Learning from Errors: A Data-Efficient Adaptation Method of Large Language Models for Code Generation

Anonymous ACL submission

Abstract

001 002 Large Language Models (LLMs) have achieved substantial advances in code generation, but they still struggle in specific code generation scenarios. These scenarios often require LLMs to be adapted to meet specific needs, but the limited training data available in practice leads to poor code generation performance. Therefore, how to effectively adapt LLMs to new scenarios with less training data is a major challenge for current code generation. In this paper, we propose a novel and effective adaptation method called DEED, which stands for **D**ata-**E**fficient adaptation based on **E**rror-**D**riven learning for code generation. DEED leverages the errors made by LLM as learning opportunities and overcomes its own shortcomings through error revision, thereby achieving efficient learning. Specifically, DEED includes identifying the erroneous code generated by LLM, using SELF-REVISE for code revision, optimizing the model with the revised code, and iteratively adapting the process for continuous improvement. Experimental results show that DEED achieves superior performance compared with mainstream fine-tuning and prompting methods using only a small amount of training data, with an average relative improvement of 54.7% on Pass@1 on multiple code generation datasets. We also verify the effectiveness of SELF-REVISE, which generates revised code that optimizes the model more efficiently compared to the code samples from datasets. Moreover, DEED consistently shows strong performance across various LLMs, highlighting its generalizability.

1 Introduction

Code generation is an important technology that can improve the efficiency and quality of software development. Given the human requirement expressed in natural language, code generation allows machines to generate executable programs that satisfy this requirement. Code generation has been a research hot topic in the fields of artificial intelligence, software engineering, and natural language processing. Recently, code generation technologies have made significant advancements in both academia and industry (Chen et al., 2021; Shen et al., 2022; Rozière et al., 2023). In particular, large language models (LLMs) demonstrate great potential in code generation tasks (Zheng et al., 2023; Nijkamp et al., 2023; Fried et al., 2022; Chen et al., 2023b; Zhang et al., 2023b; Jiang et al., 2023). However, LLMs still face significant challenges in code generation for some specific scenarios or domains (Ahmed et al., 2024; Chen et al., 2023d).

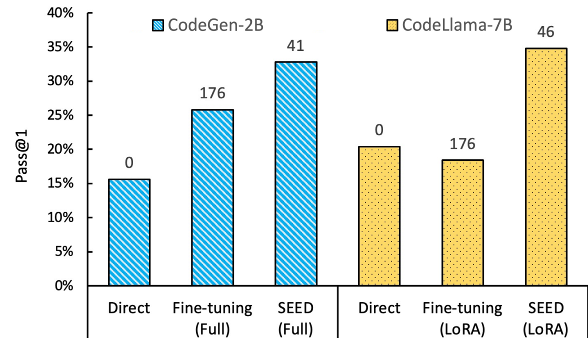

Figure 1: The performance of direct generation, fine-tuning, and our proposed DEED on MBPP dataset with few training data. The numbers on the bars indicate the training data used by different methods.

For specific code generation scenarios, fine-tuning is an essential adaptation method to ensure LLMs fulfill particular needs (Shi et al., 2023; Liu et al., 2024; Chakraborty et al., 2022; Ciniselli et al., 2022). However, in these specific scenarios, it is difficult to obtain sufficient training data for fine-tuning LLMs, due to common reasons such as industry secrecy and scarcity of resources. For example, in safety-critical scenarios like aerospace, medical devices, and transportation industries, the generated code must adhere to specific security

specifications, and accessing relevant data is often extremely difficult due to high confidentiality and strict access control. Under the circumstance of limited data, mainstream fine-tuning methods might not enable LLMs to achieve the desired code generation performance and may even lead to a degradation in model performance (Aghajanyan et al., 2021; Xu et al., 2021b; Zhang et al., 2022), as shown in Figure 1. Consequently, how to effectively adapt LLMs to specific scenarios with limited data available is a major challenge for code generation in practice.

