Assignment 7: Tiny GRPO-based RL Fine-Tuning

1 Introduction

In this assignment, you will fine-tune a **tiny language model** using **Group Relative Policy Optimization (GRPO)** on a toy task. Unlike standard Supervised Fine-Tuning (SFT) which trains on pre-written answers, Reinforcement Learning (RL) fine-tuning trains the model to optimize a reward signal.

You will implement the entire pipeline: generating a dataset, defining a reward function (rule-based or model-based), implementing the GRPO algorithm from scratch, and observing how the model's behavior shifts to maximize rewards.

Character limit: You must convert to markdown with the checker and check that your submission renders correctly and is below the character limit of 40000.

1.1 Why GRPO?

Standard RLHF methods (like PPO) usually require training a separate "Critic" (Value) model to estimate the expected reward. This effectively doubles the memory requirements.

GRPO removes the need for a Critic. Instead, it samples a **group** of outputs for a single prompt and uses the group's average reward as the baseline. This makes it highly efficient and ideal for learning RL fine-tuning concepts on limited hardware.

1.2 Core Tasks

You will implement the following components in order:

Component	Description	Constraints
Task & Data	Design a toy task (e.g., counting, pattern matching) and generate prompts.	Must be auto-evaluable. Dataset size: hundreds to thousands.
Reward Function	Implement a function that scores model outputs.	Rule-based (recommended) or LLM-as-a-judge.
GRPO Logic	Implement the loss function using group-normalized advantages.	Must sample $G \geq 2$ outputs per prompt.
Training Loop	Run the optimization loop to fine-tune a tiny pretrained model.	Use a tiny model (e.g., TinyLlama, GPT-Neo-125M, or smaller).
Analysis	Compare the model before and after fine-tuning.	Quantitative (avg reward) and qualitative analysis.

2 Technical Background: GRPO

Group Relative Policy Optimization works by estimating the "advantage" of a specific response based on how much better it is compared to other responses generated from the same prompt.

2.1 The Algorithm

For each training step:

1. **Sample Prompts:** Sample a batch of prompts x from your dataset.
2. **Generate Groups:** For each prompt x, sample a **group** of G outputs $\{y_1, y_2, \ldots, y_G\}$ from the current policy π_θ.
3. **Score:** Compute rewards for each output: $r_i = \text{Reward}(x, y_i)$.

4. **Compute Advantages:** Calculate the advantage A_i for each output by normalizing the rewards within the group:

$$A_i = \frac{r_i - \text{mean}(\{r_1, \ldots, r_G\})}{\text{std}(\{r_1, \ldots, r_G\}) + \epsilon}$$

Note: Outputs that are better than the group average get positive advantages; those worse get negative.

5. **Optimize:** Update the model parameters θ to maximize the objective:

$$\mathcal{J} = \frac{1}{G} \sum_{i=1}^{G} \left[A_i \cdot \log \pi_\theta(y_i|x) - \beta \cdot \mathbb{D}_{KL}(\pi_\theta || \pi_{ref}) \right]$$

This encourages the model to increase likelihood of above-average answers and reduce likelihood of below-average ones. *(The KL divergence term explained very briefly below is optional for this toy assignment but recommended for stability).*

3 (Optional) Technical Background: Estimating KL Divergence

In the context of RL Fine-Tuning (GRPO or PPO), the KL divergence term, $\mathbb{D}_{KL}(\pi_\theta || \pi_{ref})$, measures the "distance" between your current training policy π_θ and the original reference model π_{ref}.

The goal is to prevent the model from forgetting its original training (catastrophic forgetting) or "gaming" the reward function by producing gibberish that happens to satisfy the rule-based reward.

3.1 1. The Estimator

Computing the exact KL divergence requires summing over the entire vocabulary at every sequence position (computationally expensive) and over all possible sequences (impossible). Therefore, we estimate it using a **Monte Carlo** method based only on the tokens the model actually generated.

3.1.1 Standard Definition

The standard estimation for a specific sampled sequence y is the expected log-ratio of the probabilities:

$$D_{KL} \approx \log \pi_\theta(y|x) - \log \pi_{ref}(y|x)$$

3.1.2 Schulman's Approximation (Advanced)

Standard RLHF libraries (like Hugging Face `trl`) often use an estimator proposed by John Schulman. This "k3" estimator is strictly positive and generally has lower variance than the standard definition:

$$D_{KL} \approx \frac{(\pi_{ref}/\pi_\theta) - 1 - \log(\pi_{ref}/\pi_\theta)}{2}$$

3.2 2. Pseudo-Algorithm: Calculating Per-Token KL

To implement this efficiently, you must perform dual forward passes to compare the probability of the *generated* tokens under both the new model and the old frozen model.

Algorithm: KL Estimation Step

1. **Input:** A batch containing Prompts x and Generated Responses y.
2. **Models:** You need the Policy Model (trainable) and Reference Model (frozen/no gradients).
3. **Forward Passes:**
   - Run (x, y) through the **Policy Model** to get logits.
   - Run (x, y) through the **Reference Model** to get logits.
4. **Log-Probabilities:**
   - Convert logits from both models into log-probabilities (e.g., using LogSoftmax).
5. **Gather:**

- For every position in the sequence, select only the log-probability corresponding to the actual token present in y. Discard probabilities for tokens that were *not* selected.

6. **Compute:**
   - Calculate the difference (or Schulman's approx) between the Policy log-probs and Reference log-probs.
7. **Output:** A tensor of KL values matching the shape of the generated sequence.

