Assignment 2: GRPO

September 28, 2025

Introduction

In this assignment, you will learn how to implement a minimal, end-to-end reinforcement learning pipeline: you will explore Group Relative Policy Optimization (`GRPO`) on the Countdown task (Section 1.1). The pipeline includes building parsers and a reward function (Sections 1.2 and 1.3), computing group-normalized advantages and per-token losses (Section 1.4), and training with various RL techniques. You will also investigate the `Dr.GRPO` variant. In Problem 1, you will run experiments, report accuracy and reward trends, and analyze successes and failures. Problem 2 (Section 2) is optional extra credit for an open-ended extension.

Contents

1 GRPO for Countdown Math Problem **1**

1.1 Introduction and Experimental Setting 1

1.2 Task 1: Verifier Functions Implementation 2

1.3 Task 2: Reward Function 3

1.4 GRPO: Group Relative Policy Optimization 4

1.4.1 Task 3: `compute_group_normalized_advantages` 5

1.4.2 Task 4: `compute_loss` 6

1.5 Putting It Together: Training Loop, Grouping, and Microbatches 8

1.6 Experiments 10

1.6.1 Zero-Shot Baseline. 10

1.6.2 Main Experiments 10

2 Optional Extra Credit: Problem 2 - Open-Ended Investigation **10**

1 GRPO for Countdown Math Problem

For this problem, we will implement a **Group Relative Policy Optimization (GRPO)** training pipeline. This includes setting up the experimental environment, designing a reward function, implementing the core GRPO algorithm, and exploring its variant **Dr.GRPO**.

1.1 Introduction and Experimental Setting

Countdown Problem. The Countdown problem is a numerical reasoning task. Given a finite set of positive integers $S = \{n_1, n_2, \ldots, n_k\}$ and a target positive integer T, the language model must generate an expression E that evaluates to T. In our experiments, the size of the input number set is constrained to either three or four numbers ($k \in \{3, 4\}$).

The equation E must satisfy the following constraints:

1. **Integer Constraint:** Only the four basic arithmetic operations $+, -, *, /$ and parentheses are permitted.
2. **Usage Constraint:** Each number $n_i \in S$ must be used in E exactly once.

The model's output is the final correct mathematical equation.

```
Example #1
Input: [7, 7, 65]
Target: 51
Expected Output: (65 - 7) - 7

Example #2
Input: [34, 84, 83, 72]
Target: 39
Expected Output: (72 - 34) + (84 - 83)

Example #3
Input: [9, 4, 3, 7]
Target: 101
Expected Output: (9 * (4 * 3)) - 7
```

Experimental Setting. Our implementation will use the following model and public dataset:

- **Model:** Qwen/Qwen3-1.7B
- **Dataset:** justinphan3110/Countdown-Tasks-3to4
- **Compute Requirements:** An A100 GPU

1.2 Task 1: Verifier Functions Implementation

Given a generation $\mathcal{G}$ from our model M, we define verifier functions to extract and validate the final equation E.

_extract_answer Extract the content from the last occurrence of the `<answer>...</answer>` tag in the model's rollout generation $\mathcal{G}$.

```
def _extract_answer(solution_str: str) -> str | None:
    ### YOUR CODE HERE ###
    ...

# Input/Output Format:
rollout_response = "Okay, let's see. I need to use 79, 17, ... So the equation is
↪  <answer>(79 - (60 - 17))</answer>"
extracted_answer = _extract_answer(rollout_response)
assert extracted_answer == "(79 - (60 - 17))"
```

_validate_numbers Checks if the numbers used in the equation E are a valid **subset** of the available numbers S and that each number is used at most once.

```python
def _validate_numbers(equation_str: str, available_numbers: List[int]) -> bool:
    ### YOUR CODE HERE ###
    ...

# Input/Output Format:
# Example 1: Valid Usage (subset)
assert _validate_numbers("(64 + 14) - 16", [64, 14, 16, 7]) == True

# Example 2: Invalid Usage (number used too many times)
assert _validate_numbers("(7 + 7) + 7", [7, 7, 65]) == False

# Example 3: Invalid Usage (number not available)
assert _validate_numbers("10 + 9", [10, 8, 2]) == False
```

