TOWARDS EXECUTION-GROUNDED AUTOMATED AI RESEARCH

Chenglei Si*, **Zitong Yang***, **Yejin Choi, Emmanuel Candès, Diyi Yang, Tatsunori Hashimoto**
Stanford University
clsi@stanford.edu, zitong@berkeley.edu
*Equal contribution

ABSTRACT

Automated AI research holds great potential to accelerate scientific discovery. However, current LLMs often generate plausible-looking but ineffective ideas. Execution grounding may help, but it is unclear whether automated execution is feasible and whether LLMs can learn from the execution feedback. To investigate these, we first build an automated executor to implement ideas and launch large-scale parallel GPU experiments to verify their effectiveness. We then convert two realistic research problems – LLM pre-training and post-training – into execution environments and demonstrate that our automated executor can implement a large fraction of the ideas sampled from frontier LLMs. We analyze two methods to learn from the execution feedback: evolutionary search and reinforcement learning. Execution-guided evolutionary search is sample-efficient: it finds a method that significantly outperforms the GRPO baseline on post-training, and finds a pre-training recipe that outperforms the nanoGPT baseline on pre-training, all within just ten search epochs. Frontier LLMs often generate meaningful algorithmic ideas during search, but they tend to saturate early and only occasionally exhibit scaling trends. Reinforcement learning from execution reward, on the other hand, suffers from mode collapse. It successfully improves the average reward of the ideator model but not the upper-bound, due to models converging on simple ideas. We also thoroughly analyze the executed ideas and training dynamics.

1 INTRODUCTION

We envision automated AI research: LLMs generate research ideas to tackle important research problems, implement the ideas as code, run experiments to verify the effectiveness, and continuously learn from the execution results. If successful, these automated AI researchers could automatically develop and identify effective research ideas in a massive search space, thereby scalably converting compute into scientific discovery. Despite the promise, automated AI research is bottlenecked by the ability of LLMs to generate effective ideas. Si et al. (2025b) and Si et al. (2025a) evaluated the quality of LLM-generated research ideas through large-scale expert review and found that LLM ideas often look convincing but are ineffective after being executed by human researchers.

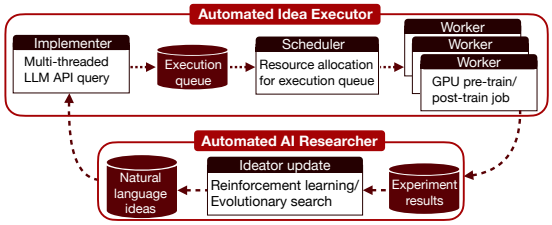


Figure 1: We build an automated idea executor involving Implementer, Scheduler, and Worker. We then use this automated executor as a reward function to teach LLMs to generate more effective ideas through evolutionary search and RL. We only update the ideator in the learning process.

This highlights the need to ground idea generation in execution. However, obtaining execution results of ideas in an automated and scalable manner is challenging, especially since we are targeting open-ended AI research where any ideas expressible in natural language are within our action space. To tackle this, we design and build a high-throughput automated idea executor that can implement

hundreds of model-generated ideas and execute them in parallel to obtain the experiment results as execution feedback.

To study the extent to which we can automate realistic LLM research, we chose two GPU-intensive research problems (LLM pre-training and post-training) that are critical for improving the capabilities of LLMs as the research environments for our automated AI researchers. For the first time, we demonstrate that our automated executor can implement a large fraction of LLM-generated ideas on these challenging open-ended research problems, with over 90% execution rates on the pre-training environment with Claude-4.5-Sonnet and Claude-4.5-Opus.

To analyze whether grounding on execution-feedback can improve LLM idea generation, we define objective performance metrics for both environments and analyze the strengths and weaknesses of two popular learning algorithms: evolutionary search and reinforcement learning.

We use our automated executor to guide evolutionary search. Within ten search epochs, this execution-guided search finds a post-training recipe that outperforms the GRPO baseline (69.4% vs 48.0%) on the task of post-training a 1.5B model for math reasoning, and a pre-training recipe that outperforms the nanoGPT baseline (19.7 minutes vs 35.9 minutes) on the task of minimizing the training wall-clock time to reach the target validation loss (Table 1). Our analysis shows that models are often generating algorithmic ideas apart from tuning hyper-parameters, and evolutionary search significantly outperforms best-of-N under the same sampling budget. However, when analyzing the scaling trend, only Claude-4.5-Opus shows a clear scaling curve, while both Claude-4.5-Sonnet and GPT-5 tend to saturate early.

We then use the automated executor as the reward function in an RL loop to finetune Qwen3-30B. We show that RL with execution reward can successfully improve the average reward of the ideator model, similar to typical RL from verifiable rewards. However, RL does not improve the max reward, which is the more important metric for scientific discovery. In fact, we reveal that RL causes the ideator model to converge on a few easy-to-implement ideas, resulting in a collapse in thinking length and idea diversity.

Table 1: Performance of our execution-guided search vs baselines and best human experts.

	Post-Training$_{\uparrow}$	Pre-Training$_{\downarrow}$
Baseline	48.0%	35.9 min
Execution-Guided Search	69.4%	19.7 min
Best Human Expert	68.8%	2.1 min

2 Automated Idea Executor

To measure the effectiveness of model-generated ideas, we build an automated executor that takes natural language research ideas as input, generates code implementations, runs the experiments on the backend, and returns the idea's benchmark performance as the final output.

2.1 Research Environments for Ideation

Our automated idea executor is grounded in specific research environments, where each environment consists of a research problem, a baseline codebase, a benchmark to measure performance on, fixed training and evaluation data, and evaluation metrics. When constructing the research environments, we aim to select research problems that are open-ended, so that there is ample room for new algorithmic innovations, and at the same time have well-established baselines and benchmarking metrics so that measuring effectiveness is straightforward. In this work, we construct both a pre-training environment and a post-training environment for the automated AI researchers to work on.

Pre-Training Task: Improving nanoGPT In the nanoGPT environment, we provide a baseline codebase adapted from the nanoGPT speedrun Jordan et al. (2024) and ask the ideator model to brainstorm possible improvements. The original speedrun task is to minimize the time to pre-train a 124M GPT-2 model Radford et al. (2019) on the FineWeb corpus Penedo et al. (2024) to reach a validation loss of 3.28 on the validation set. We did several modifications to the original speedrun setting. First, we introduce a proxy reward equal to the reciprocal of the validation loss ($\frac{1}{loss}$) when performing the search and RL experiments in later sections of the paper. This way, we can fix the training wall-clock time to be 25 minutes and ask the model to directly optimize the proxy reward under this fixed budget, so that we can avoid different runs having vastly different runtimes. We report

the validation loss or the proxy reward metric in most plots, and only measure and report the training time metric for the top solution in order to directly compare it with the human experts' solutions on the original nanoGPT speedrun leaderboard. Second, to avoid any possible reward hacking, we freeze all evaluation hyper-parameters and implement an inference function that predicts one future token at a time to prevent models from changing the attention mechanism in a way that leaks future tokens (which happened multiple times during our initial development). We use this inference function during the final validation after each training run.

Post-Training Task: Improving GRPO In the GRPO environment, the baseline is an implementation of the GRPO algorithm Shao et al. (2024) that finetunes a Qwen2.5-Math-1.5B checkpoint Yang et al. (2024) on the MATH dataset Hendrycks et al. (2021). The ideator model needs to brainstorm post-training algorithms more effective than the baseline. We specify a fixed training wall-clock time budget and use the max accuracy on the MATH validation set during training as the metric. To prevent reward hacking, we keep all validation-related code in a separate file and do not allow the automatic executor to access or modify it.

2.2 System design

The automated idea executor can be viewed as a high-level API whose input is a batch of natural language ideas, and the output is the benchmark performance of each idea. There are three core building blocks of this API (Figure 1): *Implementer* – the server that generates the code diff for the idea and applies those changes; *Scheduler* – a middle layer that receives the list of codebases and allocates resources to run experiments; *Worker* – the cluster with GPU available that runs the experiments and uploads the experiment results.

Implementer The implementer is hosted on a CPU machine with high IO capacity. First, the user submits a batch of natural language ideas. Then, for each idea, the implementer makes parallelized API calls to the code execution LLM to obtain a `diff` file that can be patched into the corresponding baseline codebase. To optimize for efficiency, we prompt the code execution LLM with both the idea and the baseline codebase to sample 10 code diff files in parallel. For each sample, if the generated diff file cannot be patched into the original codebase, we provide the patch log and ask the model to revise the original generation. We repeat this sequential self-revision for a maximum of 2 times. In the end, we return the first code diff file that can be successfully patched into the baseline codebase. The patched codebase is then submitted to a cloud bucket as a `.zip` file.

Scheduler Under a set clock frequency, the scheduler downloads the new codebases from the cloud. If the codebase has not been executed, the scheduler examines the resource requirement of the given research environment and prepares a job configuration to be submitted.

Worker Once the scheduler finds available resources, it connects the prepared job configuration with the GPU resource and initializes the worker to run the experiment. If the execution of the experiment is successful, the worker will upload the experiment logs including all performance metrics to another cloud bucket (`wandb`) along with the complete metadata: idea content, code change, execution log, etc. If the execution fails (e.g., due to bugs in code implementation), the worker halts. The user (i.e., the ideator model) can then download the execution results and see the performance of the batch of ideas they submitted with full training logs.

3 Benchmarking LLM Ideators and Executors

The prerequisite for an execution-grounded feedback loop is that current LLMs can serve as both ideators and executors, so that we can get meaningful reward signals for the models to learn from. To examine this prerequisite, we benchmark various frontier LLMs as both the ideator and the executor.

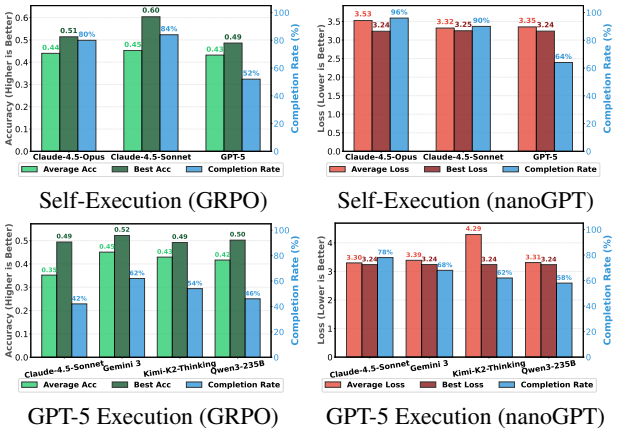
3.1 End-to-End Ideation and Execution

In the first setting, we sample ideas from an LLM, and use the same LLM as the code execution model to execute its own ideas. We sample and execute 50 ideas from Claude-4.5-Opus, Claude-4.5-Sonnet, and GPT-5, and measure several metrics: (1) completion rate: the percentage of ideas that are successfully executed with a valid (non-zero) experiment result after execution; (2) average performance: the average validation accuracy or loss for all the successfully executed ideas among the

50 samples; (3) best performance: the highest validation accuracy or lowest validation loss among all executed ideas. We present results in the top row of Figure 2. Notably, a large fraction of the sampled ideas can indeed be executed successfully, with Claude-4.5-Opus and Claude-4.5-Sonnet having a significantly higher execution rate than GPT-5. Moreover, the best-of-N performance ($N = 50$) of these models can already beat the original baseline solutions. For example, on the GRPO environment, Claude-4.5-Sonnet gets a max accuracy of 60.4% as compared to the baseline of 48.0%; on nanoGPT, Claude-4.5-Opus gets a lowest loss of 3.237 as compared to the baseline of 3.255.

3.2 Comparing Ideators with the Same Executor

In the second setting, we fix the executor model to be GPT-5 and use different ideator models to sample ideas. As shown in the bottom row of Figure 2, even when the ideator and executor are different models, the execution rate is still decent (ranging from 42% to 78%), although we do notice that the same ideas from Claude-4.5-Sonnet get a lower execution rate when executed by GPT-5 instead of itself (84% vs 42% on GRPO and 90% vs 78% on nanoGPT). Moreover, frontier open-weight models like Kimi-K2-Thinking Kimi Team (2025) and Qwen3-235B-A22B Yang et al. (2025a) can also get non-trivial completion rates and achieve best-of-N performance that outperforms the baseline solutions in this setting. For example, Qwen3-235B achieves a max accuracy of 50.2% on GRPO and min loss of 3.238 on nanoGPT with $N = 50$, both better than the baselines.



