过专家迭代学习推理。自学推理机 (STaR) [Zelikman 等人，2022] 将推理构建为引导循环：预训练模型首先对不同的思想链 (CoT) 进行采样，仅保留那些导致正确答案的思想链，然后对这些“专家”痕迹进行微调。迭代这个循环可以提高LM的推理能力和解决率。 STaR 证明，这个版本的专家迭代 [Anthony et al., 2017] 使用自动的、基于字符串匹配的生成答案验证可以引导推理技能，而无需人工编写的推理痕迹。

使用经过验证的奖励、o1 和 R1 进行推理 RL。最近的工作探索了使用更强大的强化学习算法和经过验证的奖励来提高推理性能。 OpenAI 的 o1（以及后续的 o3/o4）[OpenAI 等人，2024]、DeepSeek 的 R1 [DeepSeek-AI 等人，2025] 和 Moonshot 的 kimi k1.5 [Team 等人，2025] 使用策略梯度方法 [Sutton 等人，1999] 来训练数学和代码任务，其中字符串匹配或单元测试验证正确性，展示了在竞赛数学和编码表现。后来的工作，如 Open-R1 [Face, 2025]、SimpleRL-Zoo [Zeng et al., 2025] 和 TinyZero [Pan et al., 2025] 证实，具有经过验证的奖励的纯强化学习（即使在小至 1.5B 参数的模型上）也可以提高推理性能。

我们的设置：模型和数据集。在下面的部分中，我们将考虑逐步更复杂的方法来训练基本语言模型以逐步推理以解决数学问题。对于本次作业，我们将使用 Qwen 2.5 Math 1.5B Base 模型，该模型是根据 Qwen 2.5 1.5B 模型在高质量综合数学预训练数据上不断进行预训练的 [Yang et al., 2024]。 MATH 数据集可在位于 `/data/a5-alignment/MATH` 的 Together 集群上获取。

给开源审计员的提示：替代数据集

不幸的是，由于版权声明，MATH 数据集并未公开。如果您在家中进行操作，则可以使用以下开源数学推理数据集之一：

- Countdown [Pan et al., 2025]，可在此处获取：基于英国电视节目 Countdown 的简单合成任务，该任务已成为小规模推理 RL 的流行测试平台。
- GSM8K [Cobbe et al., 2021a]，可在此处获取：小学数学问题，这比数学更容易，但应该允许您调试正确性并熟悉推理 RL 管道。
- Tulu 3 SFT Math [Lambert et al., 2025]，可在此处获取：使用 GPT-4o 和 Claude 3.5 Sonnet 生成的综合数学问题。由于这些是综合性的，因此某些答案（甚至问题）可能不完全正确。
- 此处链接了一些其他数学 SFT 数据集。

要获取简短的真实值标签（例如，`1/2`）（如果未直接提供），您可以使用数学答案解析器（例如 Math-Verify）处理真实值列。

3 测量零样本数学性能

我们将首先在 5K MATH 示例测试集上测量基本语言模型的性能。建立此基线有助于理解后面的每种方法如何影响模型行为。

除非另有说明，对于 MATH 实验，我们将使用 DeepSeek R1-Zero 模型 [DeepSeek-AI et al., 2025] 中的以下提示。我们将其称为 `r1_zero` 提示符：

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

`r1_zero` 提示符位于文本文件 `cs336_alignment/prompts/r1_zero.prompt` 中。

在提示中，`question` 指的是我们插入的一些问题（例如，`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?`）。我们期望模型扮演助手的角色，并开始生成思考过程（因为我们已经包含了左侧思考标签 `<think>`），用 `</think>` 结束思考过程，然后在答案标签内生成最终的符号答案，例如 `<answer> 4x + 10 </answer>`。让模型生成像 `<answer> </answer>` 这样的标签的目的是为了让我们可以轻松解析模型的输出并将其与真实答案进行比较，这样当我们看到正确的答案标签 `</answer>` 时就可以停止生成响应。

注意提示选择。事实证明，`r1_zero` 提示并不是 RL 后最大化下游性能的最佳选择，因为该提示与 Qwen 2.5 Math 1.5B 模型的预训练方式不匹配。刘等人。 [2025] 发现，简单地用问题（而不是其他）提示模型，一开始就能获得非常高的准确度，例如，在 100+ 步 RL 后匹配 `r1_zero` 提示。他们的研究结果表明，Qwen 2.5 Math 1.5B 已经针对此类问答对进行了预训练。

尽管如此，我们为这项任务选择了 `r1_zero` 提示，因为使用它的 RL 在短时间内显示出明显的准确性改进，使我们能够快速了解 RL 的机制并检查正确性，即使我们没有达到最佳的最终性能。作为现实检查，您将在作业后面直接与 `question_only` 提示进行比较。

3.1 使用vLLM进行离线语言模型推理

为了评估我们的语言模型，我们必须为各种提示生成延续（响应）。虽然人们当然可以实现自己的生成函数（例如，正如您在作业 1 中所做的那样），但 RL 的有效实现需要高性能推理技术，而实现这些推理技术超出了本作业的范围。因此，在本次作业中，我们建议使用 vLLM 进行离线批量推理。 vLLM 是一种用于语言模型的高吞吐量和内存高效的推理引擎，它结合了各种有用的效率技术（例如，优化的 CUDA 内核、用于高效注意力 KV 缓存的 PagedAttention [Kwon 等人，2023] 等）。要使用 vLLM 生成提示列表的延续：[1]

```python
from vllm import LLM, SamplingParams

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

[1]Example taken from https://github.com/vllm-project/vllm/blob/main/examples/offline_inference.py.