`_evaluate_equation(equation_str)` Evaluate E using only digits, spaces, +, -, *, /, (,) characters. You can use `eval` in Python, but make sure to return `None` if the equation is invalid, such as implicit multiplication or Unicode operators.

```python
def _evaluate_equation(equation_str: str) -> float | None:
    ### YOUR CODE HERE ###
    ...

# Input/Output Format:
assert _evaluate_equation("(65 - 7) - 7") == 51.0
```

1.3 Task 2: Reward Function

`reward_fn` The reward $R(\mathcal{G})$ is calculated based on the validity of the extracted equation E and its result $\mathcal{R}$ compared to the target T. This definition directly maps to the reward structure defined in the function's docstring:

- $R(\mathcal{G}) = \mathbf{1.0}$: Assigned if the equation E is **Correct** (correct equation with correct answer).
- $R(\mathcal{G}) = \mathbf{0.1}$: Assigned if the equation E is making an attempt (**valid answer format** but wrong answer, wrong number used, or invalid equation).
- $R(\mathcal{G}) = \mathbf{0.0}$: Assigned if no `<answer>` tag is found in the generation $\mathcal{G}$.

```python
def reward_fn(generated_text: str, ground_truth: Dict) -> float:
    """
    Reward function for countdown math problems:
    - 1.0: Correct equation with right answer
    - 0.1: Valid format but wrong answer, or wrong numbers used, or invalid equation
    - 0.0: No answer tag found (completely failed format)
    """
    # Implements the reward logic using the verifier functions:
    # _extract_answer, _validate_numbers, and _evaluate_equation.

numbers = [79, 17, 60]
target = 36
ground_truth = {"numbers": numbers, "target": target}

# Example 1: Correct Answer
```

```python
rollout = "...thinking... <answer>(79 - (60 - 17))</answer>"
assert reward_fn(rollout, ground_truth) == 1.0



# Example 2: Correct usage, Wrong Answer (Partial, R = 0.1)
rollout = "...thinking...<answer>(79 + 60) + 17</answer>"
assert reward_fn(rollout, ground_truth) == 0.1  # valid format of <answer>

# Example 3: No Answer Tag (Failed Format, R = 0.0)
rollout = "...thinking...The final answer is 62."
assert reward_fn(rollout, ground_truth) == 0.0
```

1.4 GRPO: Group Relative Policy Optimization

Paper: DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models

Background: From REINFORCE to GRPO Teaching LLMs to solve mathematical problems through reinforcement learning presents unique challenges. While Supervised Fine-Tuning (SFT) requires expensive human-annotated step by step solutions, RL allows models to learn from trial and error with automatic verification. In this framework, the language model serves as a policy π_θ that, given a mathematical question q, generates a complete solution o, which receives a scalar reward $R(q, o)$ based on correctness.

The classical REINFORCE algorithm optimizes the policy using the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{o \sim \pi_\theta(\cdot|q)} \left[R(q, o) \cdot \nabla_\theta \log \pi_\theta(o|q) \right]$$

Directly weighting gradients by raw rewards can be noisy, to address this, it is common to introduce a baseline $b(q)$ and use advantages instead:

$$\nabla_\theta J(\theta) = \mathbb{E}_{o \sim \pi_\theta(\cdot|q)} \left[A(q, o) \cdot \nabla_\theta \log \pi_\theta(o|q) \right]$$

where $A(q, o) = R(q, o) - b(q)$ is the advantage function that measures how much better or worse this response is compared to the baseline.

GRPO: Group Relative Policy Optimization Traditional approaches require training a separate value function $V_\phi(q)$ to serve as the baseline, which introduces additional complexity to the training pipeline. To address this, GRPO elegantly simplifies the need for a value function through group sampling. For each question q, the policy samples a group of G responses $\{o^{(i)}\}_{i=1}^{G} \sim \pi_\theta(\cdot|q)$. Each response receives its reward $r^{(i)} = R(q, o^{(i)})$, and the group itself provides a natural baseline through its statistics.