Self-Execution (GRPO) Self-Execution (nanoGPT)

GPT-5 Execution (GRPO) GPT-5 Execution (nanoGPT)

Figure 2: Model performance comparison with self-execution (top row) vs GPT-5 execution (bottom row) on the GRPO and nanoGPT environments. The baseline accuracy for GRPO is 0.480, and the baseline loss for nanoGPT is 3.255.

These benchmarking results demonstrate the feasibility of the automated ideation and execution loop. Next, we build search scaffolds and RL training loops to examine whether models can learn from the execution feedback.

4 Execution-Guided Evolutionary Search

We develop an evolutionary search scaffold on top of frontier LLMs to optimize for effective ideas based on execution feedback. We introduce our search method that blends exploration and exploitation, its effectiveness on our two research environments, and various analyses of the generated ideas throughout the evolutionary search process.

4.1 Search Scaffold

Our search method is inspired by prior evolutionary search approaches for code optimization, such as AlphaEvolve Novikov et al. (2025). At the first search epoch, we sample a full batch of new ideas. In all subsequent epochs, we split the idea generation into exploitation and exploration subsets. For exploitation, we choose ideas from previous epochs that outperform the baseline and append them to the idea generation prompt to ask the ideator model to generate new variants that combine their strengths. For exploration, we randomly sample ideas from previous epochs to append to the idea generation prompt until reaching the max context length and instruct the ideator model to generate completely new ideas different from them. We start with 50% exploitation and 50% exploration at epoch 1 and gradually anneal the exploration rate and increase the exploitation ratio in later epochs. We use a batch size of 50 for the GRPO environment and a batch size of 80 for the nanoGPT environment.

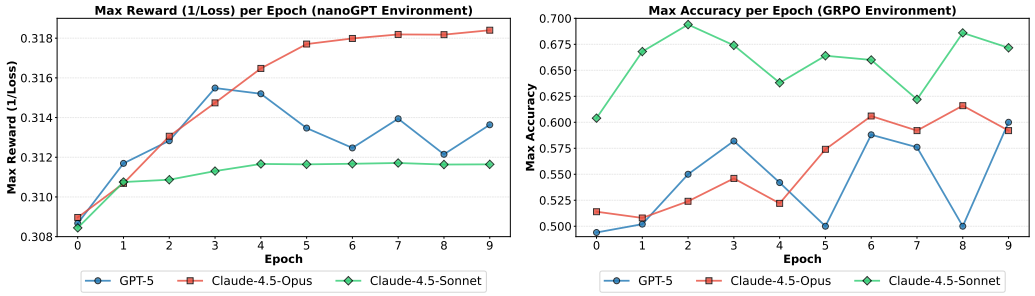


Figure 3: Best performance at each epoch when performing execution-guided search with different models. For the nanoGPT environment (left), we use the reciprocal of the validation loss as the metric; for the GRPO environment (right), we use validation accuracy as the metric. Claude-4.5-Opus exhibits a scaling trend on both environments and achieves the best performance on nanoGPT. Claude-4.5-Sonnet achieves the best performance on GRPO due to effective hyper-parameter tuning, but saturates early.

4.2 Experiment Results

For each environment, we perform execution-guided search with three different models: Claude-4.5-Opus, Claude-4.5-Sonnet, and GPT-5. For each experiment, we use the same model as both the ideator and executor (i.e., self-execution). We plot the progression of the best performance at each search epoch in Figure 3. We summarize several notable trends below.

First, we observe a scaling trend with Claude-4.5-Opus, where searching for more epochs leads to a higher upper bound. In contrast, Claude-4.5-Sonnet and GPT-5 tend to saturate early. Second, all models can find ideas that significantly outperform the baselines. On GRPO, Claude-4.5-Sonnet finds that using vanilla policy gradient with the group-average baseline without importance reweighting or clipping outperforms the standard GRPO objective in this particular experiment setup and exploits this finding in all subsequent search epochs, resulting in the best solution of 69.4% at epoch 2 with precise hyper-parameter tuning. On nanoGPT, Claude-4.5-Opus achieves the min validation loss of 3.1407 at epoch 9 by combining various architectural modifications, hyper-parameter tuning, and applying exponential moving average of intermediate checkpoints during validation (see Appendix A.4 for the full idea). We run this top solution on 8 H100s to follow the same setup as the nanoGPT speedrun, and it reaches the 3.28 target validation loss in 19.7 minutes, a significant speedup as compared to the baseline codebase, which takes 35.9 minutes of training time to reach the same target validation loss.

To better contextualize these solutions optimized by the model, we also compare the top performance of our execution-guided search to human experts. For the GRPO environment, we compare with the leaderboard of a graduate-level LLM class, which hosted the same environment as an assignment for all students to optimize the validation accuracy under the same training time budget. The best student solution achieved an accuracy of 68.8%, lower than Claude-4.5-Sonnet's top solution using our execution-guided search. For the nanoGPT environment, we directly compare with the nanoGPT speedrun leaderboard, [1] where the top human solution as of December 2025 can reach the target validation loss under 2.1 minutes, indicating significant headroom for further model capability and search method improvement on this environment.

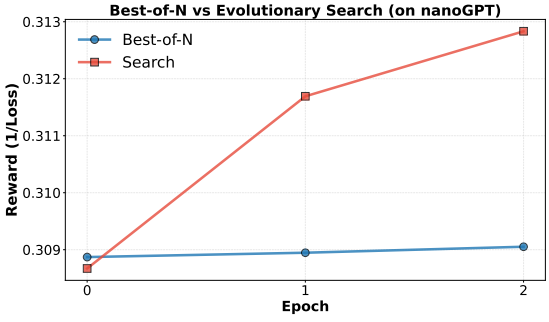


Figure 4: Comparison between best-of-N and our execution-guided search under the same sampling budget.

[1] https://github.com/KellerJordan/modded-nanogpt

Table 2: Breakdown of hyper-parameter tuning vs algorithmic ideas throughout the entire execution-guided search. We report the percentage of each type among all generated ideas of each model ($N = 500$ ideas on GRPO and $N = 800$ ideas on nanoGPT). Bold numbers every row indicate the best performance by each model. All models generate a substantial amount of algorithmic ideas apart from hyper-parameter changes, while Claude-4.5-Sonnet generates significantly more hyper-parameter ideas than other models.

	Hyper-parameter			Algorithmic		
Model name	**Percentage**	**Average**	**Best**	**Percentage**	**Average**	**Best**
GRPO environment (accuracy$_\uparrow$)						
GPT-5	5.0%	45.0%	50.2%	95.0%	44.5%	**60.0%**
Claude-4.5-Sonnet	41.1%	48.4%	**69.4%**	58.9%	45.0%	67.4%
Claude-4.5-Opus	3.7%	44.4%	50.4%	96.3%	46.5%	**61.6%**
nanoGPT environment (loss$_\downarrow$)						
GPT-5	15.4%	3.254	3.195	84.6%	3.894	**3.170**
Claude-4.5-Sonnet	31.3%	3.251	**3.208**	68.7%	3.679	**3.208**
Claude-4.5-Opus	8.7%	3.329	3.147	91.3%	3.419	**3.141**

4.3 Comparison with Best-of-N

To demonstrate the effectiveness of our search scaffold, we compare our execution-guided search with the best-of-N baseline with the same sampling budget on the nanoGPT environment. Since the batch size for our search is 80, we compare the first 3 epochs of the execution-guided search using the GPT-5 backbone with the best-of-N results of GPT-5 with $N \in \{80, 160, 240\}$. As shown in Figure 4, search and best-of-N start from similar performance at epoch 0 (they are not exactly the same due to variances from sampling), but evolutionary search significantly outperforms best-of-N from epoch 1 onward, demonstrating that the model is effectively leveraging trajectories from previous epochs to generate more effective ideas in future epochs.

4.4 Analysis of Generated Ideas

To quantitatively understand the types of ideas that models generate during the execution-guided search, we perform a stratified analysis by classifying all generated ideas into either hyper-parameter tuning (including any ideas that can be implemented via changing existing configs) or algorithmic (including all ideas that involve implementing new changes not originally supported by the baseline codebase) by using an LLM-judge. Based on Table 2, all three models generate a substantial amount of algorithmic ideas apart from hyper-parameter tuning. Interestingly, different models exhibit different patterns, where Claude-4.5-Sonnet generates significantly more hyper-parameter ideas than both Claude-4.5-Opus and GPT-5. Moreover, the most effective ideas come from algorithmic ideas in most cases, except when using Claude-4.5-Sonnet. To complement the quantitative analysis, we provide several executed ideas in Table 3, as well as Appendix A.4 and Appendix A.5.

5 Reinforcement Learning from Execution Reward

Different from evolutionary search, reinforcement learning is an alternative learning algorithm that shapes model behaviors through gradient updates. For the first time, we explore whether we can leverage the automated executor as a reward function to directly finetune LLMs to generate more effective ideas via RL. We detail our implementation, experiment setup, and analysis of the training dynamics.

5.1 Reward Design and Experiment Setup

We use Qwen3-30B-A3B Yang et al. (2025a) as the base model and finetune it using the standard GRPO algorithm Shao et al. (2024), motivated by its consistent empirical success on other verified domains. Our prompt batch size is one since we only have one prompt for each environment. In the

Figure 5: Training curves of RL from execution reward. We plot the average reward per epoch in the upper row, and the max reward per epoch in the lower row. For GRPO, reward is accuracy; for nanoGPT, reward is reciprocal loss. The average reward increases, but not the max reward.

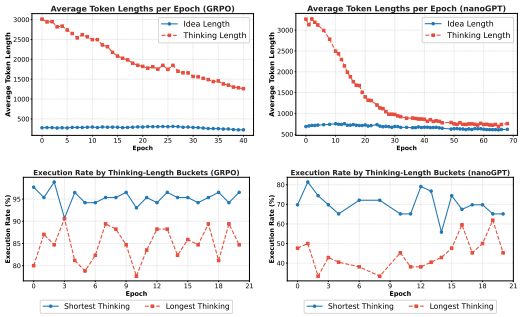


Figure 6: Upper Row: Average length of the thinking trace and the idea per training epoch. Lower Row: Execution rate of ideas with the longest versus shortest thinking traces. Ideas with longer thinking have lower execution rates; correspondingly, thinking lengths decrease in RL.

prompt, we provide the baseline codebase and ask the model to generate new ideas to improve the baseline.

We use large group sizes to stabilize training. For the post-training environment, we use a group size of 256; for the pre-training environment, we use a group size of 128. Since each idea on the GRPO environment runs on one single GPU and each idea on the nanoGPT environment runs on 8 GPUs, these group sizes correspond to parallel execution on 256 GPUs (for GRPO) or 1024 GPUs (for nanoGPT) to obtain the execution reward on each batch of rollout ideas. Each rollout is a thinking trace followed by the natural language idea. We set a max output length of 8192 tokens for rollout sampling and only feed the extracted ideas to the automated executor without the preceding thinking trace.

For the post-training environment, we use the validation set accuracy of each rollout idea after execution as the reward. For ideas without a valid accuracy (i.e., when the execution failed due to code generation errors), we assign a reward of 0. For the pre-training environment, we use the reciprocal of the validation loss as the reward ($\frac{1}{loss}$) and assign a reward of 0 to ideas with failed execution. Our experiments are based on Tinker Thinking Machines Lab (2025).

5.2 Experiment Results

Positive Training Curves for Average Reward We plot the average reward of all rollouts of each training epoch in the upper row of Figure 5. For the first time, we demonstrate that the average performance of the generated ideas can increase after sufficient training epochs for open-ended research environments. For instance, the average accuracy on the GRPO environment increases from 0.253 at the beginning to 0.343 after 40 training epochs (top left plot of Figure 5); and the average reward on the nanoGPT environment increases from 0.194 at the beginning to 0.246 after 68 epochs (top right plot of Figure 5), corresponding to a decrease in the average validation loss from 5.150 to 4.066.

The Case of Max Reward Despite successfully reproducing the positive training curves observed in other domains, we argue that there is a distinction between idea generation and other verifiable domains. For advancing scientific discovery, we often care about the upper-bound of idea generation, rather than the average quality. In our particular case, we care more about having one breakthrough idea that dominates the baselines, rather than having many safe ideas with a high average. Thus, we plot the max reward of all rollouts at each training epoch in the lower row of Figure 5. The trend here is very different – **the max reward is fluctuating throughout RL training without a clear upward trend**. This reveals the crucial limitation of the standard GRPO algorithm for improving idea generation. In the next subsection, we analyze why RL from execution reward improves the average reward but not the max.

