Extended Abstract

Motivation Current LLMs still struggle with logical reasoning related tasks as these tasks demand accuracy on a granular individual step level. Traditional solutions for these problems involve fine-tuning models which require access to model weights and significant computational resources. Test-time compute scaling methods like sampling and prompt revisioning offer an alternative low barrier approach that help increase accuracy. However, sampling methods like best-of-n suffer from false-positive issues where individual predictions cannot be relied upon while revisioning methods like traditional Chain-of-Thought (CoT) can have prompt bloat where the prompts contain redundant and duplicated information.

Method This work investigates a novel test-time compute scaling modification to the Chain-of-Thought (CoT) verification revisioning methodology involving precomputation and optimization to help address these problems. In the approach, the information garnered from each subsequent inference is aggregated and deduplicated to keep prompt size to a minimum. Additionally, precomputations are done and injected into the initial prompt which can be used by the model to prevent hallucinations and inaccuracies. The Countdown task (Gandhi et al., 2024) is used as a case study for this investigation which involves creating an equation using a list of numbers that evaluates to a target value using the +, -, *, and / operators. Each number provided has to be used exactly once and no other numbers may be used. Two baselines are explored in this work: Supervised Fine-tuning (SFT) and REINFORCE Leave One Out (RLOO) models. CoT revisioning based inference is run on these baseline models based around precomputation, extraction, verification, and deductions. As part of these four phases, model generations are analyzed, fixed, and the byproducts are kept track of in an oracle which are then fed into a revised prompt.

Implementation Qwen 2.5 0.5B is used as the base model for this work. SFT finetuning is done using the warmstart dataset (Gandhi et al., 2025) which consists of 1000 query and completions. RLOO is trained using 5000 examples from Countdown dataset which contains input numbers and the target value. 1000 Countdown problems are used as the validation set and the two stage reward score from Pan et al. (2025) is used to calculate the average reward score. A 1.0 score is given to examples with correct responses, and a 0.1 score is given to incorrect responses that have the correct format. The final best SFT and RLOO models were chosen based on the validation score after which optimized CoT was run on both to measure the performance differential.

Results The RLOO baseline model outperformed the baseline SFT model (0.397 vs. 0.274 average validation reward score). The modified CoT demonstrated substantial improvements to the performance of both RLOO and SFT for Countdown - increasing the validation score by 131% for SFT model inference and 72% for RLOO model inference. Additionally, inference costs increased by 2.55x on average for succcessful evaluations when running modified CoT inference.

Discussion While optimized CoT showed substantial results for the Countdown task, there were limitations in conveying more complex reasoning across iterations mainly owing to the fact that more sophisticated deduction mechanisms would be necessary to extract the insights. Additionally, there was still some inability in guiding the inferences towards unexplored sample spaces and avoid some of the "logical flows" from the previous inferences. As far as extending this method to other problems goes, the Countdown task allowed for extraction and deduplication of computations. Other tasks may require more complex strategies for identifying and storing useful intermediate step information. Nevertheless, this work serves as a good reference starting point for implementing optimized CoT in other domains.

Conclusion CoT test time compute scaling is a more accessible way for substantial accuracy improvements on logical deduction related tasks.The results indicate that models with varying levels of performance and accuracy can all get significant benefit with optimized CoT. There is a lot of potential for this work to go even further. An interesting avenue to explore could be to create a dataset based on CoT responses by a stronger LLM model which could then be used as a dataset for finetuning models. These methods will drive reducing dependency on the LLM's intrinsic capabilities and allow for more control, leading to higher accuracies on such logical reasoning tasks with more robustness.