The group-normalized advantage for the i-th response in GRPO is computed as:

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

Here, `advantage_eps` is a small constant added to the denominator to prevent division by zero when all responses in a group receive identical rewards.

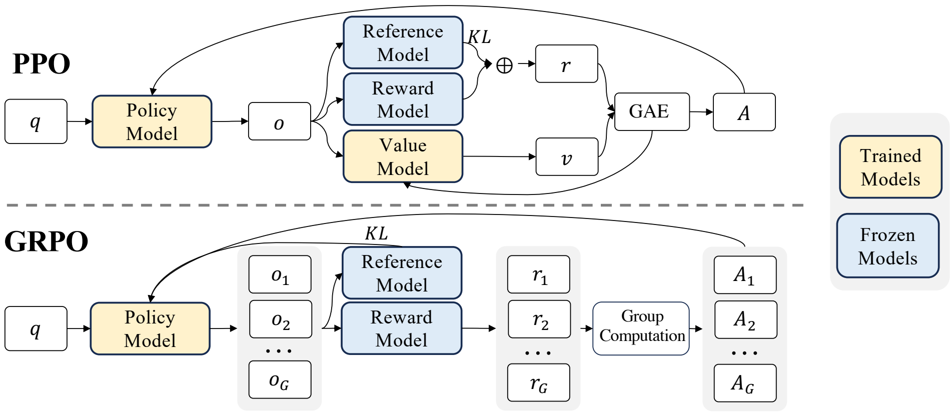

Figure 1: GRPO eliminates the need for a separate value function by using group sampling to estimate baselines.

1.4.1 Task 3: `compute_group_normalized_advantages`

Your task is to implement a function that takes the raw rewards from a batch of rollouts and transforms them into normalized advantages.

```python
def compute_group_normalized_advantages(
    rollout_responses: List[str],
    repeated_ground_truths: List[Dict],
    reward_fn: Callable[[str, Dict], float],
    group_size: int,
    advantage_eps: float,
    normalize_by_std: bool,
) -> Tuple[torch.Tensor, torch.Tensor, Dict[str, torch.Tensor]]:
    """
    Computes advantages by normalizing rewards within groups.

    Args:
        rollout_responses: List of generated responses (length = batch_size *
        ↪  group_size)
        repeated_ground_truths: Ground truth repeated for each response
        reward_fn: Function to compute rewards
        group_size: Number of responses per question (G)
        advantage_eps: Small constant for numerical stability (epsilon)
        normalize_by_std: If True, normalize by std (GRPO); if False, don't (DR-GRPO)

    Returns:
        advantages: Flattened tensor of advantages (shape: [batch_size * group_size])
        raw_rewards: Original rewards before normalization
        metadata: Dictionary with 'mean', 'std', 'max', 'min' of raw rewards
    """
    ### YOUR CODE HERE ###
    pass
```

Implementation Notes: When `normalize_by_std` is True, the advantages are computed as shown in the Eq. (1). If `normalize_by_std` is False (as in the Dr.GRPO paper), the advantages are simply calculated as the difference between each reward and the group mean, without normalizing by the standard deviation.

1.4.2 Task 4: `compute_loss`

After computing advantages, we construct a per-token loss by decomposing the conditional log-probability of the whole response. Using the chain rule, the policy factorizes as

$$\pi_\theta(o \mid q) = \prod_{t=1}^{|o|} \pi_\theta\big(o_t \mid o_{<t}, q\big)\,, \quad \Rightarrow \quad \log \pi_\theta(o \mid q) = \sum_{t=1}^{|o|} \log \pi_\theta\big(o_t \mid o_{<t}, q\big).$$

