

Extended Abstract

Motivation Language models are known to have issues with backtracking during generation. There has been a lot of recent work that explores how we can create models that think and self-reflect during inference. The goal of this work is to explore how one can incentivize self correcting behavior in language models during training so that they perform self correction when generating output. In practice, the model proposes some facts, follows logic to its conclusion, realizes there is an issue, and goes back to the root of the error before it continues.

Method We use the game, Countdown as a simplified environment for training and analysis of Qwen2.5-0.5B’s ability to learn backtracking and self-reflective behaviors. Countdown is a simple but hard game where the player is given numbers and a target, and may use +, -, *, and / to reach the target. We can quickly and easily verify that a solution is correct or incorrect. Further, we can identify why the answer is incorrect. We have devised a scheme where the model plays Countdown, submits a solution, and is given feedback in the form of self-correction. This feedback is injected into its chain of thought as if it realized the issue with its solution. Importantly we don’t tell it how to fix the issue, just what the issue is. We leave it up to the model to discover a strategy for making forward progress. We explore two variants of this scheme: SFT only and RLOO RL loop.

In our SFT experiments, we generate new datasets of positive samples that demonstrate self correction and perform SFT on the dataset. We show that we are able to improve model performance on a holdout dataset.

In the RL setup, we sample new positive and negative samples at each iteration with the new policy. This allows us to perform online reinforcement learning.

Implementation We implemented the experiments in pytorch. We created data loaders using Huggingface datasets library and use Huggingface transformers for the model implementation. The training loop and loss function calculation was manually implemented. This involves creating:

- A modified countdown verifier.
- A solution traversal routine that produces linearized search traces.
- An RL loop that creates traces and then performs gradient updates.

Results The experiment results are inconclusive, they don’t show that the proposed methods result in better model performance. We do show that the proposed methods cause the model to exhibit a small amount self correcting behavior at the expense of potentially reduced accuracy. This is an unfortunate downside of training on partially incorrect data. We suspect that we could get around this issue through more training iterations or per-thought reward weighting. This is a potential future direction.

Discussion We have seen examples of improved backtracking behavior after post training the model with verifier feedback, but were unable to outperform RLOO on correct traces from a larger model (Warmstart). It is possible that our model is not able to bootstrap this type of self corrective behavior without the help of a larger model or that this method is not very compute/sample efficient. To gain more insight we could perform the search procedure with a larger model and fine tune on the resulting dataset.

Conclusion We’ve demonstrated that we are able to train the model to generate solution traces that resemble the backtracking behavior created by our sampling procedure. We believe that the proposed methods show promise, but don’t strongly suggest that our approach results in a more generalized and performant model at this model scale.

Improving Test Time Inference via Learned Self Correction, Backtracking, and Verification

Jacob Householder

Department of Computer Science
Stanford University
jhouse@stanford.edu

Abstract

LLMs often struggle to recover from errors introduced by sampling during generation. Previous work has shown that teaching the models self-correcting behavior can lead to tangible performance improvements [2]. We build on this work by constructing self-sampled correction, backtracking, and self-verification traces. We implement a fan-out sampling/search procedure to generate thought traces that demonstrate the desired behavior. We use a verifier to provide feedback that guides the thought trace before continuing completion. It's similar to an interactive tree-of-thought style search where the model explores multiple trajectories. This approach to trace generation is beneficial as it is expensive and complex to design heuristic based traces that have high signal-to-noise ratio. Simulating correction, backtracking, and verification behavior through hybrid generation is a straightforward and extensible solution to improve LLM performance. By training on linearized fan-out-trees we encourage the model to exhibit improved self correction during inference. We show mixed improvements with this method in our experiments, but are optimistic about the general idea.

1 Introduction

The primary goal of my extension is to instill backtracking behavior into Qwen 2.5 0.5B Base through data augmentation and post-training via reinforcement learning. As demonstrated by Gandhi et al. (2024) and others, the ability for humans to backtrack is valuable to reasoning and intelligence. Humans routinely walk through a chain of reasoning, spot an error or contradiction, trace back to the source of the mistake, and continue to the correct solution. This type of behavior has been shown to improve model performance on specific tasks and may potentially lead to better generalization.

