Reinforcement Learning for LLM Reasoning Under Memory Constraints

1st Alan Lee
Department of Computer Science
University of Michigan
Ann Arbor, USA
alanleee@umich.edu

2nd Harry Tong
Department of Computer Science
University of Michigan
Ann Arbor, USA
harryt@umich.edu

***Abstract*—We explore reinforcement learning (RL) techniques to enhance reasoning within targeted problem spaces in large language models (LLMs) under memory and compute constraints. Our focus is on critic-free methods compatible with LoRA fine-tuning on a single 40GB GPU, a common limitation in academic settings. We introduce S-GRPO, a memory-efficient variant of Group Relative Policy Optimization, and T-SPMO, a token-level prefix matching strategy for fine-grained credit assignment. Despite limited resources, when used to fine-tune Qwen2-1.5B both methods significantly improve SVAMP benchmark accuracy from 46% to above 70% using LoRA training. T-SPMO also excels in multi-digit multiplication tasks, underscoring the potential of RL fine-tuning under hardware constraints. Additionally, we find that our full-token GRPO baseline under LoRA fine-tuning did not improve model performance (compared to base model) on either task, suggesting that our memory-efficient methods may act as a form of regularization that stabilizes training when only a small subset of parameters are updated. Project details are at: https://harrytong123.github.io/research.html.**

I. Introduction

Large Language Models (LLMs) have demonstrated substantial improvements in mathematical and structured problem-solving through reinforcement learning (RL) fine-tuning on specific tasks [1]. However, standard RL approaches such as Proximal Policy Optimization (PPO) [2] and Group Relative Policy Optimization (GRPO) [3] can be impractical for researchers operating under limited computational budgets, for the following reasons: (1) both methods compute loss over the entire output trajectory, (2) they are typically designed for full model fine-tuning, and (3) PPO additionally requires a critic network, further increasing memory demands.

Moreover, supervised fine-tuning approaches rely on high-quality, human-annotated chain-of-thought data, which can be expensive or infeasible to collect at scale for diverse problem domains.

In this work, we explore the question: **Can RL fine-tuning improve reasoning performance on targeted tasks using only parameter-efficient updates and a single 40GB GPU?** To this end, we develop and evaluate two critic-free methods tailored for constrained memory and computational environments. First, we introduce Stochastic-GRPO (S-GRPO), a lightweight variant of GRPO that samples tokens from output trajectories to contribute to the loss. Second, we propose Token-Specific Prefix Matching Optimization (T-SPMO), which assigns credit at a token granularity.

To evaluate these methods, we fine-tune the open-weight Qwen2-1.5B model [4] using LoRA [5] on two reasoning benchmarks: the SVAMP dataset [6] and a multi-digit arithmetic task. Despite the constrained training setup, both S-GRPO and T-SPMO raise SVAMP test accuracy from 46% to above 70%. On the multiplication benchmark, S-GRPO improves accuracy from 3.9% to 22.9%, while T-SPMO dramatically outperforms both baselines, reaching 70%. To ground our results, we also establish a full-token GRPO baseline under LoRA, which surprisingly did not lead to meaningful jumps in performance from the base model. These results suggest that carefully designed, memory-efficient RL algorithms can significantly improve LLM reasoning under compute constraints—even outperforming exhaustive full-token optimization strategies.

Our contributions are threefold:

We present S-GRPO, a lightweight variant of GRPO with minimal alterations.

We propose T-SPMO, a novel token-level RL method with fine-grained credit assignment tailored for reasoning tasks.

We demonstrate that both methods are effective when applied with LoRA-based fine-tuning under tight GPU wall-clock time and memory budgets, making RL for reasoning accessible to a broader research community.

II. Related Work

A. Reinforcement Learning for Large Language Models

Reinforcement learning has been widely used to align language model outputs with user preferences or task-specific objectives. Proximal Policy Optimization (PPO) [2] is a commonly used algorithm in reinforcement learning from human feedback (RLHF), most notably applied in training models like InstructGPT [7]. However, PPO requires training a separate value network (critic), which significantly increases memory and compute demands.