This means that maximizing the sequence log-likelihood is equivalent to maximizing the sum of per-token log-likelihoods. In GRPO, each sampled response for a prompt receives a single scalar advantage $A^{(i)}$; we broadcast it to all tokens in that response ($A_t = A^{(i)}$ for all t). The PPO-style sequence loss then sums token losses:

$$\mathcal{L}_{\text{seq}} = -\sum_{t=1}^{|o|} \min(A_t\,\text{ratio}_t,\ A_t\,\text{clip}(\text{ratio}_t, 1-\epsilon, 1+\epsilon))\,.$$

Where the probability ratio compares the new and old policies at each token:

$$\text{ratio}_t = \frac{\pi_\theta(o_t \mid o_{<t}, q)}{\pi_{\theta_{\text{old}}}(o_t \mid o_{<t}, q)} = \exp\big(\log \pi_\theta - \log \pi_{\theta_{\text{old}}}\big),$$

where $\pi_{\theta_{\text{old}}}$ is the pre-update policy or the policy that was used to generate the rollout. Following PPO, the clipping simply keeps the new policy from drifting too far from the current one, which stabilizes training even when advantages are large.

```python
def compute_loss(
    advantages: torch.Tensor,
    policy_log_probs: torch.Tensor,
    old_log_probs: torch.Tensor,
    clip_range: float,
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    """
    Computes the per-token PPO clipped surrogate loss.

    Steps:
    1. Calculate the probability ratio ratio_t
    2. Calculate the unclipped term: A_t * ratio_t
    3. Calculate the clipped term by clipping ratio_t to [1-clip_range,
    ↪  1+clip_range]
      and then multiplying by A_t.
    4. Then take the minimum between 2 terms
```

Implementation details Let batch size be B and let T be the number of tokens in each example (note that the prompt tokens are masked out).

- `policy_log_probs` $\in \mathbb{R}^{B\times T}$: computed by applying `log_softmax` to model logits and gathering at the predicted token indices; one log-prob per token.
- `old_log_probs` $\in \mathbb{R}^{B\times T}$: the same meaning but comes from the pre-update policy in one rollout.
- `advantages` $\in \mathbb{R}^{B}$: one scalar per response; broadcast along the time axis to shape $B \times T$ inside `compute_loss`.

- `response_mask` $\in \{0,1\}^{B\times T}$: masks out prompt positions so loss is computed only on response tokens.

The `compute_loss` function computes per-token probability ratios, multiplies by advantages, applies elementwise clipping, and returns a loss matrix of shape $B \times T$.

Conceptual: Clipping and Policy Regime

Question.

1. In the PPO-style loss above, is clipping more critical for **on-policy** or **off-policy** updates in this setup? Briefly justify your answer in terms of the probability ratio.

2. In the context of GRPO for this task, list **two advantages of on-policy** and **two advantages of off-policy** and give a brief explanation for each.

Task 5: `masked_mean` and `masked_mean_drgrpo` reduction

With per-token losses $L \in \mathbb{R}^{B\times T}$ computed, we must reduce them to a single scalar for optimization while respecting the response mask M. In this part, we will implement two reduction schemes: (1) the `masked_mean`, which is the original GRPO proposal where the reduction is based on the response length, and (2) `masked_mean_drgrpo`, where the reduction is instead scaled by a constant, specifically the maximum response length, as in Dr.GRPO. Interestingly, the choice of reduction also changes the per-token gradients during backpropagation, since different denominator schemes rescale token losses differently.