Our work explores this line of research by exploring how model verifier interaction can be used to generate synthetic data that can be used to train the model to have this desired self-reflective behavior. We further extend this by creating the dataset in an on policy manner.

2 Related Work

This section surveys key lines of research that inform our approach to backtracking and chain-of-thought (CoT) reasoning.

The concept of CoT reasoning was first introduced by Wei et al. (2023), where they demonstrated that prompting large language models to generate intermediate thought processes can lead to better final solutions. Building on this idea, among others, Zelikman et al. (2022) proposed StAR, a technique for self-bootstrapping reasoning language models by creating a loop that uses previous thought traces to iteratively teach a model to reason. StAR has been extended to work on a corpora of text in [Zelikman

et al. (2024)] and employed to improve the effectiveness of RL training models to backtrack and exhibit other cognitive behaviors [Gandhi et al. (2025)]. A similar line of work focuses on self correction [Welleck et al. (2022), Kumar et al. (2024)]. DeepSeek-R1 [DeepSeek-AI et al. (2025)] showed that RL techniques can yield cutting-edge reasoning capabilities. Our work extends this line of research by exploring methodologies of data augmentation to instill backtracking behavior through problem specific engineering.

Other research has shown that the idea of test time compute has the ability to improve model performance [Snell et al. (2024)]. Thus, efficiency of test time compute has become an important line of research [Arora and Zanette (2025)]. Recently, Qu et al. (2025) proposed the MRT framework, which introduces the notion of cumulative regret over the model output tokens at inference time and proposes a methodology to optimize for "progress" during test time compute. While we don't implement or extend this framework, we do introduce a similar notion by weighting paths differently depending on the number of failed solutions as detected by our verifier.

In the following sections, we will explore the details of how we will investigate and extend these insights and ideas to build more robust backtracking capabilities.

3 Method

3.1 Verifier Modification

We modify the verifier to provide additional detailed feedback when validating and evaluating proposed solutions. We can enumerate and classify the possible issues that a proposed solution has and generate a signal to feedback into training.

Case	Data for template
Missing Solution	-
Invalid Equation	-
Extra and Missing Numbers	Extra numbers, missing numbers
Extra Numbers	Extra numbers
Missing Numbers	Missing numbers
Incorrect solution	Provided equation, evaluated equation
Unbalanced Parentheses	-

Table 1: Verifier provides the following feedback.

The modified verifier provides detailed error information, as opposed to generic information, to allow the model to adjust its approach. This provides a more nuanced signal for the model to backtrack on based on its own internal representation of the problem space.

We note that the Countdown verifier is deterministic; this work can be extended to the non-deterministic verifier setting by using a LLM policy to provide textual feedback and a reward signal. This can even be implemented efficiently by reusing the hidden embedding representation and feeding to a causal language modeling head and a reward head.

3.2 Linearized Search Trace Generation

To generate search traces we have the LLM play Countdown and interact with the modified verifier. Each game play episode is given a token budget of 1024 new tokens. To start, we prompt the model and generate (N) candidate solutions. We use the modified verifier to score and classify each error scenario. From here the completion chain continues until we:

- Run out of token budget
- Hit maximum search depth
- Produce the correct answer

At each step we perform half of the amount of samples, this is an exponential back-off. We can visualize the search in figure 1. From the model's point of view, it decided to output a think tag followed by a self correction instead of the EOS token.

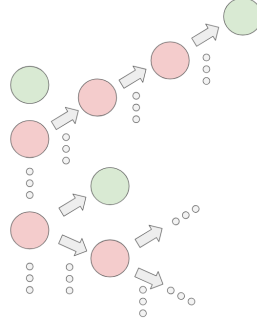


Figure 1: For each model input RL sample completions are generated iteratively.

Now that we have a verifier that tells us how our partial solution was incorrect we can generate linearized search traces that exhibit backtracking behavior. As an illustrative example, consider the "Extra and Missing Numbers" case, where we would update the thought trace as follows

```
{partial_solution}\n<think>
That was incorrect, I can only use the numbers {ground_truth.numbers} once
each number, I am missing {error.missing}, and I accidentally used
{error.extra}. Let's backtrack
```

This new partial solution would then be passed back into the model for completion. This has the effect of looking like the model realized it had made an error and self corrected. This new formatting has the important property where it has multiple tagged sections that differentiate thought and proposed answers; this will allow us to programmatically analyze the generated results.