Group Relative Policy Optimization (GRPO) [3] was introduced to remove the need for a critic by computing group-level statistics over generated samples. Although effective, GRPO still assumes access to full-model fine-tuning and typically re-

quires the storage of full trajectories for reward normalization, limiting its applicability in low-resource settings.

B. Memory-Efficient Fine-Tuning

Parameter-efficient fine-tuning methods such as Low-Rank Adaptation (LoRA) [5] have enabled researchers to adapt large models on modest hardware by injecting trainable rank-decomposed matrices into frozen weights. These techniques allow significant reductions in memory usage and training time without substantially sacrificing downstream performance. Our work builds on this line of research by evaluating reinforcement learning algorithms in the context of LoRA-based fine-tuning, rather than full backpropagation over all model weights.

C. Reasoning Benchmarks

Benchmarks such as GSM8K [8], SVAMP [6], and ASDiv [9] have been widely adopted to measure mathematical and symbolic reasoning in language models. SVAMP in particular requires verbal reasoning over numerical quantities and has been shown to be out-of-distribution (OOD) for many LLMs (such as by presenting irrelevant information in questions) [6]. Prior work has focused primarily on supervised fine-tuning or prompt engineering for these tasks, but recent interest has grown in using RL to improve step-by-step reasoning consistency [10].

III. Methods

A. Problem Setup

We focus on enhancing the reasoning capabilities of large language models (LLMs) using reinforcement learning (RL) under constrained computational resources. The task is framed as *conditional generation with reward feedback*, where the model generates a sequence of tokens $Y = (y_1, y_2, \ldots, y_T)$ in response to a prompt X, and receives a scalar reward signal $r(Y) \in \mathbb{R}$ based on the correctness or quality of the generated sequence.

Let $\pi_\theta(y_t \mid y_{<t}, X)$ denote the autoregressive policy defined by the language model with parameters θ. The objective is to optimize θ to maximize the expected reward over the distribution of completions:

$$\mathcal{J}(\theta) = \mathbb{E}_{Y \sim \pi_\theta}\left[r(Y)\right] \tag{1}$$

Traditional policy gradient methods often rely on estimating the advantage function using a separate value network, which can be computationally intensive and memory-demanding. To address these challenges, the recently popularized critic-free RL algorithm *Group Relative Policy Optimization (GRPO)* estimates advantages by comparing the rewards of multiple responses generated for the same prompt [3].

For each input prompt, GRPO generates a group $\mathcal{G}$ completions from the current policy π_θ. Each completion $y^{(i)}$ in group i is tokenized as $y^{(i)} = (y_1^{(i)}, y_2^{(i)}, \ldots, y_{T_i}^{(i)})$ and assigned a scalar reward R_i.

The advantage of each token is normalized using the group-level statistics:

$$\hat{A}_{i,t} = \frac{R_i - \mu_R}{\sigma_R} \tag{2}$$

where μ_R and σ_R are the mean and standard deviation of rewards across the group.

The policy is updated to increase the likelihood of responses with positive advantages and decrease it for those with negative advantages:

$$\begin{aligned}\mathcal{J}(\theta) = \frac{1}{|\mathcal{G}|}\sum_{i=1}^{|\mathcal{G}|}\frac{1}{|y^{(i)}|}\sum_{t=1}^{|y^{(i)}|}\Bigg\{ \\ \min\Bigg[\frac{\pi_\theta\big(y_t^{(i)} \,|\, x, y_{<t}^{(i)}\big)}{\pi_{\theta_{\text{old}}}\big(y_t^{(i)} \,|\, x, y_{<t}^{(i)}\big)}\,\hat{A}_{i,t}, \\ \text{clip}\Big(\frac{\pi_\theta\big(y_t^{(i)} \,|\, x, y_{<t}^{(i)}\big)}{\pi_{\theta_{\text{old}}}\big(y_t^{(i)} \,|\, x, y_{<t}^{(i)}\big)}, 1-\epsilon,\, 1+\epsilon\Big)\hat{A}_{i,t}\Bigg] \\ -\ \beta\,\mathbb{D}_{\text{KL}}\big[\pi_\theta \,\|\, \pi_{\text{ref}}\big]\Bigg\}\end{aligned} \tag{3}$$

