Offline Training of Language Model Agents with Functions as Learnable Weights

Shaokun Zhang [* 1]   Jieyu Zhang [* 2]   Jiale Liu [1]   Linxin Song [3]   Chi Wang [4]   Ranjay Krishna [2]   Qingyun Wu [1]

Abstract

Researchers and practitioners have recently re-framed powerful Large Language Models (LLMs) as *agents*, enabling them to automate complex tasks largely via the use of specialized functions. To facilitate the development of LLM agents, we present a novel paradigm of training LLM agents without modifying the LLM weights, which is particularly useful when the LLMs are difficult or inaccessible for modifications. Inspired by how humans continuously forge tools to adapt to real-world tasks, rather than change our biological structure to fit a static set of tools, we propose to progressively forge agent's functions to better solve the downstream tasks instead of modifying the LLM weights. By treating the functions as learnable 'agent parameters' and leveraging the fundamental idea of model training in artificial intelligence, we develop AgentOptimizer that employs the LLM to update agents' functions and devise an *agent training* algorithm with two strategies, roll-back, and early-stop, to streamline the training process. With extensive experiments, we showcase that the agent training paradigm could significantly improve the performance of representative LLM agents in various downstream tasks. We also study the behavior of the agent training regarding aspects like the learning curve and domain transferability. We have integrated our method into AutoGen library.

1. Introduction

Reframing Large Language Models (LLMs) as agents has ushered in a new paradigm of automation—one where LLMs can utilize existing functions [1] to accomplish complex

*Equal contribution [1]Pennsylvania State University [2]University of Washington [3]University of Southern California [4]Microsoft Research. Correspondence to: Qingyun Wu <qingyun.wu@psu.edu>.

[1]Note that the literature has used the term 'functions' to sometimes refer to tools or other actions.
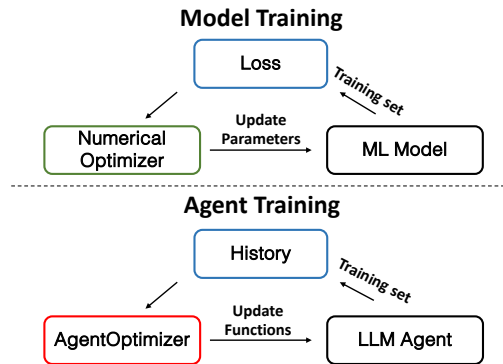

Figure 1. The comparison between model training and agent training. In model training, *numerical optimizers* (Ruder, 2016) such as SGD and Adam optimize the model weights according to the loss on the training set. In contrast, agent training iteratively updates the agents' functions according to the execution history using the proposed *AgentOptimizer*.

tasks (Xi et al., 2023; Wang et al., 2023b; Yao et al., 2023; Wang et al., 2023a; Humphreys et al., 2022; Shridhar et al., 2021). For example, LLM agents, armed with a function to 'search over Wikipedia' can answer knowledge questions; agents with the ability to 'issue SQL queries' can search large databases. Functions allow LLMs to access external knowledge sources (Peng et al., 2023), offload numerical computation (Wu et al., 2023b), search the internet (Shi et al., 2017), and much more (Qin et al., 2023).

To enable LLM agents with useful functions, users need to first manually create functions that would be helpful for specific downstream tasks. This curation process may require many iterations and, therefore, be time-consuming. Since LLMs are black boxes, researchers have found that LLMs unexpectedly fail to utilize certain kinds of functions (Qin et al., 2023). In response, researchers have tried to improve the underlying LLM's capability of using existing functions by finetuning the LLM with ground truth function calls (Qin et al., 2024; Zeng et al., 2023a). This finetuning process requires large computing resources. Worse, it limits which LLMs can be used since many LLM models are proprietary.

Inspired by the fact that human-made tools become an extension of the human user (Botvinick & Cohen, 1998) and how humans forge tools to best adapt to real-world tasks, rather than change the biological structure of the human

to fit a static set of tools, we propose a new *agent training* paradigm that 'forge' the functions for the LLM agent to use to best adapt to the training tasks. It addresses both aforementioned challenges at the same time since it does not require finetuning the underlying LLM and could start with an empty set of functions. While the LLM's parameters are never updated, its functions are optimized to maximize the agent's ability to solve tasks.

In specific, we draw an analogy between traditional model training and our agent training (Figure 1). (1) Instead of updating model parameters, our training process updates the functions for LLM agents, viewing them as the agent's 'trainable parameters'. (2) Instead of a loss calculated over a training set, our training process uses the agent's execution history and performance on training tasks as the basis for updating the agent's functions. Since we operate in the space of functions, numeric optimizers such as SGD or Adam are not applicable. Instead, we develop **AgentOptimizer**, which leverages the LLM to update the agent's functions based on the execution history and the agent-generated / ground truth answer from the current epoch. In particular, the AgentOptimizer is instructed to progressively update the current function set by performing one of the predefined function manipulation actions (add, revise, and remove), rather than regenerate the whole function set at each optimization step.

With AgentOptimizer, the overall workflow of LLM agent training is as follows: given a training set and an empty function set, at each epoch, we first evaluate the agent system against the training set and collect the execution history as well as the agent-generated / ground truth answers, then we feed this information to the AgentOptimizer to perform an optimization step to update the current function set. To avoid potential performance degradation caused by function updates, we introduce two simple strategies: roll-back and early-stop. The former is to withdraw the current function updates if the performance over the training set is degraded and roll back to the previous status, while the latter is to early terminate the optimization process when a certain number of consecutive optimization steps do not introduce any performance boost over the training set.

We conducted extensive empirical evaluations on three distinct tasks: mathematical reasoning (MATH) (Hendrycks et al., 2021), tabular processing (TabMWP) (Lu et al., 2023), and general real-world problems (GAIA) (Mialon et al., 2023). We trained two typical agent systems, GPT-4+ agent (OpenAI, 2023) and ReAct agent (Yao et al., 2023), using the agent training method. For the MATH dataset, agent training resulted in an obvious performance improvement in almost all cases. For more realistic and complex tasks GAIA and TabMWP, agent training led to an average performance improvement of 6% and 8.5% in GPT-4+ agent

and ReAct agents, respectively. We also perform ablation to demonstrate the efficacy of different components of the agent training method. In addition to ablation, we analyzed its extension to large-scale training and its transferability across different domains.

Our contributions are summarized below:

- **(Paradigm)** Inspired by the fundamental idea of model training in machine learning, we introduce a tailored paradigm for training LLM agents without modifying the LLMs to build specialized LLM agents for a given application: we establish analogies between the learnable parameters inherent in traditional models and the operational functions of LLM agents, as well as between the models' loss functions and the agents' execution history over the training set, to craft a training regime that enhances the LLM agents' capabilities;

- **(Methodology)** To realize this paradigm, we propose the AgentOptimizer as an alternative to numeric optimizers used in traditional model training. It is designed to operate in the space of the operational functions of LLM agents via the exceptional capabilities of LLMs. Based on the AgentOptimizer, we develop a training algorithm with two additional techniques (roll-back and early-stop) that streamline the training process;

- **(Experiments)** We conduct extensive experiments on three distinct tasks in training two typical agent systems, the GPT-4+ agent and the ReAct agent, to showcase the advantage of the proposed paradigm. We also provide ablation studies and detailed analysis to understand the behavior of the agent training.

2. Methodology

We begin by defining notations and setting up the research problem. We use $S_{\mathcal{F}}$ to denote any LLM agent system with function set $\mathcal{F} = \{f_1, ..., f_n | \forall i \in [n], f_i \in \mathcal{V}\}$. f_i denotes the i_{th} function that can be used by agent system $S_{\mathcal{F}}$ in the function space $\mathcal{V}$.

Throughout this work, we assume black-box LLMs such as ChatGPT (OpenAI, 2022) in the form of LLM as services. Given any task with training data $\mathcal{D}_{train}$ and test data $\mathcal{D}_{test}$, the goal of this study is to find a set of functions $\mathcal{F}^*$ that could improve the LLM agent's expected performance on unseen test data $\mathcal{D}_{test}$. To put it more formally,

$$\mathcal{F}^* = \arg\min_{\mathcal{F} \subset \mathcal{V}} E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})], \quad (1)$$

where $Loss(S_{\mathcal{F}}, \mathcal{D}_{test})$ measures the average loss of the agent $S_{\mathcal{F}}$ on test data $\mathcal{D}_{test}$. In the context of agent training throughout this paper, loss is defined as the rate of failed problem-solving attempts using agent systems.

However, the test set and its distribution are not available. In traditional machine learning model training, it is a common practice to assume that the distribution of the training and test data are the same or similar. While this assumption doesn't always hold, in machine learning practice, training loss is used ubiquitously as the primary metric for parameter selection as a compromise solution.

