Teaching Language Models to Self-Improve through Interactive Demonstrations

Xiao Yu[†] **Baolin Peng**[‡*] **Michel Galley**[‡] **Jianfeng Gao**[‡] **Zhou Yu**[†]
[†]Columbia University [‡]Microsoft Research
{xy2437,zy2416}@columbia.edu
{mgalley,jfgao}@microsoft.com

Abstract

The self-improving ability of large language models (LLMs), enabled by prompting them to analyze and revise their own outputs, has garnered significant interest in recent research. However, this ability has been shown to be absent and difficult to learn for smaller models, thus widening the performance gap between state-of-the-art LLMs and more cost-effective and faster ones. To reduce this gap, we introduce TRIPOST, a training algorithm that endows smaller models with such self-improvement ability, and show that our approach can improve LLaMA-7B's performance on math and reasoning tasks by up to 7.13%. In contrast to prior work, we achieve this by using the smaller model to interact with LLMs to collect feedback and improvements on *its own generations*. We then replay this experience to train the small model. Our experiments on four math and reasoning datasets show that the interactive experience of learning from and correcting its *own* mistakes is crucial for small models to improve their performance.

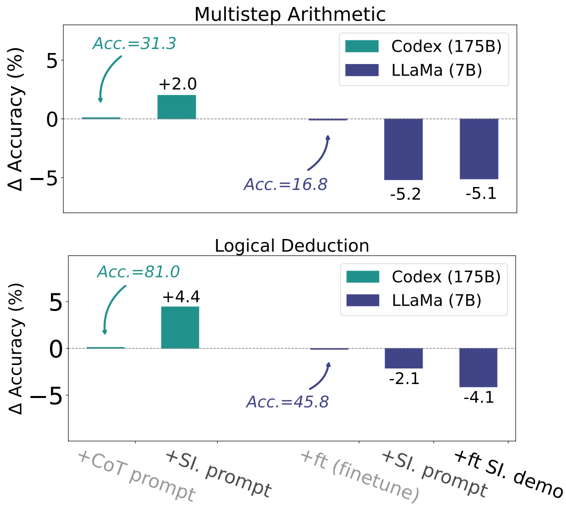


Figure 1: Compared to LLMs, smaller models have difficulty performing self-improvement on math or logical tasks, such as Multistep Arithmetics and Logical Deduction from the Big-Bench. *+ft*: finetuned on ground-truth rationales; *+SI. prompt*: prompted to perform self-improvement; *+ft SI. demo* further finetuned *+ft* on LLM self-improvement demonstrations.

1 Introduction

Large language models (OpenAI, 2023; Ouyang et al., 2022) together with techniques such as few-shot prompting (Brown et al., 2020) and Chain-of-Thought (CoT) prompting (Wei et al., 2023; Kojima et al., 2023) have been shown to be effective in achieving strong performance on various downstream language tasks. More recently, a new way to adapt LLMs to downstream tasks has captured the attention of many researchers, namely to further enhance the LLM's downstream task performance by asking the LLM to provide feedback on its own generations and then use the feedback to revise its outputs (Bai et al., 2022; Huang et al., 2023; Peng et al., 2023a; Shinn et al., 2023). This process is often called "self-improvement", and has proven to be an effective technique to make the LLM's generations more diverse, more precise, or more faithful to a given piece of knowledge (Schick et al., 2022; Madaan et al., 2023; Peng et al., 2023a).

However, Saunders et al. (2022); Huang et al. (2023) found that the ability to generate critical feedback or to self-improve is hardly evident in smaller models[1]. Similarly, Ye et al. (2023) found that fine-tuning smaller models (e.g. 7-13B) with self-improvement demonstrations from LLMs can still fail on tasks such as math, reasoning, and factuality. Following these previous works, we performed a similar study on two math and reasoning tasks in Figure 1. We compared the accuracy of the final answer generated by prompting

[*]Now at Tencent AI (baolinpeng@global.tencent.com).

[1]The distinction between small and large language models is often context-dependent (Saunders et al., 2022). In this work, we denote "small models" to those with a few billion parameters (e.g., LLaMA-7B), and LLMs as those scaled to hundreds of billions of parameters (e.g., ChatGPT).

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 5127–5149
June 16-21, 2024 ©2024 Association for Computational Linguistics

Method	Codex (175B)	LLaMA+ft (7B)
Input	Q: ((4 - 0 * -1 * 8 + 6) + (0 + 4 + 9 * 5 * -2)) =	
Attempt:	(1) This equation can be written as "(A + B)" ... (2) Let's calculate A=(4 - 0 * -1 * 8 + 6) =(4 - (0 * -1 * 8) + 6) =(4 - (0 + 6)) =(4 - 6) =-2 (3) Let's calculate B=(0 + 4 + 9 * 5 * -2) =(0 + 4 + (9 * 5 * -2)) =(0 + 4 + (-90)) =-86 (4) Then, the final eq is (A + B) = (-2 + -86) = -88. (Final response) So the answer is -88.	(1) This equation can be written as "(A + B)" ... (2) Let's calculate A=(4 - 0 * -1 * 8 + 6) =(4 - (0 * -1 * 8) + 6) =(4 - (0 + 8) + 6) =(4 - 8 + 6) =2 (3) Let's calculate B=(0 + 4 + 9 * 5 * -2) =(0 + (4 + (9 * 5))) =(0 + (13)) =13 (4) Then, the final eq is (A + B) = (2 + 13) = 15. (Final response) So the answer is 15.

Table 1: Training smaller models using self-improvement demonstrations from LLMs can be ineffective, as models of different sizes make different types and amount of mistakes (highlighted in red). Small models can make simple copying errors, while LLMs can make other arithmetic errors, such as not switching plus or minus signs when adding parentheses. See Appendix B for a more quantitative analysis.

a 175B Codex (Chen et al., 2021) to self-improve, with prompting or training a LLaMA-7B model to self-improve using demonstrations from Codex (Ye et al., 2023). In Figure 1, we surprisingly find that *smaller models performed worse* using prior self-improvement-related methods than simply training on ground-truth step-by-step rationales (*+ft*). By comparing the generated solutions from Codex-175B and LLaMA-7B, we find that smaller models, such as LLaMA-7B, not only make more mistakes, but also *different types of mistakes* compared to an LLM (Table 1 and Appendix B). Due to the smaller model's weaker math and reasoning ability, we believe training on LLM self-improvement demonstrations is less effective, as it forces the smaller model to learn from mistakes not of its own.

Motivated by this finding, we propose TRIPOST, a training algorithm that can more effectively train a small model to learn from its mistakes, generate feedback, and improve its performance on math and reasoning tasks. TRIPOST is an iterative algorithm consisting of three stages: Interactive **Tr**ajectory Ed**i**ting, Data **Pos**t-processing, and Model **T**raining. Similar to the exploration stage in reinforcement learning, TRIPOST first creates improvement demonstrations *using the small model to interact* with the expert LLMs or relevant Python scripts. Then, TRIPOST postprocesses the collected data by filtering out failed improvement attempts, and then re-balances the dataset to disincentivize the model from trying to self-"improve" when it is not needed. Finally, TRIPOST replays the post-process dataset (Andrychowicz et al., 2018; Schaul et al., 2016), and trains the smaller model using weighted supervised learning. TRIPOST repeats entire the process several times. We evaluate our approach on four maths and reasoning datasets from the BIG-Bench Hard (Suzgun et al., 2022) collection, and find that TRIPOST-trained models can use its learned self-improvement ability to improve their task performance. We also find that TRIPOST-trained models achieve better in-domain and out-of-domain performance than models trained using just the ground truth step-by-step rationales and trained using direct LLM demonstrations (Saunders et al., 2022; Ye et al., 2023). This paper makes the following contributions:

- We illustrate how prior work (Saunders et al., 2022; Ye et al., 2023) can be ineffective in training smaller models to self-improve their performance on math and reasoning tasks.
- We propose TRIPOST, an iterative training algorithm that trains a smaller language model to learn to self-improve.
- We show that TRIPOST-trained models achieve better performance than models trained using ground-truth rationales or using LLM demonstrations on four math and reasoning datasets from BIG-Bench Hard.

2 Approach

TRIPOST is an algorithm that trains a small language model to self-improve by learning from its *own mistakes*. Each iteration of TRIPOST consists of three stages. On a high level, we first collect

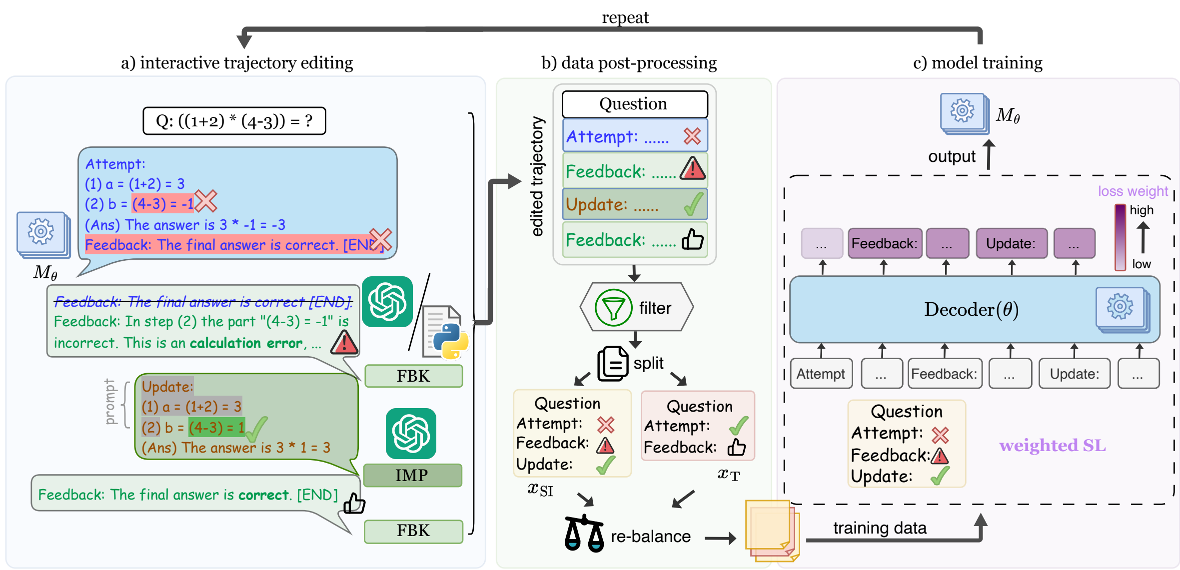


Figure 2: Overview of TRIPOST algorithm. TRIPOST consists of three stages: interactive trajectory editing where we use our FBK and IMP module to edit trajectories generated by a smaller model M_θ; data post-processing where we filter out erroneous trajectories and create a re-balanced dataset; and model training where we train M_θ using weighted supervised learning on the post-processed dataset.

a set of improving trajectories by using a smaller model M_θ to interact with LLMs. We use M_θ to generate initial attempts and then use a feedback module FBK and an improvement module IMP to edit parts of the M_θ generated attempts. This creates a trajectory that includes attempts generated by the small model, with feedbacks and improvements tailored to the small model's capability (Figure 2). Next, we post-process the collected trajectories by 1) using scripts and other heuristics to filter out failed "improvement" attempts; and 2) re-balancing the dataset using both directly correct attempts and the improving trajectories. Finally, we use weighted supervised learning to train a smaller model M_θ using the post-processed data.