During trace generation we want to account for incorrect attempts in some manner as traces that have multiple incorrect answers should have lower reward than traces that get the correct answer after one try. To solve this we introduce path depth weighted reward. We divide the verifier reward by the path depth when storing the solution trace in the replay buffer.

$$R_{\text{modified}}(x) = \frac{R_{\text{verifier}}(x)}{\text{path depth}} \quad (1)$$

This incentivizes short correct paths. It also provides positive signal for the model to continue generation after it produced the final `<answer>` tag if it suspects that the proposed answer was incorrect.

It is important to note that the trace generation procedure results in very slow iterations during the on policy RL experiments. The implementation would benefit from performance optimizations due to the large amount of generation.

3.3 Dataset Augmentation

We use the search procedure that we have previously described to generate an SFT style dataset of ground truth completion pairs:

$$y_{(i)}, \dots, y_{(k)} \sim \text{Search}(x; \pi_{\theta}, V). \quad (2)$$

Each game instance potentially contains multiple solutions if multiple were found. Note that they do not contain per reward weighting in this setup. Therefore a solution that is correct on the first attempt is as correct as a solution that is of depth three.

In our experiments we generate a dataset called Backtrack, that only contains games and solutions that were generated using a model that was SFT'd on Warmstart. This intentional decision to sample from the same model could potentially explain why we don't see dramatically increased model performance. Had we sampled thought traces using our procedure from a larger model, we could have potentially collected more high fidelity examples to further train on.

3.4 Reinforcement Learning

We can then extend the SFT setup to an RL loop! Rather than applying StAR Zelikman et al. (2022), we opted to setup RLOO for on policy training as follows:

- Each trace is weighted by its final reward divided by its depth to incentivise shorter traces and provide a more dense reward signal.
- The buffer includes the “negative” traces so that RLOO will update the model appropriately. This gives the effect of per thought reward weighting.

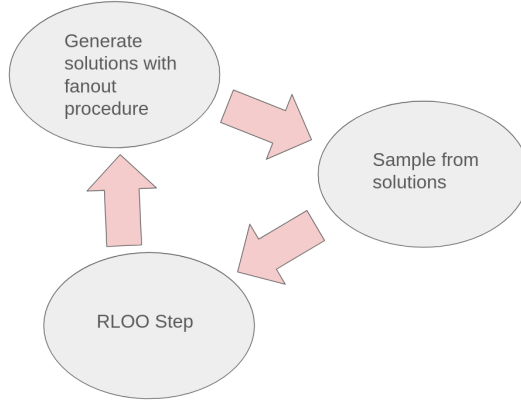


Figure 2: Method Overview.

We use the RLOO Objective

$$\frac{1}{k} \sum_{i=1}^k \left[R(y_{(i)}, x) - \frac{1}{k-1} \sum_{j \neq i} R(y_{(j)}, x) \right] \nabla \log \pi(y_{(i)}|x) \text{ for } y_{(1)}, \dots, y_{(k)} \stackrel{\text{iid}}{\sim} \pi_{\theta}(\cdot|x) \quad (3)$$

where we break independence of samples and propose a modification such that,

$$y_{(i)}, \dots, y_{(k)} \sim \text{Search}(x; \pi_{\theta}, R_{\text{with feedback}}). \quad (4)$$

This search results in lots of trajectories per game, so we sample to reduce the computational overhead. We do this so that the model is able to see a wider variety of scenarios in the same wall-clock time. Using all of the traces results in prohibitively slow iterations. Since the model is trained on policy, we sample at each iteration.

An interesting observation is that partial solutions are valid completions from the model, in the sense that we can assign a reward to them. We opted to store the partial solutions in the replay buffer along with their associated reward from the verifier. This is important so that the model is not just trained on correct completions that contain incorrect reasoning.