Following the same spirit, we also employ training data as a proxy for test data. Then optimizing the functions of the language agent in function space by minimizing the loss of training data. This approach allows us to approximate the performance of the language agent on unseen test data, i.e., $\hat{\mathcal{F}} = \arg\min_{\mathcal{F} \subset \mathcal{V}} Loss(S_{\mathcal{F}}, \mathcal{D}_{train})$, where $\hat{\mathcal{F}}$ is approximation of $\mathcal{F}^*$.

2.1. The AgentOptimizer

To obtain $\hat{\mathcal{F}}$, it is critical to develop an optimizer tailored for agent training: it should be capable of updating current functions according to the agent system's performance on the training set. In contrast to traditional model training where the optimization is conducted over a numeric model parameter space and derivative-based optimizers can be applied with a loss of choice, agent training aims to search for the optimal set of functions for the agent system and therefore existing numeric model optimizers are not applicable.

Considering these, we propose the **AgentOptimizer** which leverages LLMs' exceptional capability of understanding and generating language to iteratively update the current set of functions as an optimizer. Specifically, at each optimization step, we prompt the AgentOptimizer with the current status of the agent system and its execution history and performance on the training set and instruct it to update the functions of the agent system. Intuitively speaking, this iterative optimization paradigm could lead to the identification of optimal functions in a large language space, analogous to iteratively performing gradient descent when training traditional machine learning models.

The input to the AgentOptimizer. We use H to denote the information used to prompt LLMs, which mainly comprises the following two parts: 1) The execution history of the agents in solving each problem of the training set, including the details of how the agent uses current functions and 2) the final performance over the training data. In addition, we include the current set of functions associated with the agent system as input. This information is necessary for the AgentOptimizer to be aware of the current state of the agent system and accordingly suggest function updates.

Progressive function update. Given the inputted information H, a naive way of updating the current functions is to instruct the LLM to regenerate the whole function set

---

Algorithm 1 Progressive Function Update (`AgentOptimizer.step`)

Input: Functions to be optimized $\mathcal{F}^0$, historical information H
Output: Updated agent functions $\tilde{\mathcal{F}}$
1 **Initialization:** $\tilde{\mathcal{F}} \leftarrow \mathcal{F}^0, t \leftarrow 0$
2 **while** $t <$ *MAX_NUM* **do**
3   Action $\leftarrow$ LLM($\mathcal{F}^t$, H)
4   **if** *Action = TERMINATE* **then**
5   break
6   **else**
7   // add/revise/remove function
8           $\mathcal{F}^{t+1} =$ Action($\mathcal{F}^t$)
         $\tilde{\mathcal{F}} \leftarrow \mathcal{F}^{t+1}$
9       $t \leftarrow t + 1$
10 **Return** $\tilde{\mathcal{F}}$

---

to replace the existing one. However, such an aggressive optimization step is unwise since it overwrites all existing functions, discards useful functions already established, and requires the LLM to generate multiple functions in a single shot. In contrast, we propose to progressively update the functions via predefined *actions* within each optimization step. In particular, we adopt four actions: 1) *add_function*: add one new function that may be useful; 2) *revise_function*: revise one existing function; 3) *remove_function*: remove one existing function; and 4) *TERMINATE*: terminate the function update process. Except for the TERMINATE action, all the actions require certain *action arguments* as input; For example, to perform the *add_function* action, the LLM needs to generate the name, description, code, etc. of the function as the action arguments so that when executed, this action will add the new function to the function set. More details are presented in Appendix D.3. At each time step, the AgentOptimizer is prompted to choose one action until the maximum number of actions is reached or the AgentOptimizer chooses TERMINATE, and the resulting function set will be returned. The overall procedure of progressive function update is shown in Algorithm 1.

2.2. Agent Training

With the AgentOptimizer, we then present the overall agent training procedure. In practice, the function updates suggested by the AgentOptimizer may cause performance degradation, since the LLM is not the oracle for updating the functions. Therefore, we propose two simple strategies for the training procedure to avoid performance degradation.

Roll-back. To avoid performance degradation after function updating, we employ a simple yet effective strategy named roll-back. Specifically, at each optimization step, if the latest function update leads to a performance drop on the training set, the AgentOptimizer will withdraw this update. Moreover, considering the fact that LLMs are shown to be able to recognize patterns from in-context demonstra-

tions (Wei et al., 2022), we also record the failed updated function and the corresponding performance in a list (Line 11 of Algorithm 2). This list will be used as the prompt for the next function generations from AgentOptimizer. We expect that LLMs could use the historical failure information to generate better functions. The list will be cleared after achieving performance improvement.

Early-stop. In extreme situations, the optimization process may stuck, and rollback repeats without improving the performance. In this case, it is wise to terminate the optimization process, and we employ an early-stop strategy: the optimization process will be automatically terminated after C consecutive optimization steps without performance improvement over the training set.

Overall agent training algorithm. The pseudocode of the agent training is shown in Algorithm 2. The agent training process takes as input the following parameters: training data $\mathcal{D}_{train}$, agent system S, maximum training epoch E, and the early-stop threshold C. After an initialization step, which sets the initial functions list $\mathcal{F}_0$ and initial historical information H_0 to empty sets, the algorithm proceeds as follows: at each iteration i, the AgentOptimizer optimizes the functions list $\mathcal{F}_i$ to obtain $\mathcal{F}_{i+1}$ based on historical information H_i. The updated function set is then evaluated on the training set to obtain evaluation information for the next epoch of training. The training procedure terminates when the maximum epoch or early-stop threshold is reached.

3. Experiments

We conduct experiments to prove the superiority of the proposed method. We begin by providing the experimental settings in Section 3.1. We then evaluate the agent training method on three datasets to verify its effectiveness in Section 3.2. Finally, we perform in-depth investigations in the last three sections to provide a better understanding of the proposed agent training method.

3.1. Experimental Setup

Evaluation tasks and metrics. To evaluate the effectiveness of the proposed agent training, we conducted experiments on three distinct tasks: *Mathematical Reasoning*, *Tabular Processing*, and *General Real-World Tasks*. Due to the high cost of OpenAI models, it is impractical to evaluate the method on the complete datasets and therefore we subsample data from these datasets for training and testing, following the same settings as previous works (Yuan et al., 2024; Wu et al., 2023b). The number of training examples is set according to the LLM's context limit.

(1) Mathematical reasoning: Following a similar setting with (Yuan et al., 2024), we use a subset of MATH

Algorithm 2 Agent Training

Input: Training Data $\mathcal{D}_{train}$, agent system S, max training epoch E, early-stop threshold C
Output: Enhanced agent system $S_{\hat{\mathcal{F}}}$
11 **Initialization:** $i \leftarrow 0, r \leftarrow 0, H_0 \leftarrow \varnothing, \mathcal{F}_0 \leftarrow \varnothing$.
12 **while** $i < E$ **do**
13   **if** $H_i \neq \varnothing$ **then**
14         $\mathcal{F}_{i+1} = \texttt{AgentOptimizer.step}(\mathcal{F}_i, H_i)$
15   **else**
16         $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_i$
17     $H_{i+1} = \texttt{Eval}(S_{\mathcal{F}_{i+1}}, \mathcal{D}_{train})$
18   **if** $H_{i+1}.loss < H_i.loss$ **then**
19         $H_i.\text{fail\_record} \leftarrow \emptyset,$
        $\hat{\mathcal{F}} \leftarrow \mathcal{F}_{i+1}, i \leftarrow i + 1, r \leftarrow 0$
20   **else**
21         $H_i.\text{failure\_record} \leftarrow (\mathcal{F}_{i+1}, H_{i+1}.loss)$
        $r \leftarrow r + 1$
22   **if** $r > C$ **then**
23   // Early stop
            Break
24 **Return** $S_{\hat{\mathcal{F}}}$

---

datasets (Hendrycks et al., 2021) to evaluate the LLM agent's performance in addressing mathematical problems. For each data type (7 in total), we randomly choose 20 test examples and 80 training examples, and report the accuracy of each data type respectively.

(2) Tabular processing: The TabMWP (Lu et al., 2023) dataset evaluates agents in processing structured data in tables, where each data sample contains one table and one question in natural language. We randomly sampled 10 test examples and 100 training examples. We measured the model performance using accuracy based on the exact match.

(3) General real-world tasks: The GAIA dataset (Mialon et al., 2023) is dedicated to evaluating the LLM agents in solving unambiguous real-world questions. From its public subset, we randomly select 10 questions for training and 100 questions for testing and report the correct rate as suggested in the original paper.

Agent systems employed. We employ the proposed agent training method to train two typical LLM agent systems:

(1) GPT-4+ agent: GPT-4+ agent essentially is GPT-4 with function call and code interpreter. The GPT-4 plays the role of making reasoning decisions, while the code interpreter executes code and function calls suggested by the GPT-4.

(2) ReAct agent: The ReAct agent (Yao et al., 2023) generates both reasoning traces and task-specific actions in an interleaved manner to solve tasks. In our evaluations, we optimized the ReAct agent to improve its actions that may be taken at each action step after a reasoning process.