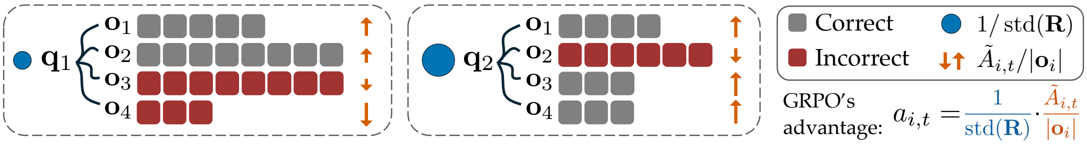
$\mathbf{q}_1$
$\mathbf{o}_1$
$\mathbf{o}_2$
$\mathbf{o}_3$
$\mathbf{o}_4$
$\mathbf{q}_2$
$\mathbf{o}_1$
$\mathbf{o}_2$
$\mathbf{o}_3$
$\mathbf{o}_4$
Correct
Incorrect
$1/\,\mathrm{std}(\mathbf{R})$
$\tilde{A}_{i,t}/|\mathbf{o}_i|$
GRPO's advantage: $a_{i,t} = \frac{1}{\mathrm{std}(\mathbf{R})} \cdot \frac{\tilde{A}_{i,t}}{|\mathbf{o}_i|}$

Figure 2: This figure illustrates response-level length bias: dividing by the response length $|o_i|$ skews gradient magnitudes. For correct responses (gray tokens), shorter responses receive larger updates (bigger orange arrows) compared to longer responses. Conversely, for negative advantages from incorrect responses (red tokens), longer responses are penalized less.

`masked_mean` (length-normalized). Average the per-token loss over each sequence's response tokens and then average across the batch.

```python
def masked_mean(tensor: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
    """
    Mean over response tokens per sequence, then mean over batch.

    Args:
        tensor: Per-token loss [B, T]
        mask:  Binary mask   [B, T] (1=response token, 0=prompt/pad)
    Returns:
        Scalar loss.
    """
```

masked_mean_drgrpo (constant-normalized). Dr.GRPO removes response-length normalization by replacing the per-sequence denominators with a fixed constant num_tokens.

```python
def masked_mean_drgrpo(tensor: torch.Tensor, mask: torch.Tensor, num_tokens: int) -> torch.Tensor:
    """
    Dr.GRPO reduction: sum over response tokens, divide by (num_tokens * B).

    Args:
        tensor: Per-token loss [B, T]
        mask:  Binary mask [B, T]
        num_tokens: A fixed constant
    Returns:
        Scalar loss.
    """
    ### YOUR CODE HERE ###
    pass
    ### END YOUR CODE ###
```

Both reductions use the mask to only calculate the loss on the response tokens. The difference is that one normalizes by the response length for each sequence, while the other uses a fixed constant.

Per-token gradients under two reductions

Question. Consider a batch with two responses: one short (3 response tokens and 3 masked tokens) and one long (6 response tokens and 0 masked tokens). Assume similar per-token log-prob ratio and advantages, and set the fixed maximum response length to 6 for the masked_mean_drgrpo reduction.

1. Apply each reduction to the batch and call backward(), then compare the gradient w.r.t the per-token log-prob ratio. What do you observe?

2. Implications: Give an example scenario where masked_mean is preferable to masked_mean_drgrpo, and one where the opposite is true. Give a brief explanation for each.

1.5 Putting It Together: Training Loop, Grouping, and Microbatches

This section is a short reading that explains how the whole pipeline fits together in practice.

One rollout-training step (high level). Each training step follows the same pattern:

1. **Sample prompts.** Pick a small set of prompts from the training set. We intend to produce multiple responses for each prompt in order to compare them within a group.

2. **Sync policy to vLLM.** Before rollout, load the current policy weights into the vLLM engine so generation uses the latest policy.

3. **Duplicate per group and generate.** For each prompt, duplicate it group_size times and send these copies to the generator (vLLM). This yields one response per copy. Duplicating is what forms a *group* of responses that all answer the same prompt.

4. **Score and compute advantages.** Use the reward_fn to score each response. Reshape the rewards into groups, subtract each group's mean to get relative scores, and (optionally) divide by the group's standard deviation. This produces one advantage value per response. Importantly, advantages are computed *before* any microbatching so that all tokens of a response share the same scalar advantage.

5. **Tokenize and mask.** Build input_ids, labels, and a response_mask that zeros out prompt tokens so the loss is only computed on generated tokens.