```
# Create a sampling params object, stopping generation on newline.
sampling_params = SamplingParams(
    temperature=1.0, top_p=1.0, max_tokens=1024, stop=["\n"]
)

# Create an LLM.
llm = LLM(model=<path to model>)

# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)

# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

在上面的示例中，可以使用 HuggingFace 模型的名称（如果本地未找到，将自动下载并缓存）或 HuggingFace 模型的路径来初始化 `LLM`。由于下载可能需要很长时间（特别是对于较大的模型，例如 70B 参数）并且为了节省集群磁盘空间（因此每个人都没有自己独立的预训练模型副本），我们在 Together 集群上的以下路径下载了以下预训练模型。请不要在 Together 集群上重新下载这些模型:

- Qwen 2.5 Math 1.5B Base（用于推理实验）：
  `/data/a5-alignment/models/Qwen2.5-Math-1.5B`
- Llama 3.1 8B Base（用于可选指令调整实验）：
  `/data/a5-alignment/models/Llama-3.1-8B`
- Llama 3.3 70B Instruct（用于可选的指令调整实验）：
  `/data/a5-alignment/models/Llama-3.3-70B-Instruct`

3.2 零样本数学基线

提示设置。为了评估 MATH 测试集上的零样本性能，我们只需加载示例并使用上面的 `r1_zero` 提示提示语言模型回答问题。

评估指标。当我们评估多项选择或二元响应任务时，评估指标很明确——我们测试模型是否输出完全正确的答案。

在数学问题中，我们假设存在已知的基本事实（例如 `0.5`），但我们不能简单地测试模型输出是否恰好为 0.5，它也可以回答 `<answer> 1/2 </answer>`。因此，当我们评估 MATH 时，我们必须解决匹配来自 LM 的语义等效响应的棘手问题。

为此，我们希望提出一些答案解析函数，该函数将模型的输出和已知的事实真相作为输入，并返回一个布尔值来指示模型的输出是否正确。例如，奖励函数可以接收模型以 `<answer> She sold 15 clips. </answer>` 结尾的字符串输出和黄金答案 `72`，如果模型的输出正确则返回 `True`，否则返回 `False`（在这种情况下，它应该返回 `False`）。

对于我们的数学实验，我们将使用最近的 RL 推理工作中使用的快速且相当准确的答案解析器 [Liu et al., 2025]。此奖励函数在 `cs336_alignment.drgrpo_grader.r1_zero_reward_fn` 中实现，除非另有说明，否则您应该使用它来评估 MATH 的表现。

生成超参数。生成响应时，我们将以温度 1.0、top-p 1.0、最大生成长度 1024 进行采样。提示要求模型以字符串 `</answer>` 结束其答案，因此我们可以指示 vLLM 在模型输出此字符串时停止：

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

问题（**`math_baseline`**）：4分

(a) 编写一个脚本来评估 Qwen 2.5 Math 1.5B 在 MATH 上的零样本性能。该脚本应该 (1) 从 `/data/a5-alignment/MATH/validation.jsonl` 加载 MATH 验证示例 ，，，，
(2) 使用 `r1_zero` 提示将它们格式化为语言模型的字符串提示，以及 (3) 为每个示例生成输出。该脚本还应该 (4) 计算评估指标并 (5) 将示例、模型生成和相应的评估分数序列化到磁盘，以便在后续问题中进行分析。

包含带有类似于以下参数的方法 `evaluate_vllm` 可能会对您的实现有所帮助，因为您稍后可以重用它：

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

可交付成果：评估基线零样本数学性能的脚本。

(b) 在 Qwen 2.5 Math 1.5B 上运行评估脚本。有多少代模型属于以下类别：(1) 格式和答案奖励 1 均正确，(2) 格式奖励 1 和答案奖励 0，(3) 格式奖励 0 和答案奖励 0？观察至少 10 个格式奖励为 0 的情况，您认为问题出在基础模型的输出上，还是解析器上？为什么？在（至少 10 种）格式奖励为 1 但答案奖励为 0 的情况下该怎么办？

可交付成果：对模型和奖励函数性能的评论，包括每个类别的示例。

(c) Qwen 2.5 Math 1.5B 零样本基线在 MATH 上的表现如何？

可交付成果：1-2 个带有评估指标的句子。

4 Supervised Finetuning for MATH

```
算法 1 监督微调 (SFT)
输入初始策略模型$\pi_{\theta_{\text{init}}}$；  SFT 数据集 $\mathcal{D}$
 1: 策略模型 $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$ 2: 对于步骤 $= 1, \cdots,$ n_sft_steps 执行 3: 从 $\mathcal{D}$ 中采样一批问题-响应对 $\mathcal{D}_b$ 4: 使用
 模型 $\pi_\theta$ 计算给定问题的响应的交叉熵损失 5: 通过相对于交叉熵损失采取梯度步骤来更新模型参数 $\theta$ 6: 结
 束
输出 $\pi_\theta$
```

推理的监督微调。在本节中，我们将在 MATH 数据集上微调我们的基本模型（算法 1）。由于我们的目标是提高模型的推理能力，而不是对其进行微调以直接预测正确答案，因此我们将对其进行微调以首先生成一条思路推理轨迹，然后生成答案。为此，我们提供了从 DeepSeek R1 DeepSeek-AI 等人获得的此类推理轨迹的数据集。 [2025]，在