5.3 Analysis of Training Dynamics

Thinking Length We first plot how the lengths of the thinking traces evolve over RL training in the upper row of Figure 6. In both environments, the thinking traces rapidly decrease in length while the idea lengths stay roughly constant. This is the opposite of the thinking emergence trend from prior RLVR work, such as DeepSeek-R1 DeepSeek-AI et al. (2025). To further understand why the thinking length decreases, we investigate the correlation between the idea execution rate and the thinking trace length. In the bottom row of Figure 6, for the first 20 epochs of the RL training, we sort all ideas in each epoch by their thinking trace lengths and plot the average execution rate of the top-30% longest thinking ideas (red line) and the bottom-30% shortest thinking ideas (blue line). We see a clear trend where ideas with longer thinking consistently have a lower execution rate. We thus hypothesize that longer thinking correlates with more complex ideas with lower execution rates, leading the model to prefer shorter thinking instead in order to maximize the reward.

Diversity Collapse Upon manual investigation of all the rollouts being sampled throughout the RL training, we also observed a diversity collapse. Specifically, the model learned to converge on a few simple ideas that can consistently get a positive reward. For example, in the nanoGPT environment, the model learned to converge towards two common ideas: (1) replacing RMSNorm with LayerNorm; and (2) performing exponential moving average (EMA) over intermediate model checkpoints. As shown in Figure 7, out of a batch of 128 sampled ideas per epoch, 51 ideas sampled from Qwen3-30B at epoch 0 are one of the two common ideas above. Towards the end of the RL training, 119 out of 128 sampled ideas at epoch 68 are one of the two common ideas, indicating a severe diversity collapse.

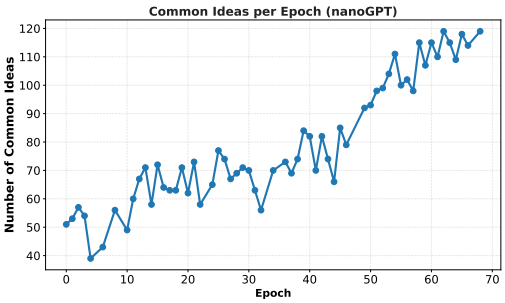


Figure 7: We plot how many ideas in each epoch are about either replacing RMSNorm with LayerNorm or doing EMA. The model converged on these two ideas after RL training.

The above analyses reveal that RL causes the models to converge on a few simple-to-implement ideas, accompanied by shrinking thinking lengths. These lead to an increase in the average reward, but do not push the upper-bound due to the lack of exploration. This phenomenon is analogous to mode-collapse observations on other verifiable domains, where the pass@k performance stagnates or even decreases after RL Yue et al. (2025); Wu et al. (2025). Avoiding such convergence and collapse is an open problem and likely requires new algorithmic interventions beyond standard GRPO, which is beyond the scope of this work. However, we do share several preliminary attempts, including: sampling and appending previous epochs' trajectories into the current epoch's prompt for rollout sampling, adding a weighted length reward in the total reward, and adding a weighted similarity penalty in the total reward. We did not observe clear gains in the initial epochs and thus early-stopped them, but we document all these results in Appendix A.3 to inform future work.

6 Conclusion

In this work, we built a large-scale parallel executor for automatically executing model-generated ideas to verify their effectiveness on open-ended LLM research problems, including both pre-training and post-training. Using this executor as a reward function, we analyzed the effectiveness of execution-guided evolutionary search, where frontier LLMs equipped with our search scaffold can significantly outperform the baseline solutions. We also analyzed the limitations of reinforcement learning with execution rewards, where models converge on simple ideas to improve the average reward but lose diversity and do not improve the upper-bound. Our empirical results demonstrate the feasibility and potential of the automated execution feedback loop and also point out the remaining limitations for future improvement.

REFERENCES

Akari Asai, Jacqueline He, Rulin Shao, Weijia Shi, Amanpreet Singh, Joseph Chee Chang, Kyle Lo, Luca Soldaini, Sergey Feldman, Mike D'Arcy, David Wadden, Matt Latzke, Minyang Tian, Pan Ji, Shengyan Liu, Hao Tong, Bohao Wu, Yanyu Xiong, Luke S. Zettlemoyer, Graham Neubig, Dan Weld, Doug Downey, Wen tau Yih, Pang Wei Koh, and Hanna Hajishirzi. OpenScholar: Synthesizing Scientific Literature with Retrieval-augmented LMs. *ArXiv*, abs/2411.14199, 2024.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mkadry. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *ICLR*, 2025.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic Discovery of Optimization Algorithms. In *NeurIPS*, 2023.

Junyan Cheng, Peter Clark, and Kyle Richardson. Language Modeling by Language Models. In *NeurIPS*, 2025.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bing-Li Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dong-Li 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, Jiong 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, M. 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, Ruiqi Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shao-Kang Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wen-Xia Yu, Wentao Zhang, Wangding 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, Xiangyu Jin, Xi-Cheng 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, Yi Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yu-Jing Zou, Yujia He, Yunfan Xiong, Yu-Wei Luo, Yu mei You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yao Li, Yi Zheng, Yuchen Zhu, Yunxiang Ma, Ying Tang, Yukun Zha, Yuting Yan, Zehui Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhen guo Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zi-An Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *Nature*, 2025.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriele Synnaeve. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. In *ICML*, 2025.

Shashwat Goel, Rishi Hazra, Dulhan Hansaja Jayalath, Timon Willi, Parag Jain, William F. Shen, Ilias Leontiadis, Francesco Barbieri, Yoram Bachrach, Jonas Geiping, and Chenxi Whitehouse. Training AI Co-Scientists Using Rubric Rewards. *ArXiv*, abs/2512.23707, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Xiaodong Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving With the MATH Dataset. In *NeurIPS*, 2021.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *ICLR*, 2025.

Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. ResearchCodeBench: Benchmarking LLMs on Implementing Novel Machine Learning Research Code. In *NeurIPS*, 2025.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-Driven Exploration in the Space of Code. *ArXiv*, abs/2502.13138, 2025.

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977. modded-nanogpt: Speedrunning the nanogpt baseline, 2024. URL `https://github.com/KellerJordan/modded-nanogpt`.

Kimi Team. Kimi K2: Open Agentic Intelligence. *ArXiv*, abs/2507.20534, 2025.

Jakub L'ala, Odhran O'Donoghue, Aleksandar Shtedritski, Sam Cox, Samuel G. Rodriques, and Andrew D. White. PaperQA: Retrieval-Augmented Generative Agent for Scientific Research. *ArXiv*, abs/2312.07559, 2023.

Boaz Lavon, Shahar Katz, and Lior Wolf. Execution Guided Line-by-Line Code Generation. In *NeurIPS*, 2025.

Weixin Liang, Yuhui Zhang, Hancheng Cao, Binglu Wang, Daisy Ding, Xinyu Yang, Kailas Vodrahalli, Siyu He, Daniel Scott Smith, Yian Yin, Daniel A. McFarland, and James Zou. Can large language models provide useful feedback on research papers? a large-scale empirical analysis. *NEJM AI*, 2024.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search. In *ICLR*, 2018.

Yixiu Liu, Yang Nan, Weixian Xu, Xiangkun Hu, Lyumanshan Ye, Zhen Qin, and Pengfei Liu. AlphaGo Moment for Model Architecture Discovery. *ArXiv*, abs/2507.18074, 2025.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Nicolaus Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. Discovering Preference Optimization Algorithms with and for Large Language Models. In *NeurIPS*, 2024a.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Nicolaus Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. *ArXiv*, abs/2408.06292, 2024b.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. DiscoveryBench: Towards Data-Driven Discovery with Large Language Models. In *ICLR*, 2025.

Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C. Landsness, Dániel L. Barabási, Siddharth Narayanan, Nicky Evans, Shriya Reddy, Martha S. Foiani, Aizad Kamal, Leah P. Shriver, Fang Cao, Asmamaw T. Wassie, Jon M. Laurent, Edwin Melville-Green, Mayk Caldas Ramos, Albert Bou, Kaleigh F. Roberts, Sladjana Zagorac, Timothy C. Orr, Miranda E. Orr, Kevin J. Zwezdaryk, Ali E. Ghareeb, Laurie McCoy, Bruna Gomes, Euan A Ashley, Karen E. Duff, Tonio Buonassisi, Tom Rainforth, Randall J. Bateman, Michael Skarlinski, Samuel G. Rodriques, Michaela M. Hinks, and Andrew D. White. Kosmos: An AI Scientist for Autonomous Discovery. *ArXiv*, abs/2511.02824, 2025.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Niko lay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. MLGym: A New Framework and Benchmark for Advancing AI Research Agents. In *COLM*, 2025.

Alexander Novikov, Ngân V~u, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav M. Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, Matej Balog, and Google Deepmind. Alphaevolve: A coding agent for scientific and algorithmic discovery. *ArXiv*, abs/2506.13131, 2025.

Guilherme Penedo, Hynek Kydlícek, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. *ArXiv*, abs/2406.17557, 2024.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

Esteban Real, Chen Liang, David R. So, and Quoc V. Le. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *ICML*, 2020.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. Agent Laboratory: Using LLM Agents as Research Assistants. In *Findings of EMNLP*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *ArXiv*, abs/2402.03300, 2024.

Chenglei Si, Tatsunori Hashimoto, and Diyi Yang. The Ideation-Execution Gap: Execution Outcomes of LLM-Generated versus Human Research Ideas. *ArXiv*, abs/2506.20803, 2025a.

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. Can LLMs Generate Novel Research Ideas? A Large-Scale Human Study with 100+ NLP Researchers. In *ICLR*, 2025b.

David R. So, Chen Liang, and Quoc V. Le. The Evolved Transformer. In *ICML*, 2019.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's Ability to Replicate AI Research. In *ICML*, 2025.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. AI-Researcher: Autonomous Scientific Innovation. In *NeurIPS*, 2025.

Thinking Machines Lab. Announcing Tinker, 2025.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yaohui Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Min Zhu, Kilian Adriano Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, E. A. Huerta, and Hao Peng. SciCode: A Research Coding Benchmark Curated by Scientists. *ArXiv*, abs/2407.13168, 2024.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Mario Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Stenetorp, Nicola Cancedda, Jakob Nicolaus Foerster, and Yoram Bachrach. AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench. *ArXiv*, abs/2507.02554, 2025.

Qingyun Wang, Doug Downey, Heng Ji, and Tom Hope. SciMON: Scientific Inspiration Machines Optimized for Novelty. In *ACL*, 2024.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. *ArXiv*, abs/2411.15114, 2024.

Fang Wu, Weihao Xuan, Ximing Lu, Zaid Harchaoui, and Yejin Choi. The Invisible Leash: Why RLVR May Not Escape Its Origin. *ArXiv*, abs/2507.14843, 2025.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Nicolaus Foerster, Jeff Clune, and David Ha. The AI Scientist-v2: Workshop-Level Automated Scientific Discovery via Agentic Tree Search. *ArXiv*, abs/2504.08066, 2025.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement. *ArXiv*, abs/2409.12122, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Jingren Zhou, Junyan Lin, Kai Dang, Keqin Bao, Ke-Pei Yang, Le Yu, Li-Chun Deng, Mei Li, Min Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shi-Qiang Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. *ArXiv*, abs/2505.09388, 2025a.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement Learning for Machine Learning Engineering Agents. *ArXiv*, abs/2509.01684, 2025b.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model? In *NeurIPS*, 2025.

Michael R. Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. Using Large Language Models for Hyperparameter Optimization. *ArXiv*, abs/2312.04528, 2023.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of ACL*, 2024.

Minjun Zhu, Yixuan Weng, Linyi Yang, and Yue Zhang. DeepReview: Improving LLM-based Paper Review with Human-like Deep Thinking Process. In *ACL*, 2025.

Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. In *ICLR*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2017.

A Appendix

A.1 Related Work

AutoML Our work has deep connections to the AutoML literature. For example, the Neural Architecture Search (NAS) line of work typically defines a constrained set of neural network operators and optimizes for architectures based on validation set performance through reinforcement learning Zoph & Le (2017); Zoph et al. (2017) or search Liu et al. (2018); So et al. (2019). More recent works explored directly using LLMs to propose architecture variants and implement them for validation Liu et al. (2025); Cheng et al. (2025). Beyond architectures, similar automatic optimizations have been applied to improve hyperparameter tuning Zhang et al. (2023), discover machine learning algorithms Real et al. (2020), improve post-training objectives Lu et al. (2024a), discover better neural network optimizers Chen et al. (2023), and design agent scaffolds Hu et al. (2025). Different from this line of work, we tackle automated AI research in a fully open-ended setting without any constraint on the type of ideas. Moreover, our goal is to improve the effectiveness of idea generation, where natural language ideas represent a higher level of abstraction than specific architecture variants or code optimizations.