We provide an overview of our algorithm in Figure 2, and detail each of the three stages in Section 2.2, Section 2.3, and Section 2.4, respectively.

2.1 Notation

We denote the entire attempt from a language model to solve a given question as a trajectory x:

$$x = (x_0^{\text{att}}, x_1^{\text{fb}}, x_1^{\text{att}}, x_2^{\text{fb}}, x_2^{\text{att}}, ..., x_m^{\text{fb}}),$$

where x_0^{att} denotes the initial attempt, and $x_i^{\text{fb}}, x_i^{\text{att}}$ denotes the i-th feedback and updated attempt, respectively. Such a trajectory ends when the last feedback x_m^{fb} contains the phrase "the final response is correct". Therefore, *directly correct* trajectories take the form of $x_{\checkmark} = (x_0^{\text{att}}, x_1^{\text{fb}})$, and *self-improving* trajectories take the form of $x_{\text{SI}} = (x_0^{\text{att}}, x_1^{\text{fb}}, x_1^{\text{att}}, ..., x_m^{\text{fb}})$ where $m > 1$.

2.2 Interactive Trajectory Editing

In our prior study in Figure 1 and Table 1, we find that it is difficult to elicit a 7B model to perform self-improvement due to its significantly weaker math and reasoning capability compared to LLMs. To address this issue, we use the smaller model M_θ to first generate an initial attempt[2], and then apply a feedback module FBK and an improvement module IMP to *rewrite parts of the M_θ trajectories*. Specifically, we first use FBK (prompting text-davinci-003 or using a Python script) to generate a feedback $x_i^{\text{fb}*}$ based on the first error step it identified for each incorrect attempt. After that, we edit the trajectory by replacing the first feedback that M_θ and FBK disagree on with the FBK-generated feedback, creating an edited trajectory:

$$(x_0^{\text{att}}, ..., x_{i-1}^{\text{att}}, x_i^{\text{fb}*}).$$

Finally, we use our improvement module IMP (prompting Codex) to generate an improved attempt $x_i^{\text{att}*}$ conditioned on the previous x_{i-1}^{att} and feedback $x_i^{\text{fb}*}$, and append it to the trajectory:

$$x_{\text{edited}} = (x_0^{\text{att}}, ..., x_{i-1}^{\text{att}}, x_i^{\text{fb}*}, x_i^{\text{att}*}).$$

[2]We also allow M_θ to attempt generating feedbacks and improvements, as self-improvement training progresses.

As an example, if feedback $x_i^{\text{fb}*}$ identifies that the first mistake in x_{i-1}^{att} appears in step 3, then step 1-2 in x_{i-1}^{att} is kept untouched, and IMP is used to generate an improved solution by only changing steps $\geq$ 3. This design is to prevent IMP from re-writing the whole attempt from scratch (e.g., generating the gold solution), which would violate our motivation to create trajectories with feedback and improvements that are incremental and tailored to the small model's capability.

We repeat this process, up to a maximum number of iterations, until the last attempt in x_{edited} is correct. Otherwise, we discard x_{edited} that failed to reach the correct answer.

2.3 Data Post-processing

After the interactive trajectory editing step, we have three types of data: 1) gold step-by-step demonstrations x_{gold} for the task, 2) directly correct trajectories $x_{\checkmark}$ generated by M_θ, and 3) edited trajectories x_{edited} created using M_θ, FBK, and IMP.

To make training easier, we first split *all data* into triplets of *single-step improvement* $x_{\text{imp}} = (x_i^{\text{att}}, x_i^{\text{fb}}, x_{i+1}^{\text{att}})$ if an attempt x_i^{att} was incorrect, or into $x_{\text{T}} = (x^{\text{att}}, x^{\text{fb}})$ where the attempt is correct and the trajectory ends with x^{fb} containing the phrase "the final response is correct". To learn from expert's correction, x_j^{att} and x_j^{fb} may be the edited $x_j^{\text{att}*}$ and $x_j^{\text{fb}*}$, respectively (see Section 2.2). Next, we filter out some x_{imp} triplets that contain incorrect feedbacks or improvement steps using some rules (see more in Appendix I). Then, we combine x_{T} and filtered x_{imp} into a single dataset, and balance them using a hyperparameter p specifying the proportion of x_{imp}. We find that this parameter is important for the model to learn to improve its attempt *only when necessary*. This is because we found that training with too many x_{imp} can cause the model to attempt self-improvement even when the last attempt is already correct, thus damaging its performance (see Section 4.2 for more details).

2.4 Model Training

Finally, we use supervised learning (SL) to train a smaller model M_θ on the combined dataset. To promote the model to focus on learning the feedback and improvement steps in x_{imp}, we use a weighted cross-entropy loss. We weight the loss for all the tokens in x_{T} with $w = 1.0$, but with $w > 1.0$ for the tokens that belong to x_i^{fb} or x_{i+1}^{att} in single-step improvement triplets x_{imp}. We note that we also experimented with masking x_i^{att} (Zheng et al., 2023), but found it to be less effective than weighted SL in our case. See Appendix E for more empirical analysis and discussions on related techniques.

2.5 TRIPOST

In Figure 2 and Algorithm 1 we summarize our TRIPOST algorithm. For each of the t iterations, we first utilize M_θ to generate its own attempts X, and then use FBK and IMP to generate and create a set of edited trajectories as described in Section 2.2. Next, we process the newly collected trajectories and the gold task demonstrations X_{gold} by first splitting them into a unified format of x_{imp} triplet or x_{T}, and then filtering out erroneous x_{imp} data (Section 2.3). Finally, we create a training dataset $\mathcal{D}$ by balancing the number of x_{imp} and x_{T} using a hyperparameter p, and finetune M_θ on $\mathcal{D}$ using weighted SL. Unless otherwise specified, we repeat this procedure for $t = 3$ iterations, and refer to the model trained using TRIPOST with t iterations as TRIPOST(t).

Algorithm 1 TRIPOST Training Algorithm

Require: Generative language model M_θ
Require: FBK and IMP modules
Require: Gold task demonstrations X_{gold}
Require: Data buffer $\mathcal{B}$
for t iterations **do**
  // interactive trajectory editing
  Gen. trajectories $X = \{X_{\checkmark}, X_{\text{✗}}\}$ with M_θ
  Add correct trajectories $X_{\checkmark}$ to $\mathcal{B}$
  for each incorrect trajectory $x_{\text{✗}} \in X_{\text{✗}}$ **do**
    Use FBK to generate feedbacks $x_i^{\text{fb}*}$
    Replace feedback from $x_{\text{✗}}$ with $x_i^{\text{fb}*}$
    Prompt IMP to generate x_{i+i}^{att}
    Repeat until termination cond. reached
    Add edited trajectory x_{edited} to $\mathcal{B}$
  end for
  // data post-processing
  Split $X_{\text{gold}} \cup \mathcal{B}$ into triplets x_{imp} or x_{T}
  Filter x_{imp}
  $\mathcal{D} = \{x_{\text{imp}}, x_{\text{T}}\}$, balanced using p
  // model training
  Train M_θ on $\mathcal{D}$ using weighted SL
end for

3 Experiments

In this section, we test if our TRIPOST can 1) help distill self-improvement ability into a smaller model M_θ, and 2) help M_θ improve performance on math and reasoning tasks.

Dataset	Criterion	Example	*seen* subtask	*unseen* subtask
Multistep Arithmetic	nesting depth (d) and number of operands (l)	Q: ((2 * 2 + 1) + (3 * 1 - 1)) // $l = 3, d = 2$	$l = \{3, 4\} \times d = \{2\}$	$l = \{3, 4\} \times d = \{3\}$ and $l = \{5, 6\} \times d = \{2, 3\}$
Word Sorting	number of words to sort (l)	Q: orange apple banana pear // $l = 4$	$l = \{2, 3, ..., 7\}$	$l = \{8, 9, ..., 16\}$
Date Understanding	number of steps to solve (l)	Q: Today is 01/02, what's the date yesterday? // $l = 1$	$l = \{1, 2\}$	$l \geq 3$
Logical Deduction	number of options (l)	Q: John runs ... Who runs fastest? Options: (A).. (B).. (C).. // $l = 3$	$l = \{3, 5\}$	$l = \{7\}$

Table 2: Categorization of the datasets into seen and unseen tasks. *seen* tasks are chosen to be easier and are used for training. Example questions are abbreviated, for complete examples please refer to Appendix A.

3.1 Dataset and Preprocessing

We utilize the BIG-Bench (Srivastava et al., 2023) benchmark to evaluate our approach. BIG-Bench is a collection of more than 200 text-based tasks including categories such as traditional NLP, mathematics, commonsense reasoning, and more.

