

Extended Abstract

Motivation Large language models (LLMs) often struggle with multi-step arithmetic reasoning, as evidenced by their performance on the Countdown task (combining a multiset of integers to reach a target value). Standard supervised fine-tuning (SFT) yields plausible “chain-of-thought” (CoT) transcripts but does not guarantee correctness; reinforcement-learning approaches like REINFORCE Leave-One-Out (RLOO) improve solve rates but treat verification as a black-box end-of-chain reward. We seek a method that can enable LLMs to self-verify their answers and indirectly achieve precise and concise arithmetic reasoning.

Method We employ Generative Reward Modeling (GenRM) on top of SFT. Essentially, we further fine-tune our best SFT model to generate special verification tokens of the format “<check>Correct/Incorrect</check>”, conditioned on the entire CoT and final expression. This reframing turns verification into a next-token prediction task, providing self-supervision: every arithmetic operator, number, and “=” sign in the CoT affects the likelihood of a correct tag. During inference, we sample multiple candidate responses and select the one with highest $\text{Pr}(\text{Correct} \mid x, y, \text{<check>})$, where x is the prompt and y is the reasoning trace followed by the final answer. Our novelty lies in using the LLM’s own generative head for verification, requiring no extra discriminator or complex reward engineering.

Implementation We start from Qwen-2.5-0.5B and perform SFT on the WarmStart dataset (1k examples) for 4 epochs with a batch size of 8, using AdamW with a learning rate of $5e-5$. We then generate a verification dataset ($\sim 1k$ examples containing the “<check>Correct/Incorrect</check>” tags) using our best SFT model and a deterministic verifier function, over which the LLM is fine-tuned further for 3 epochs with a batch size of 8, using AdamW with a learning rate of $5e-6$. At inference, we generate $N = 10$ candidate responses per prompt with temperature 0.7 and top- $k = 20$, then pick the candidate whose “Correct” probability is maximal, which can be determined from the log probabilities returned by the model.

Results On a held-out Countdown leaderboard, vanilla SFT achieves a solve rate of 30%, RLOO raises this to 52%, and SFT+GenRM alone reaches 47%. Beyond solve rate, GenRM chains show a higher arithmetic-only token ratio, fewer English sentences, 40% lower parentheses depth (implying simpler and therefore more correct computation), and 80% of examples require only one <answer> attempt. A 2D histogram of invalid-fraction vs. revision-magnitude reveals that GenRM more consistently produces valid equations with minimal edits.

Discussion GenRM fine-tuning with a single end-of-chain correctness tag drives the model to produce more arithmetic-centric reasoning with simpler, more consistent structure: it favors numbers and operators over verbose language, yields shallower and less variable nesting of expressions, and, when a nearly correct solution is found, requires only minimal revisions before declaring correctness. This single self-verification signal thus stabilizes generation, leading to concise, reliable, and reproducible chains of thought. However, we acknowledge that the need to generate multiple candidate responses can be computationally expensive and increase inference time.

Conclusion A simple modification—training the model to predict a “<check>Correct/Incorrect</check>” tag—yields substantial gains in both accuracy and reasoning quality on Countdown. This approach offers a lightweight yet powerful mechanism for self-verification, pointing toward a new paradigm for teaching LLMs precise, interpretable chains of thought in arithmetic and beyond.

Improving Chain-of-Thought Reasoning in LLMs via Generative Reward Modeling

Akhilesh Balasingam

Department of Electrical Engineering
Stanford University
avb03@stanford.edu

Axel Wennstrom

Department of Electrical Engineering
Stanford University
axelw@stanford.edu

Vishal Jain

Department of Electrical Engineering
Stanford University
jainvi@stanford.edu

Abstract

Generative Reward Modeling (GenRM) enhances arithmetic reasoning in large language models by training them to self-verify answers using inline verification tokens, rather than relying on external rewards or discriminators. Fine-tuned on a custom verification dataset derived from model-generated outputs and deterministic correctness checks, GenRM reframes verification as next-token prediction, enabling more concise, arithmetic-focused reasoning with minimal structural complexity. Compared to standard SFT, GenRM improves solve rates and yields more stable, interpretable outputs, demonstrating that a lightweight generative verification signal can significantly improve structured reasoning without architectural changes.