`/data/a5-alignment/MATH/sft.jsonl`

在实践中训练推理模型时，SFT 通常用作第二个 RL 微调步骤的热启动。造成这种情况的主要原因有两个。首先，SFT 需要高质量的注释数据（即具有预先存在的推理轨迹），而 RL 仅需要正确答案进行反馈。其次，即使在注释数据丰富的环境中，强化学习仍然可以通过找到比 SFT 数据更好的策略来释放性能提升。不幸的是，我们使用的模型不够大，无法在组合 SFT 和 RL 时显示效果，因此对于本次作业，我们将分别处理这两个阶段。

4.1 使用 HuggingFace 模型

加载 HuggingFace 模型和分词器。要从本地目录加载 HuggingFace 模型和分词器（在 `bfloat16` 中并使用 FlashAttention-2 来节省内存），您可以使用以下起始代码：

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained(
    "/data/a5-alignment/models/Qwen2.5-Math-1.5B",
    torch_dtype=torch.bfloat16,
    attn_implementation="flash_attention_2",
)
tokenizer = AutoTokenizer.from_pretrained("/data/a5-alignment/models/Qwen2.5-Math-1.5B")
```

向前传球。加载模型后，我们可以对一批输入 ID 运行前向传递，并获取输出的 logits（带有 `.logits`）属性。然后，我们可以计算模型的预测逻辑与实际标签之间的损失：

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

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

保存经过训练的模型。要在训练完成后将模型保存到目录中，可以使用 `.save_pretrained()` 函数，传入所需输出目录的路径。确保保存在 `/data/yourusername` 下，因为它们可能非常大。我们建议还保存标记生成器（即使您没有修改它），以便模型和标记生成器是独立的并且可以从单个目录加载。

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

梯度积累。尽管在 `bfloat16` 中加载模型并使用 FlashAttention-2，但即使是 80GB GPU 也没有足够的内存来支持合理的批量大小。要使用更大的批量大小，我们可以使用一种称为 *gradient accumulation* 的技术。梯度累积背后的基本思想是，我们不是在每个批次之后更新模型权重（即采取优化器步骤），而是在采取梯度步骤之前对多个批次的梯度进行 *accumulate* 处理。直观上，如果我们有更大的 GPU，我们应该一次性计算一批 32 个示例的梯度得到相同的结果，而不是将它们分成 16 批，每批 2 个示例，然后在最后取平均值。

梯度累积在 PyTorch 中实现起来很简单。回想一下，每个权重张量都有一个存储其梯度的属性 `.grad`。在我们调用 `loss.backward()` 之前，`.grad` 属性为 None。在我们调用 `loss.backward()` 之后，`.grad` 属性包含渐变。通常，我们会采取优化器步骤，然后使用 `optimizer.zero_grad()` 将梯度归零，这会重置权重张量的 `.grad` 字段：

```
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()
```

为了实现梯度累积，我们只需每 k 步调用 `optimizer.step()` 和 `optimizer.zero_grad()`，其中 k 是梯度累积步数。在调用 `loss.backward()` 之前，我们将损失除以 `gradient_accumulation_steps`，以便在梯度累积步骤中对梯度进行平均。

```
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()
```

因此，训练时的有效批量大小乘以梯度累积步数 k。

4.2 SFT 辅助方法

接下来，我们将实现一些辅助方法，您将在 SFT 和后面的 RL 实验中使用它们。关于术语的快速说明：在以下部分中，我们将在给定提示的情况下将模型的完成互换称为“输出”、“完成”或“响应”。

对提示和输出进行标记。对于每对问题和目标输出（q, o），我们将分别标记问题和输出并将它们连接起来。然后，我们可以使用我们的 SFT 模型（或者在后面的部分中，我们的 RL 策略）对输出的对数概率进行评分。此外，我们需要构造一个 `response_mask`：一个布尔掩码，对于响应中的所有标记为 **`True`**，对于所有问题和填充标记为 **`False`**。我们将在训练循环中使用这个掩码，以确保我们只计算响应令牌的损失。

问题（**`tokenize_prompt_and_output`**）：提示和输出标记化（2分）

可交付成果：实现一个方法 `tokenize_prompt_and_output`，该方法分别标记问题和输出，将它们连接在一起，并构造一个 `response_mask`。推荐使用以下接口：

`def tokenize_prompt_and_output(prompt_strs, output_strs, tokenizer):` 对提示和输出字符串进行标记，并构造一个掩码，该掩码对于响应标记为 1，对于其他标记（提示或填充）为 0。

args：

`prompt_strs: list[str]` 提示字符串列表。
`output_strs: list[str]` 输出字符串列表。

`tokenizer: PreTrainedTokenizer` 用于标记化的标记器。

返回：

`dict[str, torch.Tensor]`. 令 `prompt_and_output_lens` 为包含标记化提示和输出字符串长度的列表。然后返回的字典应该具有以下键： **`input_ids`** torch.Tensor of shape `(batch_size, max(prompt_and_output_lens) - 1)`：标记化的提示和输出字符串，最终标记被切掉。 **`labels`** torch.Tensor 形状为 `(batch_size, max(prompt_and_output_lens) - 1)`：移位后的输入 id，即没有第一个标记的输入 id。 **`response_mask`** torch.Tensor，形状为 `(batch_size, max(prompt_and_output_lens) - 1)`：标签中响应标记的掩码。

要测试您的代码，请实施 `[adapters.run_tokenize_prompt_and_output]`。然后，使用 `uv run pytest -k test_tokenize_prompt_and_output` 运行测试并确保您的实现通过它。

记录每个令牌的熵。在进行强化学习时，跟踪每个标记的熵通常很有用，以查看模型的预测分布是否变得（过度）自信。我们现在将实现这一点，并比较每种微调方法如何影响模型的预测熵。

支持 $\mathcal{X}$ 的离散分布 $p(x)$ 的熵定义为

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

给定我们的 SFT 或 RL 模型的 logits，我们将计算每个令牌的熵，即每个下一个令牌预测的熵。

问题 (**compute_entropy**)：每个令牌的熵（1 分）

可交付成果：实现一种方法 `compute_entropy`，用于计算下一个令牌预测的每个令牌熵。推荐使用以下接口：

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

获取下一个标记预测的熵（即词汇维度上的熵）。

args：

logits: torch.Tensor 形状为 `(batch_size, sequence_length, vocab_size)` 的张量，包含未归一化的 logits。

返回：

torch.Tensor 形状`(batch_size, sequence_length)`。每个下一个标记预测的熵。

注意：您应该使用数值稳定的方法（例如，使用 logsumexp）以避免溢出。

要测试您的代码，请实施 `[adapters.run_compute_entropy]`。然后运行 `uv run pytest -k test_compute_entropy` 并确保您的实现通过。

从模型中获取对数概率。从模型中获取对数概率是 SFT 和 RL 中都需要的一个原语。

对于前缀 x、LM 生成下一个令牌 logits $f_\theta(x) \in \mathbb{R}^{|\mathcal{V}|}$ 和标签 $y \in \mathcal{V}$，y 的对数概率为

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

其中符号 $[x]_y$ 表示向量 x 的第 y 个元素。

您将需要使用数值稳定的方法来计算它，并且可以自由使用 `torch.nn.functional` 中的方法。我们还建议包含一个参数来可选地计算和返回令牌熵。

问题 (**get_response_log_probs**)：响应对数概率（和熵）（2 分）

可交付成果：实现一种方法 `get_response_log_probs`，该方法从因果语言模型获取每个标记的条件对数概率（给定先前的标记），以及可选的模型的下一个标记分布的熵。推荐使用以下接口：