LLM-based Research Agents Recent works have been building end-to-end research agents Lu et al. (2024b); Yamada et al. (2025); Tang et al. (2025); Schmidgall et al. (2025) that use LLMs to generate ideas and implement them through carefully designed scaffolds. They address open-ended AI research as we do, but do not study how to learn from execution feedback. On the other hand, on more grounded benchmarks with clear performance metrics such as MLE-Bench Chan et al. (2025), RE-Bench Wijk et al. (2024), and ML-Gym Nathani et al. (2025), various works have explored how to learn from execution feedback through search Toledo et al. (2025); Jiang et al. (2025) or RL Yang et al. (2025b) to optimize performance on these targeted ML engineering tasks. While we also study algorithms for learning from execution feedback, we tackle open-ended research problems like pre-training and post-training rather than ML engineering tasks that heavily depend on feature engineering and hyper-parameter tuning.

AI for Research Apart from fully end-to-end automated AI research, many works have studied how to use LLMs for specific components of the scientific research pipeline, such as literature review Asai et al. (2024); L'ala et al. (2023), idea generation Si et al. (2025b); Wang et al. (2024), data analysis Majumder et al. (2025); Mitchener et al. (2025), experiment plan generation Goel et al. (2025), research code execution Starace et al. (2025); Hua et al. (2025); Tian et al. (2024), and paper reviewing Liang et al. (2024); Zhu et al. (2025). Our work focuses on automated idea execution and learning from the execution feedback. We consider our work complementary to many of the above works that improve other aspects of the scientific research pipeline.

Execution Grounding for Code The idea of learning from execution feedback has been explored in the code generation domain. For example, Zheng et al. (2024) curate data and train models to refine code from either human or execution feedback; Gehring et al. (2025) use end-to-end RL training to teach models to improve code based on execution feedback; Lavon et al. (2025) directly guide code generation with execution signals during inference time. In contrast, our work explores execution grounding for the application of idea generation, where the verification is more complicated and expensive.

A.2 DISCUSSION

Despite encouraging initial signals, there are still many limitations to our current set of experiments, which would be great directions for future improvement.

First, our current procedure does not test the generalizability of the generated ideas. It is possible that the best-performing ideas at the small scales may not transfer to gains at a larger scale or on other datasets. Future work should explore methods that explicitly test such generalizability and scalability, and even incorporate them as part of the optimization objectives.

Second, we have shown that RL with execution reward in our current setup can only improve the average reward but not the upper bound. There are many possible reasons, such as a lack of diversity in the base model and missing exploration incentives in the current RL objective. Future work should explore remedies and better learning algorithms for LLMs to more efficiently learn from the execution feedback. For instance, future works could explore how to exploit richer learning signals from the execution trajectories beyond just scalar rewards.

Third, our current experiment scope is bounded by the capability of the execution agent. There exist many promising model-generated ideas that could not be successfully executed by the execution agent (e.g., see the end of Appendix A.4), leading to noise in the reward signal. Future work could develop more capable execution agents and extend our setup to even more complex research problems. For instance, instead of directly prompting an LLM for code diff, future work can implement more capable coding agents with access to external tools and the ability to install new libraries in the execution environments.

Last but not least, we only explored effectiveness as the training reward in this work. There are many other, more subjective alternative metrics that could complement the effectiveness reward, such as the idea novelty and interestingness. Future work could explore how to computationally measure them and incorporate them as part of the training objective to discover more insightful ideas.

A.3 Other RL Attempts

We present several attempts to improve our RL from the execution reward recipe.

Attempt 1: Dynamic Prompt At each epoch (except the first epoch), we randomly sample different executed idea trajectories from the previous epoch and append them to the idea sampling prompt when sampling new rollouts. This merges in-context learning with RL and adds diversity to the idea sampling process. We present the experiment results on the GRPO environment in Figure 8. We did not see significant improvement in early epochs and thus early stopped.

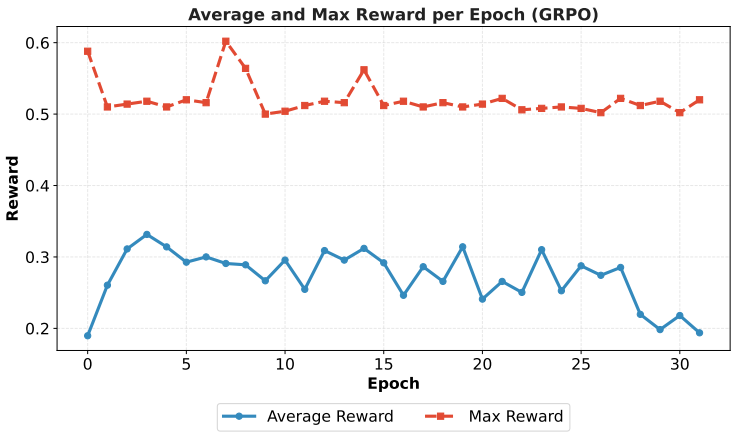


Figure 8: RL with dynamic prompt by appending trajectories from previous epochs. Results are on the GRPO environment.

Attempt 2: Length Reward Since we noted a rapid thinking length decrease in our main RL experiment, we tried a simple fix by adding a weighted length reward that counts the number of tokens in the entire rollout sequence, including the thinking trace and the idea. We cap the length reward to a maximum of 0.3 to avoid it dominating the accuracy reward. We present the experiment results on the GRPO environment in Figure 9. As shown on the right panel, the thinking length no longer decreases after adding the length reward to the total reward, but the total training reward isn't going up as shown on the left panel.

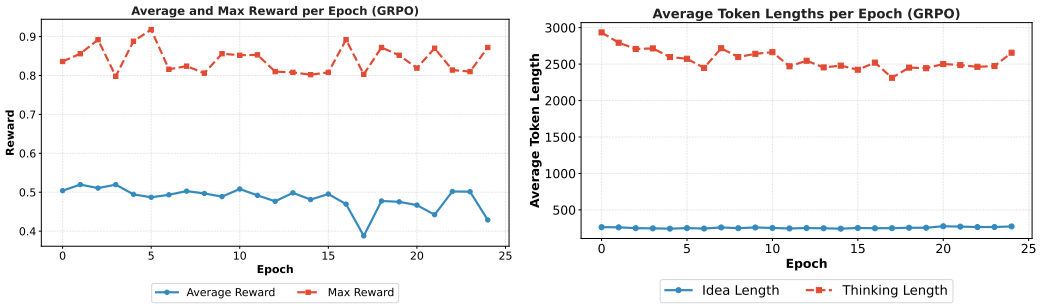


Figure 9: Average and max reward throughout RL training when adding the length reward (left), as well as the progression of the thinking and idea lengths (right).

Attempt 3: Diversity Reward We also tried adding a diversity reward in addition to the effectiveness reward. Specifically, when computing the reward for each rollout, we compute its token-level Jaccard similarity with ideas from the previous epoch and add the negative similarity as a penalty to the total reward to discourage repeating ideas that have already been generated before. In fact, this is similar to one of the post-training ideas proposed by Claude-4.5-Sonnet (see example 4 in Appendix A.5). We show the training curves on the GRPO environment in Figure 10. The model maintains a consistent idea similarity (right plot), suggesting sustained exploration. The effectiveness reward is generally showing an upward trend (left plot), but not markedly better than the main RL run with just the effectiveness reward (first sub-plot in Figure 5).

Figure 10: Effectiveness reward (left) and average idea similarity to previous epoch (right) when doing RL with diversity reward.

A.4 ADDITIONAL IDEA EXAMPLES

We provide several example ideas on the GRPO environment, including ideas with failed code execution. In most cases, code execution errors happen when the idea involves complicated changes or installing and importing external packages not supported in our execution environment. Future work should explore improvement to the execution agent so that more complicated types of ideas (e.g., training additional auxiliary models or system-level optimizations) can be implemented correctly.

Claude-4.5-Opus on GRPO	**Claude-4.5-Sonnet on GRPO**
Residual Ratio Learning with Momentum Bounds: Instead of directly using the (importance sampling) ratio, decompose it into a "base" component (EMA of batch mean ratios) and a "residual" component (ratio – base). Apply sigmoid bounding only to the residual, allowing the base to capture systematic policy drift while controlling deviations from it. Combined with momentum clip adaptation. Formula: `residual = ratio - ema_batch_ratio`, `bounded_residual = sigmoid_bound(residual, deviation)`, `effective_ratio = 1.0 + bounded_residual`. **Validation Accuracy: 61.6**	Dynamic Mathematical Problem Difficulty Balancing with Performance Feedback: Implement intelligent difficulty balancing that dynamically adjusts the mix of problem difficulties based on recent performance trends. When performance is strong, increase difficulty proportion; when struggling, provide more foundational problems. Combine with the proven hyper-parameters for optimal learning progression. **Validation Accuracy: 64.0**
Advantage Rank Difference Weighting: Instead of using absolute advantage magnitude, weight samples by how far their rank differs from their expected rank under uniform distribution. Samples that significantly outperform or underperform their "expected" position get higher weights. This is distribution-free and robust to outliers. Formula: `expected_rank = (N-1)/2`, `rank_diff = |actual_rank - expected_rank| / expected_rank`, `weight = 0.5 + 0.5 * rank_diff`. **Validation Accuracy: 59.2**	Implement token-level reward attribution by using attention weights to identify which input tokens contributed most to correct answers, then amplifying the gradient updates for those tokens during policy gradient training. **Validation Accuracy: 45.2** Create mathematical working memory simulation by maintaining a context buffer of mathematical facts, definitions, and intermediate results during problem solving. This buffer gets updated as the model works through problems and provides additional context for subsequent mathematical steps, simulating how humans maintain mathematical working memory during complex calculations. **Validation Accuracy: 58.0**

Table 3: Examples of successfully executed ideas on the GRPO environment, along with their accuracy on the MATH validation set. The baseline accuracy is 48.0% on this environment.

Successful Execution	Failed Execution
[Experiment] Sequence Position Weighted Trust Region: Apply tighter sigmoid bounds to earlier tokens in the sequence (where errors compound) and looser bounds to later tokens. Weight: `position_weight = 1 - 0.3 * (position / seq_len)`, `effective_deviation = 0.25 + 0.2 * position_weight`. This accounts for the sequential nature of autoregressive generation. **[Code Changes]** Modify `grpo.py`: Initialize `current_cliprange = 0.2`, `ema_clip_fraction = 0.15`. Standard momentum clip updates. Modify `compute_grpo_clip_loss` in `grpo_utils.py`: After computing ratio on line 91 (shape: `batch_size x seq_len`): `batch_size, seq_len = ratio.shape, positions = torch.arange(seq_len, device=ratio.device).float() .unsqueeze(0).expand(batch_size, -1), position_weight = 1.0 - 0.3 * (positions / (seq_len - 1 + 1e-6)), effective_deviation = 0.25 + 0.2 * position_weight`. Apply position-aware sigmoid: `centered_ratio = ratio - 1.0, bounded_ratio = 1.0 + (2.0 * torch.sigmoid(centered_ratio) - 1.0) * effective_deviation`. Use: `surr1 = bounded_ratio * advantages, surr2 = torch.clamp(bounded_ratio, 1 - cliprange, 1 + cliprange) * advantages, loss = -torch.min(surr1, surr2)`. Add metadata: `metadata["mean_effective_deviation"] = effective_deviation.mean().item(), metadata["early_deviation"] = effective_deviation[:, :seq_len//4].mean().item(), metadata["late_deviation"] = effective_deviation[:, -seq_len//4:].mean().item()`. **Validation Accuracy: 59.8**	**[Experiment]** Hierarchical Position-Group Trust Region: Apply trust region at two hierarchical levels – group level (shared within each response group) and position level (varies along sequence). Groups with high internal reward variance get tighter group-level bounds. Within groups, positions follow the proven decay pattern. This captures both cross-sample and within-sample structure. Formula: `group_dev = 0.4 - 0.15 * tanh(group_reward_var / 0.3)`, `position_factor = 1 - 0.2 * rel_pos`, `effective_dev = group_dev * position_factor`. **[Code Changes]** Modify `grpo.py`: Initialize `current_cliprange = 0.2`, `ema_clip_fraction = 0.15`. Standard momentum clip updates. Pass `group_size` to function. Modify `compute_grpo_clip_loss` in `grpo_utils.py`: Add parameter `group_size=8`. After computing ratio: `batch_size, seq_len = ratio.shape, n_groups = batch_size // group_size`. Compute group reward variance from advantages as proxy: `adv_grouped = advantages.view(n_groups, group_size, -1), group_adv_var = adv_grouped.var(dim=1, keepdim=True), group_adv_var_expanded = group_adv_var.expand(-1, group_size, -1).reshape(advantages.shape)`. Group-level deviation: `group_deviation = 0.4 - 0.15 * torch.tanh(group_adv_var_expanded / 0.3)`. Position factor: `positions = torch.arange(seq_len, device=ratio.device).float().unsqueeze(0) .expand(batch_size, -1), rel_pos = positions / (seq_len - 1 + 1e-6), position_factor = 1.0 - 0.2 * rel_pos`. Hierarchical deviation: `effective_deviation = group_deviation * position_factor, effective_deviation = torch.clamp(effective_deviation, 0.15, 0.45)`. Apply: `centered_ratio = ratio - 1.0, bounded_ratio = 1.0 + (2.0 * torch.sigmoid(centered_ratio) - 1.0) * effective_deviation`. Use: `surr1 = bounded_ratio * advantages, surr2 = torch.clamp(bounded_ratio, 1 - cliprange, 1 + cliprange) * advantages, loss = -torch.min(surr1, surr2)`. Add `metadata["mean_group_var"] = group_adv_var.mean().item(), metadata["mean_effective_deviation"] = effective_deviation.mean().item()`. Log to wandb.