1 Introduction

Large language models (LLMs) excel at generating fluent text, yet their ability to perform structured, multi-step reasoning—particularly in domains like arithmetic and symbolic manipulation—remains an open challenge. The Countdown task, in which a model is presented with a multiset of integers $\{n_1, n_2, \dots, n_k\}$ and must construct an arithmetic expression evaluating exactly to a target integer T , has become a canonical benchmark for probing an LLM’s chain-of-thought capabilities. In human problem solving, one typically breaks the task into successive binary combinations of numbers, annotating each intermediate result to ensure correctness. Encouraging LLMs to emulate this pattern—by producing explicit reasoning steps during generation—has been shown to yield more interpretable and accurate solutions than black-box decoding alone. Moreover, Countdown epitomizes a class of problems where reasoning is both compositional (i.e., combining basic arithmetic operations) and sequential (i.e., planning a sequence of operations), making it an ideal setting to study how to teach LLMs to “think” through a problem rather than simply memorize end-to-end mappings.

In our work, we adopt the well-established pipeline of supervised fine-tuning (SFT) followed by policy refinement via reinforcement learning—in particular, REINFORCE Leave-One-Out (RLOO)—to bootstrap the model’s initial reasoning capabilities and then optimize for correct final answers. While SFT exposes the model to gold-standard, stepwise solutions and RLOO nudges it toward higher reward (i.e., correct arithmetic expressions), these components are not the primary novelty of our contribution. Our research question was if we can find a method that can enable LLMs to self-verify their answers, without the need for extra discriminator or complex reward engineering. Inspired by Zhang et al. (2025), our approach recasts reward modeling as a next-token generation task that trains the model to

self-verify its reasoning and final answer by predicting a “<check>Correct/Incorrect</check>” tag, thereby enabling the model to internalize what “correct” means and recognize its own mistakes.

In the sections that follow, we will review the different algorithms we explored, how they were implemented with the relevant data and training setup, and how they fare against each other on the set of evaluation metrics relevant to the Countdown task, as well as attempt to understand why one approach might perform better than the other.

2 Related Work

Chain-of-Thought (CoT) prompting has emerged as a powerful technique for eliciting multi-step reasoning from large language models. Wei et al. (2022) first showed that providing few-shot examples with explicit intermediate steps can dramatically improve performance on arithmetic, commonsense, and symbolic reasoning benchmarks such as GSM8K, as models learn to decompose problems into sequential subproblems rather than directly generating final answers. Subsequent studies have applied CoT prompting to domains like logical deduction and multi-hop question answering, demonstrating that even moderately sized models can achieve strong reasoning accuracy when guided by structured exemplars (Press et al., 2022; Kojima et al., 2023). However, CoT prompting remains an inference-time strategy and does not modify model parameters to internalize reasoning patterns during training.

To provide stronger training-time supervision for reasoning, researchers have developed various verifier and reward modeling techniques. Early approaches used discriminative classifiers to score completed chain-of-thought transcripts and select the highest-scoring candidate among multiple generations (Cobbe et al., 2021; Nie et al., 2022). Although these discriminators can filter out semantically incorrect solutions, they treat verification as a post-hoc, black-box task and do not exploit the LLM’s generative capacity. Iterative self-consistency methods generate multiple reasoning traces and aggregate them via majority voting, improving robustness but still relying on discrete vote aggregation over full solutions (Wang et al., 2022; Chung et al., 2022). More recently, Zhang et al. (2025) introduced the Generative Reward Model (GenRM), which reframes verification as a next-token generation task: which could be as simple as fine-tuning a model to predict the text “Is the answer correct (Yes/No)?” followed by a “Yes/No” token. We adapt this simple yet effective approach by customizing the verification tokens for the Countdown task, and further analyzing the impact of this approach through token-level statistics.

3 Method

3.1 Dataset Construction and Preprocessing

We leverage two publicly available Countdown datasets for this work, as well as a generated verification dataset:

1. WarmStart Dataset (https://huggingface.co/datasets/Asap7772/cog_behavior_all_strategies) - This dataset contains 1000 “train” examples and 200 “test” examples of well-formatted natural language queries and completions for the Countdown task. The queries contain the relevant inputs, instructions and example formatting, while the completions contain correct reasoning traces between <think> tags and the expected final expression between <answer> tags. Thus, this dataset is ideal for vanilla SFT, and the corresponding preprocessing involves extracting the input ids, attention masks, and labels after tokenization and dynamic padding using a custom collator that checks for maximum length of input ids at a batch level.
2. TinyZero Dataset (<https://huggingface.co/datasets/Jiayi-Pan/Countdown-Tasks-3to4>) - This dataset contains 490k examples of input numbers and target number for the Countdown task. This serves as a large pool of questions that can be used to generate input prompts using the same prompt template as the WarmStart dataset, which can then be leveraged for both model evaluation and on-policy generation for RLOO.
3. Verification dataset - We also use a sample of the TinyZero dataset to generate ~ 1200 examples for our verification dataset for the GenRM approach. For every input prompt, we

save one correct and one incorrect response generated by our best SFT model, which are correspondingly appended with the appropriate verification tokens and the end of sequence token, while the prompt itself is appended with a verifier instruction (see Section 3.4 for details).