We perform experiments on four math and reasoning tasks from the challenging BIG-Bench Hard (Suzgun et al., 2022) collection. We consider two *scriptable* tasks: Multistep Arithmetic and Word Sorting, where a step-by-step solution (rationale) and a feedback can be generated using a script; and two *unscriptable* tasks: Date Understanding and Logical Deduction, where we prompt an LLM (Codex/text-davinci-003) to generate feedbacks. We prompt Codex as the IMP module for all tasks.

For each task, we first collect a set of gold step-by-step rationales by either scripting a solution for *scriptable* tasks, or using the CoT prompts from Suzgun et al. (2022) to generate a solution using LLMs. For those LLM-generated rationales, we only keep the correct ones (see Appendix A for more details) for training. Then, to better measure a model's generalization ability, we split each of the 4 tasks further into *seen* and *unseen* subtasks. We mainly categorize simpler questions as the *seen* subtasks to be used for model training. We describe our categorization method in Table 2.

3.2 Models and Baselines

Models We use LLaMA-7B as M_θ in our main experiments in Table 3. LLaMA (Touvron et al., 2023a) is a collection of foundation language models ranging from 7B to 65B that have shown strong performance compared to GPT-3 (175B) on many benchmarks (Zheng et al., 2023; Taori et al., 2023; Peng et al., 2023b). Due to the cost of training language models, we use the smallest 7B model. For results with LLaMA-2 models, see Appendix D. For training hyperparameters, see Appendix J.

Baselines We compare TRIPOST training with three baselines: fine-tuning using self-generated, self-consistent rationales (*LMSI*, Huang et al. (2023)); fine-tuning using only ground truth rationales (*ft rationale*); and fine-tuning using self-improvement demonstrations from LLMs (*ft SI. demo*, similar to Ye et al. (2023)). For better performance, we initialize with the model trained after *ft rationale* for all methods. Lastly, for a fair comparison, we restrict iterative algorithms such as TRIPOST to only have access to the same amount of input prompts as used to train baselines such as *ft rationale*. For more implementation details, see Appendix G and Appendix I.

3.3 Metrics

To measure task performance, we follow prior studies on Big-Bench (Ho et al., 2023; Huang et al., 2023) and report the accuracy of the final answer extracted from the model's output. For each task, we report the accuracy on the seen subtasks and unseen subtasks, and its overall performance. To measure the model's self-improvement ability, we mainly consider two metrics: 1) how often the model tries to self-improve (*SI. Freq.*), and 2) how much those of self-improvement attempts contribute to the model's task performance (*SI. Contrib.*). We measure *SI. Freq.* as the number of times the model attempted to self-improve divided by the size of the test set, and *SI. Contrib.* as the number of times those improvement attempts actually reached the correct final answer.

3.4 Main Results

Table 3 summarizes TRIPOST's evaluation results on the four datasets. First, we find *LMSI* (Huang et al., 2023) to be roughly on-par with *ft. rationale* only when the performance of the base model (i.e., *ft. rationale*) is already high on the training questions (the *seen* subtask). This is understandable, as *LMSI* was originally designed for LLM (e.g.,

	Method	Multistep Arithmetic†			Word Sorting†			Date Understanding			Logical Deduction		
		seen	unseen	total	seen	unseen	total	seen	unseen	total	seen	unseen	total
	LMSI	10.83	0.00	4.33	67.72	5.56	26.83	14.55	9.09	12.99	61.11	20.00	48.10
	ft rationale	39.75	1.48	16.78	73.49	5.82	28.50	33.35	21.21	29.87	62.69	8.67	45.78
	ft SI. demo	29.17	0.00	11.67	53.54	1.98	19.26	27.27	18.18	24.68	54.63	15.00	41.67
Ours	TRIPOST($t = 1$)	41.67	0.84	17.17	74.02	5.16	28.23	32.73	13.64	27.27	57.88	**22.00**	46.52
Ours	TRIPOST($t = 2$)	49.58	1.39	20.67	74.02	**7.14**	29.55	35.46	25.00	32.47	58.80	18.00	45.25
Ours	TRIPOST($t = 3$)	**52.50**	**2.50**	**22.50**	**77.17**	5.95	**29.82**	**40.00**	**29.55**	**37.01**	**63.89**	15.00	**48.42**

Table 3: Overall performance of TRIPOST on four BIG-Bench hard datasets. For each dataset, we train our models on the *seen* tasks, and evaluate their performance on both *seen* and *unseen* tasks. For all TRIPOST runs, we use the same hyperparameters (e.g., $p = 0.43$). Total accuracy (*total*) is accuracy weighted based on the number of test samples. † denotes that the task uses scripted rationale/feedback. Results are averaged over three runs.

Dataset	SI. Contrib.			Directly Correct	Total Acc.
	seen	unseen	total		
Multistep Arithmetic	1.39	0.28	1.67	20.83	22.50
Word Sorting	1.85	0.52	2.37	27.44	29.82
Date Understanding	1.95	1.29	3.25	33.76	37.01
Logical Deduction	8.23	0.63	8.86	39.56	48.52

Table 4: Analyzing how TRIPOST-trained models improved the overall task performance. Total accuracy is first decomposed into attempts that are directly correct (*Directly Correct*) and attempts with self-improvement (*SI. Contrib.*). *SI. Contrib.* is then further decomposed into its accuracy contribution on the seen and unseen subtasks.

PaLM-540B) to improve on tasks where it can already achieve a reasonable performance. Next, we find *ft SI. demo* to slightly degrade the model's performance across all tasks, which we believe is due to the capability mismatch between the LLM demonstrator and the small LM learner (Section 1). This forces the small LM to learn from "advanced" errors not from its own (Table 1 and Appendix B). Finally, we see that in all tasks, TRIPOST-trained models performs the best in all metrics. In general, we also observe improvement in the performance of TRIPOST-trained models as the number of iterations t increases.[3] We believe this is because, during the process of learning to self-improve, the model also learns to better understand the tasks by learning from its *own mistakes* (Zhang et al., 2023; Andrychowicz et al., 2018; Lightman et al., 2023). This enables the model to not only generate better initial attempts, but also improve its self-improvement ability.

In Table 4, we further explore the contribution of M_θ's self-improvement ability by describing how its overall performance improved. We find that in two out of the four datasets, TRIPOST-trained models generate an more accurate initial attempt than the baselines (denoted as *Directly Correct*), and in all cases, TRIPOST-trained models had measurable self-improvement contributions in both seen and unseen tasks (cf. Figure 1 and Table A4). This suggests that TRIPOST-training can 1) help the model better understand the tasks and generate better initial attempts, and 2) help distill self-improving ability into the model. We believe that the combination of both factors improve the model's overall performance in Table 3.

3.5 TRIPOST-auto

In Table 5, we explore another way of training M_θ with TRIPOST. Instead of re-balancing the training dataset using a fixed p as in Section 3.4, we can simply include all the edited improvement tuples x_{imp} and the directly correct attempts x_{T} generated by M_θ. We denote this method as TRIPOST-auto, as it automatically "balances" its training data to be proportional to its current performance, because p can be interpreted as how often the model's attempts were incorrect and needed editing. TRIPOST-auto training included no less x_{imp} compared to TRIPOST (but generally more x_{T}, resulting in $p < 0.43$), and we find that the model now rarely attempts to self-improve. However, this unexpectedly leads to even better overall performance, especially on *unscriptable* tasks. We believe this indicates that 1) learning to always generate a useful feedback and the corresponding improvement is

[3]For a comparison against LMSI with more than $t = 1$ iteration, please see Appendix H.

Method	Multistep Arithmetic†			Word Sorting†			Date Understanding			Logical Deduction		
	SI. Freq	SI. Cont.	total	SI. Freq	SI. Cont.	total	SI. Freq	SI. Cont.	total	SI. Freq	SI. Cont.	total
TRIPOST($t = 1$)	0.00	0.00	17.17	1.58	0.52	28.23	0.00	0.00	27.27	8.86	2.85	46.52
TRIPOST($t = 2$)	1.33	1.11	20.67	2.90	0.52	29.55	1.94	0.65	32.47	**29.72**	**11.39**	45.25
TRIPOST($t = 3$)	**3.67**	**1.67**	22.50	**4.38**	**2.37**	29.82	**10.38**	**3.25**	37.01	23.42	8.86	48.42
TRIPOST-auto($t = 1$)	0.00	0.00	20.00	0.00	0.00	30.34	0.00	0.00	32.47	1.90	0.63	51.27
TRIPOST-auto($t = 2$)	0.00	0.00	23.33	0.00	0.00	29.55	0.00	0.00	56.82	0.63	0.00	55.06
TRIPOST-auto($t = 3$)	0.00	0.00	**24.33**	0.00	0.00	**30.34**	0.00	0.00	**68.83**	0.63	0.63	**56.96**

Table 5: Overall performance of TRIPOST without explicit re-balancing. TRIPOST-auto uses the same training procedure as TRIPOST, except that the proportion of x_{imp} used for training is determined automatically using the model's current task performance.

Method	Multistep Arithmetic		Logical Deduction	
	SI. Contrib.	Total Acc.	SI. Contrib.	Total Acc.
TRIPOST	**1.67**	22.50	**8.86**	48.42
-interaction	0.28	11.67	0.00	41.67
-filtering	0.33	20.67	7.59	48.27
+auto-balance	0.00	**24.33**	0.63	**56.96**
-weighed SL	0.00	21.33	1.90	43.67

Table 6: TRIPOST ablation studies.

Dataset	p	Self-Improvement Freq.	Self-Improvement Contrib.	Total Acc.
Multistep Arithmetic	0.05	0.00	0.00	23.17
	0.20	0.00	0.00	24.33
	0.43	3.67	1.67	22.50
	0.56	8.61	2.50	20.00
	0.70	18.88	3.61	18.67
Logical Deduction	0.05	0.00	0.00	49.37
	0.20	0.63	0.00	52.63
	0.43	23.42	8.86	48.42
	0.56	20.25	7.59	45.57
	0.70	59.49	31.64	45.57

Table 7: Varying the proportion of x_{SI} used during TRIPOST training.

harder than learning to directly generate a correct attempt, and 2) using LLM-generated feedbacks, which covers more error cases than a Python script, is effective in improving a model's performance.