For both the GPT-4+ agent and ReAct agent, we initialize

Data types	P.Algebra		Algebra		I.Algebra		Geometry		C.Probability		Precalculus		N.Theory	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
GPT-4+ Agent w/o Agent Training	60.0%	78.8%	55.0%	**66.3%**	30.0%	30.0%	30.0%	40.0%	65.0%	72.5%	5.0%	32.5%	70.0%	56.3%
GPT-4+ Agent w/ Agent Training	**65.0%**	**82.6%**	**65.0%**	65.0%	**40.0%**	**38.8%**	**40.0%**	**42.5%**	65.0%	**76.3%**	**10.0%**	**35.0%**	**80.0%**	**67.5%**
ReAct Agent w/o Agent Training	55.0%	87.5%	55.0%	**83.8%**	25.0%	50.0%	5.0%	53.8%	45.0%	73.8%	5.0%	53.8%	75.0%	68.8%
ReAct Agent w/ Agent Training	**55.0%**	**87.5%**	**60.0%**	82.5%	**35.0%**	**51.3%**	**15.0%**	**58.8%**	**50.0%**	**78.8%**	**10.0%**	**62.5%**	**75.0%**	**72.5%**

Table 1. Train/Test accuracy of GPT-4+/ReAct agents with/without agent training on MATH datasets. We show the accuracy of each data type. We can observe that agent training could lead to an obviously better performance for both two agent systems in most cases.

Method	GAIA		TabMWP	
	Train	Test	Train	Test
GPT-4+ Agent w/o Agent Training	10.0%	16.0%	30.0%	51.0%
GPT-4+ Agent w/ Agent Training	**30.0%**	**23.0%**	**66.7%**	**56.0%**
ReAct Agent w/o Agent Training	20.0%	12.0%	63.3%	59.0%
ReAct Agent w/ Agent Training	**40.0%**	**18.0%**	**73.3%**	**70.0%**

Table 2. Train/Test accuracy of GPT-4+/ReAct agents with/without agent training on the GAIA and TabMWP datasets. We can observe that agent training can lead to greater performance for both GPT-4+ and ReAct agents on both two datasets.

Method	Number Theory	Intermediate Algebra	Counting and probability
No Agent Training	56.3%	30.0%	72.5%
Agent Training w/o Roll-back & Early-stop	63.8%	36.3%	72.5%
Agent Training w/o Progressive Function Update	60.0%	28.8%	70.0%
Agent Training (Ours)	**67.5%**	**38.8%**	**76.3%**

Table 3. We take the training of the GPT-4+ agent as an example and perform ablation to investigate the effect of different components of the agent training method on three data types of the MATH dataset.

them with Python as the initial function that can execute the Python code suggested by the LLMs

Models. For the more challenging tasks on the MATH and GAIA datasets, we used GPT-4-1106-preview for both AgentOptimizer and LLMs agents. For the easier task TabMWP, we chose to use GPT-3.5-turbo-1106 to construct LLMs agents and GPT-4-1106-preview to construct the AgentOptimizer. This was done to better visualize the improvement brought by the agent training and did not sacrifice the conclusions obtained from the experiments.

3.2. Main Results

Mathematical reasoning. We first evaluated the performance of GPT-4+ agent and ReAct agent on the MATH dataset, as well as their performance after agent training on train/test splits, as shown in Table 1. Across seven data types, we observed that agent training led to better performance on the test set in most cases (11 out of 14). Additionally, training performance improved in almost all cases, while in the remaining cases, it remained the same. Our results indicate that agent training could produce functions useful for unseen test tasks. Interestingly, for counting and probability problems, when training GPT-4+ agent, the training performance remains the same while test performance

improves from 72.5% to 76.3%. This suggests that in specific situations, even if the generated functions do not lead to performance improvement on the training set, they are helpful for the unseen test data.

Tabular processing and general real-world tasks. We then perform evaluations on Tabular Processing tasks TabMWP (Lu et al., 2023) and general real-world tasks GAIA (Mialon et al., 2023), as shown in Table 2. Our observations indicate that agent training led to performance improvements for both two agent systems. Since these two datasets are more realistic and complex than MATH, our results demonstrate that agent training can generate general and usable functions that increase agents' realistic task-solving capabilities, indicating that agent training is practically useful to some extent

3.3. Ablation and Analysis

3.3.1. ABLATION

We conducted ablation experiments to evaluate the effectiveness of two different components of the agent training method: (1) roll-back & early-stop, and (2) progressive function updating. To achieve this goal, we chose three data types of MATH that resulted in the largest performance improvements in training GPT-4+ agent: number theory, in-

termediate algebra, and counting and probability [2]. Specifically, to investigate (1), we removed roll-back and early-stop and trained the agent until reaching the maximum epoch number. The agent status will not roll-back when the training performance drops. For (2), we replaced the progressive function update with a one-step function generation, which directly prompted the GPT-4 in AgentOptimizer to generate the functions at each epoch. We also showed the origin GPT-4+ agent performance without agent training.

As shown in Table 3, the performance greatly dropped if either one of them was removed. Another interesting observation is that agent training without progressive function update even exhibited worse performance than the origin GPT-4+ agent without agent training. This scenario proves that prompting LLMs to generate functions is non-trivial. A bad function generation method may even lead to a negative effect. Therefore, a carefully designed function generation algorithm is desirable.

3.3.2. LEARNING CURVE

(a) Positive - I.Algebra   (b) Negative - Algebra
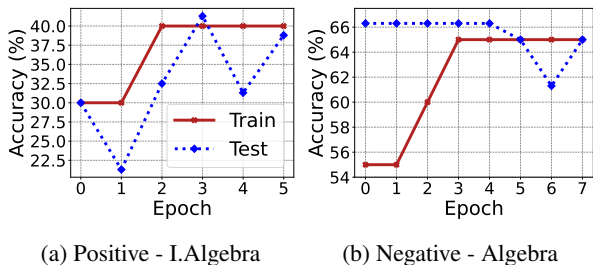
Figure 2. On the MATH dataset, we visualize the changes in train/test performance across epochs when training a GPT-4+ agent. For analysis purposes, we select one data type where the training does improve the test performance (Positive) and another that does not (Negative).

For analysis purposes, we visualize the learning curve when training GPT-4+ agent in solving mathematical problems in Figure 2. According to the types of experiment results, i.e., whether test performance improves (positive) or not (negative), we choose two data types, the only data type that failed to improve the test performance (Algebra) and one similar data type with the failed one that successfully obtained test performance improvement (Intermediate Algebra). Regarding the positive results on Intermediate Algebra in Figure 2a, we observe that when the optimization starts, the test performance is better than it was at the start time in most epochs, and the test performance is positively correlated with the training performance in general. These scenarios provide evidence to demonstrate the effectiveness of agent training.

---

[2]Pre-Algebra was not selected due to its similarity to Intermediate Algebra, despite having the same performance improvements as counting and probability.

However, we also notice that the highest test performance is not at the last epoch where the algorithm terminates. To some extent, it represents GPT-4+ agents overfitting to the training set and suffering from a test performance drop while the training performance remains the same. Regarding the negative results on Algebra in Figure 2b, we get a similar observation that the test performance drops while the training performance remains the same. We also found the scenario that the test performance remains the same while training performance improves, indicating that sometimes the generated tool may be not general enough to be useful but would not harm the performance in solving tasks.
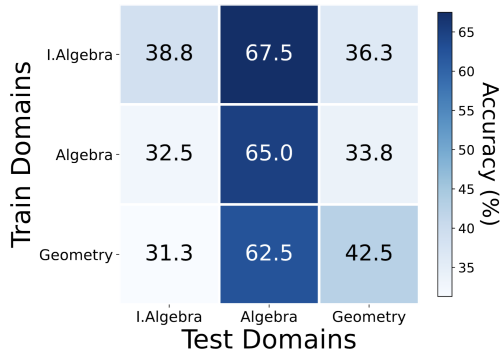
3.3.3. DOMAIN TRANSFERABILITY

Figure 3. To investigate the domain transferability of the agent training method, we show test performances of three different data types of the MATH dataset after training with different domains.

We then investigate the generalization and transferability (Zhou et al., 2022) of the agent training method when the test data and training data are not sampled from the same domain. We use three data types in MATH: algebra, intermediate algebra, and geometry. We intend to choose two data types that have similar distributions (algebra and intermediate algebra) and another data type that should have the largest semantic distance with the algebra and intermediate algebra (geometry). We then train GPT-4+ agent on these three datasets crossly using different train-test pairs and show the test performance in Figure 3. We observe that in most cases (2 out of 3), when the training and test data come from the same domain, agent training leads to the best test performance compared with training using other domains, where the results are intuitive to us. However, we observed an exception when testing on algebra. Using intermediate algebra for training led to better performance than using algebra (67.5% vs. 65.0%). This could be because intermediate algebra shares a similar distribution with algebra, and the more harder problems in intermediate algebra could be easier to learn basic and general functions that works for basic problems. Another observation is that using geometry as the training domain leads to the worst test performance in algebra and intermediate algebra. This

Method	MATH - Train	MATH - Test	TabMWP - Train	TabMWP - Test
CREATOR (Qian et al., 2023)	N/A	75.0%	N/A	30.0%
CRAFT (Yuan et al., 2024)	50.0%	73.8%	38.0%	38.5%
GPT-4+ Agent w/ Agent Training	**60.0%**	66.25%	**66.6%**	**56.0%**
ReAct Agent w/ Agent Training	**60.0%**	**77.5%**	**73.3%**	**70.0%**

Table 4. The comparisons between the trained agent systems with two typical tool-creation methods on MATH and TabMWP datasets. CREATOR doesn't involve a training stage so the training performance is unavailable. The results indicate that both GPT-4+ agent and ReAct agent trained with our method outperform tool-creation methods in most cases.

is because its distribution is far from both of the other two data types.