3.2 Supervised Fine-Tuning

In the supervised fine-tuning (SFT) stage, we adapt the pretrained *Qwen-2.5-0.5B* model π_θ to the Countdown task by maximizing the likelihood of gold CoT transcripts. Given a dataset $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$, where each prompt x_n specifies a set of integers and target T and $y_n = (y_{n,1}, \dots, y_{n,T_n})$ is the corresponding stepwise solution, we train

$$\pi_\theta(y_n \mid x_n) = \prod_{t=1}^{T_n} \pi_\theta(y_{n,t} \mid y_{n,< t}, x_n),$$

and minimize the cross-entropy loss $-\frac{1}{N} \sum_n \log \pi_\theta(y_n \mid x_n)$. Fine-tuning proceeds for a fixed number of epochs with AdamW and hyperparameters (learning rate, batch size, dropout) selected by held-out validation (Section 5.1.1). This SFT model then serves as the basis for subsequent reward-based refinements like RLOO or GenRM.

3.3 REINFORCE Leave-one-out (RLOO)

As described in the default project specification, we apply the standard RLOO update to our SFT policy π_θ on Countdown: for each prompt x , we sample K chain-of-thought transcripts $\{y^{(i)}\}_{i=1}^K \sim \pi_\theta(\cdot \mid x)$, score each with a deterministic reward $r^{(i)}$, and compute the leave-one-out baseline $\bar{r}^{(-i)}$. The resulting gradient estimator is

$$\nabla_\theta \mathcal{J}(\theta \mid x) = \frac{1}{K} \sum_{i=1}^K (r^{(i)} - \bar{r}^{(-i)}) \nabla_\theta \log \pi_\theta(y^{(i)} \mid x).$$

We then average over a minibatch of prompts and perform an AdamW step; hyperparameters (e.g. K , learning rate, batch size) are detailed in the Appendix.

3.4 Generative Reward Modeling

In standard SFT+RLOO pipelines, a separate verifier or rule-based checker assigns a binary reward to a fully generated chain-of-thought solution. By contrast, Generative Reward Modeling (GenRM) reframes verification as a next-token prediction task: given a prompt x and a candidate transcript $y = (y_1, \dots, y_T)$ (including both chain-of-thought steps and the final answer), the same LLM architecture is fine-tuned to generate a "`<check>`" tag whose next token is either "Correct" or "Incorrect". In this way, our GenRM implementation leverages the model's autoregressive scores directly to estimate a scalar reward.

Formally, let $\mathcal{D} = \{(x_i, y_i^{\text{SFT}})\}_{i=1}^N$ be the set of prompt-response pairs generated by our best SFT-only model, where each y_i^{SFT} ends with an arithmetic expression between the `<answer>` tags. For each x_i , we construct two examples: one where y_i^{SFT} yields a correct answer and one where it yields an incorrect answer. Each example is appended with a verification string and a token indicating correctness:

$$\mathcal{D}_{\text{GenRM}} = \{(x_i + I, y_i, +<\text{check}>z_i</\text{check}>)\}_{i=1}^N,$$

where $z_i \in \{\text{Correct}, \text{Incorrect}\}$, and I is a fixed instruction appended to the prompt that defines the verification task:

```
"Verify the final answer in <check> </check> tags, for example <check>
  Correct </check> if the answer is correct, and <check>Incorrect </
  check> if the answer is incorrect.\n"
```

This acts as our verification dataset using which we can perform further SFT on the model used to generate the dataset. At inference time, we score any candidate response y by computing the likelihood of predicting `Correct` given the full input sequence:

$$r_{\text{GenRM}}(x, y) = \Pr(\text{Correct} \mid x + I, y, <\text{check}>).$$

This score captures the model’s generative confidence in the correctness of the answer, allowing it to act as a scalar reward for ranking candidate responses, thus enabling best-of- N selection to choose the most likely correct solution without requiring any explicit reinforcement learner.