```
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 用于评分的 HuggingFace 模型（如果不应计算梯度，则放置在正确的设备上并处于推理模式）。**input_ids: torch.Tensor** 形状 (batch_size, sequence_length)，由您的标记化方法生成的串联提示 + 响应标记。**labels: torch.Tensor** 形状 (batch_size, sequence_length)，由标记化方法生成的标签。**return_token_entropy: bool** 如果 **True**，还通过调用 compute_entropy 返回每个令牌的熵。

返回：

dict[str, torch.Tensor].

  "log_probs" 形状 (batch_size, sequence_length)，条件对数概率 $\log p_\theta(x_t \mid x_{<t})$。**"token_entropy"** 可选，形状 (batch_size, sequence_length)，每个位置的每个标记熵（仅在 return_token_entropy=**True** 时出现）。

实施技巧：

- 使用 model(input_ids).logits 获取 logits。

要测试您的代码，请实施 [adapters.run_get_response_log_probs]。然后运行 uv run pytest -k test_get_response_log_probs 并确保测试通过。

SFT 微批次训练步骤。我们在 SFT 中最小化的损失是给定提示的目标输出的负对数似然。为了计算这个损失，我们需要计算给定提示的目标输出的对数概率以及输出中所有标记的总和，屏蔽提示中的标记和填充标记。

我们将为此实现一个辅助函数，稍后我们也会在强化学习期间使用它。

问题 (**masked_normalize**)：屏蔽标准化（1 分）

可交付成果：实现一种方法 masked_normalize，该方法对张量元素求和并通过常数进行归一化，同时遵循布尔掩码。推荐使用以下接口：

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

对维度求和并通过常数进行归一化，仅考虑其中 `mask == 1` 的元素。

args：

`tensor: torch.Tensor` 要求和并标准化的张量。

`mask: torch.Tensor` 与tensor形状相同；具有 1 的位置包含在总和中。
`normalize_constant: float` 用于标准化的常量。

`dim: int | None` 标准化之前要求和的维度。如果没有，则对所有维度求和。

返回：

`torch.Tensor` 归一化总和，其中屏蔽元素 (`mask == 0`) 对总和没有贡献。

要测试您的代码，请实施 `[adapters.run_masked_normalize]`。然后运行 `uv run pytest -k test_masked_normalize` 并确保它通过。

SFT 微批次训练步骤。我们现在准备好为 SFT 实现单个微批次训练步骤（回想一下，对于训练小批次，我们迭代许多微批次 if `gradient_accumulation_steps` > 1）。

问题 (**`sft_microbatch_train_step`**)：微批量训练步骤（3 分）

可交付成果：为 SFT 实现单个微批量更新，包括交叉熵损失、使用掩码求和以及梯度缩放。

推荐使用以下接口：

```
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]]:
```

对微批次执行前向和后向传递。
args：

`policy_log_probs` (`batch_size, sequence_length`)，来自正在训练的 SFT 策略的每个令牌的对数概率。 **`response_mask`** (`batch_size, sequence_length`)、1 用于响应令牌，0 用于提示/填充。 **`gradient_accumulation_steps`** 每个优化器步骤的微批次数。
`normalize_constant` 用于除以总和的常数。最好将其保留为 1.0。

返回：

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

损失标量张量。根据梯度累积调整微批次损失。我们返回这个以便我们可以记录它。元数据 包含来自底层丢失调用的元数据以及您可能想要记录的任何其他统计信息的字典。实施技巧 :

- 您应该在此函数中调用 `loss.backward()`。确保调整梯度累积。要测试您的代码，请实施 [`adapters.run_sft_microbatch_train_step`]。然后运行`uv run pytest -k test_sft_microbatch_train_step`并确认通过。