4 Analysis

To investigate the factors that can influence how TRIPOST-trained models learned to attempt self-improvement, we focus our analysis on the Multistep Arithmetic and Logical Deduction datatset. We also mainly study TRIPOST with $p = 0.43$, which has both a measurable self-improvement contribution and improvement in its task performance (see Table 3 and Table 4)[4].

[4]In practice, we implement p by specifying the ratio of the number of "self-improvement samples vs. directly correct samples vs. gold samples“. For example, a ratio of $1.5 : 1.0 : 1.0$ corresponds to $p = 0.43$.

4.1 Ablation Studies

We perform ablation studies for each of the three stages in TRIPOST to better understand their contribution to model's overall performance. In Table 6, we report the task accuracy when: interaction between M_θ and LLM is removed, so that M_θ is distilled with purely LLM demonstrations (*-interaction*); data filtering is removed (*-filtering*); dataset balancing is changed to using its own performance (*+auto-balance*); and the weights for SL are changed to be the same for all tokens (*-weighed SL*). We find that all components are important for TRIPOST to work well, and the choice of fixing p presents a trade-off between a model's self-improvement ability and its task performance (notibly, both TRIPOST and TRIPOST-auto improve upon the baselines).

4.2 Proportion of SI. Training Data

In Table 7, we investigate how much improvement demonstration (x_{imp}) is needed to elicit a measurable self-improvement contribution from M_θ. We find that when a large proportion (e.g. $p = 0.70$) of the training data contains x_{imp}, the model often *attempts* to self-improve but does not always result in an overall better performance. This is because many of the "improvement" attempts result in failures (e.g. changing an already correct attempt to become an incorrect one), and the best performance is achieved typically when p is low. Despite this, we find that for all other cases with $p \leq 0.43$, TRIPOST-trained model achieved a better performance than the baseline methods (see Table 4).

4.3 Number of TRIPOST Iterations

In most of our experiments, we trained TRIPOST up to $t = 3$ iterations. This is because we found that LLMs and our Python scripts start to struggle with generating feedback or improving M_θ attempts after three iterations. In Figure 3, we present

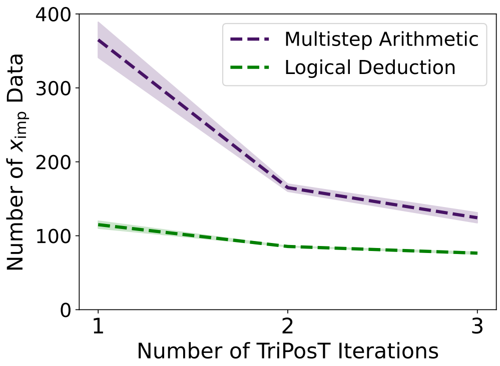


Figure 3: Improvement demonstrations become more difficult to collect as TRIPOST iteration increases.

how the number of self-improving trajectories collected (x_{imp}, after filtering) changes as TRIPOST iteration increases. We found that as M_θ improves its performance over time, it 1) poses a greater challenge for our FBK module to generate feedback and/or the IMP module to generate improvement, and 2) generates fewer incorrect attempts for TRIPOST to edit. This is especially impactful for Multistep Arithmetic, as our feedback scripts can only consider a fixed number of error types. This also shows that even LLMs can struggle at generating useful feedbacks or correct improvements, which supports our findings in Section 3.5 that learning to generate feedback and improvements may be harder than to directly generate a correct solution. Lastly, we note that TRIPOST can, in principle, be applied as an online RL algorithm, where one does not restrict the input prompts to be a fixed set as in Section 3. We believe this could be beneficial to improve the model's performance and genearlization ability beyond TRIPOST($t = 3$).

5 Related Work

Prompting LLMs to Self-Improve Recently, many work (Bai et al., 2022; Madaan et al., 2023) have discovered LLM's capability to self-improve by letting it revise its own answer after prompting it to generate feedbacks. Following these work, Yang et al. (2022); Peng et al. (2023a); Shinn et al. (2023); Schick et al. (2022); Yang et al. (2023) has utilized such a capability to improve LLM's performance on various tasks. For example, Yang et al. (2022) recursively prompts an LLM to generate a longer story, and Madaan et al. (2023) iteratively prompts an LLM to improve its answers on a wide range of tasks such as sentiment reversal and dialogue response generation. More generally, Yang et al. (2023) finds that LLMs can be prompted to act as an "optimization function", which can be used to automatically perform prompt engineering. Our work focuses on distilling the self-improvement ability of LLMs into a smaller model, which was initially not capable of self-improvement (Figure 1).

Training LMs to Self-Improve Besides prompting methods, recent work also explored approaches to train a LM to self-improve. LMSI (Huang et al., 2023) trains LMs (e.g., PaLM-540B) with self-generated, self-consistent answers to improve their task performance, yet we found such method ineffective for small LMs. Many work such as Paul et al. (2023); Welleck et al. (2022); Madaan et al. (2021); Yasunaga and Liang (2020); Du et al. (2022) considered using multiple small LMs to generate feedback and improvement, which also relates to model ensemble methods (Dietterich, 2000). For example, Welleck et al. (2022) trains a "corrector" to improve answers generated by a given fixed generator. This method gathers improved attempts by sampling from the generator and pairing high-scoring attempts with low-scoring ones. It also does not provide reasonings (e.g., feedbacks) for each improvement. Paul et al. (2023) first trains a feedback model by using a set of predefined rules that perturbs an original solution, and then trains a separate model to generate answers conditioned on the feedback. Our work leverages LLMs to train a single model capable of generating both feedback and improvement, and also does not require any predefined rules (e.g., using LLMs as the FBK module). Saunders et al. (2022); Ye et al. (2023) has attempted to equip a single small model to self-improve by training on LLM demonstrations, but found that it had little to no effect for small models on math/reasoning tasks. Our work presents analyses of how these previous methods can fail, and proposes TRIPOST that can train a small model to self-improve and achieve better task performance.

Knowledge Distillation Learning from experts' demonstrations or reasoning (e.g., from GPT-4) has shown to be successful at improving the performance of smaller models in various tasks (Mukherjee et al., 2023; Laskin et al., 2022; Peng et al., 2023b; Ho et al., 2023; Ye et al., 2023; Huang et al., 2023; Jung et al., 2024). Distillation methods (Hinton et al., 2015; Ba and Caruana, 2014) generally train a target model using expert demonstrations unaware of the target model's capability. While TRI-

POST also use LLMs to demonstrate generating a feedback or an improvement, these demonstrations are always conditioned on the output of the smaller model. In this view, our approach combines merits from reinforcement learning with knowledge distillation techniques, where small models are distilled with demonstrations that are created by its own exploration augmented by LLMs' supervision.

6 Conclusion

We introduce TRIPOST, a training algorithm that distills the ability to self-improve to a small model and help it achieve better task performance. TRIPOST first creates improving trajectories using interactions between a smaller model and an LLM, then post-process the collected trajectories, and finally train the smaller model to self-improve using weighted SL. We evaluated TRIPOST on four math and reasoning tasks from the Big-Bench Hard collection and found that it can help small models achieve better task performance. In our analysis, we find that 1) the interactive process of learning from and correcting its *own* mistakes is crucial for small models to learn to self-improve and 2) learning to always generate a useful feedback and a corresponding improvement can be much harder than learning to directly generate a correct answer. These findings suggest that other data formats, beyond the traditional (input, answer) pair, could be better suited for training a language model to solve a downstream task. We believe this also opens new possibilities for future work to leverage LLMs to improve the performance of smaller, faster models.

7 Limitations

Model Sizes In all of our experiments, we used a single A100 and mainly tested TRIPOST on 7B models, the smallest in the LLaMA-1 and LLaMA-2 family (Touvron et al., 2023a,b). However, with the recently introduced flash attention technique (Dao et al., 2022; Dao, 2023) which can be used to reduce memory usage during training, we plan to extend our experiments to use models with more than 7B parameters.

Datasets We focused our experiments on math and reasoning tasks because 1) prior work (Ye et al., 2023) had found it difficult to train a 7-13B to self-improve on those tasks and 2) measuring performance improvement is more well defined (for example, as compared to creative story writing). However, we note that as TRIPOST is task agnostic, in theory it can be applied to other tasks such as knowledge-grounded dialogue generation (Yoshino et al., 2023) or dialogue safety (Dinan et al., 2019). We intend to leave this for future work.

LLM Usage While attempts for some tasks can be parsed and evaluated using a Python script (e.g., multistep arithmetic and word sorting), it quickly becomes unmanageable for tasks where reasonings mostly take the form of free text (e.g., date understanding and logical deduction). Therefore, we use LLMs such as GPT-3 and Codex (and ChatGPT, see Appendix F), which are highly performant at a reasonable cost. Specifically, we mainly use text-davinci-003 as the feedback module and Codex as the improvement module, as we found this to be the most cost-performant configuration in our experiments.

However, since the ability of LLMs to generate feedback or improvements is *crucial* for TRIPOST to collect training data, this presents a trade-off between the cost of using more performant LLMs (e.g., GPT-4) and the training outcome of TRIPOST, for example on harder tasks such as GSM8k (Cobbe et al., 2021). We hope that with advances in making LLMs more available (Zhang et al., 2022a), such a trade-off would diminish.

8 Ethical Considerations

Our work describes an algorithm to improve small models' performance on math and reasoning tasks, by distilling them the ability to self-improve using interaction records with LLMs. Generally, while most algorithms are not designed for unethical usage, there is often potential for abuse in their applications. In our experiments, we apply TRIPOST to four math and reasoning tasks from the Big-Bench Hard collection (Suzgun et al., 2022). However, because training algorithms are typically task-agnostic, it is possible to use them for unethical tasks, such as scamming and generating harmful responses (Welbl et al., 2021; Gehman et al., 2020). We do not condone the use of TRIPOST for any unlawful or morally unjust purposes.