3.3.4. EXTEND TO LARGE SCALE TRAINING DATA - BATCH TRAINING

The proposed agent training method has one obvious bottleneck, which is that the training data size is limited to the context limit of the LLM-backed optimizer. This limitation prevents the full utilization of large-scale training data. A similar bottleneck occurs in traditional model training, where the constraint is from the GPU/CPU memory. To resolve this problem, traditional machine learning uses the concept of *batch training* (Masters & Luschi, 2018). This method divides the dataset into smaller subsets (batches) and trains the model iteratively on each batch to overcome the memory limitation.

Building on this practice, we propose a straightforward batch training method for our agent training flow. Specifically, we randomly sample one batch of training data within the LLM context limit at each training iteration from large-scale training data. Other procedures remain the same. We evaluate the Intermediate algebra of the MATH dataset on GPT-4+ agent system with 100 problems for training and 80 problems for testing where the test data is the same as it is in previous sections. We tried four different batch sizes (5, 10, 15, and 20), and set the epoch to 40, 20, 13, and 10, respectively, to ensure that the number of examples used for training is the same. We show the final test performance in Figure 4. The results show that large training data does not necessarily lead to test performance improvement in most cases, and only one case achieved a mirror improvement. Even when the batch size is set to 20, which is the same as the training data size in Figure 4, the test performance drops by 7.8%. This drop may be due to the frequent changing of training examples at each epoch, which prevents the AgentOptimizer from generating stable and effective functions.

3.4. Agent Training v.s. Tool-Creation

Tool-creation algorithms (Cai et al., 2024; Qian et al., 2023) are to prompt LLMs to create tools that are tailored to specific tasks. Since the tool-creation procedure is a one-time process that does not include subsequent optimization mechanisms based on training performance, the design philosophy emphasizes that the created tools *can be used* (without
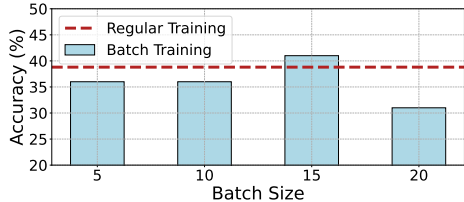
Figure 4. The comparisons between the "regular training" of our method and the extended "batch training". The batch training with an enlarged training set doesn't necessarily lead to better performance in different batch settings.

error), but not *used effectively* (improve performance).

In this section, we compare the trained GPT-4+/ReAct agents and two latest tool-creation methods, CREATOR (Qian et al., 2023) and CRAFT (Yuan et al., 2024), on MATH and TabMWP datasets. For TabMWP, we follow the same experimental setting as Section 3.2. We choose these two datasets because the baseline codes on these two datasets are available and we can make a rigorous comparison. To cover all data types of the MATH dataset, we randomly sample 20 examples for training and 80 examples for testing from all data types. As shown in Table 4, after agent training, both the GPT-4+ agent and ReAct agent exhibit better performance compared with the tool-creation method, indicating agent training is a promising paradigm to distill function/tool from advanced large language models.

3.5. Analysis of the Learned Functions

We conducted an in-depth analysis of the generated functions. First, we present a list of frequently used functions generated for all datasets in Table 5. Then, we show the number of successful function calls at the second and end epochs (the functions may not be the same) during model training in Table 6. We also present the widely adopted cyclomatic complexity (McCabe, 1994) of the generated functions. We calculate the complexity using the Lizard Python library and present the average complexity of tools for each task when optimizing both GPT-4+ agent and ReAct agent.

Our observations indicate that the number of successful function calls exhibits significant improvement in most datasets, indicating that the optimized functions are becoming more effective compared to the initial list. Considering function complexity, a good function should have a complexity of no more than 10. A less complex function is less prone to

Tasks	Top Used Functions
MATH	evaluate_expression, calculate_polynomial_roots, solve_algebraic_equation, calculate_circumference calculate_polynomial_roots, solve_algebraic_equation, calculate_complex_magnitude
GAIA	scrape_wikipedia_table, extract_pdf_text, perform_web_search, fetch_web_content
TabMWP	calculate_total_cost, analyze_stem_leaf_plot, calculate_basic_statistics, perform_table_calculation perform_arithmetic_operations, statistical_analysis

Table 5. For illustration purposes, we list frequently used (during testing) functions generated by AgentOptimizer in different tasks.

Metrics	MATH	GAIA	TabMWP
Second Epoch	11	8	19
Last Epoch	**23**	**10**	**41**
Avg. Complexity	1.2	3.7	5.0

Table 6. The number of successful function calls in the **second epoch** and the **last epoch** (functions may not be the same) of the agent training. We also show the cyclomatic complexity of the generated functions in the last row.

trigger bugs. We observed that the created functions for the three tasks exhibit relatively low complexity, indicating that the functions are reliable.

4. Related Work

There has been a growing volume of research focusing on employing LLMs to construct autonomous agents for reasoning, planning, and adapting to new observations in real-world tasks (Xi et al., 2023; Wang et al., 2023b; Hong et al., 2024; Yao et al., 2023; Wu et al., 2023a; Li et al., 2023; BabyAGI, 2023; Park et al., 2023). In such LLM agents, functions/tools/actions that LLM can leverage to interact with the environment or solve sub-tasks play a critical role, yet are often manually crafted (Yao et al., 2023). Recent works have explored automatic tool creation (Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024). Specifically, Tool-maker (Cai et al., 2024) proposes to create tools through three demonstrations and then validates the created tool using three validation examples; CREATOR (Qian et al., 2023) proposes to create tools exclusive for each query; And CRAFT (Yuan et al., 2024) first creates customizable tools tailored for specific problems and then retrieves relevant tools for user query inference time. In this work, we propose a conceptual framework that treats functions as learnable parameters in traditional AI models and develop a generic agent training paradigm to improve functions iteratively across epochs. Different from prior works, our AgentOptimizer updates the function set based on the LLM agent's execution history of the whole training set, rather than making functions according to individual query-answer pair(s); this approach not only includes the specific LLM agent's behavior into consideration for function creation (in contrast to looking at the query-answer pair only), but also tends to make generic functions that work for the whole training set. By formulating an iterative optimization process, the AgentOptimizer can continuously update the functions based on the execution history of each epoch during training in a trial-and-error manner.

Sharing a similar goal of improving LLM agents, another line of work aims to enhance agent capability by modifying the underlying LLMs (Patil et al., 2023; Qin et al., 2024; Zeng et al., 2023a). For instance, ToolLLM (Qin et al., 2024) collects a massive amount of APIs to construct instruction data to finetune LLaMA (Touvron et al., 2023) to obtain a new LLM optimized for using the collected APIs; AgentTune (Zeng et al., 2023a) proposes to enhance the agent abilities through a hybrid instruction-tuning strategy to tune the LLMs parameters. In contrast, we explore a new paradigm of training LLM agents without modifying the underlying LLM, which is particularly useful when the LLMs are online services and not available for tuning like GPT-4 or when tuning and maintaining a new LLM are expensive and time-consuming.

Besides, in this work, we leverage the exceptional capability of the LLM to build an optimizer (the AgentOptimizer) for training the agents, mimicking the numeric optimizers in model training such as SGD and Adam. Such an idea of using LLM as an optimizer has been proven effective by prior work (Yang et al., 2024; Zhang et al., 2023). While these prior works mainly leverage LLM as an optimizer for optimization problems like prompt optimization (Yang et al., 2024) and hyperparameter optimization (Zhang et al., 2023), our AgentOptimizer is particularly designed for the novel agent training paradigm and progressively update LLM agent's functions via multiple add, revise, and/or remove actions within each optimization step.

5. Conclusion

In this study, we propose a novel approach to train specialized LLM agents. The core idea is to draw an analogy between LLM agent training and traditional model training, where the learnable parameters in traditional models correspond to the operational functions of LLM agents, and the models' loss functions correspond to the historical performance metrics of the agents. Leveraging the impressive optimization capability of LLMs, we enhance the agents by updating the agent functions through the proposed AgentOptimizer. We evaluate the proposed method on multiple distinct tasks in training two typical agent systems and demonstrate that the agent training exhibits obvious performance improvement.