Where π_{ref} and π_{old} are previous versions of π_θ, which are updated to π_θ after varying iterations. ϵ is a hyperparameter for PPO-style objective clipping, and β is a hyperparameter coefficient for the KL-divergence regularization. GRPO estimates the KL-Divergence using the unbiased estimator:

$$\mathrm{D}_{\text{KL}}\left[\pi_\theta \,\|\, \pi_{\text{ref}}\right] = \frac{\pi_{\text{ref}}(y_t^{(i)} \mid x, y_{<t}^{(i)})}{\pi_\theta(y_t^{(i)} \mid x, y_{<t}^{(i)})} - \log\frac{\pi_{\text{ref}}(y_t^{(i)} \mid x, y_{<t}^{(i)})}{\pi_\theta(y_t^{(i)} \mid x, y_{<t}^{(i)})} - 1 \tag{4}$$

To further accommodate limited computational resources, we employ *Low-Rank Adaptation (LoRA)* [5] for parameter-efficient fine-tuning. LoRA allows us to fine-tune only a small subset of the model's parameters, significantly reducing memory requirements.

We evaluate our methods on two representative reasoning tasks:

- **SVAMP**, a benchmark of verbal arithmetic problems requiring symbolic and numerical reasoning [6].
- **Multi-digit multiplication**, where models must learn step-wise calculation strategies.

In the following sections, we describe two methods—**S-GRPO** and **T-SPMO**—that implement reinforcement learning updates without requiring full-sequence baselines or critic networks, making them well-suited to low-memory training environments. We then discuss the implementation of our baseline GRPO, which sacrifices wall clock time for memory efficiency using gradient accumulation.

B. S-GRPO: Stochastic Group Relative Policy Optimization

Stochastic Group Relative Policy Optimization (S-GRPO) extends GRPO to a low-memory setting by reducing the tokens that contribute to the gradient from the full response trajectory.

We also set the number of updates per problem (denoted as μ in the original GRPO paper [3]) to 1, aligning with the default setting in lightweight RL libraries such as TRL. This choice both simplifies optimization by eliminating the need for PPO-style clipped objectives and reduces computational overhead, consistent with our focus on efficient fine-tuning. Also in alignment of the default setting in TRL, we do not update π_{ref} to π_θ.

The policy objective is thus:

$$\mathcal{J}_{\text{SGRPO}}(\theta) = \frac{1}{|\mathcal{G}|}\sum_{i=1}^{\mathcal{G}}\frac{1}{|\mathcal{T}_i|}\sum_{t\in\mathcal{T}_i}\frac{\pi_\theta(y_t^{(i)} \mid x, y_{<t}^{(i)})}{\pi_\theta(y_t^{(i)} \mid x, y_{<t}^{(i)})\big|_{\text{no grad}}}\hat{A}_{i,t} - \beta\, D_{\text{KL}}\big[\pi_\theta \,\|\, \pi_{\text{ref}}\big] \tag{5}$$

Here, $\mathcal{T}_i$ denotes the set of tokens in completion i that contribute to the loss. Tokens are included in $\mathcal{T}_i$ according to:

$$t \in \mathcal{T}_i \iff \begin{cases} 1 & \text{if } t < \alpha \\ \text{Bernoulli}(P) & \text{if } t \geq \alpha \text{ and } |\mathcal{T}_i| < k \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where α is a cutoff index ensuring early tokens are always included, k is the maximum number of tokens contributing to the loss, and P is the probability of including later tokens stochastically.

Algorithm 1 presents the corresponding pseudocode.

Algorithm 1 Token Selection for Loss Calculation

Require: Token indices $t = 1, 2, \ldots, T$
Require: Cutoff index α
Require: Maximum tokens k
Require: Sampling probability P
Initialize $\mathcal{T}_i \leftarrow \emptyset$
for each token t from 1 to T **do**
  if $t < \alpha$ **then**
    Include token: $\mathcal{T}_i \leftarrow \mathcal{T}_i \cup \{t\}$
  else if $|\mathcal{T}_i| < k$ **then**
    Sample $s \sim \text{Bernoulli}(P)$
    if $s = 1$ **then**
      Include token: $\mathcal{T}_i \leftarrow \mathcal{T}_i \cup \{t\}$
    end if
  else
    break
  end if
end for
return $\mathcal{T}_i$

This hybrid rule ensures early tokens—which often contain key semantic steps—are reliably updated, while sampling later tokens only when there is enough space. This strikes a balance between learning stability and memory efficiency. This stochastic sampling mechanism allows S-GRPO to scale to larger batches without retaining full trajectories.