Improving LLM Reasoning Through Optimized Chain of Thought Revisioning and Precomputation

Akhil Vyas
Department of Computer Science
Stanford University
avyas21@stanford.edu

Abstract

Current LLMs still struggle with logical reasoning related tasks as these tasks demand accuracy on a granular individual step level. Traditional solutions for these problems involve fine-tuning models which require access to model weights and significant computational resources. Test-time compute scaling methods like sampling and prompt revisioning offer an alternative low barrier approach that help increase accuracy. However, sampling methods like best-of-n suffer from false-positive issues where individual predictions cannot be relied upon while revisioning methods like Chain-of-Thought (CoT) can have prompt bloat where the prompts contain redundant and duplicated information. In this work, a new optimized CoT methodology is proposed that takes advantage of precomputation and integrates verification of intermediate reasoning steps while minimizing prompt redundancy, enhancing both interpretability and final accuracy. Applied to the Countdown task, this method demonstrates substantial improvements - increasing the validation score by 131% for baseline SFT model inference and 72% for traditional RLOO model inference. While optimized CoT-based inference has a trade-off in the form of increased inference time (around 2.55× slower on average), our findings show that it is a practical and effective post-training strategy for improving logical reasoning in LLMs in a more cost-effective way that requires less specialized ML knowledge.

1 Introduction

Logical reasoning related tasks remain one of the most prominent challenges for even the most recent, state-of-the-art LLMs. This difficulty arises from the need for high accuracy and precision across multiple steps which can not be accounted for with just general knowledge. The inherence mechanism of LLMs which predict most likely next tokens becomes a hindrance in effectively solving these problems. Often, an approach to help address these issues is to finetune the underlying model with large amounts of training data. However, the problem is that this requires access to the underlying weights of the models and fine-tuning models is a pain-staking process that requires a specialized background on machine learning and is generally expensive requiring large scale computation. Additionally, fine-tuning requires the creation of well-curated datasets for effective training.

There is an alternative to model fine-tuning which is to have multiple passes at test time inference without altering the model weights. These methods typically involve multiple inference passes and aggregate the outputs to get a final result. This includes techniques like best of n where n candidates are generated and the best inference is chosen. The problem with these sorts of methods is that it requires having the ability to be able to determine which result is better from another which can also require some additional effort and insight.

Other approaches look to run multiple iterations sequentially instead of in parallel like Chain-of-Thought, which look to revise instructions by verifying and fixing the intermediate steps and passing the corrected input into the model. This allows models to correct mistakes and use past experience for higher accuracy. However, there are some problems with the vanilla Chain-of-Thought - there can quickly be prompt bloat with too much extraneous and redundant information being carried over to the next iteration which can lead to more accuracy issues as a result of longer contexts.

This project explores a refined Chain-of-Thought (CoT) revisioning methodology with two main goals:

1. Optimize the information retained between iterations to keep the prompt size to a minimum.

2. Precompute any information to give the model a reference to prevent hallucinations and inaccuracies.

In this method, the model's intermediate reasoning is individually verified for correctness which is then stored off into an *oracle* with duplication checks. The oracle is then used to revise and improve the model's inference while reducing redundancy. This process not only aims to increase the accuracy of intermediate steps but also improves the interpretability and robustness of the final output for every single inference.

The Countdown task (Gandhi et al., 2024) is used as a case study for this work. The Countdown task involves creating an equation using a list of numbers that evaluates to a target value using the +, -, *, and / operators. Each number provided has to be used exactly once. No other numbers may be used.

By focusing on post-training improvements, this project explores a more computationally efficient alternative to full model fine-tuning, offering a promising direction for more performant LLMs for a variety of use cases outside the scope of what was used to train the LLM initially.

2 Related Work

Recent advances in LLM performance have been significantly driven by inference time scaling exemplified by models like DeepSeek R1 (DeepSeek-AI et al., 2025). Snell et al. (2024) classifies such methods into two broad categories. The first category consists of approaches that modify the proposal distribution via modifying the input prompt through sequential inferences - CoT being a prime example. The second category consists of verifier function based approaches, where multiple samples are generated and ranked in order to pick the best response. While both of these paradigms have been explored independently in numerous works, there has been limited investigation into combining the two. **This project seeks to bridge the gap by integrating verification into intermediate outputs and using insights on those to guide and revise subsequent inference steps.**