References

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. 2018. Hindsight experience replay.

Lei Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. 2022. Constitutional ai: Harmlessness from ai feedback.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Tri Dao. 2023. FlashAttention-2: Faster attention with better parallelism and work partitioning. *ArXiv*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Thomas G. Dietterich. 2000. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*.

Emily Dinan, Samuel Humeau, Bharath Chintagunta, and Jason Weston. 2019. Build it break it fix it for dialogue safety: Robustness from adversarial human attack.

Wanyu Du, Zae Myung Kim, Vipul Raheja, Dhruv Kumar, and Dongyeop Kang. 2022. Read, revise, repeat: A system demonstration for human-in-the-loop iterative text revision. In *Proceedings of the First Workshop on Intelligent and Interactive Writing Assistants (In2Writing 2022)*, pages 96–108, Dublin, Ireland. Association for Computational Linguistics.

Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. 2020. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3356–3369, Online. Association for Computational Linguistics.

Zhiyuan He, Danchen Lin, Thomas Lau, and Mike Wu. 2019. Gradient boosting machine: A survey.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network.

Namgyu Ho, Laura Schmid, and Se-Young Yun. 2023. Large language models are reasoning teachers.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1051–1068, Singapore. Association for Computational Linguistics.

Jaehun Jung, Peter West, Liwei Jiang, Faeze Brahman, Ximing Lu, Jillian Fisher, Taylor Sorensen, and Yejin Choi. 2024. Impossible distillation: from low-quality model to high-quality dataset & model for summarization and paraphrasing.

Sekitoshi Kanai, Shin'ya Yamaguchi, Masanori Yamada, Hiroshi Takahashi, Kentaro Ohno, and Yasutoshi Ida. 2023. One-vs-the-rest loss to focus on important samples in adversarial training.

Angelos Katharopoulos and François Fleuret. 2019. Not all samples are created equal: Deep learning with importance sampling.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, Maxime Gazeau, Himanshu Sahni, Satinder Singh, and Volodymyr Mnih. 2022. In-context reinforcement learning with algorithm distillation.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Aman Madaan, Niket Tandon, Dheeraj Rajagopal, Peter Clark, Yiming Yang, and Eduard Hovy. 2021. Think about it! improving defeasible reasoning by first modeling the question scenario.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive learning from complex explanation traces of gpt-4.

OpenAI. 2022. OpenAI: Introducing ChatGPT.

OpenAI. 2023. GPT-4 technical report.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. 2023. Refiner: Reasoning feedback on intermediate representations.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. 2023a. Check your facts and try again: Improving large language models with external knowledge and automated feedback.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023b. Instruction tuning with gpt-4.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA. Association for Computing Machinery.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators.

Robert E. Schapire. 1999. A brief introduction to boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, page 1401–1406, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized experience replay.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. 2022. Peer: A collaborative language model.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, and et al. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models.

Qizhou Wang, Feng Liu, Bo Han, Tongliang Liu, Chen Gong, Gang Niu, Mingyuan Zhou, and Masashi Sugiyama. 2022. Probabilistic margins for instance reweighting in adversarial training.

Yisen Wang, Difan Zou, Jinfeng Yi, James Bailey, Xingjun Ma, and Quanquan Gu. 2020. Improving adversarial robustness requires revisiting misclassified examples. In *International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Johannes Welbl, Amelia Glaese, Jonathan Uesato, Sumanth Dathathri, John Mellor, Lisa Anne Hendricks, Kirsty Anderson, Pushmeet Kohli, Ben Coppin, and Po-Sen Huang. 2021. Challenges in detoxifying language models. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2447–2469, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers.

Kevin Yang, Yuandong Tian, Nanyun Peng, and Dan Klein. 2022. Re3: Generating longer stories with recursive reprompting and revision.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback.

Seonghyeon Ye, Yongrae Jo, Doyoung Kim, Sungdong Kim, Hyeonbin Hwang, and Minjoon Seo. 2023. Selfee: Iterative self-revising llm empowered by self-feedback generation. Blog post.

Koichiro Yoshino, Yun-Nung Chen, Paul Crook, Satwik Kottur, Jinchao Li, Behnam Hedayatnia, Seungwhan Moon, Zhengcong Fei, Zekang Li, Jinchao Zhang, Yang Feng, Jie Zhou, Seokhwan Kim, Yang Liu, Di Jin, Alexandros Papangelis, Karthik Gopalakrishnan, Dilek Hakkani-Tur, Babak Damavandi, Alborz Geramifard, Chiori Hori, Ankit Shah, Chen Zhang, Haizhou Li, João Sedoc, Luis F. D'Haro, Rafael Banchs, and Alexander Rudnicky. 2023. Overview of the tenth dialog system technology challenge: Dstc10. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, pages 1–14.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022a. Opt: Open pre-trained transformer language models.

Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E. Gonzalez. 2023. The wisdom of hindsight makes language models better instruction followers.

Zhisong Zhang, Emma Strubell, and Eduard Hovy. 2022b. A survey of active learning for natural language processing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 6166–6190, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena.

A More Details on Datasets and Preprocessing

We use four tasks from the Big-Bench Hard collection (Suzgun et al., 2022) for our experiments: *multistep arithmetic*, *word sorting*, *date understanding*, and *logical deduction*. Since these tasks do not provide ground truth step-by-step rationale, we either generate them using a script (for *multistep arithmetic* and *word sorting*), or prompt Codex (Chen et al., 2021) in a few-shot setting using examples from Suzgun et al. (2022). For rationales generated using prompting, we only keep the ones that reached the correct answer and passed a simple consistency check (e.g. for multiple choice questions, we ensure that the final selected choice in the last step appeared in the second last step). We provide example rationales used for each task in Table A8, Table A9, Table A10, and Table A11. Since Big-Bench (Srivastava et al., 2023) did not provide an official training/validation/test split, we generated our own splits with statistics shown in Table A1.

Dataset	Train	Validation	Test
Multistep Arithmetics	550	50	300
Word Sorting	433	40	379
Date Understanding	191	20	87
Logical Deduction	360	40	158

Table A1: Number of training, validation, and test samples used for the four tasks from the Big-Bench Hard collection (Suzgun et al., 2022).

B Analyzing Errors Made by Codex and LLaMA-7B

To detail the different type and amount of errors made by an LLM (e.g., Codex) and a smaller model (e.g., LLaMA-7B), we manually examine incorrect attempts generated by the two models in the Multistep Arithmetics dataset. We use Codex with few-shot prompting, and LLaMA-7B after supervised finetuning on ground-truth step-by-step solutions (denoted as *LLaMA+ft*). We randomly sample 50 generated attempts with incorrect answers, and carefully review each step in those attempts. For each incorrect step, we apply the principle of error-carried-forward and categorize the first error encountered according to Table A2.

We present our analysis in Figure A1 and Table A3. Figure A1 shows that calculation errors take up more than 50% of the time for both Codex and the finetuned LLaMA-7B. However, Codex also makes many algebriac errors (such as forgetting to change sign after adding brackets), while LLaMA-7B often hallucinates by adding or deleting terms from previous calculations. Furthermore, Table A3 shows that, compared to the fine-tuned LLaMA-7B, Codex generates longer solutions while producing fewer errors per step. These findings suggest that supervised finetuning a smaller LM (e.g., LLaMA-7B) based on correcting LLM-generated errors may be inefficient, as it forces the smaller model to learn from attempts and mistakes very different from its own (see Section 1 and Appendix C for more details).

C More Details on the Prior Study

In the prior study mentioned in Section 1, we experimented with distilling a smaller model (e.g. LLaMA-7B) with self-improvement demonstration using just the LLMs. We found that not only can the smaller model *not* self-improve by few-shot prompting, they also still fail to do so after training on the LLM self-improvement demonstrations (also discussed in Section 1). In Figure 1 we presented the performance gap between prompting Codex (175B) and finetuning/prompting LLaMA (7B) with self-improvement demonstrations, and in Table A4 we show the detailed numerical results.

D Additional Results on LLaMA-2

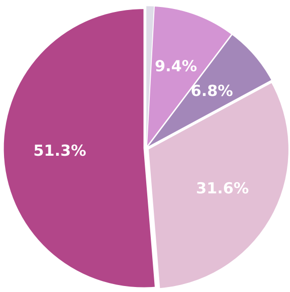
In Table A5 we present the results of using the LLaMA-2 7B model (Touvron et al., 2023b) for TRIPOST training. We used the same procedure as testing with the LLaMA-1 model in our main experiments (Section 3), except that we used $p = 0.26$ across all settings with LLaMA-2 instead of $p = 0.43$. This is because we found that the LLaMA-2 baseline (*ft rationale*) achieves almost twice the performance compared to its LLaMA-1 counterpart. As the LLaMA-2 models make fewer mistakes, we decrease p accordingly to prevent TRIPOST from terminating early due to lack of data. In general, Table A5 shows a similar trend as discussed in Section 3 that 1) fine-tuning on LLM demonstrations of self-improvement did not help improve math/reasoning task performance, and 2) TRIPOST can further improve upon the baselines.

E Effect of Weighted SL

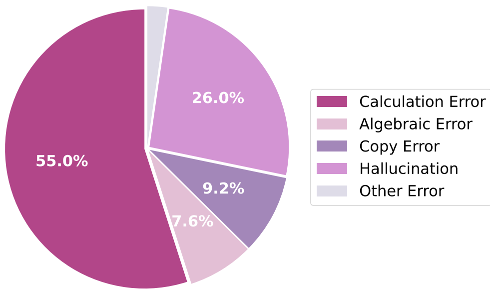
Besides balancing the training dataset, we also found it important to use a weighted cross-entropy

Error Name	Definition	Example
Calculation Error	errors in performing basic arithmetic operations (addition, subtraction, multiplication, division)	$2+3=7$
Algebraic Error	errors in algebraic manipulation, such as forgetting to change signs when adding brackets or forgetting the correct order of operations	$1-2+3=1-(2+3)$
Copy Error	mis-copying an operand or an operator from previous steps	$7+1+(...)=7-1+(...)$
Hallucation	adding or deleting an operand or an operator from previous steps	$7+(...)=7-1+(...)$
Other Error	errors that do not fall into the above categories	

Table A2: Categorization of errors commonly made by Codex or LLaMA-7B in the Multistep Arithmetics dataset.