Note that in GRPO, all tokens t are guaranteed to be in $\mathcal{T}_i$ [3].

C. T-SPMO: Token-Specific Prefix Matching Optimization

Token-Specific Prefix Matching Optimization (T-SPMO) is a token-level RL algorithm that enables fine-grained credit assignment without trajectory-wide statistics. Similarly to GRPO, we sample $|\mathcal{G}|$ completions per prompt. From these completions, we construct a prefix tree to identify all unique (p, v) token transitions, where p is a prefix of tokens and v is the next token.

Each (p, v) pair is assigned a token-level advantage based on the expected change in reward if v is appended to p:

$$A(v \mid p) = \mathbb{E}[R \mid p \circ v] - \mathbb{E}[R \mid p] \tag{7}$$

We approximate these expectations using empirical averages over the sampled completions. Let $\mathcal{G}$ be the set of all sampled completions for a given prompt. We define:

$$\mathbb{E}[R \mid p] = \frac{1}{|\mathcal{G}_p|}\sum_{y\in\mathcal{G}_p} R(y) \tag{8}$$

$$\mathbb{E}[R \mid p \circ v] = \frac{1}{|\mathcal{G}_{p\circ v}|}\sum_{y\in\mathcal{G}_{p\circ v}} R(y) \tag{9}$$

where $\mathcal{G}_p$ is the subset of completions in which p appears as a prefix, and $\mathcal{G}_{p\circ v}$ is the subset in which the prefix is extended by token v.

The objective used to update the policy is:

$$\mathcal{J}_{\text{TSPMO}}(\theta) = \frac{1}{|\mathcal{U}|}\sum_{(p,v)\in\mathcal{U}}\pi_\theta(v \mid p)\cdot A(v \mid p) - \lambda \sum_{W\in\mathcal{W}_{\text{LoRA}}}\|W\|_2^2 \tag{10}$$

where $\mathcal{U}$ is the set of unique $(\text{prefix}, \text{token})$ pairs extracted from all completions, and λ is a regularization coefficient applied to the LoRA parameters $\mathcal{W}_{\text{LoRA}}$.

Consistent with our implementation of S-GRPO, we update the policy once per problem, and forego PPO-style objective clipping.

Replay-Based Resampling: As most generations will diverge from one another near the start of the generation, we introduce a configurable replay mechanic that serves to build prefix trees in later sections of completions. This allows the model to learn strategies past the beginning of generation.

Let $\mathcal{R}$ denote the set of all previously generated completions, each associated with a scalar reward R. We define a threshold value $r \in \mathbb{R}$ and classify completions into:

- **Successful completions:** $\mathcal{R}_{\text{success}} = \{y \in \mathcal{R} \mid R(y) \geq r\}$
- **Failed completions:** $\mathcal{R}_{\text{fail}} = \{y \in \mathcal{R} \mid R(y) < r\}$

At each update step, we may configure replay to sample from either subset. Given a user-defined replay budget of C_{success} and C_{failure} completions, we select C_{success}, C_{failure} completions from $\mathcal{R}_{\text{success}}$ and $\mathcal{R}_{\text{fail}}$ respectively. For each selected completion, we randomly sample a token position t such that

$$t \in [T_{\text{prompt}},\ T_{\text{total}}]\,,$$

where T_{prompt} is the index of the last token in the input question, and T_{total} is the final token index of the full completion. This ensures that replay focuses on the model-generated portion of the sequence.