3.3 3. Applying it in GRPO

Once you have calculated the per-token KL, you incorporate it into your training loop in one of two ways:

1. **As a Reward Penalty:** Subtract the KL term from the reward *before* normalization.

$$R_{total} = R_{task} - \beta \cdot D_{KL}$$

2. **As a Loss Term:** Add the mean KL directly to the loss function.

$$Loss = Loss_{GRPO} + \beta \cdot \text{mean}(D_{KL})$$

For the "Tiny GRPO" assignment, **Method 2** is usually preferred as it keeps the reward signal clean and interpretable.

---

4 Notebook Structure

Your submission must be a single Jupyter notebook organized as follows.

4.1 Cell 0: Quarto YAML Header (Markdown)

```yaml
---
format:
  html:
    code-fold: true
jupyter: python3
---
```

4.2 Cell 1: Task Definition and Plan (Markdown)

Describe your experiment setup.

- **Tiny Model:** Which model are you using? (Must be a pretrained decoder-only model).
- **Toy Task:** Define the prompt structure and the desired output.
  - *Example:* "Prompt: 'Count the dots: ...', Output: '3' ".
- **Reward Logic:** How will you score the outputs? (e.g., "1.0 if correct, 0.0 otherwise").
- **Hypothesis:** What behavior do you expect to see change?

4.3 Cell 2: Setup and Model Loading (Code)

- Import libraries (PyTorch, Transformers, etc.).
- Load the **pretrained model** and **tokenizer**.
- Ensure the model is on the correct device (GPU/CPU).
- **Output:** Print the model architecture or parameter count to verify loading.

4.4 Cell 3: Dataset Generation (Code)

- Programmatically generate your **Training Set** and **Evaluation Set**.
- The dataset should consist of prompts (strings) tailored to your toy task.
- **Output:** Print 3 example prompts from your dataset.

4.5 Cell 4: Reward Function Implementation (Code)

- Implement a function `get_reward(prompts, responses)` that returns a list/tensor of scalar rewards.
- **Sanity Check:** Create a small manual list of prompts and paired "good" and "bad" responses. Run them through your function.
- **Output:** Print the prompt, response, and calculated reward for your manual sanity check.

4.6 Cell 5: The GRPO Step (Code)

- Implement a helper function (or class) that performs a single GRPO update step.
- **Input:** A batch of prompts.
- **Logic:**
  1. **Generation:** Generate G completions per prompt (where $G \geq 2$).
  2. **Scoring:** Calculate rewards for all completions.
  3. **Advantage:** Compute group-wise normalized advantages (mean 0, std 1).
  4. **Loss:** Compute the policy gradient loss ($-\text{Advantage} \times \text{LogProb}$).
  5. **(Optional) KL:** If implementing KL regularization, calculate the per-token KL estimate using a frozen reference model.
- **Constraint:** You must calculate the log-probabilities of the generated tokens to compute the loss.
- **Output:** Run this function on *one* single batch of data (without an optimizer step) and print the shape of the advantages and the initial loss value.

4.7 Cell 6: Training Loop (Code)

- Initialize your optimizer (e.g., AdamW).
- Run the training loop for a fixed number of steps/epochs.
- In each step, call your GRPO logic, compute gradients, and update the model.
- **Logging:** Print the **average reward** and **average loss** every N steps.
- **Output:** The training logs showing the evolution of the reward.

4.8 Cell 7: Evaluation and Generation (Code)

- **Quantitative:** Run the fine-tuned model on your held-out **Evaluation Set**. Compute the mean reward and compare it to the baseline (untrained) performance.
- **Qualitative:** Generate responses for 5 specific prompts using the fine-tuned model.
- **Output:**
  - A printed comparison of Baseline Reward vs. Fine-tuned Reward.
  - Five examples showing: Prompt → Model Output → Reward.

4.9 Cell 8: Analysis (Markdown)

Provide a brief analysis of the results.

- **Success:** Did the model learn the task? (Did rewards go up?)
- **Dynamics:** Did you observe any "reward hacking" (model finding a shortcut to get rewards without solving the task properly)?
- **Group Size:** How might changing the group size G affect the stability of the training?

---

5 Rubric

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Task & Rewards	Task is precise and auto-evaluable. Reward function is strictly verified with sanity checks and edge cases.	Task is clear. Reward function is implemented and verified but lacks comprehensive edge-case testing.	Task is defined but reward function is vague or verification is minimal.	Task is defined, but reward function has logical flaws or is inconsistent.	Task is missing or reward function is illogical/unusable.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
GRPO Implementation	Correctly implements group sampling, advantage normalization, and policy loss. Code is clean and modular.	Implements logic correctly, but code is slightly disorganized or inefficient.	Implements logic but misses minor details (e.g., epsilon in normalization) or hardcodes values.	Logic is implemented but has significant errors (e.g., wrong advantage math, no grouping).	Logic is fundamentally incorrect (e.g., standard supervised learning).
Training Pipeline	Loop runs successfully, logs are informative/scannable, and model shows clear signs of learning.	Loop runs and model learns, but logging is messy or hard to interpret.	Loop runs but model struggles to learn due to minor setup issues or hyperparams.	Loop crashes frequently or produces NaNs; no successful training run.	Training loop does not run.
Evaluation	Clear quantitative (before/after) and qualitative analysis. Insightful discussion on dynamics/limitations.	Good comparison of before/after, but analysis of "why" is superficial.	Basic evaluation metrics present but lacks direct comparison or qualitative examples.	Evaluation is present but confusing, incorrect, or missing key metrics.	No evaluation provided.