There have been several studies that have explored verifier and CoT approaches in the context of natural language math problems. Cobbe et al. (2021) evaluated both finetuning and verifier methods on the GSM8K grade school math natural language problems dataset. Their approach involved generating multiple completions, ranking them using a verifier model, and selecting the top scoring completion. A notable finding of this work was that verification scaled more effectively than a finetuning baseline with increased data. Separately, Wang et al. (2023) proposed a self-consistency methodology for CoT that was explored for arithmetic and commonsense reasoning benchmarks that instead of greedily decoding samples, also generated various samples and then picked the most consistent answer. However, both of these works focus on verifying the final result or the entire reasoning for ranking and selection which leads to more flakiness for the intermediate steps and more false positives. Additionally, using best of n leads to less confidence for single predictions while also requiring functions to be able to rank the many samples.**This work looks to make single prediction chain more robust without the need to generate multiple samples by being able to garner all the information learnt from past passes**. Additionally, other CoT approaches focus mainly on correcting and continuing the train of thought. This is not fully efficient and can lead to prompt bloat, introducing repetitive or irrelevant content that harms efficiency and model focus. In this work, **information is extracted and condensed before being passed forward with a more optimized prompt.**

3 Method

3.1 Baselines

We evaluate our approach on two widely used learning strategies: Supervised Fine-Tuning (SFT) and REINFORCE Leave-One-Out (RLOO) which is a popular fine-tuning method.

3.1.1 Supervised Fine-Tuning (SFT)

SFT uses a labeled training set to maximize the likelihood of completion y given query x. Training objective optimized over queries x and completions y. SFT optimizes the following objective function:

$$\max_{\theta} \mathbb{E}_{x,y\in D} \sum_{t=1}^{|y|} \log \pi_\theta(y_t|x, y_{<t}) \tag{1}$$

Here π_θ is the model's output distribution parameterized by θ and the loss for SFT is computed using the token-wise log probabilities across the entirety of completion y while ignoring pad tokens. The log-probabilities are computed using the softmax activation on the model's logits at each timestep, and gradients are backpropagated through the sequence so that higher probabilities are assigned to correct tokens for the generation.

3.1.2 REINFORCE Leave One-Out (RLOO)

RLOO is a policy gradient estimator where multiple samples are generated using the policy and the weighted average is taken based on the rewards of the generations to reduce the variance of the samples overall based on a leave one out baseline. This is the RLOO objective per prompt with K generations:

$$\frac{1}{K}\sum_{i=1}^{K}\left(r_i - \frac{1}{K-1}\sum_{j\neq i} r_j\right)\log \pi_\theta(y_i \mid x) \tag{2}$$

The loss for RLOO is calculated as follows:

1. Using the current RLOO policy, K samples are generated and the rewards are computed on each sample.
2. The original prompt is concatenated to these generations and a forward pass is done on the RLOO policy.
3. The log probabilities are calculated only on the output tokens and not the input tokens or padding.
4. The mean of the log probabilities is taken per sample generation and a weighted average is done using the reward multipliers and the log probs across the K generations to get the final RLOO loss.

3.2 Chain of Thought (CoT) Revisioning for SFT and RLOO

Both SFT and RLOO are augmented with CoT test time inference scaling. This is the overall algorithm for CoT revisioning with precompute as visualized in fig. 1:

1. **Precomputation** For each prompt, some preliminary calculations are done between the numbers using the +,-,*,/ operators. The results of those operations are stored in an oracle.
2. The oracle is fed into the prompt and passed into the model.
3. **Verification and Extraction** After sampling generation from the model, the overall reward for that generation is computed. If reward is 1.0, the process stops. Otherwise, all the equations in the model generation are extracted and verified / fixed.
4. **Deductions** Deductions are done on wrong responses and fed into the prompt.