Table 4: Additional examples on the GRPO environment. Ideas are generated by Claude-4.5-Opus during evolutionary search.

While the best-performing ideas on nanoGPT tend to be heavily optimized with extensive hyperparameter tuning mixed with various architecture tweaks, we pick a few more "atomic" algorithmic ideas that are successfully executed.

Examples from Claude-4.5-Opus

- **Head-Wise Attention Output Scaling** Add learnable per-head scaling factors to attention, allowing different heads to contribute with different magnitudes to the output.
  Validation Loss: 3.2386
- **Learned Residual Connection Weights** Add learnable scalar weights for each residual connection that are initialized to 1.0, allowing the model to learn optimal residual scaling during training.
  Validation Loss: 3.2517
- **Mixture of Embeddings with Position** Learn to mix token embeddings and position embeddings with a content-dependent weight, allowing the model to dynamically balance positional vs semantic information per token.
  Validation Loss: 3.2497
- **Shared Input-Output Embedding with Learned Asymmetry** Keep weight tying but add a small learned transformation on the output side, providing the benefits of weight tying while allowing output-specific adaptation.
  Validation Loss: 3.2499
- **Gated Final Normalization** Replace the final RMSNorm before `lm_head` with a gated version where a learned gate controls how much normalization is applied vs passing the raw representation.
  Validation Loss: 3.2503
- **Position-Aware MLP Gating** Gate the MLP output based on position information, allowing the model to learn position-dependent processing depth.
  Validation Loss: 3.2506
- **Learned Residual Connection Weights** Add learnable scalar weights for each residual connection that are initialized to 1.0, allowing the model to learn optimal residual scaling during training.
  Validation Loss: 3.2517
- **Grouped Token Embeddings** Group the vocabulary into clusters and add a learned embedding per cluster on top of token embeddings, providing hierarchical vocabulary structure.
  Validation Loss: 3.2521

Lastly, we present several interesting ideas on the nanoGPT environment that didn't get successfully executed. These examples are generated by Claude-4.5-Opus.

- **Soft Layer Repetition** Allow the model to softly repeat computation through layers by adding a learned gate that mixes the current layer's input back into its output, simulating variable depth.
- **Causal Context Compression** Before each attention layer, apply a learned compression that mixes local context (previous 2-3 tokens) into the current representation, providing implicit local context without convolutions.
- **Attention Head Specialization via Orthogonal Loss** Add a soft penalty that encourages different attention heads to attend to different patterns by penalizing similarity between head outputs.
- **Skip Connections with Learned Residual Weights** Combine skip connections with learned residual weights. The skip connections provide alternative gradient paths while learned weights allow adaptive scaling.
- **Token Difficulty-Aware Loss Weighting** Weight the loss contribution of each token based on the model's uncertainty (entropy) at that position, focusing learning on difficult tokens while not over-optimizing easy ones.

A.5 CODE EXECUTION EXAMPLES

We present a few ideas with their full code execution to demonstrate the full end-to-end trajectories. All examples below are from Claude-4.5-Sonnet on the GRPO environment. For each example, we first present the natural language idea, followed by the code implementation generated by Claude-4.5-Sonnet.

Example 1

[Experiment] Create mathematical working memory simulation by maintaining a context buffer of mathematical facts, definitions, and intermediate results during problem solving. This buffer gets updated as the model works through problems and provides additional context for subsequent mathematical steps, simulating how humans maintain mathematical working memory during complex calculations.

[Code Changes] Modify the prompt template in `sample.py` to include a "Mathematical Context" section that accumulates relevant mathematical facts during problem solving. Implement context buffer management in `train_loop` in `grpo.py` that maintains recent mathematical definitions, formulas, and intermediate calculations across problems in the same batch. Update the buffer with key mathematical expressions extracted from each response. Use enhanced prompts that include relevant context from the buffer. Set `-learning_rate 2.5e-5` and `-loss_type reinforce_with_baseline`.

Validation Accuracy: 58.0

Example 1 code diff

```
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/sample.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/sample.py
@@ -2,6 +2,7 @@ from vllm import LLM, SamplingParams
 import json
 from tqdm import tqdm
 import random
+import re

 def load_prompt_template(prompt_path = "prompts/r1_zero.prompt"):
     with open(prompt_path, 'r') as f:
@@ -20,12 +21,52 @@ def get_model_and_sampling_params(model_path = "Qwen/Qwen2.5-Math-1.5B", token
     return llm, sampling_params

-def load_dataset_with_prompt_template(prompt_template, dataset_path="/juice5b/scr5b/nlp/aihinton/CS336_data/a5-alignment/MATH/test.jsonl"):
+def extract_mathematical_expressions(text):
+    """Extract mathematical expressions, formulas, and key facts from text."""
+    expressions = []
+
+    # Extract LaTeX math expressions
+    latex_patterns = [
+        r'\$[^$]+\$',  # Inline math
+        r'\\[a-zA-Z]+\{[^}]+\}',  # LaTeX commands
+        r'\\frac\{[^}]+\}\{[^}]+\}',  # Fractions
+        r'\\sqrt\{[^}]+\}',  # Square roots
+        r'[a-zA-Z]\s*=\s*[^,\n\.]+',  # Variable assignments
+    ]
```

```
+
+    for pattern in latex_patterns:
+        matches = re.findall(pattern, text)
+        expressions.extend(matches)
+
+    # Extract definitions and key facts
+    definition_patterns = [
+        r'[A-Z][a-z]+ is defined as [^\.]+\.',
+        r'Let [a-zA-Z] be [^\.]+\.',
+        r'Given that [^\.]+\.',
+        r'Since [^\.]+\.',
+        r'Therefore [^\.]+\.',
+    ]
+
+    for pattern in definition_patterns:
+        matches = re.findall(pattern, text)
+        expressions.extend(matches)
+
+    return list(set(expressions))  # Remove duplicates
+
+def create_context_prompt(base_template, context_items):
+    """Create a prompt with mathematical context."""
+    if context_items:
+        context_section = "Mathematical Context:\n" + "\n".join(f"- {item}" for item in context_items[-10:]) + "\n\n"
+        return context_section + base_template
+    return base_template
+
+def load_dataset_with_prompt_template(prompt_template, dataset_path="/juice5b/scr5b/nlp/aihinton/CS336_data/a5-alignment/MATH/test.jsonl", context_items=None):
     with open(dataset_path, 'r') as f:
         dataset = [json.loads(line) for line in f]
     print(f"Loaded {len(dataset)} examples from {dataset_path}")

-    prompts = [prompt_template.format(question=example["problem"]) for example in dataset]
+    enhanced_template = create_context_prompt(prompt_template, context_items)
+    prompts = [enhanced_template.format(question=example["problem"]) for example in dataset]
     answers = [example["answer"] for example in dataset]
     return prompts, answers

@@ -93,4 +134,42 @@
     if return_rewards:
         return batched_prompts, batched_answers, batched_responses, batched_rewards
     return results
-
+
+class MathContextBuffer:
+    """Buffer to maintain mathematical working memory across problems."""
+
+    def __init__(self, max_size=50):
+        self.buffer = []
+        self.max_size = max_size
+
+    def add_expressions(self, expressions):
+        """Add new mathematical expressions to the buffer."""
+        for expr in expressions:
```

```
+            if expr and len(expr.strip()) > 3:  # Filter out very short expressions
+                self.buffer.append(expr.strip())
+
+        # Keep only the most recent items
+        if len(self.buffer) > self.max_size:
+            self.buffer = self.buffer[-self.max_size:]
+
+    def get_relevant_context(self, query_text, max_items=10):
+        """Get the most relevant context items for a given query."""
+        if not self.buffer:
+            return []
+
+        # Simple relevance scoring based on shared mathematical terms
+        scored_items = []
+        query_lower = query_text.lower()
+
+        for item in self.buffer[-20:]:  # Consider recent items
+            score = sum(1 for word in item.lower().split() if word in query_lower)
+            if score > 0:
+                scored_items.append((score, item))
+
+        # Return top items sorted by relevance
+        scored_items.sort(reverse=True, key=lambda x: x[0])
+        return [item[1] for item in scored_items[:max_items]]
+
+    def clear(self):
+        """Clear the context buffer."""
+        self.buffer = []
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/grpo.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/grpo.py
@@ -1,6 +1,6 @@
 from transformers import AutoModelForCausalLM, AutoTokenizer
 from utils import tokenize_prompt_and_output, get_response_log_probs
-from sample import load_prompt_template, load_dataset_with_prompt_template, sample_rollout
+from sample import load_prompt_template, load_dataset_with_prompt_template, sample_rollout, MathContextBuffer, extract_mathematical_expressions, create_context_prompt
 from drgrpo_grader import r1_zero_reward_fn_train
 from evaluate import r1_zero_reward_fn_eval, evaluate_vllm
 from grpo_utils import compute_group_normalized_rewards, grpo_microbatch_train_step
@@ -66,6 +66,9 @@ def evaluate_model(policy_model, vllm_model, eval_prompts, eval_answers, eval_s
 def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps, train_steps_per_rollout, output_dir, batch_size, gradient_accumulation_steps = 4, group_size = 2, rollout_subset_size = 256, device = "cuda", logging_steps = 20, saving_steps = 4000, eval_epochs = 5, eval_prompts = None, eval_answers = None, sampling_params = None, eval_vllm_model = None, cliprange = 0.2, loss_type = "reinforce_with_baseline"):
     model.to(device)
     training_steps = grpo_steps
+
```

```
125 +    # Initialize mathematical context buffer
126 +    context_buffer = MathContextBuffer(max_size=100)
127      optimizer = torch.optim.AdamW(model.parameters(), lr=
     learning_rate, weight_decay=0.0, betas=(0.9, 0.95))
128      global_step = 0  # Initialize global step counter
129
130 @@ -85,8 +88,31 @@ def train_loop(model, train_prompts,
     train_answers, learning_rate, grpo_steps,
131          load_policy_into_vllm_instance(model, vllm_model)
132
133          ## sample rollouts
134 +        # Get enhanced prompts with mathematical context for
     this epoch
135          print ("Sampling rollouts for epoch: ", epoch)
136 -        rollout_prompts, rollout_answers, rollout_responses,
     rollout_rewards = sample_rollout(vllm_model,
     r1_zero_reward_fn_train, train_prompts, train_answers, G=
     group_size, eval_sampling_params=eval_sampling_params,
     subset_size=rollout_subset_size, return_rewards=True,
     batch_size=512)
137 +
138 +        # Create enhanced prompts with context for this batch
139 +        enhanced_prompts = []
140 +        base_template = load_prompt_template()
141 +
142 +        # Select subset of problems for this epoch
143 +        if rollout_subset_size is not None:
144 +            indices = random.sample(range(len(train_prompts)),
     rollout_subset_size)
145 +            epoch_prompts = [train_prompts[i] for i in indices]
146 +            epoch_answers = [train_answers[i] for i in indices]
147 +        else:
148 +            epoch_prompts = train_prompts
149 +            epoch_answers = train_answers
150 +
151 +        # Create context-enhanced prompts
152 +        for prompt in epoch_prompts:
153 +            relevant_context = context_buffer.
     get_relevant_context(prompt, max_items=8)
154 +            enhanced_prompt = create_context_prompt(
     base_template.format(question=prompt.split("Question: ")[-1]),
      relevant_context)
155 +            enhanced_prompts.append(enhanced_prompt)
156 +
157 +        # Sample with enhanced prompts (need to adapt
     sample_rollout for direct prompt input)
158 +        rollout_prompts, rollout_answers, rollout_responses,
     rollout_rewards = sample_rollout_with_prompts(vllm_model,
     r1_zero_reward_fn_train, enhanced_prompts, epoch_answers, G=
     group_size, eval_sampling_params=eval_sampling_params,
     return_rewards=True, batch_size=512)
159 +
160          # Randomly sample 2 rollouts to print
161          indices = random.sample(range(len(rollout_prompts)), 2)
162          print ("Example rollouts:")
163 @@ -96,6 +122,15 @@ def train_loop(model, train_prompts,
     train_answers, learning_rate, grpo_steps,
164              print(f"Response: {rollout_responses[idx]}")
165              print(f"Reward: {rollout_rewards[idx]}")
166              print(f"Ground truth: {rollout_answers[idx]}")
167 +
168 +        # Update context buffer with mathematical expressions
     from responses
```

```
+        print("Updating mathematical context buffer...")
+        for response in rollout_responses:
+            if rollout_rewards[rollout_responses.index(response)
    ] > 0:  # Only add from correct responses
+                mathematical_expressions =
    extract_mathematical_expressions(response)
+                if mathematical_expressions:
+                    context_buffer.add_expressions(
    mathematical_expressions)
+
          rollout_tokenized = tokenize_prompt_and_output(
    rollout_prompts, rollout_responses, tokenizer)
          rollout_data_loader = create_data_loader(
    rollout_tokenized, batch_size=batch_size, shuffle=False)

@@ -196,6 +231,47 @@ def train_loop(model, train_prompts,
    train_answers, learning_rate, grpo_steps,
                      }, step=global_step)
                      # print ("Global Step: ", global_step, "Loss:
     ", loss.item(), "Entropy: ", entropy.mean().item(), "Clip
    fraction: ", metadata.get("clip_fraction", 0.0))

+def sample_rollout_with_prompts(
+    vllm_model,
+    reward_fn,
+    prompts,
+    answers,
+    G,
+    eval_sampling_params,
+    return_rewards=False,
+    batch_size=64
+):
+    """Sample rollouts using provided prompts directly."""
+    # Create batched prompts by repeating each prompt G times
+    batched_prompts = []
+    batched_answers = []
+    for prompt, answer in zip(prompts, answers):
+        batched_prompts.extend([prompt] * G)
+        batched_answers.extend([answer] * G)
+
+    # Process in batches to avoid OOM
+    all_outputs = []
+    for i in range(0, len(batched_prompts), batch_size):
+        batch_prompts = batched_prompts[i:i + batch_size]
+        batch_outputs = vllm_model.generate(batch_prompts,
    eval_sampling_params)
+        all_outputs.extend(batch_outputs)
+
+    # Process results
+    batched_responses = []
+    batched_rewards = []
+    total_rewards = 0
+
+    for output, answer in tqdm(zip(all_outputs, batched_answers)
    ):
+        generated_answer = output.outputs[0].text
+        reward = reward_fn(generated_answer, answer)
+        batched_responses.append(generated_answer)
+        batched_rewards.append(reward["reward"])
+        total_rewards += reward["reward"]
+
```

```
+    print(f"Accuracy of sampled rollouts: {total_rewards}/{len(batched_prompts)} = {total_rewards / len(batched_prompts) * 100}%")
+
+    return batched_prompts, batched_answers, batched_responses, batched_rewards
+
 if __name__ == "__main__":
     parser = argparse.ArgumentParser()
     parser.add_argument("--model_path", type=str, default="Qwen/Qwen2.5-Math-1.5B")
@@ -203,7 +279,7 @@ if __name__ == "__main__":
     parser.add_argument("--train_dataset_path", type=str, default="../MATH/train.jsonl")
     parser.add_argument("--eval_dataset_path", type=str, default="../MATH/test.jsonl")
     parser.add_argument("--output_dir", type=str, default="ckpts/")
-    parser.add_argument("--learning_rate", type=float, default=1e-5)
+    parser.add_argument("--learning_rate", type=float, default=2.5e-5)
     parser.add_argument("--grpo_steps", type=int, default=200)
     parser.add_argument("--group_size", type=int, default=8)
     parser.add_argument("--rollout_subset_size", type=int, default=256)
@@ -212,7 +288,7 @@ if __name__ == "__main__":
     parser.add_argument("--batch_size", type=int, default=4)
     parser.add_argument("--cliprange", type=float, default=0.2)
-    parser.add_argument("--loss_type", type=str, default="grpo_clip")
+    parser.add_argument("--loss_type", type=str, default="reinforce_with_baseline")
     parser.add_argument("--wandb_project", type=str, default="grpo-math")
     parser.add_argument("--wandb_name", type=str, default="grpo_clip_1")
     args = parser.parse_args()
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/run_job.sh
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_43/run_job.sh
@@ -21,7 +21,7 @@ timeout 2h uv run   \
     --index https://download.pytorch.org/whl/cu128 \
     --index-strategy unsafe-best-match \
     python grpo.py \
-        --learning_rate 1e-5 \
+        --learning_rate 2.5e-5 \
         --grpo_steps 20 \
         --group_size 8 \
         --rollout_subset_size 128 \
@@ -30,7 +30,7 @@ timeout 2h uv run   \
         --gradient_accumulation_steps 16 \
         --batch_size 4 \
         --cliprange 0.2 \
-        --loss_type grpo_clip \
+        --loss_type reinforce_with_baseline \
         --wandb_name $wandb_name

 echo "Experiment finished successfully!"
```