在循环中记录各代。进行一些涉及模型生成的循环内日志记录始终是一种很好的做法，推理 SFT/RL 也不例外。编写一个函数 `log_generations`，它将提示您的模型为某些给定的提示生成响应（例如，从验证集中采样）。最好为每个示例至少记录以下内容:

1.输入提示。 2. SFT/RL 模型生成的响应。 3. 真实答案。 4.奖励信息，包括格式、答案、奖励总额。 5. 响应的平均标记熵。 6. 平均回答长度、正确回答的平均回答长度、错误回答的平均回答长度。

问题 (`log_generations`)：记录生成（1 分）

可交付成果：实现一个函数 `log_generations`，可用于记录模型中的生成。

4.3 SFT实验

使用上面的部分，您现在将实现完整的 SFT 过程（算法 1），以在 MATH 数据集上微调 Qwen 2.5 Math 1.5 B 基础模型。 `/data/a5-alignment/MATH/sft.jsonl` 中的每个示例都由格式化提示和目标响应组成，其中目标响应包括思路推理轨迹和最终答案。特别是，每个示例都是类型为 `{"prompt": str, "response": str}` 的 JSON 元素。

为了跟踪模型在训练过程中的进度，您应该定期在 MATH 验证集上对其进行评估。您应该使用 2 个 GP U 运行脚本，其中一个 GPU 用于策略模型，另一个用于 vLLM 实例以评估策略。为了使其正常工作，这里有一些起始代码，用于初始化 vLLM 并在每个推出阶段之前将策略权重加载到 vLLM 实例中:

```
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())
```

您可能会发现记录有关训练和验证步骤的指标很有帮助（这在以后的 RL 实验中也很有用）。要在 `wandb` 中执行此操作，您可以使用以下代码：

```
# 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")
```

最后，我们建议您使用剪辑值为 1.0 的渐变剪辑。

问题 (`sft_experiment`)：在 MATH 数据集上运行 SFT（2 分）（2 H100 小时）

---

1. 使用 Qwen 2.5 Math 1.5B 基本模型对推理 SFT 示例（在 `/data/a5-alignment/MATH/sft.jsonl` 中提供）运行 SFT，改变 SFT 的唯一示例数量

范围 {128, 256, 512, 1024}，以及使用完整的数据集。使用完整数据集时，调整学习率和批量大小以实现至少 15% 的验证准确度。可交付成果：与不同数据集大小相关的验证精度曲线。 2. 过滤推理 SFT 示例，仅包含产生正确答案的示例。在（完整）过滤的数据集上运行 SFT，并报告过滤后的数据集的大小以及您实现的验证准确性。可交付成果：报告数据集的大小以及您实现的验证准确性曲线。将您的发现与之前的 SFT 实验进行比较。

5 数学专家迭代

在上一节中，我们观察到可以通过从 SFT 数据中过滤掉不良示例来提高 SFT 模型的性能。在本节中，我们将更进一步：我们将将此过滤过程应用于从基本模型本身生成的推理跟踪。这个过程在文献中被称为 *expert iteration* [Anthony et al., 2017]，并且 Cobbe 等人在语言模型的背景下进行了探索。 [2021b]，Zelikman 等人。 [2022]，Dohan 等人。 [2022]，Gulcehre 等人。 [2023]。

```
算法2 专家迭代（EI）
输入初始策略模型$\pi_{\theta_{\text{init}}}$；奖励函数$R$；任务问题 $\mathcal{D}$
1: 策略模型 $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$ 2: 对于步骤 $= 1, \cdots,$ n_ei_steps 执行 3: 从 $\mathcal{D}$ 中采样一批问题 $\mathcal{D}_b$ 4: 设置旧的策略模型 $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$ 5: 对每个问题 $q \in \mathcal{D}_b$ 采样 $G$ 输出 $\{o^{(i)}\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ 6: 通过运行奖励函数 $R(q, o^{(i)})$ 计算每个采样输出 $o^{(i)}$ 的奖励 $\{r^{(i)}\}_{i=1}^G$ 7: 过滤掉错误的输出（即，$o^{(i)}$ 与 $r^{(i)} = 0$）获得正确问题-回答对 8 的数据集 $\mathcal{D}_{\text{sft}}$：$\pi_\theta \leftarrow$ SFT($\pi_\theta, \mathcal{D}_{\text{sft}}$)（算法 1）

9：结束
输出 $\pi_\theta$
```

接下来，我们将对 MATH 数据集运行专家迭代。

作为提示，您应该将 `min_tokens` 值传递给 vLLM `SamplingParams`，这将确保您不会生成空字符串（这可能会导致下游出现 NaN，具体取决于您的实现）。这可以通过以下方式完成

```
sampling_min_tokens = 4
sampling_params = SamplingParams(
      temperature=sampling_temperature,
      max_tokens=sampling_max_tokens,
      min_tokens=sampling_min_tokens,
      n=G,
      seed=seed,
  )
```

与 SFT 中一样，您应该使用剪辑值为 1.0 的渐变剪辑。

问题 (`expert_iteration_experiment`)：在 MATH 数据集上运行专家迭代（2 分）（6 H100 小时）

Run expert iteration on the MATH dataset (provided at `/data/a5-alignment/MATH/train.jsonl`)

使用 Qwen 2.5 Math 1.5B 基础模型，改变每个问题的推出次数 G 和 SFT 步骤中使用的纪元数，并使用 `n_ei_steps` $= 5$. 改变 {512, 1024, 2048} 中每个专家迭代步骤的批量大小（即 $\mathcal{D}_b$ 的大小）。（您不需要尝试这些超参数的所有可能组合。只要对每个超参数得出结论就可以了。）记录模型在训练过程中的响应的熵。确保让 vLLM 在第二个答案标记 `</answer>` 处终止生成，如 SFT 部分中所做的那样。