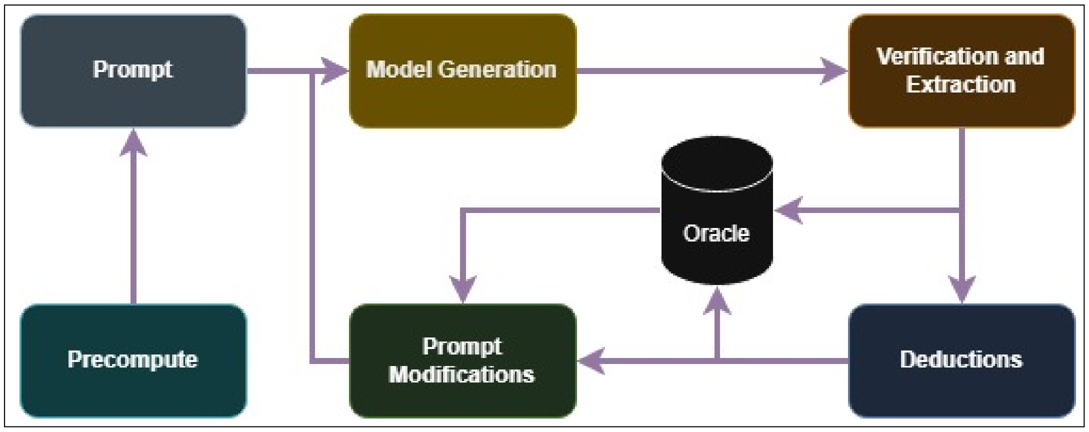

Figure 1: CoT architecture utilizing precomputation, verification, extraction, and deductions to augment the prompt with multiple iterations

5. **Prompt Modification** The updated oracle and the deductions garnered are fed into a new revised prompt.
6. Repeat Steps 3-5.

Each component is described in more detail below.

3.2.1 Oracle and Precomputation

We have the concept of an oracle which stores all preliminary results. For simplicity, only results consisting of 2 numbers and the +, -, *, and / operators are used. Each element in the oracle consists of an operator and 2 numbers along with the result of using the operator between the 2 numbers and an optional note. This storage structure allows for easy deduplication of all the equations. For this work, that allows optimizations on the same equations not being stored twice. For example, operators like + and * which are cumulative operations would not have both num2 + num1 and num2 + num1 stored.

During the precomputation phase, for all combinations of the input numbers, computations are performed using the four operators and stored into the oracle. Division operators are only stored if the result is an integer. If the result from an operation is the target or a factor/multiple of the target, a special note is stored with the operation which is then also injected into the prompt.

Precomputation is useful because it allows the model to reference calculations instead of generating them which is very vulnerable to mistakes as LLMs don't actually perform calculations but just predict tokens. An example of precomputation for a problem with nums [2,3, 1] and target 5 would be "operator": "+", "num1":2, "num2":3, "result": 5, "note": "This is equal to the target!" which would be input to the prompt as "2 + 3 = 5 (This is equal to the target!)"

3.2.2 Extraction and Verification

After model generation on the initial prompt, complicated regex matching is used to garner all the equations present in the model response. Only equations with the format "num1 <operator> num2 = <result>" are gathered to only provide simple calculations to the model and avoid noise in the reference data. These equations are then verified and fixed. So for example, a hallucinated calculation like 128 / 32 = 8 would then be fixed to 128 / 32 = 4 and then stored into the oracle while avoiding duplication which allows the subsequent prompt to not be overblown in length.

3.2.3 Deduction

Deductions are made on the final expression returned by the model for each iteration. There are four kinds of feedback provided:

- **Unused numbers** If any number from the input is not present in the output expression.

- **Numbers used multiple times** If a number from the input is used more times than it shows up in the input.
- **Invalid numbers** If a number from outside the input shows up in the output expression.
- **Wrong result** When the evaluation of the submitted expression does not equal the target.

The model is also instructed to not return the same expression again. One thing to note is that only the answer from the previous iteration and its corresponding deductions are injected into the prompt to keep the prompt length small.