Impact Statement

This paper presents research aimed at advancing the field of language agents. Our work has several potential societal consequences, both positive and negative, that we feel need to be highlighted. On the positive side, language agents could be the core of many real-life applications (Hosseini et al., 2023; Cai et al., 2019), and our work could greatly benefit these applications by enhancing the agents. For instance, it could be the core of an industrial robot (Zeng et al., 2023b), and our work could potentially enhance working efficiency. On the negative side, the development of language agents raises the possibility of negative use of enhanced agents, such as using language agents to generate misinformation or harmful content (Navigli et al., 2023) in social media for illegal purposes. Another concern is allowing language models to make changes in external environments (Tian et al., 2023). For instance, allowing language models to perform code execution in the computer may lead to unintended consequences (Liu et al., 2024).

References

BabyAGI. Github — babyagi. https://github.com/yoheinakajima/babyagi, 2023.

Botvinick, M. and Cohen, J. Rubber hands 'feel' touch that eyes see. *Nature*, 391(6669):756–756, 1998.

Cai, C. J., Winter, S., Steiner, D., Wilcox, L., and Terry, M. " hello ai": uncovering the onboarding needs of medical practitioners for human-ai collaborative decision-making. *Proceedings of the ACM on Human-computer Interaction*, 3(CSCW):1–24, 2019.

Cai, T., Wang, X., Ma, T., Chen, X., and Zhou, D. Large language models as tool makers. *ICLR*, 2024.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. In *NeurIPS*, 2021.

Hoeffding, W. Probability inequalities for sums of bounded random variables. In *The collected works of Wassily Hoeffding*, pp. 409–426. Springer, 1994.

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., et al. Metagpt: Meta programming for multi-agent collaborative framework. 2024.

Hosseini, M., Gao, C. A., Liebovitz, D. M., Carvalho, A. M., Ahmad, F. S., Luo, Y., MacDonald, N., Holmes, K. L., and Kho, A. An exploratory survey about using chatgpt in education, healthcare, and research. *medRxiv*, pp. 2023–03, 2023.

Humphreys, P. C., Raposo, D., Pohlen, T., Thornton, G., Chhaparia, R., Muldal, A., Abramson, J., Georgiev, P., Santoro, A., and Lillicrap, T. A data-driven approach for learning to control computers. In *ICML*, pp. 9466–9482. PMLR, 2022.

Jayaseelan, N. Llama 2: The new open source language model.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., and Ghanem, B. Camel: Communicative agents for" mind" exploration of large language model society. In *NeurIPS*, 2023.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.

Liu, M., Wang, J., Lin, T., Ma, Q., Fang, Z., and Wu, Y. An empirical study of the code generation of safety-critical software using llms. *Applied Sciences*, 2024.

Lu, P., Qiu, L., Chang, K.-W., Wu, Y. N., Zhu, S.-C., Rajpurohit, T., Clark, P., and Kalyan, A. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *ICLR*, 2023.

Masters, D. and Luschi, C. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

McCabe, T. J. Software complexity, crosstalk. *Journal of Defense Software Engineering*, 1994.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023.

Navigli, R., Conia, S., and Ross, B. Biases in large language models: Origins, inventory and discussion. *ACM Journal of Data and Information Quality*, 2023.

OpenAI. Introducing ChatGPT, 2022. URL https://openai.com/blog/chatgpt. (Accessed on Jun 18, 2023).

OpenAI. Gpt-4 technical report, 2023.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. In *UIST*, pp. 1–22, 2023.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden, L., Yu, Z., Chen, W., et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

Qian, C., Han, C., Fung, Y., Qin, Y., Liu, Z., and Ji, H. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *EMNLP*, pp. 6922–6939, 2023.

Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., Zeng, Z., Huang, Y., Xiao, C., Han, C., et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ICLR*, 2024.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Shi, T., Karpathy, A., Fan, L., Hernandez, J., and Liang, P. World of bits: An open-domain platform for web-based agents. In *ICML*. PMLR, 2017.

Shridhar, M., Yuan, X., Côté, M.-A., Bisk, Y., Trischler, A., and Hausknecht, M. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *ICLR*, 2021.

Tian, Y., Yang, X., Zhang, J., Dong, Y., and Su, H. Evil geniuses: Delving into the safety of llm-based agents. *arXiv preprint arXiv:2311.11855*, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023b.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. 35: 24824–24837, 2022.

Wu, Q., Wang, C., and Huang, S. Frugal optimization for cost-related hyperparameters. In *AAAI*, 2021.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023a.

Wu, Y., Jia, F., Zhang, S., Wu, Q., Li, H., Zhu, E., Wang, Y., Lee, Y. T., Peng, R., and Wang, C. An empirical study on challenging math problem solving with gpt-4. *arXiv preprint arXiv:2306.01337*, 2023b.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. 2023.

Yuan, L., Chen, Y., Wang, X., Fung, Y. R., Peng, H., and Ji, H. Craft: Customizing llms by creating and retrieving from specialized toolsets. *ICLR*, 2024.

Zeng, A., Liu, M., Lu, R., Wang, B., Liu, X., Dong, Y., and Tang, J. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023a.

Zeng, F., Gan, W., Wang, Y., Liu, N., and Yu, P. S. Large language models for robotics: A survey. *arXiv preprint arXiv:2311.07226*, 2023b.

Zhang, M. R., Desai, N., Bae, J., Lorraine, J., and Ba, J. Using large language models for hyperparameter optimization. *arXiv e-prints*, pp. arXiv–2312, 2023.

Zhang, S., Jia, F., Wang, C., and Wu, Q. Targeted hyperparameter optimization with lexicographic preferences over multiple objectives. In *ICLR*, 2022.

Zhou, K., Liu, Z., Qiao, Y., Xiang, T., and Loy, C. C. Domain generalization: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

A. Supplementary Theoretical Analysis

In this section, we attempt to provide a theoretical analysis of the proposed agent training method. The objective is to provide an upper bound for the expected test loss difference between the trained agent function and the global optimal function. As an initial attempt, our analysis on the generalization bound of the agent training requires the following two strong assumptions. We leave the relaxation of these two assumptions to future work.

Assumption A.1. In the agent training scenario, the training data $\mathcal{D}_{train}$ and test data $\mathcal{D}_{test}$ come from the same distribution $\mathbb{P}$, i.e., $\mathcal{D}_{train}, \mathcal{D}_{test} \in \mathbb{P}$.

In classical machine learning model training, it is a common practice to assume that the distribution of the training and test data are the same or similar, then use training loss as the primary metric for parameters selection.

Assumption A.2. Given training data $\mathcal{D}_{train}$, the proposed agent training method could identify the function set $\hat{\mathcal{F}}$ which achieves the smallest loss in $\mathcal{D}_{train}$ after agent training.

$$\hat{\mathcal{F}} = \underset{\mathcal{F} \subset \mathcal{V}}{\arg\min} \, Loss(S_{\mathcal{F}}, \mathcal{D}_{train}). \tag{2}$$

Lemma A.3. *Under Assumption A.1, for any agent system $S_{\mathcal{F}}$ with function set $\mathcal{F}$, with probability at least $1 - \delta$ ($\delta \in (0, 1)$), we have:*

$$|Loss(S_{\mathcal{F}}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]| \leq \sqrt{\frac{\beta \ln(1/\delta)}{2|\mathcal{D}_{train}|}},$$

in which β represents the distance between the largest and the lowest loss value on any data instance. Specifically, for any data instance $d \in \mathbb{P}$, $l_{S_{\mathcal{F}}}(d) < \beta$, where $l_{S_{\mathcal{F}}}$ denotes the loss function, which measures the loss of each data instance for agent system $S_{\mathcal{F}}$.

Proof of Lemma A.3. For any training data set $\mathcal{D}_{train}$ and potential test data set $\mathcal{D}_{test}$ from the data distribution $\mathbb{P}$, we have

$$|Loss(S_{\mathcal{F}}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]| = |\frac{1}{|\mathcal{D}_{train}|} \sum_{i=1}^{|\mathcal{D}_{train}|} l_{S_{\mathcal{F}}}(d_i) - E_{d \sim \mathbb{P}}[l_{S_{\mathcal{F}}}(d)]|. \tag{3}$$

According to Hoeffding's inequality (Hoeffding, 1994), we have:

$$P(|Loss(S_{\mathcal{F}}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]| > \epsilon) = P(|\frac{1}{|\mathcal{D}_{train}|} \sum_{i=1}^{|\mathcal{D}_{train}|} l_{S_{\mathcal{F}}}(d_i) - E_{d \sim \mathbb{P}}[l_{S_{\mathcal{F}}}(d)]| > \epsilon) \tag{4}$$

$$\leq 2\exp\frac{-2|\mathcal{D}_{train}|\epsilon^2}{\frac{1}{|\mathcal{D}_{train}|}\sum_{i=1}^{|\mathcal{D}_{train}|}\beta} = 2\exp\frac{-2|\mathcal{D}_{train}|\epsilon^2}{\beta}.$$

Then with probability at least $1 - 2\exp\frac{-2|\mathcal{D}_{train}|\epsilon^2}{\beta}$, we have:

$$|Loss(S_{\mathcal{F}}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]| \leq \epsilon. \tag{5}$$

Taking $\delta = 2\exp\frac{-2|\mathcal{D}_{train}|\epsilon^2}{\beta}$, we have:

$$\epsilon = \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}} \tag{6}$$

Combining Equation 5 and Equation 6, with probability at least $1 - \delta$, we have:

$$|Loss(S_{\mathcal{F}}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]| \leq \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}. \tag{7}$$

Which completes the proof. □

Theorem A.4. *Under Assumption A.1 and Assumption A.2, with probability at least $1 - \delta$ ($\delta \in (0,1)$), the trained agent system $S_{\hat{\mathcal{F}}}$ with trained functionl list $\hat{\mathcal{F}}$ satisfies:*

$$E[Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{test})] - E[Loss(S_{\mathcal{F}^*}, \mathcal{D}_{test})] \le 2\sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}, \tag{8}$$

where $\mathcal{F}^*$ denotes the optimal function in the function space $\mathcal{V}$, i.e., $\mathcal{F}^* = \arg\min_{\mathcal{F} \subset \mathcal{V}} E[Loss(S_{\mathcal{F}}, \mathcal{D}_{test})]$.