For each selected completion and its sampled replay position t, we restart generation from the partial sequence corresponding to the prefix $P = (y_1, y_2, \ldots, y_t)$, where y_t is the token at the chosen replay point. From this prefix, we again generate $|\mathcal{G}|$ new completions using the current policy π_θ. These completions are then evaluated with the reward function, a new independent set of $(\text{prefix}, \text{token})$ advantage pairs are collected, and finally the policy model is updated again using the T-SPMO objective function.

D. Full GRPO Baseline

To isolate the impacts of sparse-token sampling, we implement the original GRPO objective (Eq. 3). We again set the number of iterations per sample to 1. The full GRPO objective is thus equivalent to the S-GRPO objective (Eq. 5), except all tokens are included in $\mathcal{T}_i$. We adapt GRPO to our memory constraints using gradient accumulation: We generate $\mathcal{G}$ without gradient enabled, then for each completion in $\mathcal{G}$ we run a forward and backwards pass. See Algorithm 2 for details.

Algorithm 2 Memory Efficient GRPO Training (Per-Completion Backward)

Require: Prompt x, number of completions G, maximum completion length T, optimizer

Sample G completions $\{y^{(i)}\}_{i=1}^{G}$ from policy π_θ without gradients
Compute reward $r(y^{(i)})$ for each completion
Calculate advantages $\hat{A}_i$
optimizer.zero_grad()
for each sampled completion $y^{(i)}$ **do**
    Build input by concatenating x and $y^{(i)}$
    Forward policy π_θ and reference π_{ref} over full input
    Compute GRPO loss using logits of generated tokens, $\hat{A}_i$, and KL-divergence
    Divide loss by G
    loss.backward()
end for
optimizer.step()
optimizer.zero_grad()

IV. Experiments and Results

A. Experiment Setup

We evaluate our reinforcement learning methods on two reasoning benchmarks: the SVAMP dataset and a custom multi-digit multiplication task. Each dataset is handled as a separate training run, with distinct hyperparameter configurations tuned for the nature of the task.

Model and Fine-Tuning Method: All experiments use the Qwen2-1.5B model fine-tuned with Low-Rank Adaptation (LoRA), with only adapter parameters updated and base weights frozen. We insert LoRA modules into the query and value projections of the final attention layers: Last 1/3 attention layers, rank = 16 for all SVAMP experiments, last 1/4 attention layers, rank = 8 for all multiplication experiments. Based on early experiments, we used a higher LoRA rank for SVAMP, reflecting its greater semantic variability relative to multiplication. Fine-tuning is performed on a single partitioned A100 40GB GPU using AdamW optimizer with a learning rate of 1×10^{-4} and regularization coefficients $\beta = \lambda = 0.01$. Training is conducted in float32 precision.

Training Configuration: For both benchmarks, responses are limited to a maximum of 300 tokens, with temperature set to 0.3. Reward is based on exact match accuracy, where the final integer extracted from the model's output is treated as the prediction.

We use an effective batch size of 1 unless otherwise noted. Gradient accumulation is disabled by default.

S-GRPO and the baseline GRPO samples $|\mathcal{G}| = 8$ completions per prompt, while T-SPMO uses $|\mathcal{G}| = 50$ completions.

Other hyperparameter settings were selected from our ablation experiments and listed in the Ablations section.

Training Duration: Training steps for each method–dataset pair were selected based on early-stopping criteria (plateau detection) and manual selection of the checkpoint with peak performance for multiplication. Full details are provided in the Ablations section. Training steps for the baseline GRPO were selected to match those of S-GRPO.

B. Results

We report accuracy on the SVAMP testset and a generated multiplication testset.

Method	Accuracy (%)
Base Qwen2-1.5B	45.0
GRPO	45.3
S-GRPO	70.3
T-SPMO	71.6

TABLE I
SVAMP benchmark test-set (n=300) results showing accuracy and approximate wall-clock training time for each method.

Table 1 shows both S-GRPO and T-SPMO substantially improved performance on SVAMP over the base model. We also find that GRPO did not meaningfully enhance the model's performance compared to the base model.

Method	Accuracy (%)
Base Qwen2-1.5B	3.9
GRPO	4.0
S-GRPO	22.9
T-SPMO	70.0

TABLE II
3 digit x 3 digit multiplication test-set (n=3000) results showing accuracy and approximate wall-clock time for each method.