This setup makes the RLOO act like a thought-weighted reward without explicitly associating reward weight to each thought. The tokens associated to each thought and answer receive different reward (and advantage) depending on if they are correct. In further detail, the incorrect partial solutions will help cancel out the positive weighting of correct solutions that include the incorrect partial solution. In effect this prevents the model from learning wrong thoughts too much even if the trace gets the right answer in the end.

4 Experimental Setup

The experimental setup builds up from standard training methodologies on the provided datasets. These experiments are representative of state of the art methods for LLM fine tuning within reason. The parameter and configuration was chosen based on other similar works and represents a conservative starting point maximizing model performance. Due to time and resource constraints we did not tune the parameters beyond an arbitrary success threshold.

The experiments are built up from the baseline Qwen2.5-0.4B model. I first SFT on Warmstart, where this model is then used as the starting point for the RL experiments. The SFT parameter configuration was chosen based on [star]. Additionally, I explored continued training, but this did not yield an improvement in model performance.

After training, we sample the model on a holdout dataset using vLLM and evaluate the performance using the countdown verifier. Working in the verifier context provides a nice signal of model performance, which simplifies our analysis.

	SFT	RLOO	RL experiment
Dataset	Warmstart	Countdown 3-to-4	Countdown 3-to-4
Optimizer	AdamW	AdamW	AdamW
Epochs	5	1	1
LR Schedule	Cosine Annealing	Cosine Annealing	Cosine Annealing
LR	0.0001	0.00003	0.00003
Batch Size	2	32	1
Gradient accumulation	-	8	8 (due to wallclock time)
Effective batch size	2	256	8

Table 2: Configuration and hyperparameters.

During training we measure loss and reward to get a picture of model convergence and performance. To measure model generalization we compare average reward on two hold-out datasets prided by the course staff.

5 Results

5.1 Quantitative Evaluation

We will evaluate model performance on the two course-provided holdout datasets. These are game examples that were not seen during training and represent two levels of problem difficulty. The numbers used in the 1st dataset are considerably more in training distribution than in the 2nd dataset. This is evidenced by manual inspection as well as through performance metrics. Out of distribution data, like in the 2nd dataset, tests the model’s ability to manipulate abstract symbols (in our case numbers) as opposed to memorizing facts. Therefore, when we see reduced performance on the 2nd dataset we can infer that the model didn’t generalize well.

Method	Mean Test Reward (1st holdout)	Mean Test Reward (2nd holdout)
SFT	0.4180	0.1949
SFT Backtrack	0.5540	0.1604
RLOO	0.6370	0.3097
RLOO Backtrack	0.7430	0.1945
My RL Backtrack	0.6107	0.2460

Table 3: Performance Comparison

We see increased performance with SFT on Backtrack compared to the baseline SFT on WarmStart. Further, we see that RLOO on Backtrack results in increased performance on the 1st hold out dataset, but does not on the 2nd hold out dataset.

RL increases performance beyond SFT when primed with Backtrack. We see in the sample trace (from the holdout dataset) that the model generates a candidate solution, identifies the issue instead of terminating generation, and proceeds to fix its answer.

This behavior is not overwhelmingly present in solutions, but it leads to a tangible increase in performance. We analyze the frequency of this behavior by counting the think and answer tags from the generated solutions to dataset 2.

We see that when the model is trained on Backtrack it produces more think and answer tags on average! This shows that the model does at least stylistically mirror the linearized traces. Despite this

	<think>	<\think>	<answer>	<\answer>
SFT	1.007	1.007	1.007	1.007
SFT Backtrack	1.416	1.431	1.394	1.394
RLOO	1.0	1.0	1.0	1.0
RLOO Backtrack	1.435	1.451	1.404	1.404
My RL	1.0057	1.0057	1.0057	1.0057

Table 4: Mean number of tags in correct generated solutions on dataset 2.

	<think>	<\think>	<answer>	<\answer>
SFT	1.011	1.011	1.030	1.030
SFT Backtrack	2.253	2.269	2.179	2.179
RLOO	1.003	1.003	1.006	1.006
RLOO Backtrack	2.423	2.453	2.369	2.369
My RL	1.004	1.006	1.006	1.006

Table 5: Mean number of tags in incorrect generated solutions on dataset 2.