6. **Microbatch and accumulate gradients.** Split the rollout batch into several microbatches to fit in memory. For each microbatch:

   - Compute per-token log-probabilities of the policy.
   - Broadcast the response-level advantage across its tokens.
   - Form the PPO-style clipped per-token loss using the probability ratio.
   - Reduce to a scalar loss with either masked_mean or masked_mean_drgrpo.
   - Divide the scalar loss by the number of microbatches (i.e., gradient accumulation steps) and call backward() to accumulate gradients.

7. **Step optimizer and scheduler.** After all microbatches are processed, clip gradient norm, then call optimizer.step() and scheduler.step(). Gradients are zeroed *once* per rollout step, not per microbatch.

8. **Evaluate and log periodically.** Every few steps, generate on a held-out set, compute accuracy and reward statistics, and log short examples for quick inspection.

Why use vLLM instead of model.generate? vLLM is more optimized, faster, and more memory-efficient than model.generate, especially for batched inference. If you have more than one GPU, you can dedicate one for training and another for generating rollouts. This way, training and generation do not compete for memory, and you can allocate more memory for the vLLM kvcache during generation.

Why microbatches? Large rollout batches often do not fit into GPU memory. Microbatching keeps memory usage manageable while preserving the effect of a larger batch: each microbatch adds to the same gradient buffers, and we scale the loss so that the accumulated gradient matches what a single big batch would produce (up to small numerical differences).

Takeaway. Think of the pipeline as: sample prompts → sync policy → duplicate to form groups → generate/rollout → score & compute advantages → microbatch forward/loss/backward with accumulation → optimizer step & periodic eval. Once this flow is clear, switching between GRPO and Dr.GRPO is mostly about whether you normalize advantages by the group std and which reduction you use for the per-token losses.

1.6 Experiments

1.6.1 Zero-Shot Baseline.

Now, before running our GRPO pipeline. We can sanity check the `Qwen/Qwen3-1.7B` model performance on zero-shot (just prompting). Running `zero_shot_eval.py`, for 3 settings of MAX_TOKENS L_{256}, L_{512}, L_{1024} you will get the following accuracy values:

- $L_{256} \approx 0.1\%$
- $L_{512} \approx 15\%$
- $L_{1024} \approx 37\%$

Main Experiments

1.6.2 Main Experiments

For our upcoming RL experiments, our target is to train the `Qwen/Qwen3-1.7B` model at L_{256} and L_{512} to outperform zero-shot performance obtained at L_{1024} ($\approx 37\%$).
Using your implemented GRPO and Dr.GRPO pipeline, run a full training loop for models with maximum generation lengths of $\mathbf{L}_{256}$ and $\mathbf{L}_{512}$.
Experimental Setup:

1. Train with a maximum generation length of L_{256}.
2. Train with a maximum generation length of L_{512}.

Report Performance: Tabulate and plot the final task accuracy for both models (L_{256} and L_{512}), comparing them with zero-shot baselines.
Analysis: Write a short analysis discussing the results of the runs. This should include an interpretation of the successful and failed cases of the trained models. An extra hypothesis or implementation of what can still be improved in the current implementation or reward function is highly recommended.

2 Optional Extra Credit: Problem 2 - Open-Ended Investigation

This problem is designed as an optional extension to investigate deeper to the RL training pipeline. The primary objective is to demonstrate strong problem-solving and analytical thinking skills. **Your analysis of the approach, whether successful or failed, is the most important component.**

Below are some examples of possible starting point for your investigation:

- **Keep improving current results:** Show your LLM's training skill by keeping improving current results and settings.
- **RL training stability:** Investigate and attempt to resolve the current instability issues encountered when training beyond $\approx 100 - 120$ optimization steps. Stable RL training is an ongoing research problem even at frontier labs.
- **Adaptation:** Adapt the current GRPO/Dr.GRPO implementation to a conceptually similar reasoning task. Be creative.
- ...

Acknowledgments

We are grateful to TinyZero for their Countdown task, which provided an interesting introduction to RLVR. We also thank the Stanford CS336 staff for their excellent work on assignment 5, which served as a basis for this project.