Proof of Theorem A.4. Taking $\hat{\mathcal{F}}$ into Lemma A.3, with probability at least $1 - \delta$ ($\delta \in (0,1)$), we have:

$$|Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{train}) - E[Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{test})]| \le \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}. \tag{9}$$

Considering $\hat{\mathcal{F}} = \arg\min_{\mathcal{F} \subset \mathcal{V}} Loss(S_{\mathcal{F}}, \mathcal{D}_{train})$, we have:

$$Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{train}) < Loss(S_{\mathcal{F}^*}, \mathcal{D}_{train}). \tag{10}$$

Combing Equation 9 and Equation 10, we have:

$$E[Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{test})] \le Loss(S_{\mathcal{F}^*}, \mathcal{D}_{train}) + \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}. \tag{11}$$

Taking $S_{\mathcal{F}^*}$ into Lemma A.3, we have:

$$|Loss(S_{\mathcal{F}^*}, \mathcal{D}_{train}) - E[Loss(S_{\mathcal{F}^*}, \mathcal{D}_{test})]| \le \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}. \tag{12}$$

Combining Equation 11 and Equation 13, with probability at least $1 - \delta$, we have:

$$E[Loss(S_{\hat{\mathcal{F}}}, \mathcal{D}_{test})] - E[Loss(S_{\mathcal{F}^*}, \mathcal{D}_{test})] \le 2 * \sqrt{\frac{\beta \ln(2/\delta)}{2|\mathcal{D}_{train}|}}, \tag{13}$$

which completes the proof. $\qquad\square$

Theorem A.4 provides an upper bound on the expected test loss difference between the trained agent function $\hat{\mathcal{F}}$ and the global optimal function $\mathcal{F}^*$. We observe from Equation 13 that a larger training set could lead to a narrower upper bound. However, the training set is limited by the LLM's context limit. This limitation inspires us to investigate a better way of extending the training dataset, rather than relying on the straightforward batch training approach described in Section 3.3.4.

B. Supplementary Experimental Results

B.1. Evaluations on Other Language Models

	Code-Llama-34B	Mixtral-8x7B	GPT-3.5-turbo-1106
Before Training	7.5%	23.8%	25.0%
After Training	**11.3%**	**28.8%**	**28.8%**

Table 7. The performance of agents backed by other language models is evaluated before and after agent training on the MATH dataset. The results indicate that agent training still leads to significant performance improvements.

In this section, we conducted experiments to evaluate the performance of agents backed by various language models after agent training, including GPT-3.5-turbo-1106 (OpenAI, 2022), and open-source models Mixtral-8x7B (Jiang et al., 2024; 2023) and Code-Llama-34B (Roziere et al., 2023; Jayaseelan). The LLM that backed the AgentOptimizer was GPT-4-1106-preview.

We performed experiments on the MATH dataset using the same settings as described in Section 3.4. The results are presented in Table 7. Our findings indicate that agent training leads to better performance on all three models, demonstrating that agent training is agnostic to the LLMs that backed the agent.

B.2. More Experimental Results after Removing Roll-back & Early-stop
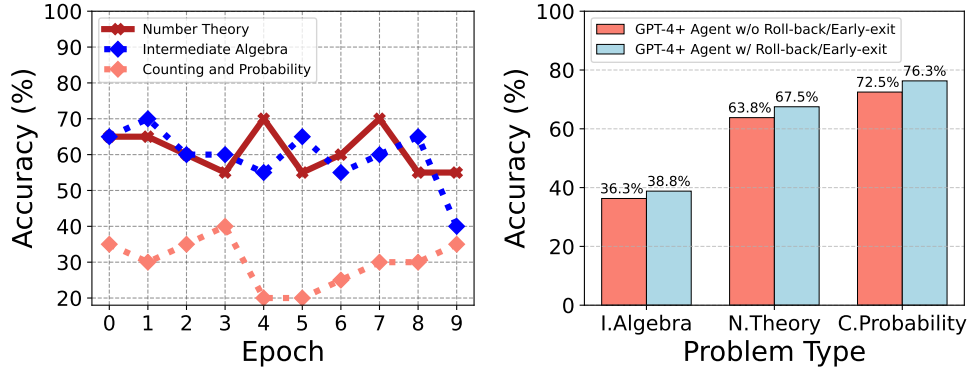
(a) Training performance w/o roll-back & early-stop (b) Test performance w/o roll-back & early-stop

Figure 5. After removing the roll-back and early-exit mechanisms, the learning curve of the training performance and the final test performance of GPT-4+ Agents.

We present additional experimental results in Figure 5 after removing roll-back and early-stop. Specifically, we further illustrate the training performance curve in Figure 5a. We observe that the training performance fluctuated with the number of training epochs, indicating that the learned functions are not stable and may not necessarily lead to improved training performance at each epoch. This unstable function optimization leads to a drop in test performance, as shown in Figure 5b.

C. Supplementary Analysis of Agent Training versus Model Training

	Optimizer	Target	Human Interpretable	Access to Model/LLM Weights
Model Training	SGD etc.	Model Weights	✗	✓
Agent Training	LLMs	Functions	✓	✗

Table 8. Comparing Model Training and Agent Training: Model training relies on an optimizer such as SGD. It is not human-interpretable and requires access to model parameters. In contrast, agent training uses LLMs as the optimizer, which is interpretable in natural language and generated functions. Furthermore, agent training does not require access to model parameters.

Table 8 summarizes the differences between these two training paradigms. Although both paradigms have a similar workflow of improving from training data leveraging their optimizers, they have different features. Specifically, the optimizers in traditional model training are gradient descent optimization algorithms, which update the model parameters in the opposite direction of the gradient of the loss function. However, the complex parameters updating logic is not interpretable to humans, and model training requires accessible parameters. In contrast, the optimizers in agent training are LLMs, which prompt the update of agent functions using natural language at each optimization step. The optimization is interpretable to humans (functions and natural language), and it doesn't require accessible parameters.

D. Implementations Details

D.1. Prompt Design for AgentOptimizer

You are a function optimizer. Your task is to maintain a list of functions for the assistant according to the existing function set and conversation history that happens between the assistant and the user.

You can perform one of the following four actions to manipulate the function set using the functions you have:

1. Revise one existing function (using revise_function). 2. Remove one existing function (using remove_function). 3. Add one new function (using add_function). 4. Directly return "TERMINATE" to me if no more actions are needed for the current function set.

Below are the principles that you need to follow for taking these four actions.

(1) Revise one existing function: 1. Pay more attention to the failed tasks and corresponding error information, and optimize the function used in these tasks according to the conversation history if needed. 2. A failed function call can occur due to incorrect input arguments (missing arguments) or an incorrect function code implementation. You should focus more on the function code implementation and make it easy to get success function call. 3. Do not revise the function that you think works well and plays a critical role in solving the problems according to the conversation history. Only making revisions if needed. 4. Sometimes, a NameError may occur. To fix this error, you can either revise the name of the function in the code implementation or revise the name of the function call to make these two names consistent.

(2) Remove one existing function: 1. Only remove the function that you think is not needed anymore in future tasks.

(3) Add one new function: 1. The added function should be general enough to be used in future tasks. For instance, if you encounter a problem that this function can solve, or one step of it, you can use the generated function directly instead of starting from scratch 2. The added new function should solve a higher-level question that encompasses the original query and extend the code's functionality to make it more versatile and widely applicable. 3. Replace specific strings or variable names with general variables to enhance the tool's applicability to various queries. All names used inside the function should be passed in as arguments. Below is an example of a function that potentially deserves to be added, which can be used to solve a higher-level question:

```
{{
    "name": "evaluate_expression",
    "description": "Evaluate arithmetic or mathematical expressions provided as strings.",
    "arguments": {{
        "expression": {{
            "type": "string",
            "description": "The mathematical expression to evaluate."
        }}
    }},
    "packages": "sympy",
    "code": "from sympy import sympify, SympifyError\n\n def evaluate_expression(expression):\n    try:\n        result = sympify(
    expression)\n        if result.is_number:\n            result = float(result)\n        else:\n            result = str(result)\n
        return result\n    except SympifyError as e:\n        return str(e)"
}}
```

(4) Directly return "TERMINATE": If you think there is no need to perform any other actions for the current function set since the current list is optimal more actions will harm the performance in future tasks. Please directly reply to me with "TERMINATE".