3.2.4 Prompt Modification

All the data points in the oracle are then provided to the prompt as reference calculations along with instructions to reuse calculations along with the deduction based feedback. The main problem is also re-referenced to keep the inference tied to the original problem and prevent further hallucinations. The overall format of a revised prompt is as follows:

```
"A conversation between User and Assistant.  The user asks a question, and
the Assistant solves it...
CALCULATIONS FOR REFERENCE:
....
User:  Using the numbers {nums}, create an equation that equals {target}.
Use above calculations as reference do not redo same computations....
You previously submitted {Previous Wrong Answer} which is incorrect (DON'T
SUBMIT THIS AGAIN)
<MODEL RESPONSE>
```

4 Experimental Setup

Qwen 2.5 0.5B was used as the base model. For all experiments, a validation set of 1000 Countdown problems was used. The two stage reward score from Pan et al. (2025) was used to calculate the average reward score on the validation set. A 1.0 score is given to examples with correct responses, and a 0.1 score is given to incorrect responses that have the correct format. For model generations, the prompt was tokenized to a length of 256, and generations were limited to 1024 tokens. A top k value of 20, a top p value of 0.95, and a temperature value of 0.7 was used for the generations for both validation and RLOO training.

- **SFT** The training set consisted of the warmstart dataset (Gandhi et al., 2025) which consists of 1000 prompts and completions. A learning rate of 1e-6 was used with the AdamW optimizer and training was run for ~20 epochs. The validation score was calculated after each epoch.
- **RLOO** The training was done on the Countdown dataset (consists of nums and a target number) with a training set of 5000 points for 1 epoch. A learning rate 1e-6 was used with the AdamW optimizer. 16 generations were done per prompt. Gradient accumulation was done such that model weights were updated every 32 prompts X 16 generations. The validation score was calculated every 10 gradient steps.
- **CoT Based Inference** A max of 10 iterations of CoT revisioning were run on each prompt.

5 Results

5.1 Training - SFT and RLOO

Figs. 2 and 3 show the training curves for both SFT and RLOO. We can see the training for both SFT and RLOO was quite noisy and that the validation accuracy fell sharply after just a few steps. SFT training loss trended downwards but the validation score started to decrease due to overfitting. However, an interesting aspect of RLOO training was that the training loss started to drop as well. There could be several reasons for this - overfitting, distribution shift in the prompts, general high variance in RLOO training. The final model chosen for both SFT and RLOO was the one corresponding to the highest validation score.