Example 2

[Experiment] Implement mathematical solution robustness training through systematic perturbation testing that teaches the model to solve mathematically equivalent problems with varied presentations, notation styles, and problem phrasings. Combine this with proven reward shaping by creating robustness-aware rewards that encourage mathematical understanding that generalizes across problem variations.

[Code Changes] Create problem perturbation system in `sample.py` that generates equivalent mathematical problems with varied notation, different variable names, alternative problem phrasing, and equivalent mathematical formulations. Track solution consistency across perturbations. Enhance `r1_zero_reward_fn_train` in `drgrpo_grader.py` to reward robustness: `robustness_bonus = 0.12 * consistency_score_across_perturbations` where consistency measures solution method stability across equivalent problems. Apply generalization bonuses: `+0.08` for solving problems with unfamiliar notation, `+0.06` for handling varied problem presentations. Modify `train_loop` in `grpo.py` to include perturbation-based training batches and track robustness improvement. Use robustness-aware learning rate with cosine annealing from `3.4e-5` to `1.8e-5`. Implement perturbation-complexity gradient clipping. Set `-loss_type reinforce_with_baseline`.

Validation Accuracy: 56.2

Example 2 code diff

```
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/sample.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/sample.py
@@ -2,6 +2,7 @@ from vllm import LLM, SamplingParams
 import json
 from tqdm import tqdm
 import random
+import re

 def load_prompt_template(prompt_path = "prompts/r1_zero.prompt"):
     with open(prompt_path, 'r') as f:
@@ -29,6 +30,125 @@ def load_dataset_with_prompt_template(prompt_template, dataset_path="/juice5b/s
     answers = [example["answer"] for example in dataset]
     return prompts, answers

+def create_notation_perturbations(problem_text):
+    """Create mathematical notation variations"""
+    perturbations = []
+
+    # Variable substitutions
+    var_substitutions = [
+        ('x', 'a'), ('y', 'b'), ('z', 'c'),
+        ('n', 'm'), ('k', 'j'), ('p', 'q'),
+        ('r', 's'), ('t', 'u'), ('v', 'w')
+    ]
+
+    # Create variable name variations
+    for old_var, new_var in var_substitutions:
+        if old_var in problem_text:
+            new_text = problem_text.replace(old_var, new_var)
+            if new_text != problem_text:
+                perturbations.append(("notation_var", new_text))
+
```

```
+    # Mathematical symbol variations
+    symbol_substitutions = [
+        ('\\cdot', '\\times'),
+        ('\\times', '\\cdot'),
+        ('\\frac', '\\div'),
+        ('\\le', '\\leq'),
+        ('\\ge', '\\geq'),
+    ]
+
+    for old_sym, new_sym in symbol_substitutions:
+        if old_sym in problem_text:
+            new_text = problem_text.replace(old_sym, new_sym)
+            perturbations.append(("notation_symbol", new_text))
+
+    return perturbations
+
+def create_phrasing_perturbations(problem_text):
+    """Create alternative problem phrasings"""
+    perturbations = []
+
+    # Common phrasing substitutions
+    phrasing_patterns = [
+        (r"Find the value of", "Determine"),
+        (r"What is", "Calculate"),
+        (r"Solve for", "Find"),
+        (r"How many", "What is the number of"),
+        (r"Compute", "Find"),
+        (r"Evaluate", "Calculate"),
+    ]
+
+    for pattern, replacement in phrasing_patterns:
+        if re.search(pattern, problem_text, re.IGNORECASE):
+            new_text = re.sub(pattern, replacement, problem_text,
     flags=re.IGNORECASE)
+            if new_text != problem_text:
+                perturbations.append(("phrasing", new_text))
+
+    return perturbations
+
+def create_formulation_perturbations(problem_text):
+    """Create equivalent mathematical formulations"""
+    perturbations = []
+
+    # Simple algebraic reformulations
+    reformulations = [
+        (r"(\w+) = (\d+) \+ (\d+)", r"\1 - \2 = \3"),  # a = b +
     c -> a - b = c
+        (r"(\w+) \+ (\d+) = (\d+)", r"\1 = \3 - \2"),  # x + a =
     b -> x = b - a
+        (r"(\d+) - (\w+) = (\d+)", r"\2 = \1 - \3"),  # a - x =
    b -> x = a - b
+    ]
+
+    for pattern, replacement in reformulations:
+        if re.search(pattern, problem_text):
+            new_text = re.sub(pattern, replacement, problem_text
    )
+            if new_text != problem_text:
+                perturbations.append(("formulation", new_text))
+
+    return perturbations
+
```

```
+def generate_problem_perturbations(problem_text, max_perturbations=3):
+    """Generate various perturbations of a mathematical problem """
+    all_perturbations = []
+
+    # Generate different types of perturbations
+    all_perturbations.extend(create_notation_perturbations(problem_text))
+    all_perturbations.extend(create_phrasing_perturbations(problem_text))
+    all_perturbations.extend(create_formulation_perturbations(problem_text))
+
+    # Randomly sample up to max_perturbations
+    if len(all_perturbations) > max_perturbations:
+        all_perturbations = random.sample(all_perturbations, max_perturbations)
+
+    return all_perturbations
+
+def sample_with_perturbations(prompts, answers, perturbation_ratio=0.3):
+    """Sample problems with perturbations mixed in"""
+    perturbed_prompts = []
+    perturbed_answers = []
+    perturbation_metadata = []
+
+    for prompt, answer in zip(prompts, answers):
+        # Always include original
+        perturbed_prompts.append(prompt)
+        perturbed_answers.append(answer)
+        perturbation_metadata.append({"type": "original", "source_idx": len(perturbed_prompts)-1})
+
+        # Add perturbations with some probability
+        if random.random() < perturbation_ratio:
+            # Extract problem text from prompt (assuming it's in a specific format)
+            problem_text = prompt.split("question:")[-1] if "question:" in prompt else prompt
+            perturbations = generate_problem_perturbations(problem_text)
+
+            for pert_type, pert_text in perturbations:
+                # Reconstruct full prompt with perturbed problem
+                pert_prompt = prompt.replace(problem_text, pert_text)
+                perturbed_prompts.append(pert_prompt)
+                perturbed_answers.append(answer)  # Same answer for equivalent problem
+                perturbation_metadata.append({
+                    "type": pert_type,
+                    "source_idx": len(perturbed_prompts)-len(perturbations)-1
+                })
+
+    return perturbed_prompts, perturbed_answers, perturbation_metadata

 def sample_rollout(
     vllm_model,
```

```
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/drgrpo_grader.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/drgrpo_grader.py
@@ -1025,3 +1025,83 @@ def r1_zero_reward_fn_train(response, ground_truth, fast=True):
                "reward": 0.0
           }
 
+def compute_consistency_score(responses, ground_truth, perturbation_metadata):
+    """Compute consistency score across perturbations"""
+    if not perturbation_metadata:
+        return 0.0
+
+    # Group responses by source problem
+    source_groups = {}
+    for i, meta in enumerate(perturbation_metadata):
+        source_idx = meta.get("source_idx", i)
+        if source_idx not in source_groups:
+            source_groups[source_idx] = []
+        source_groups[source_idx].append((i, responses[i], meta))
+
+    total_consistency = 0.0
+    valid_groups = 0
+
+    for source_idx, group_data in source_groups.items():
+        if len(group_data) > 1:  # Only consider groups with multiple responses
+            # Get correctness for each response in the group
+            correctness_scores = []
+            for _, response, meta in group_data:
+                reward_result = r1_zero_reward_fn_train(response, ground_truth)
+                correctness_scores.append(reward_result["reward"])
+
+            # Consistency is measured as how often the model gets the same correctness
+            if correctness_scores:
+                # If all responses have same correctness (all correct or all incorrect), high consistency
+                unique_scores = set(correctness_scores)
+                if len(unique_scores) == 1:
+                    consistency = 1.0
+                else:
+                    # Partial consistency based on agreement
+                    most_common = max(set(correctness_scores), key=correctness_scores.count)
+                    consistency = correctness_scores.count(most_common) / len(correctness_scores)
+
+                total_consistency += consistency
+                valid_groups += 1
+
+    if valid_groups == 0:
+        return 0.0
+
+    return total_consistency / valid_groups
+
+def r1_zero_reward_fn_train_robust(response, ground_truth, perturbation_metadata=None, responses_batch=None, fast=True):
```

```
+    """Enhanced reward function with robustness bonuses"""
+    # Get base reward
+    base_reward = r1_zero_reward_fn_train(response, ground_truth, fast)
+
+    # Initialize bonus components
+    robustness_bonus = 0.0
+    notation_bonus = 0.0
+    presentation_bonus = 0.0
+
+    # Compute robustness bonus if perturbation data available
+    if perturbation_metadata and responses_batch:
+        consistency_score = compute_consistency_score(responses_batch, ground_truth, perturbation_metadata)
+        robustness_bonus = 0.12 * consistency_score
+
+    # Check for notation and presentation bonuses from metadata
+    if perturbation_metadata:
+        current_meta = perturbation_metadata[0] if perturbation_metadata else {}
+        pert_type = current_meta.get("type", "original")
+
+        if pert_type in ["notation_var", "notation_symbol"] and base_reward["reward"] > 0:
+            notation_bonus = 0.08
+
+        if pert_type in ["phrasing", "formulation"] and base_reward["reward"] > 0:
+            presentation_bonus = 0.06
+
+    # Combine all rewards
+    total_reward = base_reward["reward"] + robustness_bonus + notation_bonus + presentation_bonus
+
+    return {
+        "format_reward": base_reward["format_reward"],
+        "answer_reward": base_reward["answer_reward"],
+        "robustness_bonus": robustness_bonus,
+        "notation_bonus": notation_bonus,
+        "presentation_bonus": presentation_bonus,
+        "reward": total_reward
+    }
--- repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/grpo.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch7/idea_37/grpo.py