3.4.1 Format

To implement GenRM, we experimented with several tagging conventions before settling on the format below:

$\underbrace{<\text{think}> \dots </\text{think}>}_{\text{chain-of-thought}} \underbrace{<\text{answer}> \dots </\text{answer}>}_{\text{final arithmetic expression}} \underbrace{<\text{check}> \{\text{Correct, Incorrect}\} </\text{check}>}_{\text{verification token}}$

Each record in $\mathcal{D}_{\text{GenRM}}$ thus looks like:

$x \parallel <\text{think}> y_1 y_2 \dots y_{T_i} </\text{think}> <\text{answer}> \alpha_i </\text{answer}> <\text{check}> z_i </\text{check}>$,

where:

- x is the prompt (i.e., “Using the numbers $\{n_1, \dots, n_k\}$, reach target $T \dots$ ”).
- y_1, \dots, y_{T_i} is the human-like chain of arithmetic steps (sampled from SFT).
- α_i is the final arithmetic expression that should evaluate to T .
- $z_i \in \{\text{Correct, Incorrect}\}$ is computed by plugging α_i into a deterministic checker.

An example of a different format we experimented with was following Zhang et al. (2025) directly and using the instruction “Is the answer correct (Yes/No)?” followed by a “Yes/No” token, but we observed that the models trained on this dataset did not perform very well. We hypothesize that the $<\text{check}>$ tags are more “in-distribution” with the $<\text{think}>$ and $<\text{answer}>$ tags, and hence are easier to learn.

4 Experimental Setup

Our optimal training configurations for SFT, RLOO, and GenRM variants are discussed in the sections below. We trained on L40S and H100 NVIDIA GPUs provided by the Stanford Sherlock cluster, and used optimizations like gradient accumulation and gradient checkpointing to work around out-of-memory issues for larger batch sizes.

5 Results

5.1 Quantitative Evaluation

Table 1: Performance Comparison on Held-out Leaderboard Set

Method	Solve Rate
Baseline	0.00
SFT	0.30
RLOO	0.52
GenRM + SFT	0.47

5.1.1 SFT and RLOO

For SFT, we ran a sweep over learning rates between $1e-4$ and $1e-6$ (keeping the optimizer fixed as AdamW with a weight decay of 0.01), and found that a learning rate of $5e-5$ proved most effective (*Perky Shrew*). For readability, we only include the training loss and validation curves for two training runs in Figure 1b. For RLOO, the average reward curves on our heldout-validation set are shown in Figure 1a for various configurations of batch sizes and rollouts. We found that a batch size $B = 64$ and rollouts $K = 16$ proved most effective in stabilizing loss. Our Appendix

provides more details on the training configuration. In Table 1, we see that both SFT and RLOO give substantial improvements in score compared to the base Qwen-2.5-0.5B model. RLOO builds on SFT’s initial improvement by reinforcing on examples with bigger rewards (i.e., more correct examples).

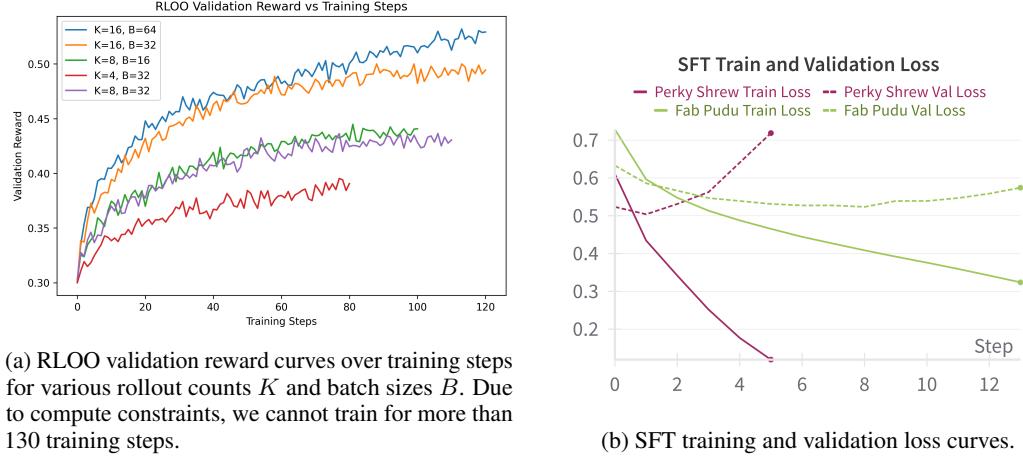


Figure 1: Training metrics for SFT and RLOO.