As demonstrated in Table 2, the base Qwen2-1.5B model struggled significantly on 3 digit x 3 digit multiplication. Our baseline GRPO also did not improve performance. We see noticeable improvement after training with S-GRPO, but an early plateau in training resulted in low albeit improved test

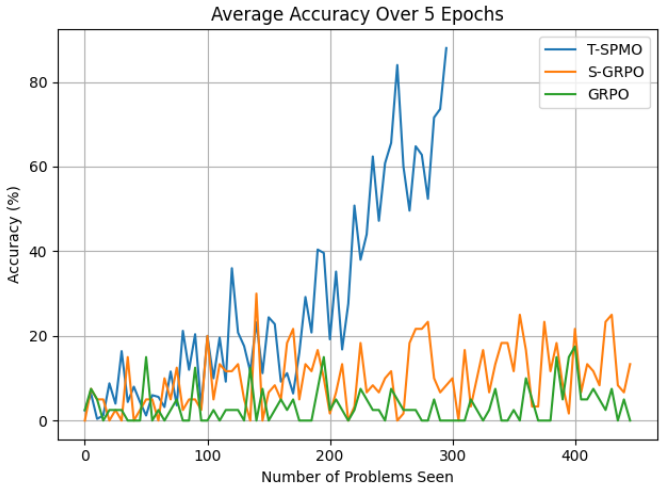

Fig. 1. Training plots of T-SPMO, S-GRPO, GRPO on multiplication of 3 digit x 3 digit integers

set accuracy. T-SPMO on the other hand excelled on this benchmark, significantly outperforming the baseline and S-GRPO.

We include training curves for the multiplication task in Figure 1, where the performance divergence between T-SPMO and S-GRPO is very pronounced.

C. Ablations

We conducted ablations to measure the sensitivity of S-GRPO and T-SPMO to key hyper-parameters. For each algorithm–dataset pair, a pilot sweep with handcrafted settings determined an early-stopping point.

SVAMP: Reward plateauing yielded 400 steps for T-SPMO and 325 for S-GRPO.

3-digit multiplication (more out-of-distribution): we chose the checkpoint with peak validation accuracy after plateauing—350 steps for T-SPMO and 450 for S-GRPO. We also extended T-SPMO training beyond the plateau to verify that additional optimization did not close the performance gap.

All subsequent ablations reused the step counts established for each algorithm–dataset pair to ensure a consistent compute budget and isolate the effect of each hyperparameter.

For S-GRPO, we varied the cutoff α and the maximum number of contributing tokens k. The probability parameter P was not ablated independently, as it is inherently tied to α and k — higher P values are only meaningful when combined higher k and lower α. We also do not ablate the group size $|\mathcal{G}|$, as memory usage (or wall-clock time when using gradient accumulation) scales directly with $|\mathcal{G}|$, making larger values infeasible and out of scope to test. Furthermore, $|\mathcal{G}|$ is not unique to S-GRPO and is a shared hyperparameter with GRPO, making it a lower priority for ablation in our study.

(α, k)	SVAMP Accuracy (%)	Multiplication Accuracy (%)
(0, 100)	66.0	13.0
(50, 100)	70.0	22.9
(100, 100)	70.3	12.7
(25, 50)	68.7	12.6
(0, 25)	69.0	20.9

TABLE III
S-GRPO ablations on α and k values. Test set accuracies on SVAMP (n=300) and multiplication (n=3000).

Across SVAMP, S-GRPO was relatively robust to changes in α and k, with accuracies remaining between 66% and 70.3%.

On the multiplication benchmark, the influence of hyper-parameters was more pronounced, but without a clear pattern. Surprisingly, we find that ($\alpha = 50$, $k = 100$) and ($\alpha = 0$, $k = 25$) achieved stronger performance compared to intermediate values. These results suggest that two distinct strategies may be effective for S-GRPO in challenging problem spaces: (1) handcrafting token selection to align with task-specific structure, or (2) conservatively subsampling a small number of tokens across the completion to reduce noise. In contrast, indiscriminately including a large number of tokens without regard to response structure appears to introduce excessive variance and impair optimization.