The mainstream fine-tuning methods use a large number of data gathered under specific scenarios for training (Xu et al., 2021a). They enable the model to exhaustively learn the features present in these data and thus adapt to the specific scenarios. However, they have two disadvantages. First, compelling LLMs to relearn the entire code data of new scenarios is inefficient. Considering that LLMs are pre-trained on large-scale and diverse data, it's reasonably assumed that they possess a certain level of general knowledge, lacking only particular information for application in specific scenarios. Second, when faced with insufficient data volume or data drift, the model may learn certain undesirable features (such as inaccurate or irrelevant programming knowledge and patterns), thereby affecting its learning efficiency and negatively impacting its final performance.

To overcome the disadvantages of mainstream fine-tuning methods, we take inspiration from the error-driven learning observed in humans. 1) Error-driven learning requires learners to identify their errors through testing. It helps learners to identify what they have mastered and what they still need to learn, allowing them to narrow the scope of learning and avoid wasting efforts on irrelevancies. 2) Through error revision, learners can understand their deficiencies and make targeted improvements, thus enhancing learning efficiency and effectiveness. This motivates us to explore methods to achieve data-efficient adaptation of LLMs for code generation guided by error-driven learning.

In this paper, we propose DEED, a **D**ata-**E**fficient adaptation based on **E**rror-**D**riven learning for code generation. DEED aims to alleviate the problem of poor code generation performance of fine-tuning LLMs in scenarios with few training data. Following the error-driven learning, our method proceeds in four steps: ❶ **Error Code Collection.** We identify and collect error codes generated by LLMs, aiming to mine the weaknesses of LLMs. ❷ **Automatic Code Revision.** To obtain revisions of error codes in a low-cost way, we design SELF-REVISE to realize automatic revision leveraging information in the original dataset and code execution feedback. ❸ **Model Optimization.** We optimize the LLMs using the revised code, making them focus on learning the revision of these critical errors, thereby improving the learning efficiency of LLMs. ❹ **Iterative Adaptation.** We adopt an iterative strategy, which involves repeating the preceding three steps, to continuously optimize and improve the performance of LLMs. Extensive experimental results demonstrate the superiority and generalizability of DEED in the data-efficient adaptation of LLMs for specific code generation scenarios. To summarize, the main contributions of this paper are:

- We propose error-driven learning for LLMs adaptation is better, i.e., utilizing revisions of LLMs' erroneous outputs for training has higher learning efficiency than original data.
- Based on the principle of error-driven learning, we propose a data-efficient adaptation method of LLMs for code generation, named DEED, which can effectively adapt model to specific scenarios with limited data.
- DEED outperforms the mainstream fine-tuning and prompting methods on three code generation datasets across various LLMs.

2 Methodology

In this section, we describe our proposed DEED in detail. Specifically, given a code generation scenario/domain with a limited-sample training dataset $\mathcal{D}_{train} = \{(r, c)\}$, where each data pair (r, c) consists of an input requirement r and an associated example of desired output code c. For a pre-trained LLM $\mathcal{M}_\theta$ with parameter θ, we aim to adapt $\mathcal{M}_\theta$ to the specific scenario of $\mathcal{D}_{train}$. DEED achieves data-efficient adaptation of LLMs through four steps: Error Code Collection (§2.1), Automatic Code Revision (§2.2), Model Optimization (§2.3), and Iterative Adaptation (§2.4). The overview of DEED and its differences from traditional fine-tuning are shown in Figure 2.

2.1 Error Code Collection

In this step, we systematically identify and collect erroneous output of LLMs using testing as criteria.