@@ -1,7 +1,7 @@
 from transformers import AutoModelForCausalLM, AutoTokenizer
 from utils import tokenize_prompt_and_output, get_response_log_probs
-from sample import load_prompt_template, load_dataset_with_prompt_template, sample_rollout
-from drgrpo_grader import r1_zero_reward_fn_train
+from sample import load_prompt_template, load_dataset_with_prompt_template, sample_rollout, sample_with_perturbations
+from drgrpo_grader import r1_zero_reward_fn_train, r1_zero_reward_fn_train_robust
 from evaluate import r1_zero_reward_fn_eval, evaluate_vllm
 from grpo_utils import compute_group_normalized_rewards, grpo_microbatch_train_step
 from torch.utils.data import DataLoader, Dataset
@@ -12,6 +12,7 @@ from tqdm import tqdm
```

```
 from vllm import LLM, SamplingParams
 import wandb
 import random
+import math

 def load_policy_into_vllm_instance(policy, llm):
     state_dict = policy.state_dict()
@@ -63,11 +64,23 @@
     metrics = evaluate_vllm(vllm_model, r1_zero_reward_fn_eval, eval_prompts, eval_answers, eval_sampling_params, output_path=output_path)
     return metrics

-def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps, train_steps_per_rollout, output_dir, batch_size, gradient_accumulation_steps = 4, group_size = 2, rollout_subset_size = 256, device = "cuda", logging_steps = 20, saving_steps = 4000, eval_epochs = 5, eval_prompts = None, eval_answers = None, sampling_params = None, eval_vllm_model = None, cliprange = 0.2, loss_type = "reinforce_with_baseline"):
+def get_cosine_annealing_lr(step, total_steps, lr_max=3.4e-5, lr_min=1.8e-5):
+    """Cosine annealing learning rate schedule"""
+    return lr_min + (lr_max - lr_min) * 0.5 * (1 + math.cos(math.pi * step / total_steps))
+
+def compute_perturbation_complexity_clip_norm(model, complexity_factor=1.0):
+    """Compute gradient clipping norm based on perturbation complexity"""
+    base_clip_norm = 1.0
+    return base_clip_norm * complexity_factor
+
+def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps, train_steps_per_rollout, output_dir, batch_size, gradient_accumulation_steps = 4, group_size = 2, rollout_subset_size = 256, device = "cuda", logging_steps = 20, saving_steps = 4000, eval_epochs = 5, eval_prompts = None, eval_answers = None, sampling_params = None, eval_vllm_model = None, cliprange = 0.2, loss_type = "reinforce_with_baseline", perturbation_ratio = 0.3):
     model.to(device)
     training_steps = grpo_steps
-    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=0.0, betas=(0.9, 0.95))
+    # Start with robustness-aware learning rate
+    initial_lr = get_cosine_annealing_lr(0, grpo_steps)
+    optimizer = torch.optim.AdamW(model.parameters(), lr=initial_lr, weight_decay=0.0, betas=(0.9, 0.95))
     global_step = 0  # Initialize global step counter
+    robustness_scores = []

     for epoch in range(grpo_steps):
         model.train()
@@ -82,21 +95,50 @@ def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps,

             model.train()

+        # Update learning rate with cosine annealing
+        current_lr = get_cosine_annealing_lr(epoch, grpo_steps)
+        for param_group in optimizer.param_groups:
```

```
276 +            param_group['lr'] = current_lr
277 +
278          ## load the current policy model to vllm for sampling
    rollouts
279          load_policy_into_vllm_instance(model, vllm_model)
280
281 +        # Sample with perturbations for robustness training
282 +        perturbed_prompts, perturbed_answers,
    perturbation_metadata = sample_with_perturbations(
283 +            train_prompts, train_answers, perturbation_ratio=
    perturbation_ratio
284 +        )
285 +
286          ## sample rollouts
287          print ("Sampling rollouts for epoch: ", epoch)
288 -        rollout_prompts, rollout_answers, rollout_responses,
    rollout_rewards = sample_rollout(vllm_model,
    r1_zero_reward_fn_train, train_prompts, train_answers, G=
    group_size, eval_sampling_params=eval_sampling_params,
    subset_size=rollout_subset_size, return_rewards=True,
    batch_size=512)
289 +
290 +        # Use subset of perturbed prompts for training
291 +        subset_size = min(rollout_subset_size, len(
    perturbed_prompts))
292 +        if subset_size < len(perturbed_prompts):
293 +            indices = random.sample(range(len(perturbed_prompts)
    ), subset_size)
294 +            subset_prompts = [perturbed_prompts[i] for i in
    indices]
295 +            subset_answers = [perturbed_answers[i] for i in
    indices]
296 +            subset_metadata = [perturbation_metadata[i] for i in
     indices]
297 +        else:
298 +            subset_prompts = perturbed_prompts
299 +            subset_answers = perturbed_answers
300 +            subset_metadata = perturbation_metadata
301 +
302 +        rollout_prompts, rollout_answers, rollout_responses,
    rollout_rewards = sample_rollout(
303 +            vllm_model, r1_zero_reward_fn_train, subset_prompts,
     subset_answers,
304 +            G=group_size, eval_sampling_params=
    eval_sampling_params,
305 +            subset_size=None, return_rewards=True, batch_size
    =512
306 +        )
307 +
308          # Randomly sample 2 rollouts to print
309          indices = random.sample(range(len(rollout_prompts)), 2)
310          print ("Example rollouts:")
311          for idx in indices:
312              print(f"\nRollout {idx}:")
313 -            print(f"Prompt: {rollout_prompts[idx]}")
314 +            print(f"Prompt: {rollout_prompts[idx][:200]}...")
315              print(f"Response: {rollout_responses[idx]}")
316              print(f"Reward: {rollout_rewards[idx]}")
317 -            print(f"Ground truth: {rollout_answers[idx]}")
318 +            print(f"Ground truth: {rollout_answers[idx
    ][:100]}...")
319 +
```

```
320           rollout_tokenized = tokenize_prompt_and_output(
      rollout_prompts, rollout_responses, tokenizer)
321           rollout_data_loader = create_data_loader(
      rollout_tokenized, batch_size=batch_size, shuffle=False)
322
323   @@ -126,15 +168,32 @@ def train_loop(model, train_prompts,
      train_answers, learning_rate, grpo_steps,
324
325           # Compute advantages using group normalization - no
      gradients needed
326           with torch.no_grad():
327   -           advantages, raw_rewards, metadata =
      compute_group_normalized_rewards(
328   -               reward_fn=r1_zero_reward_fn_train,
329   +           # Create enhanced reward function with robustness
330   +           def robust_reward_fn(response, ground_truth):
331   +               # Find corresponding metadata for this response
332   +               response_idx = rollout_responses.index(response)
       if response in rollout_responses else 0
333   +               meta_idx = min(response_idx, len(subset_metadata
      ) - 1)
334   +               current_meta = [subset_metadata[meta_idx]] if
      subset_metadata else None
335   +
336   +               return r1_zero_reward_fn_train_robust(
337   +                   response, ground_truth,
338   +                   perturbation_metadata=current_meta,
339   +                   responses_batch=rollout_responses
340   +               )
341   +
342   +           advantages, raw_rewards, metadata =
      compute_group_normalized_rewards(
343   +               reward_fn=robust_reward_fn,
344                   rollout_responses=rollout_responses,
345                   repeated_ground_truths=rollout_answers,
346                   group_size=group_size,
347                   advantage_eps=1e-6,
348                   normalize_by_std=True
349               )
350               advantages = advantages.to(device)
351   +
352   +           # Track robustness improvement
353   +           current_robustness = metadata.get('mean_reward',
      0.0)
354   +           robustness_scores.append(current_robustness)
355
356           # Log raw rewards statistics
357           print("\nGRPO epoch: ", epoch)
358   @@ -145,11 +204,20 @@
359               wandb.log({
360                   "eval/mean_reward": eval_mean_reward,
361                   "train/mean_reward": metadata["mean_reward"],
362   +               "train/learning_rate": current_lr,
363   +               "train/robustness_score": current_robustness,
364               }, step=global_step)
365           else:
366               wandb.log({
367                   "train/mean_reward": metadata["mean_reward"],
368   +               "train/learning_rate": current_lr,
369   +               "train/robustness_score": current_robustness,
370               }, step=global_step)
371   +
372   +       # Compute perturbation complexity for gradient clipping
```

```
+         perturbation_types = set(meta.get("type", "original") for meta in subset_metadata)
+         complexity_factor = 1.0 + 0.1 * len(perturbation_types)  # More complex with more perturbation types
+         clip_norm = compute_perturbation_complexity_clip_norm(model, complexity_factor)


          ## train on this rollout batch for train_steps_per_rollout steps
@@ -185,6 +252,9 @@ def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps,
                  )

                  if (batch_idx + 1) % gradient_accumulation_steps == 0:
+                     # Apply perturbation-complexity gradient clipping
+                     torch.nn.utils.clip_grad_norm_(model.parameters(), clip_norm)
+
                      optimizer.step()
                      optimizer.zero_grad()

@@ -211,7 +281,7 @@ if __name__ == "__main__":
      parser.add_argument("--gradient_accumulation_steps", type=int, default=16)
      parser.add_argument("--batch_size", type=int, default=4)
      parser.add_argument("--cliprange", type=float, default=0.2)
-     parser.add_argument("--loss_type", type=str, default="grpo_clip")
+     parser.add_argument("--loss_type", type=str, default="reinforce_with_baseline")
      parser.add_argument("--wandb_project", type=str, default="grpo-math")
      parser.add_argument("--wandb_name", type=str, default="grpo_clip_1")
      args = parser.parse_args()
@@ -266,7 +336,8 @@ if __name__ == "__main__":
          sampling_params=eval_sampling_params,
          eval_vllm_model=vllm_model,
          cliprange=args.cliprange,
-         loss_type=args.loss_type
+         loss_type=args.loss_type,
+         perturbation_ratio=0.3
      )
```

Example 3

[Experiment] Implement token-level reward attribution by using attention weights to identify which input tokens contributed most to correct answers, then amplifying the gradient updates for those tokens during policy gradient training.