5.1.2 SFT + GenRM

To maximize performance of the SFT + GenRM model, we evaluate the performance using different combinations of sampling parameters, see Figure 2. Out of the tested combinations, the optimal one has a temperature of 0.7, top-k of 20 and 10 generations per prompt. In general, we see that more generations per prompt gives better results. This is expected, as more generations give a higher probability of finding one that is correct. It is also clear that temperature plays a bigger role for performance than top-k. Temperatures between 0.3 and 0.7 perform the best, which also aligns with expectations, as a too low temperature will reduce creativity and variety in outputs, and a too high temperature will make the model perform worse overall due to too much randomness.

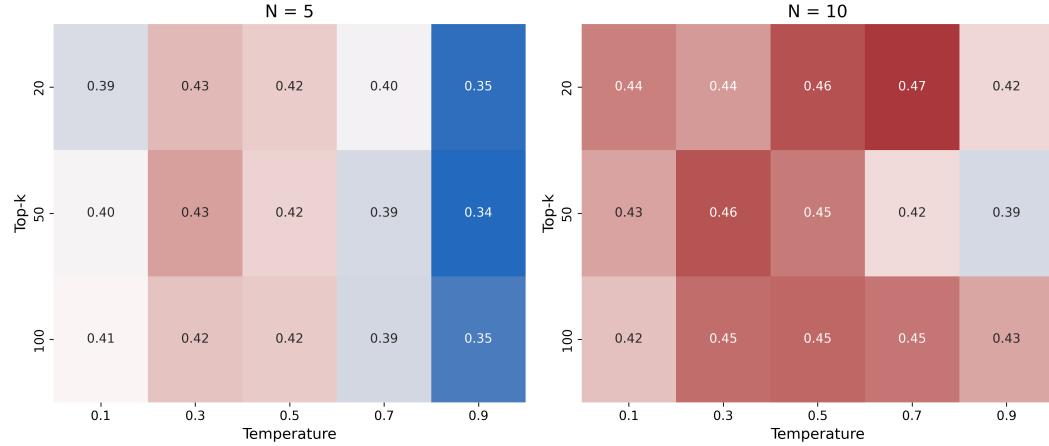


Figure 2: Solve rate heatmaps. N is the number of generations per prompt.

5.2 Qualitative Analysis

5.2.1 SFT and RLOO

Sample model outputs are shown in the Appendix. Under supervised fine-tuning, the model generates detailed but often meandering chains, relying on lengthy English explanations and exploratory

arithmetic that frequently detours through incorrect sub-expressions before arriving at a solution. Applying RLOO sharpens this behavior by reinforcing correct final answers, so the model more quickly abandons completely unpromising chains, yet it still produces verbose rationales and requires several trial-and-error attempts in many cases. In both settings, there is no internal verification during generation, so the CoTs remain dominated by natural-language elaboration rather than concise, self-validated arithmetic steps.

5.2.2 SFT + GenRM

CoT Related Metrics Here we compare the vanilla SFT output against the single best-of- N GenRM generation (chosen by highest $\Pr(\text{Correct} \mid x, y, \langle \text{check} \rangle)$) on the Countdown leaderboard. Both models produce chain-of-thought (CoT) transcripts and final arithmetic expressions, and we measure the arithmetic-only token ratio: we split each CoT on whitespace, count how many tokens consist solely of numbers or arithmetic symbols (e.g. digits, $+$, $,$, \times , \div , parentheses), and divide by the total token count. Figure 3 illustrates how GenRM’s best attempt yields shorter, more arithmetic-focused, and more verifiable CoTs compared to SFT.

We also quantify structural simplicity by measuring the parentheses-nesting depth of each final arithmetic expression. Finally, to assess the magnitude of corrections when the model backtracks (i.e., “I apologize...”), we compute a revision magnitude by taking the Levenshtein (edit) distance between each pair of such revisions. We then average this distance by the length of the chain-of-thought between them.

Figure 4 further demonstrates that GenRM chains not only exhibit shallower nesting and therefore simpler logic, but also concentrate in the low-invalid-fraction, low-revision-magnitude regime, confirming minimal backtracking and higher confidence in a single pass. More discussion can be found in the following section.