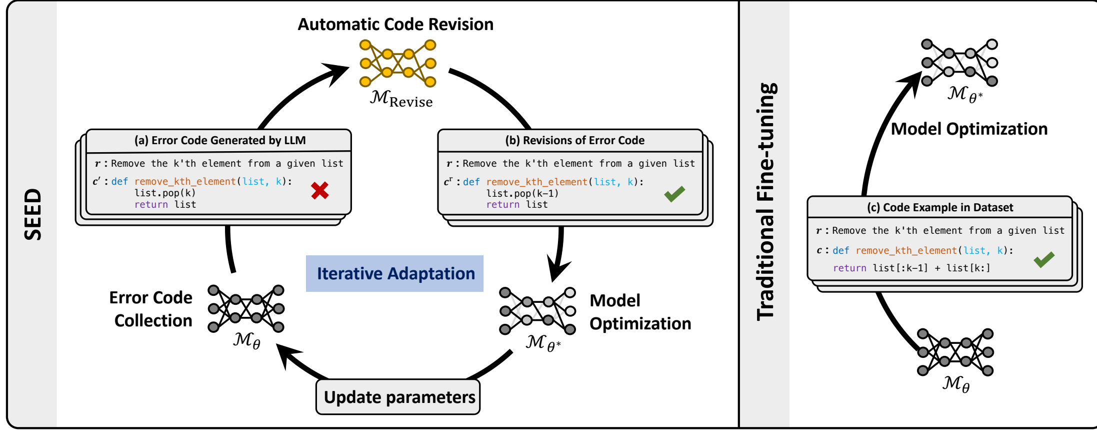

Figure 2: An overview of the proposed DEED and its differences from traditional fine-tuning methods.

168 We employ rejection sampling (Casella et al., 2004) to draw error code samples from the distribution produced by $\mathcal{M}_\theta$. For each requirement $r \in \mathcal{D}_{train}$, we sample

$$c' \sim \mathcal{M}_\theta(r) \mid \neg f, \quad (1)$$

where we sample multiple times and employ the criterion function f to determine the retention of the code sample. Specifically, the error code sample c' is retained when $f(r, c') = 0$, where $f(r, c') = 0$ if the rejection condition is satisfied, otherwise 1.

We define f as a test evaluation function since testing is the criterion for judging the correctness of code in practice:

$$\textsc{TestEval}(r, c') =:= \begin{cases} 0, & \text{if } c' \text{ fails } S_r, \\ 1, & \text{otherwise,} \end{cases} \quad (2)$$

where S_r is a suit of test cases under the requirement r and is equipped by code generation datasets. When collecting error codes for test failures, we can keep the failed test cases and error messages simultaneously for further error diagnosis.

To gain insights into the propensity of $\mathcal{M}_\theta$ to make certain errors, it is advisable to select error code sample c' for which the model demonstrates relatively high confidence. Therefore, among multiple error codes collected for the same r, we select the one with the highest generation probability [1].

2.2 Automatic Code Revision

In this step, we perform automatic revision for error codes using our Self-Revise method. Based on the LLM $\mathcal{M}_\theta$ itself, Self-Revise revises the error code by providing the information in the original dataset and pointing out the error with code execution feedback. Our objective is to derive a revised code that fixes the critical bug of the error code. As illustrated by examples (a), (b), and (c) in Figure 2, although there is already a correct code c in the dataset, it may differ significantly from the error code, leading to the critical bug being unclear. The pipeline of automatic code revision is shown in Figure 3.

Specifically, we leverage the following parts as the input of Self-Revise: 1) **Requirement (r)**: Clarify the requirement that needs to be addressed; 2) **Correct Solution (g)**: Provide a type of correct solution as a reference to reduce the difficulty of revision. The correct solution used here is the code sample c in the training dataset; 3) **Error Code (c′)**: Give the error code that needs to be revised. The error code is generated by $\mathcal{M}_\theta$ under r; 4)**Error Messages (m) and Failed Test Cases (t)**: Point out the error messages received during execution and the specific test cases where the error code fails, allowing for more focused troubleshooting and revision. These parts are combined as the input of Self-Revise according to the template:

$$T = \text{Template}(r, g, c', m, t) \quad (3)$$

where Template is shown in Figure 3.

Following previous work (Zhang et al., 2023a; Dong et al., 2023b), we use two settings for Self-Revise, i.e., fine-tuning (FT) and few-shot prompting (FSP), to get $\mathcal{M}_{\text{Revise}}$ for revising error codes.

Self-Revise (FT) entails the process of fine-tuning $\mathcal{M}_\theta$ with a small number of data for the purpose of automatic code revision. The training

[1] We determine the probability of the generated code by averaging the probabilities of each generated token.