可交付成果：与不同推出配置相关的验证精度曲线。尝试至少 2 种不同的推出计数和纪元计数。

可交付成果：数学验证准确率至少达到 15% 的模型。
可交付成果：与您的 SFT 表现以及跨 EI 步骤的表现进行比较的简短 2 句话讨论。

可交付成果：模型在训练过程中的响应熵图。

6 政策梯度入门

语言模型研究中一个令人兴奋的新发现是，使用强大的基础模型针对经过验证的奖励执行强化学习可以显着提高其推理能力和性能[OpenAI et al., 2024, DeepSeek-AI et al., 2025]。最强大的开放推理模型，例如 DeepSeek R1 和 Kimi k1.5 [Team et al., 2025]，是使用策略梯度进行训练的，策略梯度是一种强大的强化学习算法，可以优化任意奖励函数。

我们在下面简要介绍了语言模型上 RL 的策略梯度。我们的演示紧密基于一些更深入地介绍这些概念的优秀资源：OpenAI 的 Spinning Up in Deep RL [Achiam，2018a] 和 Nathan Lambert 的《人类反馈强化学习 (RLHF)》一书 [Lambert，2024]。

6.1 作为策略的语言模型

具有参数 θ 的因果语言模型 (LM) 定义了给定当前文本前缀 s_t（状态/观察值）的下一个标记 $a_t \in \mathcal{V}$ 的概率分布。在 RL 的上下文中，我们将下一个标记 a_t 视为 *action*，将当前文本前缀 s_t 视为 *state*。因此，LM 是一个 *categorical stochastic policy*

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

使用策略梯度优化策略时需要两个原始操作：

1. *Sampling from the policy*：从上面的分类分布中得出一个动作a_t；
2. *Scoring the log-likelihood of an action*：评估日志$\pi_\theta(a_t \mid s_t)$。

通常，当使用 LLM 进行 RL 时，s_t 是迄今为止产生的部分完成/解决方案，每个 a_t 是解决方案的下一个标记；当发出文本结束标记时，该剧集结束，例如 `<|end_of_text|>` 或 `</answer>`（在我们的 `r1_zero` 提示中）。

6.2 轨迹

（有限范围）轨迹是代理经历的状态和动作的交错序列：

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

其中 T 是轨迹的长度，即 a_T 是文本结束标记，或者我们已达到标记的最大生成预算。

初始状态取自起始分布 $s_0 \sim \rho_0(s_0)$；在 RL 与 LLM 的情况下，$\rho_0(s_0)$ 是格式化提示的分布。在一般设置中，状态转换遵循一定的环境

动态$s_{t+1} \sim P(\cdot \mid s_t, a_t)$。在使用 LLM 的 RL 中，环境是确定性的：下一个状态是与发出的令牌 $s_{t+1} = s_t \| a_t$ 连接的旧前缀。轨迹也称为*episodes*或*rollouts*；我们将互换使用这些术语。

6.3 奖励与回报

标量奖励 $r_t = R(s_t, a_t)$ 判断在状态 s_t 下采取的行动的直接质量。对于已验证域上的强化学习，标准做法是为中间步骤分配零奖励，为最终操作分配 *verified reward*

$r_T = R(s_T, a_T) :_= \begin{cases} 1 \text{ 如果轨迹 } s_T \| a_T \text{ 根据我们的奖励函数匹配真实值 } 0 \text{ 否则.} \\ \end{cases}$

return $R(\tau)$ 沿着轨迹聚合奖励。两个常见的选择是 *finite-horizon undis- counted* 返回

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

和 *infinite-horizon discounted* 返回

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

在我们的例子中，我们将使用未折扣的公式，因为剧集有一个自然的终止点（文本结束或最大生成长度）。

t 代理人的目标是最大化期望收益 RN

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

导致优化问题

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

6.4 普通政策梯度

接下来，让我们尝试在预期回报上学习策略参数 θ 和 *gradient ascent*:

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

我们将使用的核心身份是 REINFORCE 策略梯度，如下所示。

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

推导政策梯度。我们是如何得到这个方程的？为了完整起见，我们将在下面给出这个恒等式的推导。我们将使用一些身份。

1. 轨迹的概率由下式给出

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

因此，轨迹的对数概率为：

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

2. 对数导数技巧：

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

3. θ 中的环境项是恒定的。 ρ_0、$P(\cdot \mid \cdot)$ 和 $R(\tau)$ 不依赖于策略参数，因此

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

应用上述事实：

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

因此，代入轨迹的对数概率并利用环境项在 θ 中恒定的事实，我们得到 $vanilla$ 或 REINFORCE 策略梯度：

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

直观上，这个梯度将增加具有高回报的轨迹中每个动作的对数概率，否则会减少它们。

梯度的样本估计。给定通过采样起始状态 $s_0^{(i)} \sim \rho_0(s_0)$ 收集的一批 N 推出 $\mathcal{D} = \{\tau^{(i)}\}_{i=1}^{N}$，然后在环境中运行策略 π_θ，我们形成梯度的无偏估计器：

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

该向量用于梯度上升更新$\theta \leftarrow \theta + \alpha \widehat{g}$。

6.5 政策梯度基线

普通策略梯度的主要问题是梯度估计的高方差。缓解这种情况的常用技术是从奖励中减去仅依赖于状态的 $baseline$ 函数 b。这是一种 $control\ variate$ [Ross, 2022]：其想法是通过减去与其相关的项来减少估计量的方差，而不引入偏差。