(a) Codex



(b) LLaMA+ft (7B)

Figure A1: LMs of different sizes make different types of errors. In the Multistep Arithmetics dataset, more than half of the errors made by Codex or a finetuned LLaMA-7B belong to *Calculation Error*. However, the second most common error is *Arithmetic Error* for Codex, and *Copy Error* for LLaMA-7B.

	Codex	LLaMA+ft (7B)
Avg. Char per Question	113.8	102.4
Avg. Char per Attempt	920.0	650.1
Percent Steps with Errors	31.7	35.1

Table A3: LMs of different sizes make different amount of errors. In the Multistep Arithmetics dataset, Codex makes less errors per step compared to a finetuned LLaMA-7B, while answering longer questions and generating longer solutions.

Dataset	Method	SI. Contrib.	Total Acc.
MS.A.	Codex (175B)	-	31.33
	+ SI. prompting	2.00	33.33 ↑
	LLaMA+ft (7B)	-	16.78
	+ SI. prompting	0.00	11.60 ↓
	+ ft SI. demo	0.28	11.67 ↓
L.D.	Codex (175B)	-	81.01
	+ SI. prompting	4.43	85.44 ↑
	LLaMA+ft (7B)	-	45.78
	+ SI. prompting	0.00	43.67 ↓
	+ ft SI. demo	0.00	41.67 ↓

Table A4: Compared to LLMs, smaller models have difficulty performing self-improvement (*SI.*) on mathematical/logical tasks, such as Multistep Arithmetics (*MS.A.*) and Logical Deduction (*L.D.*).

loss to emphasize learning the improvement-related tokens (x_i^{fb} or x_{i+1}^{att}) of each training sample. In Table A6, we find that using a weight too low ($w = 1.0$) can result in the model rarely attempting to self-improve, while using a weight too high ($w = 3.0$) does not result in better performance. We believe that this has a similar effect of adjusting p in Section 4.2: some incentive is needed for the model to learn to self-improve, while too much emphasis on trying to self-improve can result in a worse performance.

While we also experimented with alternatives such as masking easier tokens (x_i^{att} in a single-step improvement triplet), we believe there is a rich set of techniques that can be used to train the model to focus on harder inputs. This includes boosting algorithms (Schapire, 1999; He et al., 2019), automatic loss reweighing methods (Kanai et al., 2023; Wang et al., 2022, 2020), as well as importance-sampling based methods (Katharopoulos and Fleuret, 2019). We leave this for future work as it is orthogonal to our main contributions.

F Prompting Details

Besides prompting to generate rationales (e.g. for *date understanding*), we also use prompting to gen-

	Method	Multistep Arithmetics†			Logical Deduction		
		seen	unseen	total	seen	unseen	total
LLaMA-1 (7B)	ft rationale	38.75	1.48	16.78	62.69	8.67	45.78
	ft SI. demo	29.17	0.00	11.67	54.63	15.00	41.67
	TriPosT($t = 1$)	41.67	0.84	17.17	57.88	**22.00**	46.52
	TriPosT($t = 2$)	49.58	1.39	20.67	58.80	18.00	45.25
	TriPosT($t = 3$)	**52.50**	**2.50**	**22.50**	**63.89**	15.00	**48.42**
LLaMA-2 (7B)	ft rationale	72.50	5.00	32.00	**87.04**	34.00	70.25
	ft SI. demo	51.67	2.22	22.00	80.56	42.00	68.35
	TriPosT($t = 1$)	71.67	3.89	31.00	83.33	**52.00**	**73.42**
	TriPosT($t = 2$)	**75.00**	**6.11**	**33.67**	83.33	48.00	72.15
	TriPosT($t = 3$)	72.22	5.19	32.00	71.67	50.00	72.78

Table A5: Using TriPosT with LLaMA-2 7B model. Overall, LLaMA-2 performs better than its LLaMA-1 counterpart, and TriPosT further improves LLaMA-2's task performance.

Dataset	w	Self-Improvement Freq.	Self-Improvement Contrib.	Total Acc.
Multistep Arithmetic	1.0	0.00	0.00	21.33
	1.5	3.67	1.67	22.50
	3.0	3.33	1.38	22.00
Logical Deduction	1.0	10.13	1.90	43.67
	1.5	23.42	8.86	48.42
	3.0	19.62	9.49	46.84

Table A6: Varying the SL weights w used during TriPosT training.

erate feedbacks and improvements given the initial attempt. For scriptable tasks such as *multistep arithmetic* and *word sorting*, we use a script to generate the feedback by first parsing each step in the attempt, and check their correctness/consistency with other steps using a set of predefined rules. This is similar to Welleck et al. (2022), but we also generalize this to unscriptable tasks such as *date understanding* and *logical deduction* by few-shot prompting GPT-3 (text-davinci-003) (Brown et al., 2020) and Codex (Chen et al., 2021) to generate feedbacks and improvements. We found that being able to generate useful feedback is critical for gathering successful improvement trajectories, and we discovered that ChatGPT (OpenAI, 2022) is less effective than GPT-3 or Codex in our case. We provide examples of the feedbacks generated for each task in Table A12, and the prompts used to generate feedback or improvements in Table A13, Table A14, Table A15, and Table A16. Note that we used a form-type of prompting for generating feedback because it can more easily ensure that our (formatted) feedback will contain all the elements we need.

When an answer is correct, we manually attach the phrase "Step 1 to step x is correct, and the final response is also correct." as the termination feedback, where "x" is the last step number. This termination condition is also used during inference.

G More Details on Baselines

LMSI Huang et al. (2023) proposed LMSI, a method to improve PaLM-540B (Chowdhery et al., 2022) on math and reasoning tasks by training it on self-generated and consistent step-by-step rationales. First, LMSI generates multiple step-by-step solutions using a high temperature ($\tau = 1.2$). Then, LMSI only keeps the answers that are self-consistent (by majority voting) in the final answer. Finally, LMSI further augments these solutions with mixed formats, such as removing all the intermediate steps and only keep the final answer. To be comparable with other methods in Table 3 that have access to the ground truth answer, we modify the second step to only keep the answers that are correct. In addition, since small models such as LLaMA-7B performed poorly in these tasks without fine-tuning, we perform LMSI after training the model on the collected silver step-by-step solutions in Appendix A.

ft. SI demo Following Ye et al. (2023), *ft. SI demo* finetunes a model on LLM-generated self-improvement demonstrations. For all tasks, we experimented with LLMs $\in$ {ChatGPT, Codex} and reported one with better performance (often Codex). In details, we first prompt a LLM (e.g. Codex) to generate an initial attempt, and then reused TriPosT with the same LLM as the FBK and IMP to generate a feedback and an improvement. For a fair comparison in Table 3, we also balanced the collected data using the same $p = 0.43$ as with TRIPOST. Finally, train the small LM using (unweighted) SL on the collected data.

H Running LMSI($t > 1$)

LMSI described in (Huang et al., 2023) was not applied as an iterative algorithm. However, since LMSI training only relies on self-generated and self-consistent answers, it can be *ran iteratively* similar to TRIPOST. We present this comparison in Table A7, and find that LMSI($t \geq 1$) struggles when the base model (*ft rationale*) has a weak task performance. We believe this is because LMSI is mainly a self-training algorithm designed for LLMs such as PaLM-540B (Chowdhery et al., 2022), which can often generate correct or near-correct solutions. However, TRIPOST is a training algorithm designed for smaller LMs, where models learns to self-improve from its interaction records with expert LLMs.

Method	Multistep Arithmetic	Date Understanding
ft rationale	16.78	29.87
LMSI($t = 1$)	4.33	12.99
LMSI($t = 2$)	2.50	11.69
TRIPOST($t = 1$)	17.17	27.27
TRIPOST($t = 2$)	**20.67**	**37.01**

Table A7: Comparing TRIPOST($t > 1$) with LMSI($t > 1$). For simplicity, we show total accuracy for each task.

I Implementation Details

We combine techniques from prompting-based self-improvement (Madaan et al., 2023; Bai et al., 2022) and active learning (Zhang et al., 2022b; Lightman et al., 2023) to collect a set of self-improving trajectories. Specifically, we first either use a script or few-shot prompting (see Appendix F for more details) to gather *feedbacks* on a given attempt, and then use prompting to generate *improvements* conditioned on the previous attempt, the feedback, and all the steps in the previous attempt before the first error step (see Tables A13 to A16 for example). This is to ensure that the improved attempt is making modifications on the previous attempt, rather than creating an entirely new attempt.

To edit the original attempt given the script/LLM-generated feedback, we 1) find the first $x_i^{\text{fb}*}$ feedback that differs from the M_θ-generated feedback x_i^{fb} (usually $i = 1$); 2) replace $x_i^{\text{fb}*}$ with x_i^{fb}; 3) remove all the attempts, feedback, and improvement after after x_i^{fb} from the trajectory. After this, we prompt an LLM in the improvement module IMP to generate an improvement as described above and in Appendix F.

To filter out some of the unhelpful feedbacks or incorrectly "improved" attempts, we mainly check 1) whether the final attempt reached the correct answer; 2) if there is at least one difference between the previous attempt and the improved attempt; and 3) if the final answer is consistent with the second last step. We only keep the data that have passed all checks. The effect of this filtering is discussed in our ablation studies in Section 4.1.

J Model/Training hyperparameters

In our main experiments, we used a single A100 GPU with DeepSpeed (Rasley et al., 2020) Zero-2 optimization. We used AdamW (Loshchilov and Hutter, 2019) as the optimizer. For each iteration of the TRIPOST algorithm, we train the model for 4 epochs (line 17). We use a linear learning rate schedule with 20% of warmup steps and a peak learning rate of 1e-6. We use a maximum sequence length of 1024 tokens, batch size = 1 and gradient accumulation step = 4. On average, three iterations of TRIPOST take about 12 hours to train.