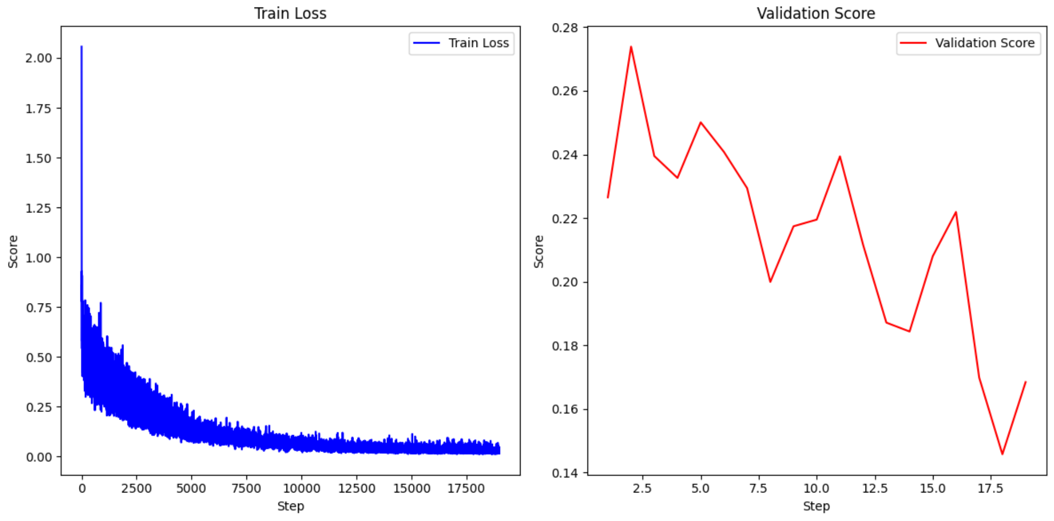

Figure 2: SFT training loss and validation score

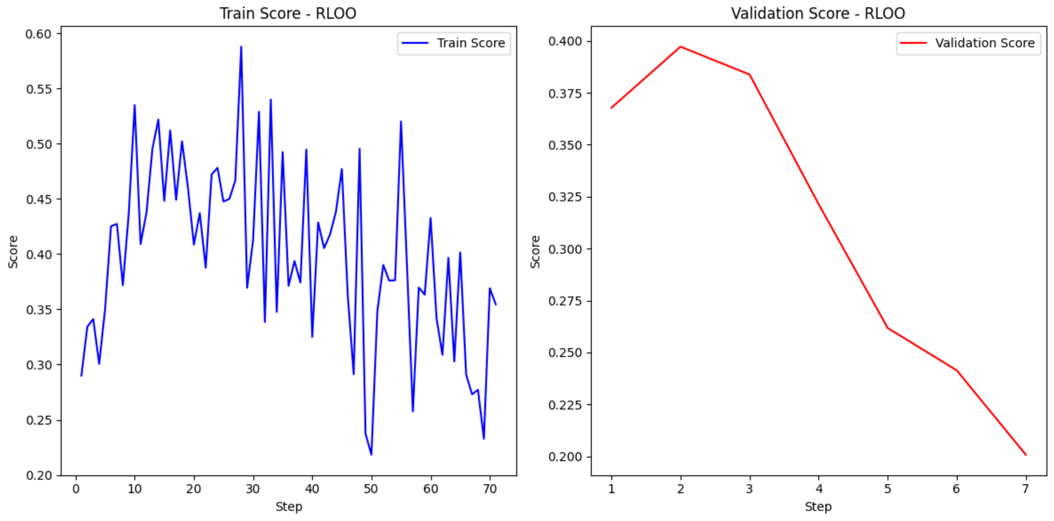

Figure 3: RLOO training and validation rewards

5.2 Quantitative Evaluation

Table 1: Performance Comparison of SFT and RLOO with and without Chain-of-Thought (CoT)

Method	Validation Score	Avg. Success Num Passes
SFT	0.274	1
SFT with CoT	0.584	2.89
RLOO	0.397	1
RLOO with CoT	0.686	2.22

Table 1 shows the comparison of SFT and RLOO methods with and without CoT revisioning. We see that baseline RLOO outperforms SFT by a score of 12.3%. Additionally, we can see that CoT gives a huge boost in performance. **We see that CoT improved performance by 131% for SFT and 72% for RLOO**. However, we see that CoT took 2.89 and 2.22 passes on average for SFT and RLOO compared to a single pass on the baseline models to get to the correct response.