One function signature includes the following five elements: 1. Function name 2. Function description 3. JSON schema of arguments encoded as a string 4. A list of package names imported by the function packages 5. The code implementation

Below are the signatures of the current functions.

List A: {current_function_signature}

The success rate (performance) with this function set is {success_rate}. The following list are the function signatures that you have after taking {actions_num} actions in our previous conversations.

List B: {updated_function_signature}.

We also provide more examples for different functions and their corresponding success rates. The following function signatures are arranged in are arranged in ascending order based on their success rates, where higher success rates indicate better quality.

{historical_fail_functions}

Here are {conversation_num} conversation histories of solving {conversation_num} tasks.

History: {history}

The following table shows the statistical information for solving each task in each conversation and indicates whether each task was successfully solved. 1 represents correct. 0 represents wrong.

statistic: {statistic}

According to the information I provide, please take one of four actions to manipulate list B using the functions you know. Instead of returning TERMINATE directly or taking no action, you should try your best to optimize the function set. Only

take no action if you really think the current list is optimal, as more actions will harm performance in future tasks. Even adding a general function that can substitute the assistant's repeated suggestions of Python code with the same functionality could also be helpful.

D.2. Prompt Design for ReAct

Answer the following question using your coding skills. Below is a list of the tools you can use and their detailed descriptions:

{tool_descriptions}

You should always follow the below template, when you respond you should provide one (Thought, Action, Action Input) triplet and wait for observation before proceeding to the next round, unless you have reached a FINAL ANSWER.

YOUR FINAL ANSWER should be a number OR as few words as possible OR a comma separated list of numbers and/or strings.

If you are asked for a number, don't use comma to write your number neither use units such as $ or percent sign unless specified otherwise. If you are asked for a string, don't use articles, neither abbreviations (e.g. for cities), and write the digits in plain text unless specified otherwise. If you are asked for a comma separated list, apply the above rules depending of whether the element to be put in the list is a number or a string.

TEMPLATE:

Question: the input question you must answer

Thought: your reasoning about the current situation

Action 1: the action to take, should be one of [{tool_names}]

Action 1 Input: the arguments passed to action 1

Observation 1: the result of action 1

Action 2: the action to take, should be one of [{tool_names}]

Action 2 Input: the input to action 2

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

FINAL ANSWER: the final answer to the original input question

D.3. Function calls of LLM backed AgentOptimizer

Add_function: add a new function that may be used in future tasks.

```
ADD_FUNC = {
    "type": "function",
    "function": {
        "name": "add_function",
        "description": "Add a function in the context of the conversation. Necessary Python packages must be declared. The name of
the function MUST be the same with the function name in the code you generated.",
        "parameters": {
            "type": "object",
            "properties": {
                "name": {
                    "type": "string",
                    "description": "The name of the function in the code implementation."
                },
                "description": {
                    "type": "string",
                    "description": "A short description of the function."
                },
                "arguments": {
                    "type": "string",
                    "description": "JSON schema of arguments encoded as a string. Please note that the JSON schema only supports
specific types including string, integer, object, array, boolean. (do not have float type) For example: { \"url\": { \"type\": \"
string\", \"description\": \"The URL\", }}. Please avoid the error 'array schema missing items' when using array type."
                },
                "packages": {
                    "type": "string",
                    "description": "A list of package names imported by the function, and that need to be installed with pip prior
to invoking the function. This solves ModuleNotFoundError. It should be string, not list."
```

```
            },
            "code": {
                "type": "string",
                "description": "The implementation in Python. Do not include the function declaration."
            }
        },
        "required": ["name", "description", "arguments", "packages", "code"]
    }
}
```

Revise_function: revise one existing function.

```
REVISE_FUNC = {
    "type": "function",
    "function": {
        "name": "revise_function",
        "description": "Revise a function in the context of the conversation. Necessary Python packages must be declared. The name
of the function MUST be the same with the function name in the code you generated.",
        "parameters": {
            "type": "object",
            "properties": {
                "name": {
                    "type": "string",
                    "description": "The name of the function in the code implementation."
                },
                "description": {
                    "type": "string",
                    "description": "A short description of the function."
                },
                "arguments": {
                    "type": "string",
                    "description": "JSON schema of arguments encoded as a string. Please note that the JSON schema only supports
specific types including string, integer, object, array, boolean. (do not have float type) For example: { \"url\": { \"type\": \"
string\", \"description\": \"The URL\", }}. Please avoid the error 'array schema missing items' when using array type."
                },
                "packages": {
                    "type": "string",
                    "description": "A list of package names imported by the function, and that need to be installed with pip prior
to invoking the function. This solves ModuleNotFoundError. It should be string, not list."
                },
                "code": {
                    "type": "string",
                    "description": "The implementation in Python. Do not include the function declaration."
                }
            },
            "required": ["name", "description", "arguments", "packages", "code"]
        }
    }
}
```

Remove_function: remove one existing function.

```
REMOVE_FUNC = {
    "type": "function",
    "function": {
        "name": "remove_function",
        "description": "Remove one function in the context of the conversation. Once remove one function, the assistant will not
use this function in future conversation.",
        "parameters": {
            "type": "object",
            "properties": {
                "name": {
                    "type": "string",
                    "description": "The name of the function in the code implementation."
                }
            },
            "required": ["name"]
        }
    }
}
```

E. Generated Functions

Here we provide a list of typical function examples generated by AgentOptimizer for all three datasets.

E.1. Trained Functions in MATH

GPT-4+ Agent

1. evaluate_expression: evaluate arithmetic or mathematical expressions provided as strings.

```
from sympy import sympify, SympifyError

def evaluate_expression(expression):
    try:
        result = sympify(expression)
        if result.is_number:
            result = float(result)
        else:
            result = str(result)
        return result
    except SympifyError as e:
        return str(e)
```

2. calculate_polynomial_roots: find all real roots of a given polynomial.

```
from sympy import solve, sympify, SympifyError

def calculate_polynomial_roots(polynomial):
    try:
        # Parse the polynomial and solve for its roots
        roots = solve(sympify(polynomial), dict=True)
        # Convert roots to a standard list format
        roots_list = [str(root) for root in roots]\n
        return roots_list
    except SympifyError as e:
        return str(e)
```

3. solve_algebraic_equation: revises the existing function that solves a single variable algebraic equation to handle equations more robustly with variable terms on both sides.

```
from sympy import symbols, Eq, solve, parse_expr

def solve_algebraic_equation(equation, variable):
    # Create a symbolic variable
    symbol = symbols(variable)
    # Parse the equation string into a sympy expression
    left_part, right_part = equation.split('=')
    eq = Eq(parse_expr(left_part), parse_expr(right_part))
    # Solve the equation for the variable
    solution = solve(eq, symbol)
    # Return the solution
    return solution
```

ReAct Agent

1. convert_decimal_to_fraction: convert a decimal number to its equivalent in the simplest fractional form.

```
from fractions import Fraction

def convert_decimal_to_fraction(decimal_number):
    try:
        # Convert string to a floating point number
        decimal_number = float(decimal_number)
        # Create a Fraction from the decimal number
        fraction_result = Fraction(decimal_number).limit_denominator()
        # Return the fraction as a string in the form 'numerator/denominator'
        return str(fraction_result)
    except ValueError as e:
        return str(e)
```

2. evaluate_math_expression: evaluate a wide range of mathematical expressions provided as strings, including basic arithmetic, factorial, combinations, and permutations.

```
from sympy import sympify, factorial, binomial

def evaluate_math_expression(expression):
    try:
        # Extend the namespace with factorial and binomial functions
        local_dict = {'factorial': factorial, 'comb': binomial}
        # Evaluate the expression using sympy's sympify function
        result = sympify(expression, locals=local_dict)
        if result.is_number:
            return float(result)
        else:
            return str(result)
    except Exception as e:
        return str(e)
```

3. get_polynomial_degree: given a polynomial expression as a string, return the degree of the polynomial.

```
from sympy import Poly, SympifyError

def get_polynomial_degree(expression):
```

17

```
try:
    # Convert the string expression into a polynomial
    poly = Poly(expression)
    # Return the degree of the polynomial
    return poly.degree()
except SympifyError as e:
    return str(e)
```

