Assignment 11: Tiny GRPO-based RL Fine-Tuning

1 Instructions

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, implementing the GRPO algorithm from scratch, and observing how the model's behavior shifts to maximize rewards.

1.1 Submission Requirements

You must submit two components to Gradescope:

1. **The Report:** A plaintext Markdown document containing the 5 sections described in Part 2.
    - **No Images:** Describe your findings with text and data tables.
    - **Character Limit:** 7,500 characters.
2. **Supplemental Material:** Your `.ipynb` notebook containing all code, plots, and raw results.

---

2 Reference Guide: The Mathematics of GRPO

Because we do not cover GRPO extensively in class, use this guide as your mathematical and intuitive foundation for the implementation.

2.1 1. Intuition: Why GRPO?

Standard RLHF methods (like Proximal Policy Optimization, or PPO) usually require training a separate "Critic" (Value) model to estimate the expected reward of a state. This Critic acts as a baseline to determine if an action was better or worse than expected. However, keeping a second model in memory effectively doubles the hardware requirements.

Group Relative Policy Optimization (GRPO) removes the need for a Critic model entirely. Instead, it estimates the "advantage" of a specific response based on how much better it is compared to *other responses generated from the very same prompt.* By sampling a **group** of outputs for a single prompt, GRPO uses the group's average reward as the baseline. This makes it highly memory-efficient and ideal for learning RL fine-tuning concepts on limited hardware.

2.2 2. The GRPO Algorithm and Math

For each training step, the algorithm follows this mathematical progression:

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 $\{y_1, y_2, \dots, y_G\}$ from the current policy π_θ.
3. **Score:** Compute rewards for each output using your reward function: $r_i = \text{Reward}(x, y_i)$.
4. **Compute Advantages:** Calculate the advantage A_i for each output by normalizing the rewards within the group. Outputs that are better than the group average get positive advantages; those worse get negative:

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

5. **Optimize:** Update the model parameters θ to maximize the policy 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 objective function encourages the model to increase the likelihood of above-average answers and reduce the likelihood of below-average ones.

2.3 3. The KL Divergence Penalty: Preventing Collapse

Intuition: In Reinforcement Learning, models are notoriously good at "reward hacking." If your reward function simply checks if the output contains the word "apple," the model might learn to output "apple apple apple" repeatedly. It maximizes the reward but destroys its ability to speak coherent English. The KL divergence term, $\mathbb{D}_{KL}(\pi_\theta || \pi_{ref})$, acts as an elastic tether. It measures the "distance" between your current training policy π_θ and the original, frozen reference model π_{ref}. By penalizing large deviations, we force the model to solve the task while maintaining its foundational language distribution.

Mathematics & Estimators: Computing the exact KL divergence requires calculating the probability of every single word in the vocabulary at every generation step, which is computationally

infeasible. Instead, we use a **Monte Carlo** approximation based only on the sequence of tokens y the model actually generated for prompt x.

The standard mathematical definition for a sampled sequence is the log-ratio of the probabilities:

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

However, standard RLHF libraries often prefer an estimator proposed by John Schulman (often called the "k3" estimator). It ensures the KL value is strictly positive and typically exhibits lower variance during training:

$$D_{KL} \approx \frac{1}{2}\left(\frac{\pi_{ref}(y|x)}{\pi_\theta(y|x)} - 1 - \log \frac{\pi_{ref}(y|x)}{\pi_\theta(y|x)}\right)$$

Implementation (Dual Forward Passes): To calculate this practically during the training loop:

1. Pass the generated sequence through your active policy model π_θ to get the log-probabilities of the generated tokens.
2. Pass the exact same sequence through your frozen reference model π_{ref} to get its log-probabilities.
3. Compute the per-token KL divergence using the estimator above.
4. Add the mean KL penalty across all tokens directly to your loss function:

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

(Note: β is a tuning parameter, often starting around 0.01 to 0.1).

---

3 Part 1: Implementation (The Notebook)

You will author a Jupyter Notebook (`.ipynb`) to perform the following experiments. It is highly recommended to use Google Colab with a GPU runtime.

Global Setup:

- Fix the random seeds for reproducibility (e.g., `torch.manual_seed(42)`).
- Configure your device dynamically: `device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")`.
- Load a tiny pretrained decoder-only model (e.g., TinyLlama, GPT-Neo-125M, or smaller) and its tokenizer.

3.1 Experiment 1: Task Definition & Dataset Generation

Concept: To train a model via RL, we need an environment or task where outputs can be evaluated automatically.

3.1.1 Task 1.1: Toy Task Setup

- Design a toy task that is strictly auto-evaluable. Because we are relying on a programmatic reward function instead of an LLM-as-a-judge (to save memory), the task must be objectively verifiable using pure Python logic.
- **Examples of Verifiable Toy Tasks:**
  - **Strict Formatting (JSON):** Prompt the model to extract information and output it strictly in JSON. *Reward:* +1.0 if `json.loads(output)` parses without error and contains the required keys; 0.0 otherwise.
  - **The Lipogram (Constraint Satisfaction):** Prompt the model to answer a question without using the letter 'e'. *Reward:* +1.0 if the answer is relevant (passes a basic length check) AND does not contain 'e' or 'E'.
  - **Regex Extraction (Math/Logic):** Provide a paragraph of text and ask the model to sum the numbers found within it, outputting the final answer in `<ans>X</ans>` tags. *Reward:* +1.0 if regex successfully extracts the tag and the integer matches the ground truth.
  - **Sentiment Reversal:** Provide a positive review and ask the model to rewrite it to be negative. *Reward:* Pass the output through a tiny, pre-trained sentiment classifier (like a DistilBERT pipeline) and use the probability of the "negative" class as the continuous reward signal.
- Programmatically generate your **Training Set** (hundreds to thousands of examples) and **Evaluation Set**.