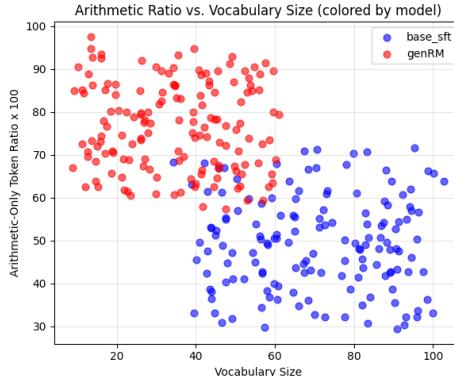


Figure 3: Arithmetic-Only Token Ratio vs. Vocabulary Size (defined as distinct whitespace-separated words)

Analysis of Generated Samples Sample generations can be found in the appendix. Inspecting the raw GenRM generations (Appendix A) reveals that the model often enters a brief self-correcting loop—making small arithmetic edits between successive ‘`<answer>`’ attempts marked “`<check>Incorrect</check>`” before arriving at a final “`<check>Correct</check>`”. These transcripts are dominated by pure numeric steps (e.g., “90 75 = 15; 15 + 5 = 20; 20 + 1 = 21”) with minimal English commentary, and they always append the verification tag immediately after the valid expression, demonstrating concise, self-verified reasoning.

6 Discussion

6.1 Baseline SFT and RLOO Performance

The base SFT model, trained to imitate human-annotated chain-of-thought transcripts, achieves only modest solve rates on the Countdown leaderboard. Although its CoTs mimic the multi-sentence,

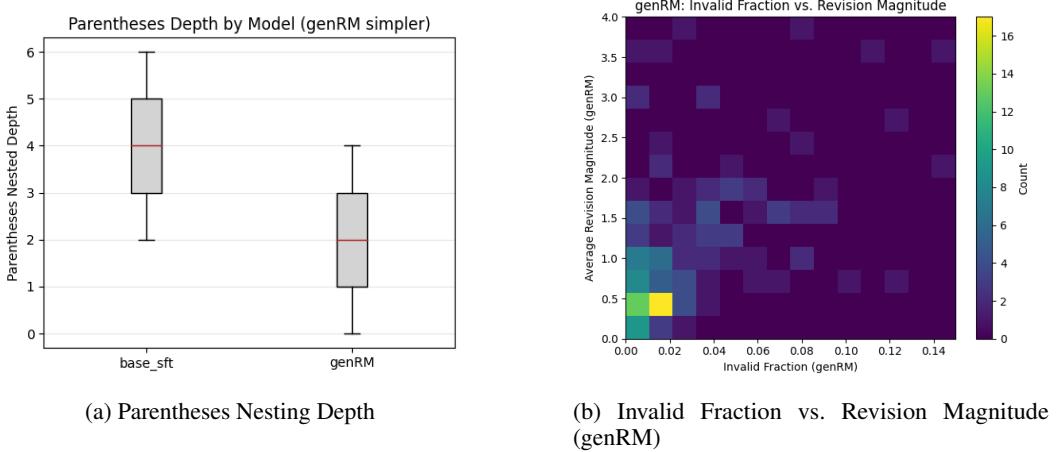


Figure 4: Quantitative comparisons of key CoT-related metrics. (a) genRM produces shallower (or simpler) arithmetic expressions (lower average parentheses nesting depth). (b) In genRM alone, most chains lie in the lower-invalid-fraction, lower-revision-magnitude quadrant, reflecting fewer incorrect or heavily revised attempts.

heavily verbal style seen in training data, they often terminate in incorrect arithmetic or deeply nested, error-prone expressions. Applying RLOO on top of SFT yields a measurable improvement in final answer correctness (Table 1), since RLOO explicitly optimizes for the reward (correct vs. incorrect) at the end of each generated chain. However, RLOO still relies on a black-box reward computed after full generation, which means that intermediate reasoning steps are not directly supervised for correctness. As a result, RLOO’s CoTs remain relatively verbose (high sentence counts and large vocabulary) and occasionally use multiple attempts before converging. In other words, while RLOO helps the model prefer final answers that evaluate correctly more often, it does not fundamentally change the way the model distributes probability mass over arithmetic tokens vs. natural-language tokens or over shallow vs. deeply nested expressions.

6.2 GenRM + SFT Improvements

GenRM fine-tuning, which appends a single `<check>` tag at the end of each chain and selects the highest-scoring completion, produces markedly more arithmetic-focused reasoning. As shown in Figure 3, GenRM chains concentrate in the upper-left: they achieve high arithmetic-only token ratios (i.e. most tokens are numbers or operators) while using relatively small vocabularies, whereas SFT chains scatter toward lower ratios and larger vocab sizes.