	<think>	<\think>	<answer>	<\answer>
SFT	1.471	0.7353	0.5221	0.471
SFT Backtrack	2.686	1.833	1.721	1.668
RLOO	1.126	0.255	0.198	0.123
RLOO Backtrack	2.842	2.103	1.944	1.845
My RL	1.0	0.045	0.027	0.0

Table 6: Mean number of tags in improperly formatted generated solutions on dataset 2.

positive result, we see mixed performance improvement when training on Backtrack vs Warmstart. We see that Backtrack improves performance on the easier problems, but does not lead to increased performance on the harder problems. While disappointing, we still believe that there is value to our proposed approach as we could extend Backtrack to include more difficult game configurations.

The "My RL" experiments are actually quite interesting despite not producing a large number of traces with multiple tags; the model did produce 1 correct multi thought solution and 5 correctly formatted but incorrect multi-thought solutions. It is hard to say if we can draw any conclusions from this, but it does appear that the model retains its single thought behavior from Warmstart more with this training method. This is potentially a positive and could be due to the simulated thought reward weighting induced by including both partial solutions and full solutions in the replay buffer.

5.2 Qualitative Analysis

The goal of this work is to train the model to identify inconsistencies in its proposed solution. We've thought a bit about the difference between training the model to mechanistically jump back to marked points in its chain of thought versus training it to generate introspective phrases followed by more generated output. We decided that it would be a more promising direction to explore introspective phrases as they embed semantic meaning more so than mechanistic jumps. We view this as the difference between teaching an algorithm as opposed to teaching to think.

We see in the generated outputs that the model generates chain of thought traces that contain multiple think/answer tags, and self reflective behavior in similar flavor to the templates that were used during training. We believe that this formatting is beneficial for analysis and interoperability reasons. Unfortunately, the model does not output each individual thought in thought tags, but entire search/exploration attempts inside the tags. This limits the utility of tag based analysis to measure the impact of post-training on formatting. Potentially, more structured post-training could result in more fine grained thought tags that are easier to parse.

In figure 3 we see that RLOO leads to increased reward during training in both experiments, but we don't see the same behavior on our search based RL setup. We associate this instability with the smaller gradient accumulation size for the experiment. This leads us to believe that we could improve our model through additional hyperparameter tuning.

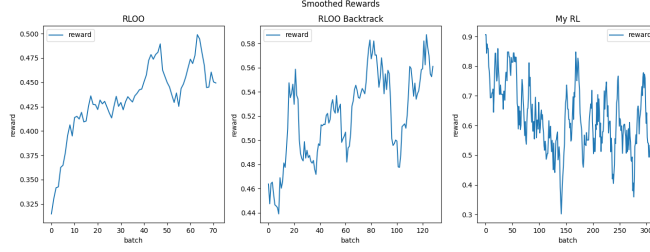


Figure 3: Moving average of training reward (k=8).

6 Discussion

We have demonstrated that we are able to post-train the model to generate text that is similar to the linearized search traces. We have seen both through manual inspection and through analysis of output format tags that this was successful. It is not clear that the backtracking behavior is a net positive, due to continued training on partially incorrect rollouts. It is possible that the model is learning to make incorrect spatial solutions due to the sequence weighting of RLOO. We suspect that thought/answer weighted training would likely result in increased performance as we could weight chunks, denoted by tags, differently depending on their factuality and contribution to the correct solution. Currently the training setup may update the model parameters to generate sequences that don’t entail the correct solution.

Similar to StAR Zelikman et al. (2022), we generate samples from the model, interact with the verifier/critic, and then perform a gradient update. Similar to SoS Gandhi et al. (2024), we use linearized search traces to iteratively improve model performance. Instead of SFT in a loop with dataset generation we use RLOO on batched samples to form the policy gradient. Using SFT in the loop felt like a natural choice as we were exploring this idea, but using RLOO seemed interesting as it uses advantage estimates to update the policy. There are a variety of interesting implementation details that likely affect performance, like how to best batch and compute advantage.

This method has a very natural extension to the non-verifier setting, where a “critic LLM” is used to generate the textual suggestions. The model can be asked to provide new thoughts with additional context and inject them into the CoT as if the model under training had output them.