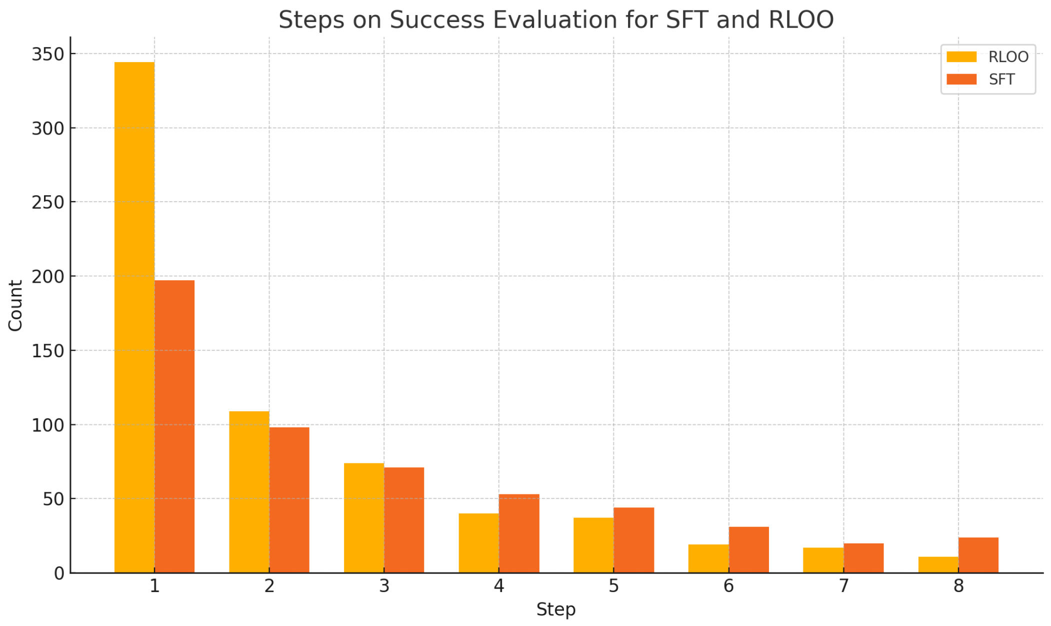

Figure 4: Step Counts for Success Evaluations for SFT and RLOO

From fig. 4 we see the distribution for the number of steps needed for successful expressions. We see that the the accuracy of the base model contributes greatly to the accuracy of the "first pass" of the model. However, we see that post the first pass, the number of problems solved correctly are equal in number across SFT and RLOO. Analyzing further, we see that for SFT 341 out of the 538 (63%) total successes come after the first pass while for RLOO 307 out of the 651 total successes (47%) come after the first pass. This is in line with expectations as the worse the underlying model is, the more benefit that model will get from CoT. **A counter-intuitive result** though was that the first pass accuracy with CoT was equal to or lower than the base SFT and RLOO models meaning that the precomputations supplied to the model did not lead to benefits on the first turn. This also indicates that there is more benefit from using the deduction step in tandem with the precompute step as that allows the model to build upon its previous attempts in a more deterministic way. The feedback provided to the model gives input on the exact areas that are incorrect like numbers not being present or numbers being multiple times which can be used as an effective starting direction.

5.3 Qualitative Analysis

5.3.1 SFT vs. RLOO

The reason RLOO outperforms SFT on the Countdown task is related to the underlying training algorithms. SFT tries to mimic the examples in the training set. That is problematic for the Countdown task because the warmstart dataset used to train SFT has numerous examples that have wrong answers and hallucinated reasoning. SFT does not use any insights on how good or accurate the response actually is as it does not incorporate any reward functions for its loss calculations. Finally, SFT is more prone to overfitting due to the limited number of 1000 training examples which impacts generalization.

A component that gives RLOO an advantage for the Countdown task is reward weighting. More accurate sample generations are given more weight during RLOO training. Not all samples generated from the RLOO policy solve the problem correctly - there is a huge variance at the start of the training but the RLOO algorithm is able to bias towards better responses.

5.3.2 Non-CoT vs. CoT

This is an example of how CoT revisioning helps amend the subsequent answers from the model:

For problem with nums [42, 34, 65, 76] and target 65:

1. First pass solution: (42 - 76) + 34
2. Feedback provided back to prompt - "This expression does not use 65 ... This expression is equal to 0 which is not equal to the target we want which is 65."
3. Second pass solution: (((42 - 76) + 34) - ((65 - (42 - 76))))
4. Feedback provided back to prompt - "This expression uses the numbers 42,76 more than once, You should only use numbers from [42, 34, 65, 76] exactly once ... This expression is equal to -99 which is not equal to the target we want which is 65"
5. Third pass solution: ((42 - 76) + 34) + 65 (Correct)

We can see how the next iteration directly builds upon the previous iteration. The final solution directly addresses the feedback from the previous turn. Looking at the prompt for the third iteration, verbiage indicating the same is present before the final answer is output: "This uses each number only once and is equal to the target!".