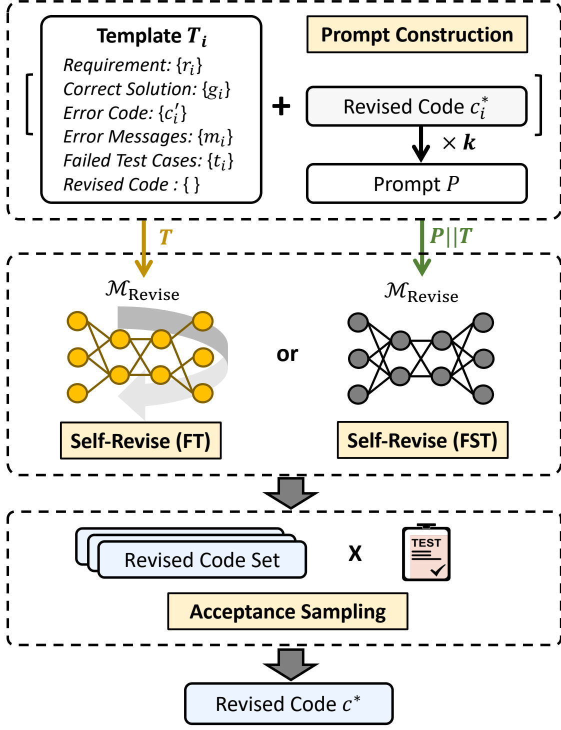

Figure 3: Illustration of automatic code revision

objective is to minimize $L(\theta)$:

$$L(\theta) = \sum_i l_{ce}(\mathcal{M}_\theta(T_i), c_i^*) \tag{4}$$

where l_{ce} represents the standard cross-entropy loss, and we update the parameters initialized with $\mathcal{M}_\theta$ to obtain $\mathcal{M}_{\text{Revise}}$ in SELF-REVISE (FT).

SELF-REVISE (FSP) adopts the few-shot prompting technique and leverages k examples of T_i and c_i^* to construct the prompt P for aligning $\mathcal{M}_\theta$ to automatic code revision. In SELF-REVISE (FSP), $\mathcal{M}_{\text{Revise}}(\cdot)$ is defined as $\mathcal{M}_\theta(P \mid\mid \cdot\,)$, where $||$ denotes the concatenation operation.

In contrast to the previous error code collection step, for each error code c', we construct T and use acceptance sampling to obtain the revised code c^*:

$$c^* \sim \mathcal{M}_{\text{Revise}}(T) \mid f. \tag{5}$$

where c^* is retained if $\text{TESTEVAL}(r, c^*) = 1$ in Eq. (2), i.e., the revised code c^* passes its test cases. We sample multiple times and it is sufficient if $\mathcal{M}_{\text{Revise}}$ could correctly revise the error code once. To prevent $\mathcal{M}_{\text{Revise}}$ from simply replicating the provided correct solution g, we exclude the output identical to g. Subsequently, for each requirement r, and select the version that is most similar to the error code among the remaining code revisions.

2.3 Model Optimization

In this step, we employ pairs of the requirement r and its revised code c^* to further fine-tune the model $\mathcal{M}_\theta$. This process leads to the enhanced version of $\mathcal{M}_\theta$, referred to as $\mathcal{M}_{\theta^*}$, in the specific scenario of dataset $\mathcal{D}_{train}$.