让我们将基线策略梯度定义为：

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

例如，合理的基线是在策略价值函数 $V^\pi(s) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)|s_t = s]$，即如果我们从 $s_t = s$ 开始并从那里遵循策略 π_θ 的预期回报。那么，量（$R(\tau) - V^\pi(s_t)$）直观上就是实现的轨迹比预期好多少。

只要基线仅取决于状态，则基线政策梯度就是无偏的。我们可以通过将基线策略梯度重写为

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

关注基线项，我们看到

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

一般来说，得分函数的期望为零：$\mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0$。因此，式(1)中的表达式为： 24 是零并且

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

因此我们得出的结论是，基线政策梯度是无偏的。我们稍后将进行一个实验，看看基线是否可以提高下游性能。

关于政策梯度“损失”的说明。当我们在像 PyTorch 这样的框架中实现策略梯度方法时，我们将定义所谓的策略梯度损失 `pg_loss`，以便调用 `pg_loss.backward()` 将用我们的近似策略梯度 $\widehat{g}$ 填充模型参数的梯度缓冲区。在数学上，可以表示为

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

`pg_loss` 不是规范意义上的损失——在训练集或验证集上报告 `pg_loss` 作为评估指标是没有意义的，良好的验证 `pg_loss` 并不表明我们的模型概括得很好。 `pg_loss` 实际上只是一些标量，因此当我们调用 `pg_loss.backward()` 时，我们通过反向传播获得的梯度是近似策略梯度 $\widehat{g}$。

在进行强化学习时，您应该始终记录并报告训练和验证奖励。这些是“有意义的”评估指标，也是我们试图用策略梯度方法优化的指标。

6.6 离策略策略梯度

REINFORCE 是一种 *on-policy* 算法：训练数据是通过我们正在优化的相同策略收集的。为了了解这一点，让我们写出 REINFORCE 算法：

1. 从当前政策 π_θ 中抽取一批推出 $\{\tau^{(i)}\}_{i=1}^{N}$。
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. 使用计算出的梯度更新策略参数：$\theta \leftarrow \theta + \alpha\widehat{g}$。

我们需要做大量的推理来对一批新的推出进行采样，只需要采取一个梯度步骤。 LM 的行为通常不会在一步中发生显着变化，因此这种同策略方法的效率非常低。

离政策政策梯度。在离策略学习中，我们会从我们正在优化的策略之外的某些策略中进行采样。流行策略梯度算法的离策略变体（例如 PPO 和 GRPO）使用先前版本的策略 $\pi_{\theta_{\text{old}}}$ 的推出来优化当前策略 π_θ。离政策政策梯度估计为

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

这看起来像是普通策略梯度的重要性采样版本，具有重新加权项$\frac{\pi_\theta(a_t^{(i)}|s_t^{(i)})}{\pi_{\theta_{\text{老的}}}(a_t^{(i)}|s_t^{(i)})}$ 。
事实上，等式。 27 可以通过重要性采样并应用合理的近似值（只要 π_θ 和 $\pi_{\theta_{\text{old}}}$ 差异不大）得出：参见 Degri
s 等人。 [2013] 了解更多相关信息。

7 集团相关政策优化

接下来，我们将描述组相对策略优化 (GRPO)，它是策略梯度的变体，您将使用它来实现和实验以解决数学问题。

7.1 GRPO算法

优势估计。 GRPO 的核心思想是从策略 π_θ 中对每个问题的许多输出进行采样，并使用它们来计算基线。这很方便，因为我们避免了学习神经值函数 $V_\phi(s)$ 的需要，该函数很难训练，并且从系统角度来看很麻烦。对于问题 q 和组输出 $\{o^{(i)}\}_{i=1}^{G} \sim \pi_\theta(\cdot|q)$，令 $r^{(i)} = R(q, o^{(i)})$ 为第 i 个输出的奖励。 DeepSeekMath [Shao et al., 2024] 和 DeepSeek R1 [DeepSeek-AI et al., 2025] 将第 i 个输出的组归一化奖励计算为

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

其中 `advantage_eps` 是一个小常数，用于防止被零除。请注意，这个 $advantage\ A^{(i)}$ 对于响应中的每个标记都是相同的，即 $A_t^{(i)} = A^{(i)}, \forall t \in 1, \ldots, |o^{(i)}|$，因此我们在下面删除 t 下标。

高级算法。在我们深入研究 GRPO 目标之前，让我们首先通过编写 Shao 等人的算法来了解训练循环。 [202
4] 算法 3.[2]

GRPO 目标。 GRPO 目标结合了三个想法：

1. 离政策政策梯度，如式（1）所示。 27.
2. 使用组归一化计算优势 $A^{(i)}$，如式（1）所示。 28.
3. 裁剪机制，如近端策略优化（PPO，Schulman 等人[2017]）。

裁剪的目的是在单批卷展中采取许多梯度步骤时保持稳定性。它的工作原理是防止政策 π_θ 偏离旧政策太远
。

[2]This is a special case of DeepSeekMath's GRPO with a verified reward function, no KL term, and no iterative update of the reference and reward model.