Example Question and Generated Rationale for Multistep Arithmetic
_Q: ((7 - -1 + -1 + -4) - (-7 + 7 * 3 * -9)) =_ _Answer: Let's think step by step. Recall that the order of operations in mathematics is as follows: (1) Parentheses, (2) exponents, (3) multiplication and division (from left to right), (4) addition and multiplication (from left to right). So, remember to always compute the expressions inside parentheses or brackets first._ (1) This equation can be written as "(A - B)", where A = (7 - -1 + -1 + -4) and B = (-7 + 7 * 3 * -9). (2) Let's calculate A = (7 - -1 + -1 + -4) = (((7 - -1) + -1) + -4) = ((8 + -1) + -4) = (7 + -4) = 3. (3) Let's calculate B = (-7 + 7 * 3 * -9) = (-7 + ((7 * 3) * -9)) = (-7 + (21 * -9)) = (-7 + -189) = -196. (4) Then, the final equation is (A - B) = (3 - -196) = 199. (Final response) So the answer is 199.

Table A8: Examples of generated rationales for Multistep Arithmetic using Python scripts. Question and prompts are written in _gray italics_, and the generated step-by-step rationale is shown in black.

Example Question and Generated Rationale for Word Sorting
Q: Sort the following words alphabetically: _List: sioux fortescue purloin percept helmsman_ _Attempt: Let's think step by step._ (1) The first letter: "sioux"="s", "fortescue"="f" (6), "purloin"="p" (16), "percept"="p" (16), "helmsman"="h" (8) (2) We now have: (6) "fortescue" < (8) "helmsman" < (16) ["purloin" ? "percept"] < (19) "sioux". (3) Now sort this subpart ["purloin" ? "percept"] by looking at their second letters: "purloin"="u" (21), "percept"="e" (5). (3.1) We now have: (5) "percept" < (21) "purloin". (3.2) Hence, we have "percept" < "purloin". (4) Hence, we have "fortescue" < "helmsman" < "percept" < "purloin" < "sioux". (Final response) So the answer is: fortescue helmsman percept purloin sioux.

Table A9: Examples of generated rationales for Word Sorting using Python scripts. Question and prompts are written in _gray italics_, and the generated step-by-step rationale is shown in black.

Example Question and Generated Rationale for Date Understanding
Q: Jane scheduled 3 appointments with 5 poeple for tomorrow (Tue, 7/9/1972). _What is the date one year ago from today in MM/DD/YYYY?_ _Options:_ _(A) 07/01/1971_ _(B) 07/08/1971_ _(C) 07/15/1971_ _(D) 07/07/1971_ _(E) 07/09/1971_ _(F) 07/08/1910_ _Attempt: Let's think step by step._ (1) If Jane scheduled 3 appointments with 5 people for tomorrow (Tuesday, 7/9/1972), then today's date is Monday, 7/8/1972. (2) The date one year ago from today is 7/8/1971. (Final response) So the answer is (B).

Table A10: Examples of generated rationales for Date Understanding by prompting Codex (Chen et al., 2021). Question and prompts are written in _gray italics_, and the generated step-by-step rationale is shown in black.

Example Question and Generated Rationale for Logical Deduction
Q: The following paragraphs each describe a set of three objects arranged in a fixed order. The statements are logically consistent within each paragraph. On a shelf, there are three books: a white book, a green book, and an orange book. The green book is to the right of the white book. The orange book is the rightmost. *Options:* *(A) The white book is the leftmost.* *(B) The green book is the leftmost.* *(C) The orange book is the leftmost.* *Attempt: Let's think step by step. Let "??" represent 0 or more objects, and "?" represent exactly 1 object.* (1) The green book is to the right of the white book: "(left) ?? white ?? green ?? (right)". (2) The orange book is the rightmost: "(left) ?? orange (right)". (3) There are in total three books: a white book, a green book, and an orange book. (4) Combining (1) and (2) we get the following ordering: "(left) ?? white ?? green ?? orange (right)". (5) Combining (3) and (4) we get the following ordering: "(left) white green orange (right)". (6) According to this ordering, the leftmost book is the white book. (Final response) So the answer is (A).

Table A11: Examples of generated rationales for Logical Deduction by prompting Codex (Chen et al., 2021). Question and prompts are written in *gray italics*, and the generated step-by-step rationale is shown in black.

Task	Example
Multistep Arithmetic†	Q: ((-8 + -2 * 1 * 3) * (-2 * -1 * 5 - 2)) = Answer: Let's think step by step. Recall that the order of operations in mathematics ... (1) This equation can be written as "(A * B)", where A = (-8 + -2 * 1 * 3) and B = ... (2) Let's calculate A = (-8 + -2 * 1 * 3) = ... = -14. (3) Let's calculate B = (-2 * -1 * 5 - 2) = ... = (2 * 5 - 2) = 10. Then, the final equation is (A * B) = (-14 * 10) = -140. (Final response) So the answer is -140. Feedback: Step (1) to (2) are correct. In step (3) the part " (2 * 5 - 2) = 10 " is incorrect. This is because there is a calculation error, since (2 * 5 - 2) is not equal to 10.
Word Sorting†	Q: Sort the following words alphabetically: List: cole estonia brian redneck coda Answer: Let's think step by step. (1) The first letter: "cole"="c" (3), "estonia"="e" (5), ..., "coda"="c" (3). (2) We now have: (2) "brian" < (3) ["cole" ? "coda"] < (5) "estonia" < (18) "redneck". (3) Now sort this subpart ["cole" ? "coda"] by looking at their second letters: "cole"="o" (15), "coda"="o" (15). (3.1) We now have: (15) ["cole" ? "coda"]. (3.2) Now sort this subpart ["cole" ? "coda"] by looking at their third letters: "cole"="l" (12), "coda"="d" (4). (3.2.1) We now have: (12) "cole" < (4) "coda". ... (Final response) So the answer is: cole coda estonia brian redneck. Feedback: Step (1) to (3.2) are correct. In step (3.2.1) the part " (12) "cole" < (4) "coda" " is incorrect. This is because words are not sorted in ascending order. Using results from step (3.2), the first and smallest number should be 4.
Date Understanding	Q: Jane thinks today is 6/18/2019, but John thinks today is 6/19/2019. Jane is correct. What is the date one week from today in MM/DD/YYYY? Options: (A) 04/21/2019 (B) 06/25/2019 (C) 06/26/2019 (D) 06/24/2019 (E) 05/28/2019 (F) 07/04/2019 Answer: Let's think step by step. (1) If Jane thinks today is 6/18/2019, but John thinks today is 6/19/2019, then today's date is 6/19/2019. (2) The date one week from today is 6/26/2019, that is 06/26/2019. (Final response) So the answer is (D). Feedback: In step (1) the part "today's date is 6/19/2019" is incorrect. This is because Jane is correct, so today's date should be 6/18/2019.
Logical Deduction	Q: The following paragraphs each describe a set of three objects arranged in a fixed order. The statements are logically consistent within each paragraph. In an antique car show, there are three vehicles: a motorcycle, a bus, and a tractor. The motorcycle is the oldest. The bus is newer than the tractor. Options: (A) The motorcycle is the newest. (B) The bus is the newest. (C) The tractor is the newest. Answer: Let's think step by step. Let "??" represent 0 or more objects, and "?" represent exactly 1 object. (1) The motorcycle is the oldest: "(oldest) motorcycle ?? (newest)". (2) The bus is newer than the tractor: "(newest) bus ?? tractor ?? (oldest)". (3) There are in total three vehicles: a motorcycle, a bus, and a tractor. (4) Combining (1) and (2) we get the following ordering: "(newest) bus ?? tractor ?? (oldest)" (5) Combining (3) and (4) we get the following ordering: "(oldest) motorcycle bus (newest) tractor". (6) According to this ordering, the vehicle that is the newest is the tractor. (Final response) So the answer is (C). Feedback: Step (1) to (2) are correct. In step (3) the part "(newest) bus ?? tractor ?? (oldest)" is incorrect. This is because it is missing the motorcycle from step (1).

Table A12: Examples of an incorrect attempt generated by a small model, followed by its feedback generated by either an LLM (Codex or text-davinci-003) or a script (indicated by †) for each task. Input question is written in black, a generated attempt is written in blue, and its feedback is written in cyan.

Prompt Type	Prompt Example
Get Improvement	Q: (((-9 * 3 + -6) * (-2 + 7 * -6)) - ((-4 - -6 + 6) * (4 * -4 + 8))) = Answer: Let's think step by step. Recall that the order of operations in mathematics is as follows: (1) Parentheses, (2) exponents, (3) multiplication and division (from left to right), (4) addition and multiplication (from left to right). So, remember to always compute the expressions inside parentheses or brackets first. (1) This equation can be written as "((A * B) - (C * D))", where A = (-9 * 3 + -6), B = (-2 + 7 * -6), C = (-4 - -6 + 6) and D = (4 * -4 + 8). (2) Let's calculate A = (-9 * 3 + -6) = ((-9 * 3) + -6) = (-27 + -6) = -33. (3) Let's calculate B = (-2 + 7 * -6) = (-2 + (7 * -6)) = (-2 + -42) = -44. (4) Let's calculate C = (-4 - -6 + 6) = ((-4 - -6) + 6) = (2 + 6) = 8. (5) Let's calculate D = (4 * -4 + 8) = ((4 * -4) + 8) = (-16 + 8) = -8. (6) Then, the final equation is ((A * B) - (C * D)) = ((-33 * -44) - (8 * -8)) = (1424 - (8 * -8)) = (1424 - 64) = 1360. (Final response) So the answer is 1360. Feedback: Step (1) to step (5) are correct. In step (6) the part "((-33 * -44) - (8 * -8)) = (1424 - (8 * -8))" is incorrect. This is because there is a calculation error, since (-33 * -44) should be 1452. Updated Answer: Let's think step by step. Recall that the order of operations in mathematics is as follows: (1) Parentheses, (2) exponents, (3) multiplication and division (from left to right), (4) addition and multiplication (from left to right). So, remember to always compute the expressions inside parentheses or brackets first. (1) This equation can be written as "((A * B) - (C * D))", where A = (-9 * 3 + -6), B = (-2 + 7 * -6), C = (-4 - -6 + 6) and D = (4 * -4 + 8). (2) Let's calculate A = (-9 * 3 + -6) = ((-9 * 3) + -6) = (-27 + -6) = -33. (3) Let's calculate B = (-2 + 7 * -6) = (-2 + (7 * -6)) = (-2 + -42) = -44. (4) Let's calculate C = (-4 - -6 + 6) = ((-4 - -6) + 6) = (2 + 6) = 8. (5) Let's calculate D = (4 * -4 + 8) = ((4 * -4) + 8) = (-16 + 8) = -8. (6) Then, the final equation is ((A * B) - (C * D)) = ((-33 * -44) - (8 * -8)) = (1452 - (8 * -8)) = (1452 - -64) = 1516. (Final response) So the answer is 1516. [END]

Table A13: Prompt used for generating improvement for Multistep Arithmetic. The generated feedback is then formatted as "Step 1 to step {earliest error step - 1} is correct. In step {earliest error step} the part '{error segment}' is incorrect. This is because '{error reason}'." In general, we used three-shot prompting. Parts that will be generated are highlighted in blue. **Due to limited space, we present one example used for each task.** Please refer to our code repository for the full prompt.