For fine-tuning (Devlin et al., 2019), we update all parameter θ of LLMs as

$$\theta^* = \arg\min_\theta \sum_{(r,c^*)} l_{ce}(\mathcal{M}_\theta(r), c^*), \tag{6}$$

When the computational resources are insufficient, we employ Low-Rank Adaptation (LoRA) (Hu et al., 2022) to fine-tune LLMs. For a weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA represents its update with a low-rank decomposition:

$$W + \Delta W = W + \Delta\alpha W_{\text{down}} W_{\text{up}}, \tag{7}$$

where α is a tunable scalar hyperparameter, $W_{\text{down}} \in \mathbb{R}^{d\times r}$, $W_{\text{up}} \in \mathbb{R}^{r\times k}$, and $r << min(r,k)$. In LoRA, we update parameter θ^* as

$$\theta^* = \theta + \Delta\theta, \tag{8}$$

$$\Delta\theta = \arg\min_{\Delta\theta} \sum_{(r,c^*)} l_{ce}(\mathcal{M}_{\theta+\Delta\theta}(r), c^*). \tag{9}$$

2.4 Iterative Adaptation

The preceding three steps can go through multiple iterations until a certain number of rounds is reached or the revised code no longer increases.

For l-th iteration that $l > 1$, we initialize its initial model $\mathcal{M}_{\theta_l}$ as the enhanced model of the previous iteration $\mathcal{M}_{\theta^*_{l-1}}$. Based on $\mathcal{M}_{\theta_l}$, we repeat the process in steps of error code collection and automatic code revision to sample error codes $\{c'\}_l$ and revised codes $\{c^*\}_l$, respectively. Inspired by experience replay (Mnih et al., 2015) in reinforcement learning, we use the union of collected data in each iteration $\{(r, c^*)\}_{1:l}$ to stabilize learning process and improve data utilization efficiency, that is,

$$\{(r, c^*)\}_1 \cup \cdots \cup \{(r, c^*)\}_i \cdots \cup \{(r, c^*)\}_l, \tag{10}$$

to update parameters in the model optimization step, thereby yielding the enhanced model of the l-th iteration $\mathcal{M}_{\theta^*_l}$. At each iteration, the model is trained to convergence.

This iteration is a step-by-step optimization designed to continuously improve the adaptability of models to the specific scenario. The complete process of DEED is summarized in Appendix, Algorithm 1.

3 Evaluation

We present extensive experiments that span three representative code generation datasets, two fine-tuning settings, and five different LLMs of varying sizes or series. We aim to investigate six research questions: 1) How does DEED perform compared to the mainstream baselines? 2) How does DEED perform when applied to various LLMs? 3) What kind of training data has the better training effect? 4) How does the number of iterations affect the effectiveness of DEED? 5) What is the impact of implementing the automatic code revision component of DEED in conjunction with alternative LLMs? 6) How does each input component of SELF-REVISE contribute to the effectiveness?

Datasets We use three public code generation datasets, i.e., MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and DS-pandas (Lai et al., 2023), to simulate the specific scenario with limited data. We sample $\min(200, 40\% * \mathcal{D})$ problems from all datasets as $\mathcal{D}_{train}$, while the remaining problems serve as $\mathcal{D}_{test}$.

Implementation Details We use a single A6000 GPU to conduct all experiments. Our base model is selected as CodeGen-2B (Nijkamp et al., 2023) by default, which is a well-known open-source LLM for code and is suitable for full fine-tuning within our computational resource constraints. $\mathcal{M}_\theta$ is initialized to our base model, and $\mathcal{M}_{\text{Revise}}$ is derived from $\mathcal{M}_\theta$ though SELF-REVISE (§2.2).

Metrics In final evaluations, we set temperature to 0.8 and generate $n = 50$ samples, which are then used to calculate unbiased Pass@k (Chen et al., 2021) in all experiments. All evaluation results are averaged over five test runs.

Detailed descriptions of datasets, implementation details, and metrics can be found in Appendix C, D, and E, respectively.

3.1 Effectiveness of DEED

Baselines. In this section, we evaluate the effectiveness of DEED by comparing it against other six methods, including **Direct Generation**, **Fine-tuning (Full)**, **Fine-tuning (LoRA)**, **Few-shot Prompting**, **Self-refine** (Madaan et al., 2023) and **Self-debug** (Chen et al., 2023e). Among them, Self-refine and Self-debug iteratively refine the generated code through prompting techniques. Considering some baselines involve full-parameter fine-tuning, CodeGen-2B is uniformly selected as the base model in this experiment. For DEED, we use $30\%{*}\mathcal{D}_{train}$ for SELF-REVISE (FT)[2], while the remaining $70\%{*}\mathcal{D}_{train}$ is employed for model optimization, where we use full-parameter fine-tuning.