Algorithm 3 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. 28)
 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. 29)
10:     end for
11: end for
Output $\pi_\theta$
```

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

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

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

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

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 and EI sections will also be reused for GRPO.

Computing advantages (group-normalized rewards). First, we will implement the logic to compute advantages for each example in a rollout batch, i.e., the group-normalized rewards. We will consider two possible ways to obtain group-normalized rewards: the approach presented above in Eq. 28, and a recent simplified approach.

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

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

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

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

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

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

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:

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

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

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

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

And here are a few additional tips:

- Remember to use the `r1_zero` prompt, and direct vLLM to stop generation at the second answer tag `</answer>`, as in the previous experiments.
- We suggest using `typer` for argument parsing.
- Use gradient clipping with clip value 1.0.
- You should routinely log validation rewards (e.g., every 5 or 10 steps). You should evaluate on at least 1024 validation examples to compare hyperparameters, as CoT/RL evaluations can be noisy.
- With our implementation of the losses, GRPO-Clip should only be used when off-policy (since it requires the old log-probabilities).
- In the off-policy setting with multiple epochs of gradient updates per rollout batch, it would be wasteful to recompute the old log-probabilities for each epoch. Instead, we can compute the old log-probabilities

once and reuse them for each epoch.

- You should not differentiate with respect to the old log-probabilities.
- You should log some or all of the following for each optimizer update:
  - The loss.
  - Gradient norm.
  - Token entropy.
  - Clip fraction, if off-policy.
  - Train rewards (total, format, and answer).
  - Anything else you think could be useful for debugging.

Problem (`grpo_train_loop`): GRPO train loop (5 points)

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.

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 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 (`grpo_learning_rate`): Tune the learning rate (2 points) (6 H100 hrs)

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: Validation reward curves associated with multiple learning rates.

Deliverable: A model that achieves validation accuracy of at least 25% on MATH.

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) (2 H100 hrs)

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) (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 implemented in `compute_group_normalized_rewards` and will now test.

Problem (`grpo_group_standard_deviation`): 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 $\widehat{g}$ to the policy gradient (besides the length and advantage normalization choices mentioned above).

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

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

Problem (`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.

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 (`grpo_off_policy_sweep`): 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

$$-\frac{\pi_\theta(o_t|q, o_{<t})}{\pi_{\theta_{\text{old}}}(o_t|q, o_{<t})}A_t. \tag{34}$$

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 `cs336_alignment/prompts/r1_zero.prompt`, we will instead use an extremely simple prompt at `cs336_alignment/prompts/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 `cs336_alignment/drgrpo_grader.py`.

Problem (grpo_prompt_ablation): 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.

9 Leaderboard: GRPO on MATH

As the last part of the (mandatory) assignment, you will experiment with approaches to obtain the highest validation rewards possible within 4 hours of training on 2 H100 GPUs.

Model. We will continue using the Qwen 2.5 Math 1.5B Base model.

Dataset. We will continue using the MATH train and validation dataset available on the cluster at `/data/a5-alignment/MATH/train.jsonl` and `/data/a5-alignment/MATH/validation.jsonl`. You are not allowed to use any other data or do SFT on reasoning chains from stronger models, etc. You must report validation accuracy on the entire validation set (all 5K examples), using the sampling hyperparameters given above (temperature 1.0, max tokens 1024). You are allowed to filter the train set, or design a curriculum over the data, as you desire. You must use the R1-Zero prompt for validation, and during validation, you must use exactly the `r1_zero_reward_fn` reward function provided in the starter code (you are allowed to develop another reward function for use during training if you wish).

Algorithm. You are free to tune hyperparameters or change the training algorithm entirely, as long as you do not use any extraneous data or another model (you are free to use more copies of the model if you want).

Systems optimizations. You might observe that in our simple GRPO implementation above, at least one GPU is always idle. You will likely find notable improvements by improving the systems characteristics of our pipeline. For example, you might consider lower precision for rollouts or training, `torch.compile`, and other systems optimizations. You are definitely not constrained to placing vLLM on a single device and the train policy on another device, and are encouraged to think of better ways to parallelize.

Ideas. For some ideas on possible improvements, see the following repos:

- veRL
- trl
- torchtune
- oat

On KL divergence. We also note that in the above experiments, we did not include a KL divergence term with respect to some reference model (usually this is a frozen SFT or pretrained checkpoint). In our experiments and others from the literature [Liu et al., 2025, NTT123, 2025], we found that omitting the KL term had no impact on performance while saving GPU memory (no need to store a reference model). However, many GRPO repos include it by default and you are encouraged to experiment with KL or other forms of regularization, **as long as you use Qwen 2.5 Math 1.5B Base or some model obtained through your algorithm for it**.

Problem (`leaderboard`): Leaderboard (16 points) (16 H100 hrs)

Deliverable: Report a validation accuracy obtained within 4 hours of training on 2 H100 GPUs and a screenshot of your validation accuracy with respect to wall-clock time, where the x-axis ends at $\leq$ 4 hours. As a reminder, we place the following constraints on your evaluation:

1. Your validation accuracy should be the average accuracy over the entire MATH validation set (all 5K examples).
2. You must use the R1-Zero prompt at validation time.
3. You must use temperature 1.0 and max tokens 1024 with vLLM for evaluation.
4. You must calculate validation accuracy by averaging the answer rewards produced by the `r1_zero_reward_fn` reward function provided in the starter code.

10 Epilogue

Congratulations on finishing the last assignment of the class! You should be proud of your hard work. We hope you enjoyed learning the foundations underlying modern language models by building their main components from scratch.

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`.

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`.

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`.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL `https://arxiv.org/abs/2203.14465`.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search, 2017. URL `https://arxiv.org/abs/1705.08439`.

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`.

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.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021a. URL https://arxiv.org/abs/2110.14168.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

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.

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.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021b. arXiv:2110.14168.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language model cascades, 2022. URL https://arxiv.org/abs/2207.10342.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (rest) for language modeling, 2023. URL https://arxiv.org/abs/2308.08998.

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.

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

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`.

NTT123. Grpo-zero. https://github.com/policy-gradient/GRPO-Zero, 2025. Accessed: 2025-05-22.