Prompt Type	Prompt Example
Get Improvement	Q: Sort the following words alphabetically: List: cosh inferno every foyer coterie mcfadden halifax Answer: Let’s think step by step. (1) The first letter: “cosh”=“c” (3), “inferno”=“i” (9), “every”=“e” (5), “foyer”=“f” (6), “coterie”=“c” (3), “mcfadden”=“m” (13), “halifax”=“h” (8). (2) We now have: (3) “cosh” ? “coterie” < (5) “every” < (6) “foyer” < (8) “halifax” < (9) “inferno” < (13) “mcfadden”. (3) Now sort this subpart ["cosh" ? "coterie"] by looking at their second letters: "cosh"="o" (15), "coterie"="o" (15). (3.1) We now have: (15) ["cosh" ? "coterie"]. (3.2) Now sort this subpart ["cosh" ? "coterie"] by looking at their third letters: "cosh"="s" (19), "coterie"="c" (3). (3.3) We now have: (3) "coterie" < (19) "cosh". (3.4) Hence, we have "coterie" < "cosh". (4) Hence, we have "coterie" < "cosh" < "every" < "foyer" < "halifax" < "inferno" < "mcfadden". (Final response) So the answer is: coterie cosh every foyer halifax inferno mcfadden. [END] Feedback: Step (1) to (3.1) are correct. In step (3.2) the part " "coterie"="c" (3) " is incorrect. This is because the third letter of "coterie" should be "t". Updated Answer: Let’s think step by step. (1) The first letter: "cosh"="c" (3), "inferno"="i" (9), "every"="e" (5), "foyer"="f" (6), "coterie"="c" (3), "mcfadden"="m" (13), "halifax"="h" (8). (2) We now have: (3) ["cosh" ? "coterie"] < (5) "every" < (6) "foyer" < (8) "halifax" < (9) "inferno" < (13) "mcfadden". (3) Now sort this subpart ["cosh" ? "coterie"] by looking at their second letters: "cosh"="o" (15), "coterie"="o" (15). (3.1) We now have: (15) ["cosh" ? "coterie"]. (3.2) Now sort this subpart ["cosh" ? "coterie"] by looking at their third letters: "cosh"="s" (19), "coterie"="t" (20). (3.3) We now have: (19) "cosh" < (20) "coterie". (3.4) Hence, we have "cosh" < "coterie". (4) Hence, we have "cosh" < "coterie" < "every" < "foyer" < "halifax" < "inferno" < "mcfadden". (Final response) So the answer is: cosh coterie every foyer halifax inferno mcfadden. [END]

Table A14: Prompt used for generating improvement for Word Sorting. The generated feedback is then formatted as “Step 1 to step {earliest error step - 1} is correct. In step {earliest error step} the part ‘{error segment}’ is incorrect. This is because ‘{error reason}’.” In general, we used three-shot prompting. Parts that will be generated are highlighted in blue. **Due to limited space, we present one example used for each task.** Please refer to our code repository for the full prompt.

Prompt Type	Prompt Example
Get Feedback	Q: Yesterday was 12/31/1929. Today could not be 12/32/1929 because December has only 31 days. What is the date tomorrow in MM/DD/YYYY? Options: (A) 12/12/1929 (B) 01/01/1930 (C) 01/02/1998 (D) 01/02/1885 (E) 01/02/1930 (F) 12/23/1929 Answer: Let's think step by step. (1) If yesterday was 12/31/1929, then today is 01/01/1930. (2) The date tomorrow is 01/02/1930. (Final response) So the answer is (F). Earliest error step: (Final response) Error segment: "the answer is (F)" Error reason: (F) 12/23/1929 is inconsistent with the result "01/02/1930" in step (2). [END]
Get Improvement	Q: Yesterday was 12/31/1929. Today could not be 12/32/1929 because December has only 31 days. What is the date tomorrow in MM/DD/YYYY? Options: (A) 12/12/1929 (B) 01/01/1930 (C) 01/02/1998 (D) 01/02/1885 (E) 01/02/1930 (F) 12/23/1929 Answer: Let's think step by step. (1) If yesterday was 12/31/1929, then today is 01/01/1930. (2) The date tomorrow is 01/02/1930. (Final response) So the answer is (F). Feedback: Step (1) to step (2) are correct. In step (Final response) the part "the answer is (F)" is incorrect. This is because (F) 12/23/1929 is inconsistent with the result "01/02/1930" in step (2). Updated Answer: Let's think step by step. (1) If yesterday was 12/31/1929, then today is 01/01/1930. (2) The date tomorrow is 01/02/1930. (Final response) So the answer is (B). [END]

Table A15: Prompt used for generating feedback and improvement for Date Understanding. The generated feedback is then formatted as "Step 1 to step {first error step - 1} is correct. In step {first error step} the part '{error part}' is incorrect. This is because '{error reason}'." In general, we used three-shot prompting. Parts that will be generated are highlighted in blue. **Due to limited space, we present one example used for each task.** Please refer to our code repository for the full prompt.

Prompt Type	Prompt Example
Get Feedback	Q: The following paragraphs each describe a set of three objects arranged in a fixed order. The statements are logically consistent within each paragraph. On a branch, there are three birds: a hummingbird, an owl, and a falcon. The falcon is to the right of the owl. The hummingbird is to the left of the owl. Options: (A) The hummingbird is the second from the left. (B) The owl is the second from the left. (C) The falcon is the second from the left. Answer: Let's think step by step. Let "??" represents 0 or more objects, and "?" represents exactly 1 object. (1) The falcon is to the right of the owl: "(left) ?? owl ?? falcon ?? (right)". (2) The hummingbird is to the left of the owl: "(left) ?? hummingbird ?? owl ?? (right)". (3) There are in total three birds: a hummingbird, an owl, and a falcon. (4) Combining (1) and (2) we get the following ordering: "(left) ?? hummingbird ?? owl ?? falcon ?? (right)". (5) Combining (3) and (4) we get the following ordering: "(left) hummingbird owl falcon (right)". (6) According to this ordering, the bird that is second from the left is the falcon. (Final response) The falcon is the second from the left. So the answer is (C). Earliest error step: (6) Error segment: "the bird that is second from the left is the falcon" Error reason: It is inconsistent with the ordering in step (5), which established that "(left) hummingbird owl falcon (right)". [END]
Get Improvement	Q: The following paragraphs each describe a set of three objects arranged in a fixed order. The statements are logically consistent within each paragraph. On a branch, there are three birds: a hummingbird, an owl, and a falcon. The falcon is to the right of the owl. The hummingbird is to the left of the owl. Options: (A) The hummingbird is the second from the left. (B) The owl is the second from the left. (C) The falcon is the second from the left. Answer: Let's think step by step. Let "??" represents 0 or more objects, and "?" represents exactly 1 object. (1) The falcon is to the right of the owl: "(left) ?? owl ?? falcon ?? (right)". (2) The hummingbird is to the left of the owl: "(left) ?? hummingbird ?? owl ?? (right)". (3) There are in total three birds: a hummingbird, an owl, and a falcon. (4) Combining (1) and (2) we get the following ordering: "(left) ?? hummingbird ?? owl ?? falcon ?? (right)". (5) Combining (3) and (4) we get the following ordering: "(left) hummingbird owl falcon (right)". (6) According to this ordering, the bird that is second from the left is the falcon. (Final response) The falcon is the second from the left. So the answer is (C). Feedback: Step (1) to (5) are correct. In step (6) the part "the bird that is second from the left is the falcon" is incorrect. This is because it is inconsistent with the ordering in step (5), which established that "(left) hummingbird owl falcon (right)". Updated Answer: Let's think step by step. Let "??" represents 0 or more objects, and "?" represents exactly 1 object. (1) The falcon is to the right of the owl: "(left) ?? owl ?? falcon ?? (right)". (2) The hummingbird is to the left of the owl: "(left) ?? hummingbird ?? owl ?? (right)". (3) There are in total three birds: a hummingbird, an owl, and a falcon. (4) Combining (1) and (2) we get the following ordering: "(left) ?? hummingbird ?? owl ?? falcon ?? (right)". (5) Combining (3) and (4) we get the following ordering: "(left) hummingbird owl falcon (right)". (6) According to this ordering, the bird that is second from the left is the owl. (Final response) The owl is the second from the left. So the answer is (B). [END]

Table A16: Prompt used for generating feedback and improvement for Logical Deduction. The generated feedback is then formatted as "Step 1 to step {first error step - 1} is correct. In step {first error step} the part '{error part}' is incorrect. This is because '{error reason}'." In general, we used three-shot prompting. Parts that will be generated are highlighted in blue. **Due to limited space, we present one example used for each task.** Please refer to our code repository for the full prompt.