Another interesting benefit seen from CoT is that the reasoning capabilities improve greatly per step. With non-CoT in a lot of examples it would just directly output the final answer after just a few computations with no evidence presented for the final expression. However, for the CoT steps after the first one, the logic would start with some reasoning relating to how to fix the deduction feedback provided as part of the prompt. CoT keeps the inference grounded and makes sure that the original problem is not lost due to the feedback that keeps reinforcing the problem.

6 Discussion

The most challenging part of the project was getting the training done for the RLOO model which took ~15-24 hours for each iteration which made it hard to run more experiments and finetune the hyperparameters. Additionally, the difficulty of the algorithm itself led to multiple iterations needed to finish the implementation.

Another major challenge involved extracting and revisioning the prompt for CoT. There is a lot of variance in model responses and for CoT to be functional, there needed to be handling around a variety of edge case scenarios (like dividing by 0 or making sure factor/multiples of 1 were not injected into the prompt).

While the work in the project has demonstrated great results from using CoT - there were some limitations that prevented from conveying more complex ideas and inject more information into subsequent inferences. For example, instead of just giving access to reference calculations, directions can be provided on the actual "logical paths" taken before to take uncovered paths rather than redo some of the same logical paths as before. However, that would require more substantial deduction mechanisms based on past computations. Additionally, the oracle for this work consisted mainly of simple calculations between 2 numbers but there could have been bigger benefits from storing away more complex equations in an optimized fashion.

As far as extending this method to other problems outside Countdown goes, this project benefited from Countdown involving simple computations which could be easily captured and de-duplicated from the model response. Other problems would not have such an easy to extract structure which would require more thought into what to store for subsequent revisions and the most effective way to store those would be. However, this work does serve as a starting reference point for designing more optimized CoT for more complicated problems.

7 Conclusion

This project has shown how CoT test time compute scaling is a lower barrier way for substantial accuracy improvements on logical deduction related tasks. We saw how with CoT average scores on the SFT model increased from 0.274 to 0.584 (a 131% increase) while RLOO with CoT saw an improvement from 0.397 to 0.686 (a 72% increase) over traditional RLOO. This showcases that while there are some diminishing returns, models with varying levels of performance and accuracy can all get significant benefit with optimized CoT. While there are numerous advantages from an accuracy perspective, CoT does come with the added cost of more inference time - we saw an average of 2.55x the amount of time to make a prediction compared to traditional models.

There is a lot of potential for this work to go even further. An interesting avenue to explore could be to create a dataset based on CoT responses by a stronger LLM model which could then be used as a dataset for finetuning models. Additionally, another area to explore could be trying to effectively generate samples in the direction of unexplored samples spaces based on information present in past inferences. All these methods will drive reducing dependency on the LLM's intrinsic capabilities and allow for more control, leading to higher accuracies on such logical reasoning tasks.

8 Team Contributions

- **Akhil Vyas:** All design, implementation, and experimentation.

Changes from Proposal The original proposal aimed at using a self-verification mechanism which scored each intermediate step. This turned out to be difficult to implement since the LLM output would contain reasoning in a variety of formats (free text, equations, etc.). Additionally, the LLM was making inaccurate logical leaps and so relying on it as a judge was not viable. Instead external verification was used for this project. The original proposal also looked to generate multiple samples and then running CoT on all the generated samples. After more research, it was found this doesn't offer too much in the way of novelty but it is also very computationally expensive. Instead, the focus for this work was to make a single inference pass as accurate as possible.

References

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168 [cs.LG] `https://arxiv.org/abs/2110.14168`

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] `https://arxiv.org/abs/2501.12948`

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs. arXiv:2503.01307 [cs.CL] `https://arxiv.org/abs/2503.01307`

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. 2024. Stream of Search (SoS): Learning to Search in Language. arXiv:2404.03683 [cs.LG] `https://arxiv.org/abs/2404.03683`

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. TinyZero. `https://github.com/Jiayi-Pan/TinyZero`. Accessed: 2025-05.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv:2408.03314 [cs.LG] `https://arxiv.org/abs/2408.03314`

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL] `https://arxiv.org/abs/2203.11171`