E.2. Trained Functions in GAIA

GPT-4+ Agent

1. perform_web_search: performs a web search using Bing Search API and returns the top search results including URLs and snippets.

```
import os
import requests

def perform_web_search(query):
    subscription_key = os.environ['BING_SEARCH_V7_SUBSCRIPTION_KEY']
    endpoint = os.environ['BING_SEARCH_V7_ENDPOINT'] + '/v7.0/search'
    headers = {'Ocp-Apim-Subscription-Key': subscription_key}
    params = {'q': query, 'textDecorations': True, 'textFormat': 'HTML'}
    response = requests.get(endpoint, headers=headers, params=params)
    response.raise_for_status()
    search_results = response.json()
    top_results = [{'url': result['url'], 'snippet': result['snippet']} for result in search_results.get('webPages', {}).get('value',
      [])]
    return top_results
```

2. scrape_wikipedia_table: scrapes data from a table on a Wikipedia page based on a header keyword.

```
import requests
from bs4 import BeautifulSoup

def scrape_wikipedia_table(url, header_keyword):
    response = requests.get(url)
    response.raise_for_status()
    soup = BeautifulSoup(response.content, 'html.parser')
    headers = soup.find_all(['h1', 'h2', 'h3', 'h4', 'h5', 'h6'])
    data = []
    for header in headers:
        if header_keyword.lower() in header.text.lower():
            table = header.find_next_sibling('table', class_='wikitable')
            if table:
                rows = table.find_all('tr')
                for row in rows:
                    cols = row.find_all(['th', 'td'])
                    cols = [ele.text.strip() for ele in cols]
                    data.append([ele for ele in cols if ele])
                break
    return data
```

3. extract_pdf_text: extracts text from a PDF file.

```
import fitz  # PyMuPDF

def extract_pdf_text(file_path):
    # Open the PDF file
    with fitz.open(file_path) as pdf:
        text = ''
        # Iterate over each page
        for page_num in range(len(pdf)):
            page = pdf[page_num]
            text += page.get_text()
        return text
```

React Agent

1. fetch_webpage_content: retrieve the HTML content of a given webpage URL.

```
import requests

def fetch_webpage_content(url):
    response = requests.get(url)
    response.raise_for_status()
    return response.text
```

2. fetch_bing_search_results: retrieve search results from Bing Web Search API.

```
import os
import requests
```

```
def fetch_bing_search_results(query):
    subscription_key = os.environ['BING_SEARCH_V7_SUBSCRIPTION_KEY']
    endpoint = os.environ['BING_SEARCH_V7_ENDPOINT'] + "/v7.0/search"

    headers = {'Ocp-Apim-Subscription-Key': subscription_key}
    params = {'q': query, 'textDecorations': True, 'textFormat': 'HTML'}

    response = requests.get(endpoint, headers=headers, params=params)
    response.raise_for_status()
    return response.json()
```

3. extract_text_from_pdf: extracts all text from a given PDF file.

```
import fitz  # PyMuPDF

def extract_text_from_pdf(file_path):
    try:
        # Open the PDF file
        with fitz.open(file_path) as pdf:
            text = ''
            # Extract text from each page in the PDF
            for page in pdf:
                text += page.get_text()
            return text
    except Exception as e:
        return f'An error occurred: {str(e)}'
```

E.3. Trained Functions in TabMWP

GPT-4+ Agent

1. perform_arithmetic_operations: perform basic arithmetic operations such as sum, average, maximum, minimum, difference, and rate of change on a given list of numbers.

```
def perform_arithmetic_operations(numbers, operation):
    result = None
    if operation == 'sum':
        result = sum(numbers)
    elif operation == 'avg':
        result = sum(numbers) / len(numbers) if numbers else None
    elif operation == 'max':
        result = max(numbers) if numbers else None
    elif operation == 'min':
        result = min(numbers) if numbers else None
    elif operation == 'diff' and len(numbers) > 1:
        result = numbers[0] - numbers[1]
    elif operation == 'rate_of_change' and len(numbers) > 1 and numbers[1] != 0:
        result = ((numbers[0] - numbers[1]) / abs(numbers[1])) * 100
    return result
```

2. analyze_stem_leaf_plot Analyze a given stem-leaf plot to calculate the total count of values within a specified range.

```
def analyze_stem_leaf_plot(stem_leaf_data, min_value, max_value):
    count = 0
    for stem, leaves in stem_leaf_data.items():
        for leaf in leaves:
            value = int(stem) * 10 + leaf
            if min_value <= value < max_value:
                count += 1
    return count
```

3. calculate_range Calculate the range (difference between the maximum and minimum) of a list of numbers.

```
def calculate_range(numbers):
    return max(numbers) - min(numbers)
```

React Agent

1. calculate_total_cost_general: Calculate the total cost given a unit price and quantity, supporting both the quantity as a string or an integer.

```
def calculate_total_cost_general(unit_price, quantity):
    return float(unit_price) * (int(quantity) if isinstance(quantity, str) else quantity)
```

F. Case Study

We present then three case studies for the trained GPT-4+ agent on three different datasets, to identify why the well-optimized learned function leads to the correct result in each case study.

F.1. Case Study for MATH

Figure 6. Comparisons of GPT-4+ Agent agents before and after agent training. After the training, the well-optimized learned function leads to correct result compared to real-time generated python code.

F.2. Case Study for GAIA

Figure 7. Comparisons of GPT-4+ Agent task-solving trajectory on GAIA before and after agent training. After training, the agent can successfully leverage the well-optimized function to handle complex web-scraping problems and solve the otherwise coding-heavy task. The suggested python code by assistant is truncated for simplicity.
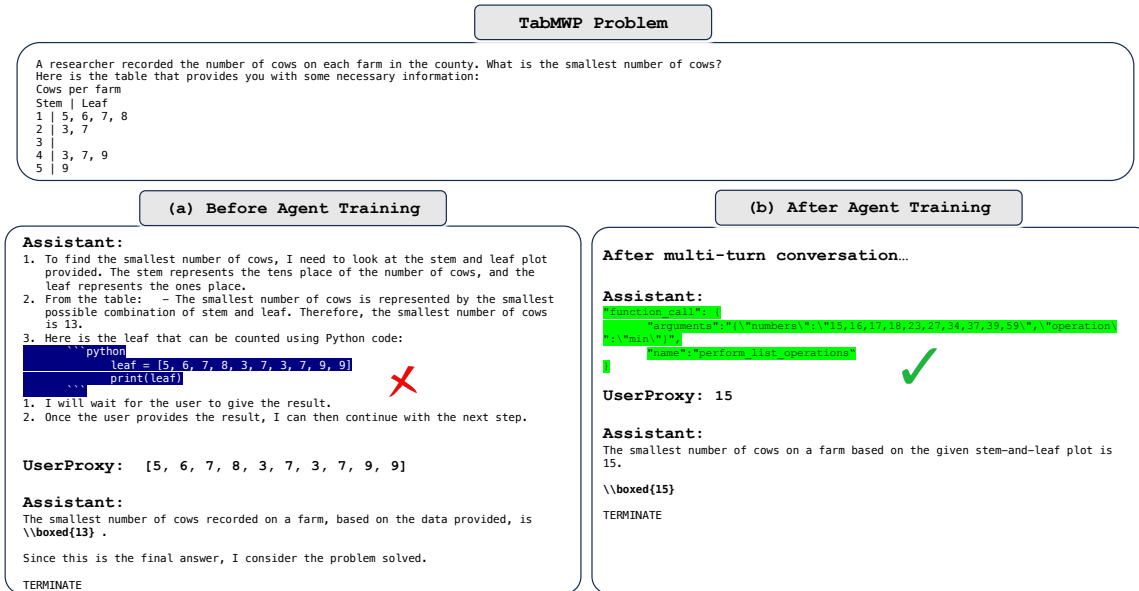
F.3. Case Study for TabMWP

Figure 8. Comparisons of GPT-3.5-turbo + Agent task-solving trajectory on TabMWP before and after agent training. After training, the agent can successfully leverage the well-optimized function to obtain an accurate result.

G. Hyperparameters Settings

The proposed agent training method involves several hyperparameters, including the training epoch, early-stop threshold, and maximum number of actions. In our empirical experiments across all three datasets, we consistently utilized the same hyperparameter configuration for the proposed agent training algorithm. Specifically: (1) We set the training epoch to 10 for all experiments. (2) An early stopping criterion was established with a threshold of 10 epochs. If there were 10 consecutive epochs without any improvement in training performance, the training process terminated. (3) Additionally, we restricted the maximum number of actions taken during each function update step to 3. It is essential to recognize that optimal hyperparameter settings can vary based on the specific problem and task. However, for our research, we kept these parameters fixed to ensure a consistent experimental setup. Combining our algorithm with hyperparameter tuning techniques from previous work (Li et al., 2018; Wu et al., 2021; Zhang et al., 2022) may further enhance performance.

H. Limitations

A significant bottleneck in the agent training algorithm arises from that the size of training data is limited by the LLM context limit. This constraint severely restricts its applicability to large-scale training scenarios. Furthermore, in Section 3.3.4, we empirically demonstrate that directly applying batch training techniques from traditional machine learning to agent training is ineffective and presents a non-trivial challenge. We regard addressing the limitations as our follow-up work.