7 Conclusion

We’ve demonstrated and discussed mixed but narrowly positive results for our experiments. We were unsure if the model has the capacity to learn the advanced reasoning capabilities needed to capture the semantics of the self-correction behavior. I believe that this hypothesis could be fully explored in a more rigorous and thorough manner. I believe that with decreased sampling latency through more complex engineering that the overhead of the online search could become a more tractable approach.

8 Team Contributions

- **Jacob Householder:** Hi!

Changes from Proposal

- Reduction in scope from Countdown and Ultra-Feedback to just Countdown.
- Shift from MRT to RLOO + online data augmentation.

References

Daman Arora and Andrea Zanette. 2025. Training Language Models to Reason Efficiently. arXiv:2502.04463 [cs.LG] <https://arxiv.org/abs/2502.04463>

- 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, Wanbiao 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, Yudian 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>
- Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. 2024. Training Language Models to Self-Correct via Reinforcement Learning. arXiv:2409.12917 [cs.LG] <https://arxiv.org/abs/2409.12917>
- Yuxiao Qu, Matthew Y. R. Yang, Amrith Setlur, Lewis Tunstall, Edward Emanuel Beeching, Ruslan Salakhutdinov, and Aviral Kumar. 2025. Optimizing Test-Time Compute via Meta Reinforcement Fine-Tuning. arXiv:2503.07572 [cs.LG] <https://arxiv.org/abs/2503.07572>
- 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>
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] <https://arxiv.org/abs/2201.11903>
- Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating Sequences by Learning to Self-Correct. arXiv:2211.00053 [cs.CL] <https://arxiv.org/abs/2211.00053>

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. 2024. Quiet-STaR: Language Models Can Teach Themselves to Think Before Speaking. arXiv:2403.09629 [cs.CL] <https://arxiv.org/abs/2403.09629>

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. STaR: Bootstrapping Reasoning With Reasoning. arXiv:2203.14465 [cs.LG] <https://arxiv.org/abs/2203.14465>

A Reproduction steps

To train run

```
uv run main.py train ./cfgs/train/countdown/sft.toml
```

Train will save the model to a unique path in `./artifacts`.

Then update `model_name` in

```
./cfgs/evaluate/countdown/sft.toml
```

to

```
./artifacts/<model_id>/models/<epoch_idx>
```

and run evaluation

```
uv run main.py evaluate ./cfgs/evaluate/countdown/sft.toml
```

The `"logs.jsonl"` file in the unique artifacts dir has the events for the run. To view the logs use the following commands:

```
uv run plot.py train ./artifacts/<model_id>/logs.jsonl
uv run plot.py reward ./artifacts/<model_id>/logs.jsonl
```

B Sample Trace

`<think>`

Let me analyze this step by step:

1. First, let's look at numbers close to 82:

- 89 is closest, being 7 more than target
- 61 is next closest, being 11 less than target
- 48 is 14 less than target
- 20 is much smaller

2. Since 89 is close to our target 82, let's try to get from 89 to 82:

- We need to subtract 7 somehow
- We could try to create 7 using the other numbers

3. Looking at the remaining numbers (61, 48, 20):

- $61 - 48 = 13$
- $13 - 20 = -7$
- If we subtract this from 89, we'll get 82!

4. Let's verify:

- First: $61 - 48 = 13$
- Then: $13 - 20 = -7$
- Finally: $89 - 7 = 82$

This gives us our target of 82.

`</think>`

`<answer> (61 - 48) - 20 </answer>`

`<think>`

That was incorrect, I need to use 89. Let me try again.

1. Let's look at it differently:

- $89 - 20 = 69$
- $69 + 48 = 117$
- $117 - 89 = 38$

2. Another approach:

- $89 - 61 = 28$
- $28 + 48 = 76$
- $76 + 20 = 96$

3. Let's try:

- $89 - 48 = 41$
- $41 + 61 = 102$
- $102 - 20 = 82$

This works perfectly! Let's use each number exactly once and reach 82.

</think>

<answer> $(89 - 48 + 61) - 20$ </answer>