Table 1: Pass@k (%) of DEED and baselines on MBPP, HumanEval, and DS-pandas datasets, and the teal number after ↑ denotes the relative improvement of DEED over the second-highest score.

Datasets	Method	Pass@1	Pass@5	Pass@10
MBPP	Direct Generation	15.6%	31.4%	40.2%
	Fine-tuning (Full)	25.8%	46.2%	57.6%
	Fine-tuning (LoRA)	19.8%	39.8%	55.2%
	Few-shot Prompting	24.4%	38.0%	49.4%
	Self-Refine	25.6%	38.8%	50.2%
	Self-Debug	20.2%	34.5%	40.6%
	DEED	**32.8%** (↑ 27.2%)	**46.8%**	**64.0%**
HumanEval	Direct Generation	24.8%	44.7%	51.8%
	Fine-tuning (Full)	29.8%	47.9%	56.4%
	Fine-tuning (LoRA)	27.4%	46.9%	53.9%
	Few-shot Prompting	25.2%	45.8%	53.1%
	Self-Refine	25.3%	45.2%	51.9%
	Self-Debug	26.4%	46.4%	54.2%
	DEED	**38.6%** (↑ 32.9%)	**54.7%**	**62.2%**
DS-pandas	Direct Generation	0.8%	3.1%	5.6%
	Fine-tuning (Full)	2.6%	6.5%	9.6%
	Fine-tuning (LoRA)	2.2%	6.0%	8.9%
	Few-shot Prompting	1.9%	4.5%	5.7%
	Self-Refine	2.1%	4.7%	5.8%
	Self-Debug	1.2%	2.8%	4.1%
	DEED	**5.3%** (↑ 103.9%)	**9.5%**	**12.3%**

Results. We conducted experiments on three public datasets, i.e., MBPP, HumanEval, and DS-pandas. The experimental results are summarized in Table 1. This comparison yielded four insightful observations: **1) Significant superiority of DEED**: Our proposed DEED performs significantly better than the other six baselines on the three datasets. Notably, DEED exhibits significant relative improvements of 27.2%, 32.9%, and 103.9%, respectively, when compared to the best-performing baseline Fine-tuning (Full). Self-refine and Self-debug underperform on small LLMs like codegen-2B. Self-debug excels over Self-refine only on the HumanEval dataset, where public test cases and results are available. On other datasets, Self-debug relies on the LLM's generated code explanations and feedback. Moreover, we find that DEED not only surpasses Self-refine and Self-debug in terms of performance but also in speed. These methods have a significant disadvantage in speed as they require iterative refinements for each sample. Their time cost is directly proportional to the number of samples and the number of iterations, while

[2] In addition to MBPP dataset, for other two datasets (i.e., HumanEval and DS-pandas), we generate one error code per sample in a subset comprising 30% of the training set, using CodeGen-2B. Subsequently, authors collaboratively apply the minimal necessary revisions to correct these error codes.

DEED is free of these two factors. **2) Worst performance of Direct Generation**: The performance of Direct Generation is significantly lower than the Fine-tuning (Full), Fine-tuning (LoRA), and Prompt baselines. This result suggests that directly applying LLMs for evaluation may be less suitable for specific scenarios, resulting in performance differences. **3) Fine-tuning (LoRA) is less effective than Fine-tuning (Full)**: Although Fine-tuning (LoRA) offers the advantage of reduced computational resource requirements for fine-tuning LLMs, it trades off the performance. **4) Less improvement of Few-shot Prompting**: Few-shot prompting is the most commonly used prompting technique, but its main limitation lies in its difficulty in imparting new knowledge or developing new capabilities in the model. It primarily assists the model in adjusting its outputs to better align with expected results, therefore its adaptability is limited.