Figure 4(a) confirms that GenRM also yields structurally simpler expressions: average parentheses nesting depth is roughly halved compared to SFT. Finally, the heatmap in Figure 4(b) shows most GenRM chains lie in the low-invalid-fraction, low-revision-magnitude quadrant, indicating that once a nearly correct expression is found, only minimal edits are needed before the model confidently emits “Correct.” Together, these patterns demonstrate that even a single end-of-chain correctness tag can steer the model toward concise, self-verified arithmetic reasoning.

Beyond the central tendencies, we also observe that GenRM dramatically reduces variability in reasoning structure. In Figure 4(a), the interquartile range of parentheses depth for GenRM is much narrower than for SFT, indicating that GenRM not only simplifies expressions but does so consistently across examples. Likewise, the sparse upper-right region of the invalid-fraction vs. revision-magnitude heatmap (Figure 4(b)) shows very few outliers—GenRM almost never produces chains that both fail the countdown rules frequently and require large rewrites. These patterns underscore that a single end-of-chain correctness signal not only drives arithmetic-focused reasoning, but also stabilizes the entire generation process, yielding reliable and reproducible chain-of-thoughts.

6.3 Reflection on Challenges Faced

During this project, we encountered several practical hurdles. The RLOO implementation proved especially finicky—high gradient variance required extensive debugging in office hours, careful tuning of batch sizes and learning rates, and tracking down subtle indexing bugs in our loss calculation. Incessant out-of-memory issues meant limiting batch sizes to as low as 1, implying extremely long training runtimes, for which the AWS Spot instances proved to be very unreliable. This made us resort to the Sherlock cluster for training and evaluation, however, compute constraints meant we could only reserve a GPU for a maximum of 8 hours of working, which meant we needed to add scaffolding to our train code to save the current state of the model and other hyperparameters like learning rate, to train for longer windows. This also forced us to compress experiments and only allowed us to run a minimal sweep of hyperparameters compared to the full grid-search we desired.

7 Conclusion

We have demonstrated that appending a simple end-of-chain “`<check>Correct/Incorrect</check>`” tag to an SFT-trained LLM—and using its likelihood to select best of N generated chains—yields substantial improvements on the Countdown arithmetic task. Without any additional discriminator or complex reward engineering, our GenRM+SFT model achieves higher solve rates, more concise and arithmetic-focused CoTs, and fewer error-prone rewrites as compared to vanilla SFT. Qualitative analyses confirm that a single correctness signal at the end of generation is sufficient to guide the model toward succinct, reliable reasoning traces. Future work can extend this lightweight verification scheme to other symbolic and logical reasoning benchmarks.

8 Team Contributions

- **Akhilesh Balasingam:** RLOO experiments
- **Vishal Jain:** GenRM+SFT experiments
- **Axel Wennstrom:** GenRM+SFT experiments
- **Common:** Data setup and evaluation

Changes from Proposal We now chose to focus on the Generative Reward Model extension, which we found as a more effective way to improve solve rates compared to calculator tool use. Quantitative and qualitative analysis of SFT and RLOO results suggest that incorrect computation itself was not the main roadblock to high solve rates; it was the approach to the problem—the chain-of-thought reasoning—that required improvement.

References

- Hyung Won Chung, Jean-Baptiste Perez, Bhargav Jain, Tomas Kociský, and Mikkel Ringgaard. 2022. Ensemble Methods for Chain-of-Thought Reasoning. In *Proceedings of the 2022 Conference of the North American Chapter of the ACL*. 250–263. <https://aclanthology.org/2022.naacl-main.17>
- Karl Cobbe, Ilya Kostrikov, Jacob Hilgard, Edward D. Grefenstette, and John Schulman. 2021. Leveraging Procedural Knowledge for Zero-Shot Generalization. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=IY0qgY0qT2>
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models Are Zero-Shot Reasoners. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=ZNzpoRTbuZ>
- Yixin Nie, Yonatan Belinkov, and Yoshua Bengio. 2022. Analyzing and Improving MMLU Performance of Chain-of-Thought Prompting. *Transactions of the Association for Computational Linguistics* 10 (2022), 837–852. <https://aclanthology.org/2022.tacl-1.56>

Ofir Press, James Minervini, and Luke Lewis. 2022. Learning to Reason: Combining Chain of Thought Prompting with Structured Symbolic Representations. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2127–2137. <https://aclanthology.org/2022.emnlp-main.150>