We also ablated the effective batch size for S-GRPO by accumulating gradients across multiple prompts before each optimization step (with total problems seen held constant):

Effective Batch Size	SVAMP Acc (%)	Multiplication Acc (%)
1	70.3	22.9
8	62.7	15.0

TABLE IV
S-GRPO ablations on effective batch size. Test set accuracies on SVAMP and multiplication benchmarks.

Increasing the effective batch size to 8 led to a significant degradation in performance across both tasks, likely because updates to the policy occur less frequently.

Finally, for T-SPMO, we ablated the group size $|\mathcal{G}|$ and whether completions were replayed based on success or failure:

($\|\mathcal{G}\|$, C_{Success}, C_{Failure})	SVAMP Acc (%)	Multiplication Acc (%)
(50, 1, 1)	71.0	62.0
(50, 1, 0)	70.0	70.0
(50, 0, 1)	68.7	24.5
(50, 0, 0)	71.6	19.5
(25, 1, 1)	70.0	47.8
(8, 1, 1)	69.3	11.1

TABLE V
T-SPMO ablations on group size and replay strategy. Test set accuracies on SVAMP and multiplication benchmarks.

T-SPMO showed strong performance on SVAMP across all configurations. The multiplication benchmark proved to be more sensitive to hyper-parameters. Decreasing $|\mathcal{G}|$ consistently reducing performance on multiplication, likely due

to insufficient exploration of alternative token transitions during optimization. Running T-SPMO without replay from a successful completion also led to significant performance degradation on multiplication, likely because T-SPMO does not optimize the model for later tokens in completions. Interestingly, for multiplication, including failed completions for replay degraded performance, indicating that forcing the model to recover from early errors may inject too much variance into training.

Overall, these ablations suggest that both S-GRPO and T-SPMO are relatively robust to moderate hyperparameter changes, but benefit from careful tuning on more out-of-distribution tasks.

D. Discussion

We hypothesize that T-SPMO outperforms S-GRPO on the multi-digit multiplication benchmark due to differences in how each method handles token-level credit assignment and partial correctness. While both encourage chain-of-thought reasoning, S-GRPO applies group-level updates across sampled tokens, making it sensitive to isolated errors: a single mistake can disproportionately penalize an otherwise accurate sequence. Our observations suggest that this broad attribution can cause useful subskills to be forgotten, destabilizing intermediate token accuracy.

T-SPMO addresses this by computing advantages for specific (prefix, token) pairs, allowing correct decisions to be reinforced even when later errors occur. This finer-grained credit assignment likely underpins the substantial token-level and final-answer accuracy gains we observe for T-SPMO over S-GRPO on multiplication.

Extending these insights, our GRPO baseline did not meaningfully improve model performance over the base model on either benchmark. This suggests that GRPO may share the sensitivity issues of S-GRPO in an even more pronounced form, particularly when only a small subset of parameters is updated. We caution against interpreting this as evidence that GRPO is an inferior algorithm broadly; rather, it may overfit under LoRA fine-tuning or require more extensive hyperparameter exploration to stabilize.

Overall, we recommend T-SPMO for tasks where token-level correctness is critical, such as arithmetic reasoning, and S-GRPO for settings where broader output structures matter more than local accuracy. Practitioners should weigh the tradeoffs between selective token-level optimization and group-level coherence depending on the demands of the task.

V. Limitations

Our study has several limitations. First, we do not compare against full model fine-tuning baselines, as this was infeasible given our single 40GB GPU constraint. It remains an open question whether full fine-tuning would mitigate some of the challenges we observed with group-level credit assignment.

Second, while we benchmarked a GRPO baseline adapted for LoRA, we did not perform extensive hyperparameter sweeps. GRPO's reliance on gradient accumulation through more forward and backward passes increases wall-clock time, making robust tuning impractical in our setup. As a result, we cannot conclusively claim that our methods will outperform GRPO in more optimized regimes.

Finally, our experiments focus on a single base model (Qwen2-1.5B). Although this choice enables controlled comparisons, future work should validate the generality of our findings across a broader range of model architectures and scales.