Notebook Output:

- Print 3 example prompts from your dataset.
- Print the model architecture or parameter count to verify loading.

3.2 Experiment 2: The Reward Function

Concept: The reward function replaces the ground-truth labels used in standard supervised learning. It must programmatically score the model's generated text.

3.2.1 Task 2.1: Implementation and Verification

- Implement a function `get_reward(prompts, responses)` that returns a list/tensor of scalar rewards (e.g., `1.0` if correct, `0.0` otherwise).
- **Sanity Check:** Create a small manual list of prompts and paired "good" and "bad" responses. Run them through your function to ensure the logic holds.

Notebook Output:

- Print the prompt, response, and calculated reward for your manual sanity check examples.

3.3 Experiment 3: Group Relative Policy Optimization (GRPO)

Concept: You will implement the core mathematical steps from the Reference Guide to calculate group-relative advantages and the resulting policy loss.

3.3.1 Task 3.1: The GRPO Step Logic

Implement a helper function that performs a single GRPO update step for a batch of prompts:

1. **Generation:** Generate G completions per prompt (where $G \geq 2$).
2. **Scoring:** Calculate rewards for all completions using your reward function.
3. **Advantage:** Compute group-wise normalized advantages.
4. **Loss:** Compute the policy gradient loss ($-\text{Advantage} \times \text{LogProb}$). You must calculate the log-probabilities of the generated tokens to compute the loss.

3.3.2 Task 3.2 (Optional but Recommended): KL Divergence

If implementing KL regularization, calculate the per-token KL estimate using Schulman's approximation by passing the generated tokens through a frozen reference model.

Notebook Output:

- Run your GRPO step function on *one* single batch of data (without an optimizer step) and print the shape of the advantages and the initial loss value.

3.4 Experiment 4: Training Loop and Evaluation

3.4.1 Task 4.1: The Training Loop

- 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.

3.4.2 Task 4.2: Evaluation

- Run the fine-tuned model on your held-out **Evaluation Set**. Compute the mean reward and compare it to the baseline (untrained) performance.

Notebook Output:

- Training logs showing the average reward and average loss every N steps.
- A printed comparison of Baseline Reward vs. Fine-tuned Reward on the evaluation set.
- Five qualitative examples showing: Prompt $\rightarrow$ Fine-tuned Model Output $\rightarrow$ Reward.

---

4 Part 2: Content Requirements (The Report)

Your report must be organized into **exactly five sections** with Markdown headers.

4.1 Section 1: Executive Summary

- **One-Sentence Takeaway:** Summarize the core finding regarding the model's ability to optimize the reward signal through GRPO compared to its baseline state.
- **Summary Paragraph:** Outline the progression of the assignment, highlighting the transition from defining auto-evaluable tasks to implementing group-based advantage normalization and observing policy shifts.

4.2 Section 2: Task & Reward Design

- **Task Definition:** Describe the toy task you designed. Why was it suitable for RL fine-tuning?
- **Reward Mechanics:** Explain your reward function logic. How did you ensure it correctly penalized bad outputs and rewarded desired behavior?

4.3 Section 3: GRPO Mechanics & Advantage

- **The Critic:** Explain mechanically why GRPO does not require a separate Critic (Value) model like PPO does.
- **Advantage Calculation:** Discuss how normalizing the rewards within a sampled group of size G provides a reliable baseline for the policy gradient update.

4.4 Section 4: KL Divergence & Stability

- *(If implemented):* Explain the purpose of the KL divergence penalty. Why is Schulman's approximation calculated only on the *generated* tokens rather than the entire vocabulary?
- *(If not implemented):* Discuss the theoretical risks of training an RL model without a KL penalty against a reference model (e.g., reward hacking, catastrophic forgetting).

4.5 Section 5: Reflection

- **Training Dynamics:** Did your model learn the task successfully? Did you observe any instances of "reward hacking" where the model found a shortcut to get rewards without solving the task properly?
- **Group Size:** Reflect on the hyperparameter G (Group size). How might changing the number of generated responses per prompt affect the variance and stability of your training?

---

5 Grading Rubric

Each of the five sections is weighted equally (**20% each**).

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 1: Summary	Takeaway is specific and insightful. Summary captures the full pipeline from task creation to policy optimization.	Takeaway is clear. Summary covers the main assignment arc.	Takeaway is generic. Summary is vague.	Summary misses key elements of the analysis.	Missing.
Section 2: Task & Reward	Deep explanation of the task constraints and robust edge-case handling in the reward logic.	Good explanation of the task and reward function.	Describes the task but the mechanical "why" behind the reward logic is weak.	Descriptions are incorrect or analysis is missing.	Missing.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 3: GRPO Mechanics	Excellent, mathematically sound explanation of group sampling and why it eliminates the need for a Critic model.	Good explanation of advantage normalization and baselines.	Explains the method but lacks a discussion on why the baseline works.	Fails to grasp the concept of group relative advantages.	Missing.
Section 4: KL & Stability	Deep, mechanical analysis of the KL estimator, token-level probabilities, and reward hacking prevention.	Good explanation of KL divergence and model degradation.	Mentions KL is needed but lacks technical reasoning on how it is calculated.	Evaluates tasks but ignores the stability discussion entirely.	Missing.
Section 5: Reflection	Thoughtful reflection on specific training dynamics, reward hacking instances, and the variance impact of group size G.	Good reflection on training success and hyperparameters.	Generic reflection (e.g., "RL is hard to train").	Minimal effort.	Missing.