Andrew Wang, Mark Yatskar, Yichong Zhao, Yejin Choi, and Veselin Stoyanov. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *Advances in Neural Information Processing Systems*. <https://proceedings.neurips.cc/paper/2022/file/43af4f5be1b1279e76c0b3f3d3c4d54d-Paper.pdf>

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Chi, Quoc Le, Ed Song, Shang-Wen Xia, Xuezhi Zhou, and Tong Luan. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*. <https://proceedings.neurips.cc/paper/2022/file/8147c6a10a7727f3d9eff10acf5a04b-Paper.pdf>

Yizhe Zhang, Lin Tran, Rishabh Pang, and Dan Roth. 2025. Generative Reward Modeling for Fine-Grained Verification in Large Language Models. *arXiv preprint arXiv:2408.15240* (2025). <https://arxiv.org/abs/2408.15240>

A Implementation Details

RLOO Hyperparameters Best performing hyperparameters were B=64, K=16, TEMP=0.6, TOP_P=0.95, with AdamW, LR=1e-6, cosine scheduler with 0 warmup steps.

B SFT vs SFT+GenRM Sample Generation

B.1 SFT Generation (Incorrect)

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.

User: Using the numbers [12, 71, 17, 99], create an equation that equals 1. You can use basic arithmetic operations (+, -, *, /) and each number can only be used once. Show your work in `<think> </think>` tags. And return the final answer in `<answer> </answer>` tags, for example `<answer> (1 + 2) / 3 </answer>`.

Assistant: Let me solve this step by step.

Assistant: `<think>`
Let me try to find a way to get to 1 using these numbers.
First, let me look for factors or multiples of 1 that could help.
1 is very small compared to all the given numbers.

Let me try some operations:
71 - 17 = 54 (too big)
99 - 71 = 28 (too small)
12 * 17 = 168 (too big)

Let me try a different approach:
99 - 71 = 28
28 * 12 = 336 (too big)

Let me work backwards from 1:
To get 1, I need operations that can reduce these large numbers.
What if I try division with the largest numbers?
99/99 = 1

But I can't use 99 twice.

Let me try another approach:
71 - 17 = 54
99 - 71 = 28
54/28 1.86 (not helpful)

New approach:
17 * 12 = 204
99 - 71 = 28
204/28 6.87 (not helpful)

Let's try:
99 - 71 = 28
28 * 12 = 336
336/17 19.29 (not helpful)

Final attempt:
71 - 17 = 54
99 - 54 = 45
45 * 12 = 540
540/17 32.94 (not helpful)

I need to try a different sequence:
17 * 12 = 204
99 - 71 = 28
204/28 6.87

Let me try one more time:
99 - 71 = 28
28 * 12 = 336
336/17 19.29

I need to find a sequence that gives exactly 1.
Let me try:
71 - 17 = 54
99 - 54 = 45
45 * 12 = 540
540/17 32.94

Final attempt:
17 * 12 = 204
99 - 71 = 28
204/28 6.87

I apologize, but I cannot find a valid sequence of operations that will give exactly 1 using these numbers with the given operations.

</think>
<answer> (99 - 71) * 12 / 17 </answer>

B.2 SFT+GenRM Generation (Correct)

<think>
Let me analyze this step by step:

1. First, let's try to get to 1 by working with the numbers:
- 71 is much larger than 1, so we'll need to reduce it
- 99 is also larger than 1

- 12 and 17 are smaller numbers that could help adjust

2. Let's try dividing 99 by something:
 $99/3 = 33$ (but we don't have 3)

3. Let's try subtracting numbers:
 $71-17 = 54$ (getting closer to 1)
 $71-12 = 59$ (still too high)

4. Let's try multiplication with small numbers:
 $12*17 = 204$ (too high)

5. What about subtraction of the largest numbers:

$99-71 = 28$

$99-17 = 82$

$71-12 = 59$

6. Let's work backwards from 1:

- To get 1, we need operations that can reduce these large numbers
- One approach could be to divide the largest numbers and subtract
- $(99-71) = 28$
- $(99-17) = 82$
- $(71-12) = 59$
- $82-28 = 54$
- This isn't helping reach 1

7. Let's try a different approach:

$71+17 = 88$

$99-12 = 87$

$88-87 = 1$

This works! We've found it!

</think>

<answer> $(71+17)-((99-12))$ </answer>

<check>Correct</check>