[Code Changes] Modify `get_response_log_probs` in `utils.py` to also return attention weights from the last layer. In `grpo_microbatch_train_step`, compute token importance scores by averaging attention weights across heads, then multiply the policy gradient loss by `(1 + importance_score)` for tokens with high attention to mathematical content.

Validation Accuracy: 45.2

Example 3 code diff

```
--- repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/utils.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/utils.py
@@ -41,12 +41,13 @@ def compute_entropy(logits):
     entropy = -torch.sum(probs * log_probs, dim=-1)
     return entropy

-def get_response_log_probs(model, input_ids, labels, return_token_entropy=False, no_grad=True):
+def get_response_log_probs(model, input_ids, labels, return_token_entropy=False, return_attention=False, no_grad=True):
     if no_grad:
         with torch.no_grad():
-            outputs = model(input_ids, labels=labels)
+            outputs = model(input_ids, labels=labels, output_attentions=return_attention)
             logits = outputs.logits # (batch_size, seq_len, vocab_size)
             log_probs = torch.log_softmax(logits, dim=-1) # (batch_size, seq_len, vocab_size)
+            attentions = outputs.attentions if return_attention else None
             # Get log probs of the actual label tokens
             batch_size, seq_len = labels.shape # (batch_size, seq_len)
             log_probs = torch.gather(log_probs, dim=-1, index=labels.unsqueeze(-1)).squeeze(-1)
@@ -55,8 +56,9 @@ def get_response_log_probs(model, input_ids, labels, return_token_entropy=False
                 entropy = None
     else:
-        outputs = model(input_ids, labels=labels)
+        outputs = model(input_ids, labels=labels, output_attentions=return_attention)
         logits = outputs.logits # (batch_size, seq_len, vocab_size)
+        attentions = outputs.attentions if return_attention else None
         log_probs = torch.log_softmax(logits, dim=-1) # (batch_size, seq_len, vocab_size)
         # Get log probs of the actual label tokens
         batch_size, seq_len = labels.shape # (batch_size, seq_len)
```

```
@@ -65,9 +67,17 @@ def get_response_log_probs(model, input_ids, labels, return_token_entropy=False
                entropy = compute_entropy(logits)
            else:
                entropy = None
-
-    return {
+
+    result = {
            "log_probs": log_probs,
            "token_entropy": entropy
        }
+
+    if return_attention and attentions is not None:
+        # Return attention weights from the last layer, averaged across heads
+        last_layer_attention = attentions[-1]  # Shape: (batch_size, num_heads, seq_len, seq_len)
+        averaged_attention = last_layer_attention.mean(dim=1)  # Average across heads: (batch_size, seq_len, seq_len)
+        result["attention_weights"] = averaged_attention
+
+    return result

--- repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/grpo_utils.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/grpo_utils.py
@@ -170,6 +170,7 @@ def grpo_microbatch_train_step(
     advantages: torch.Tensor | None = None,
     old_log_probs: torch.Tensor | None = None,
     cliprange: float | None = None,
+    attention_weights: torch.Tensor | None = None,
 ) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
     '''
     Return:
@@ -180,6 +181,20 @@ def grpo_microbatch_train_step(
     You should call loss.backward() in this function. Make sure to adjust for gradient accumulation.
     '''
     loss, metadata = compute_policy_gradient_loss(policy_log_probs, loss_type, raw_rewards, advantages, old_log_probs, cliprange) # (batch_size, sequence_length)
+
+    # Apply attention-based weighting if attention weights are provided
+    if attention_weights is not None:
+        # Compute importance scores by averaging attention weights to earlier tokens
+        # attention_weights shape: (batch_size, seq_len, seq_len)
+        # For each token, sum attention weights to all previous tokens (including itself)
+        importance_scores = torch.sum(attention_weights, dim=-1)  # (batch_size, seq_len)
+
+        # Normalize importance scores to [0, 1] range per sequence
+        importance_scores = importance_scores / (importance_scores.max(dim=-1, keepdim=True)[0] + 1e-8)
+
+        # Amplify loss for tokens with high importance: multiply by (1 + importance_score)
```

```
+        loss = loss * (1.0 + importance_scores)
+
     loss = masked_mean(loss, response_mask)
     loss = loss / gradient_accumulation_steps
     loss.backward()
--- repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/grpo.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch1/idea_32/grpo.py
@@ -109,6 +109,7 @@ def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps,
                     model,
                     input_ids,
                     labels,
                     return_token_entropy=False,
+                    return_attention=False,
                     no_grad=True
                 )
@@ -163,11 +164,13 @@
                     model,
                     input_ids,
                     labels,
                     return_token_entropy=True,
+                    return_attention=True,
                     no_grad=False
                 )
                 policy_log_probs = response_log_probs["log_probs"]
                 entropy = response_log_probs["token_entropy"]
+                attention_weights = response_log_probs.get("attention_weights")

                 # Calculate data index for advantages/old_log_probs
                 batch_idx_total = batch_idx * batch_size
@@ -177,7 +180,8 @@ def train_loop(model, train_prompts, train_answers, learning_rate, grpo_steps,
                     loss_type=loss_type,
                     advantages=batch_advantages,
                     old_log_probs=batch_old_log_probs,
-                    cliprange=cliprange
+                    cliprange=cliprange,
+                    attention_weights=attention_weights
                 )

                 if (batch_idx + 1) % gradient_accumulation_steps == 0:
```

Example 4

[Experiment] Implement response diversity rewards within groups where responses to the same prompt receive bonus rewards (0.05-0.15) for being dissimilar to other responses in their group, encouraging exploration of different solution paths while maintaining the proven `group_size=8` and `3e-5` learning rate combination.

[Code Changes] Modify `compute_group_normalized_rewards` in `grpo_utils.py` to compute pairwise similarity between responses in each group using token-level Jaccard similarity. Add diversity bonus: `diversity_reward = 0.15 * (1 - max_similarity_in_group)` to each response's reward before advantage computation. Reshape responses into groups, compute similarities, and add bonuses before advantage normalization. Set `--learning_rate 3e-5`, `--loss_type reinforce_with_baseline`, `--group_size 8`.

Validation Accuracy: 19.2

Example 4 code diff

```
--- repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/grpo_utils.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/grpo_utils.py
@@ -1,5 +1,25 @@
 import torch
 from typing import Literal

+def compute_jaccard_similarity(response1: str, response2: str) -> float:
+    """
+    Compute token-level Jaccard similarity between two responses.

+    Jaccard similarity = |intersection| / |union|
+    """
+    # Simple tokenization by splitting on whitespace
+    tokens1 = set(response1.split())
+    tokens2 = set(response2.split())
+
+    if len(tokens1) == 0 and len(tokens2) == 0:
+        return 1.0
+
+    intersection = tokens1.intersection(tokens2)
+    union = tokens1.union(tokens2)
+
+    if len(union) == 0:
+        return 1.0
+
+    return len(intersection) / len(union)
+
 def compute_group_normalized_rewards(
     reward_fn,
@@ -30,31 +49,62 @@
         raw_rewards.append(reward_dict["reward"])
     raw_rewards = torch.tensor(raw_rewards)

+    # Compute diversity bonuses within each group
+    n_groups = len(rollout_responses) // group_size
+    diversity_bonuses = []
+
+    for group_idx in range(n_groups):
+        group_start = group_idx * group_size
```

```
+         group_end = group_start + group_size
+         group_responses = rollout_responses[group_start:group_end]
+
+         # Compute pairwise similarities within the group
+         for i in range(group_size):
+             max_similarity = 0.0
+             for j in range(group_size):
+                 if i != j:
+                     similarity = compute_jaccard_similarity(group_responses[i], group_responses[j])
+                     max_similarity = max(max_similarity, similarity)
+
+             # Diversity bonus: higher reward for more dissimilar responses
+             diversity_bonus = 0.15 * (1 - max_similarity)
+             diversity_bonuses.append(diversity_bonus)
+
+     diversity_bonuses = torch.tensor(diversity_bonuses)
+
      # Reshape rewards into groups
      n_groups = len(raw_rewards) // group_size
      grouped_rewards = raw_rewards.view(n_groups, group_size)
+     grouped_diversity_bonuses = diversity_bonuses.view(n_groups, group_size)
+
+     # Add diversity bonuses to raw rewards before advantage computation
+     grouped_rewards_with_diversity = grouped_rewards + grouped_diversity_bonuses

      # Compute group statistics
-     group_means = grouped_rewards.mean(dim=1, keepdim=True)
+     group_means = grouped_rewards_with_diversity.mean(dim=1, keepdim=True)
      if normalize_by_std:
-         group_stds = grouped_rewards.std(dim=1, keepdim=True) + advantage_eps
-         advantages = (grouped_rewards - group_means) / group_stds
+         group_stds = grouped_rewards_with_diversity.std(dim=1, keepdim=True) + advantage_eps
+         advantages = (grouped_rewards_with_diversity - group_means) / group_stds
      else:
-         advantages = grouped_rewards - group_means
+         advantages = grouped_rewards_with_diversity - group_means

      # Flatten advantages back to original shape
      advantages = advantages.view(-1)
+
+     # Update raw rewards to include diversity bonuses for metadata
+     raw_rewards_with_diversity = raw_rewards + diversity_bonuses

      # Compute metadata statistics
      metadata = {
-         "mean_reward": raw_rewards.mean().item(),
-         "std_reward": raw_rewards.std().item(),
-         "max_reward": raw_rewards.max().item(),
-         "min_reward": raw_rewards.min().item(),
```

```
+        "mean_reward": raw_rewards_with_diversity.mean().item(),
+        "std_reward": raw_rewards_with_diversity.std().item(),
+        "max_reward": raw_rewards_with_diversity.max().item(),
+        "min_reward": raw_rewards_with_diversity.min().item(),
         "mean_advantage": advantages.mean().item(),
         "std_advantage": advantages.std().item(),
+        "mean_diversity_bonus": diversity_bonuses.mean().item(),
     }

-    return advantages, raw_rewards, metadata
+    return advantages, raw_rewards_with_diversity, metadata

 def compute_naive_policy_gradient_loss(
--- repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/grpo.py
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/grpo.py
@@ -203,6 +203,6 @@
     parser.add_argument("--eval_dataset_path", type=str, default="../MATH/test.jsonl")
     parser.add_argument("--output_dir", type=str, default="ckpts/")
-    parser.add_argument("--learning_rate", type=float, default=1e-5)
+    parser.add_argument("--learning_rate", type=float, default=3e-5)
     parser.add_argument("--grpo_steps", type=int, default=200)
     parser.add_argument("--group_size", type=int, default=8)
     parser.add_argument("--rollout_subset_size", type=int, default=256)
@@ -212,7 +212,7 @@ if __name__ == "__main__":
     parser.add_argument("--gradient_accumulation_steps", type=int, default=16)
     parser.add_argument("--batch_size", type=int, default=4)
     parser.add_argument("--cliprange", type=float, default=0.2)
-    parser.add_argument("--loss_type", type=str, default="grpo_clip")
+    parser.add_argument("--loss_type", type=str, default="reinforce_with_baseline")
     parser.add_argument("--wandb_project", type=str, default="grpo-math")
     parser.add_argument("--wandb_name", type=str, default="grpo_clip_1")
     args = parser.parse_args()
--- repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/run_job.sh
+++ repo_variants_full_pipeline_bsz50_fixed2_epoch3/idea_22/run_job.sh
@@ -21,7 +21,7 @@ timeout 2h uv run   \
     --index https://download.pytorch.org/whl/cu128 \
     --index-strategy unsafe-best-match \
     python grpo.py \
-        --learning_rate 1e-5 \
+        --learning_rate 3e-5 \
         --grpo_steps 20 \
         --group_size 8 \
         --rollout_subset_size 128 \
@@ -30,7 +30,7 @@ timeout 2h uv run   \
         --gradient_accumulation_steps 16 \
         --batch_size 4 \
         --cliprange 0.2 \
-        --loss_type grpo_clip \
+        --loss_type reinforce_with_baseline \
```

```
137             --wandb_name $wandb_name
138
139   echo "Experiment finished successfully!"
```