VI. Future Work

A promising direction for future research is to evaluate T-SPMO on tasks involving longer and more semantically diverse chains of thought, such as those found in the benchmark MATH. In these settings, surface-level token matching may be insufficient for effective credit assignment. We hypothesize that augmenting the prefix-matching mechanism in T-SPMO with a semantic-aware structure—such as a fuzzy-matching prefix tree based on latent representations or paraphrase clustering—could improve the generalizibility of token-level reinforcement learning and reduce sensitivity to lexical variation.

As for S-GRPO, next steps could include variations on the sampling algorithm for tokens that contribute to loss. While we found that sampling with a strong bias towards the beginning of a completion is a simple and effective method, exploring more sophisticated algorithms would likely be a promising research direction.

VII. Conclusions

We introduced two reinforcement learning methods—Stochastic Group Relative Policy Optimization (S-GRPO) and Token-Specific Prefix Matching Optimization (T-SPMO)—designed to improve reasoning performance in large language models under constrained computational budgets. Both methods are critic-free and operate within a LoRA fine-tuning framework, making them practical for training on a single 40GB GPU. Across two benchmarks, SVAMP and multi-digit multiplication, our approaches significantly outperform zero-shot baselines, with T-SPMO achieving particularly strong results on arithmetic tasks.

In addition, our GRPO baseline, which uses full-trajectory loss attribution, did not meaningfully improve model performance over the base model, suggesting that GRPO may exhibit similar sensitivity issues as S-GRPO, but in a more pronounced form when used with parameter-efficient fine-tuning. This highlights the importance of careful reward attribution under constrained settings, and the potential need for additional stabilization or tuning when adapting full-model RL algorithms to LoRA setups.

Our work targets a complementary problem setting: enabling reinforcement learning for reasoning tasks where full fine-tuning is not feasible. We believe this lightweight and efficient approach fills a critical gap for researchers and practitioners working with limited hardware. Future work will explore extending these techniques to more semantically diverse tasks and investigating scaling strategies that preserve

the efficiency and fine-grained credit assignment benefits of token-level optimization.

REFERENCES

[1] J. et al., "Openai o1 system card," *arXiv preprint arXiv:2412.16720*, 2024.
[2] J. Schulman *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[3] S. et al., "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.
[4] Y. et al., "Qwen2 technical report," *arXiv preprint arXiv:2407.10671*, 2024.
[5] H. et al., "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.
[6] A. Patel, S. Bhattamishra, and N. Goyal, "Are nlp models really able to solve simple math word problems?" in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, 2021, pp. 2080–2094. [Online]. Available: https://aclanthology.org/2021.naacl-main.168/
[7] O. et al., "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.
[8] C. et al., "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.
[9] M. et al., "A diverse dataset for algebra word problems," *arXiv preprint arXiv:2106.15772*, 2021.

APPENDIX A
PROMPT TEMPLATES

A. Multiplication Task

The following template was used for multiplication problems:

```
System: You are a helpful AI
assistant.
User: What's [Number A] multiplied
by [Number B]? Don't use a
calculator.
Assistant: Let's break this problem
down step by step.
```

Here, `[Number A]` and `[Number B]` are placeholders that were replaced with randomly sampled 3-digit integers (between 101 and 999).

B. SVAMP Task

The following prompt template was used for SVAMP problems:

```
Problem: [SVAMP Question]
Answer:
```

In this template, `[SVAMP Question]` denotes a natural language math word problem drawn from the SVAMP dataset. The model was tasked with generating only the final numerical answer following the "Answer:" cue, without explaining intermediate steps.

APPENDIX B
ANSWER PARSING METHODOLOGY

To evaluate model responses, we employed a simple parsing strategy designed to robustly extract the final numerical answer:

- **Comma Removal:** All commas were stripped from the model output to handle large numbers formatted with thousands separators (e.g., "1,234" becomes "1234").
- **Integer Extraction:** After comma removal, we used a regular expression to locate all integers in the response. Specifically, we selected the **last integer** found as the predicted answer.

This method ensures that extraneous text or intermediate calculations produced by the model do not interfere with accuracy evaluation, yet still mostly isolates correctness over formatting to optimize. Only the final numerical answer is considered for grading.