3.2 The Effect of Different LLMs

Baselines. We employ several different series and sizes of representative LLMs to perform DEED, including **CodeGen-2B and CodeGen-6B** (Nijkamp et al., 2023), **Llama-7B** (Touvron et al., 2023), and **CodeLlama-7B** (Rozière et al., 2023). Among them, CodeGen-2B uses full fine-tuning, and the remaining LLMs with larger parameters use parameter-efficient fine-tuning with LoRA. Each LLM has four baselines, i.e., **Direct Generation**, **Fine-tuning**, and **Few-shot Prompting**.

Table 2: Pass@k (%) of DEED and baselines with different LLMs, and the teal number after ↑ denotes the relative improvement of DEED over Fine-tuning.

Models	Method	Pass@1	Pass@5	Pass@10
CodeGen-2B	Direct Generation	15.6%	31.4%	40.2%
	Fine-tuning (Full)	25.8%	46.2%	57.6%
	Few-shot Prompting	24.4%	38.0%	49.4%
	DEED (Full)	**32.8%** ↑ 27.2%	**46.8%**	**64.0%**
CodeGen-6B	Direct Generation	19.6%	40.2%	60.8%
	Fine-tuning (LoRA)	26.6%	46.8%	63.0%
	Few-shot Prompting	26.2%	45.2%	60.2%
	DEED (LoRA)	**33.4%** ↑ 25.6%	**47.4%**	**67.6%**
Llama-7B	Direct Generation	13.4%	29.8%	37.4%
	Fine-tuning (LoRA)	15.2%	27.4%	34.0%
	Few-shot Prompting	16.6%	26.2%	33.8%
	DEED (LoRA)	**22.0%** ↑ 44.7%	**30.4%**	**40.8%**
CodeLlama-7B	Direct Generation	20.4%	43.8%	52.8%
	Fine-tuning (LoRA)	19.9%	42.4%	53.2%
	Few-shot Prompting	27.8%	46.6%	64.8%
	DEED (LoRA)	**34.8%** ↑ 74.9%	**49.2%**	**65.8%**

Results. The results of applying DEED to different LLMs are shown in Table 2. From the results, we observed that DEED consistently achieves improvements across different series (CodeGen, Llama, and CodeLlama) and various sizes (2B, 6B, and 7B), outperforming the Direct Generation, Fine-tuning, and Few-shot Prompting baselines. This indicates that DEED generalizes well to different LLMs.

3.3 The Effect of Training Data Variants

Baselines. We investigate the influence of different training data on the final adapted model $\mathcal{M}_{\theta^*}$ to validate the effectiveness of using revisions of model's erroneous output for training. The different variants of training data include: **W/o Training** (Direct generation without any training data), **Raw $\mathcal{D}_{train}$** (All raw samples in $\mathcal{D}_{train}$), **$\mathcal{D}_{train} \cap$ DEED** (The samples of the same problem as DEED in $\mathcal{D}_{train}$), **$\mathcal{D}_{train} \cup$ DEED** (Include not only samples of problems obtained through SELF-REVISE, but also samples of other problems in $\mathcal{D}_{train}$), **Human-revised $\mathcal{D}_{train}$** (Samples obtained human revision), and **DEED** (Samples obtained through SELF-REVISE).

Table 3: Comparison of the effect of different training data variants.

Variants	Pass@1	Pass@5	Pass@10
W/o Training	15.6%	31.4%	40.2%
Raw $\mathcal{D}_{train}$	25.8%	46.2%	57.6%
$\mathcal{D}_{train} \cap$ DEED	22.4%	33.8%	42.8%
DEED $\cup$ $\mathcal{D}_{train}$	29.2%	44.2%	58.0%
Human-revised $\mathcal{D}_{train}$	28.0%	46.2%	59.8%
DEED	32.8%	46.8%	64.0%

Results. As shown in Table 3, we discover that: **1) DEED exceeds Raw $\mathcal{D}_{train}$, despite Raw $\mathcal{D}_{train}$ having more training data.** This proves that training using revisions produced by SELF-REVISE is more efficient compared to using samples in the dataset. **2) The effect of $\mathcal{D}_{train} \cap$ DEED is comparatively weaker**, which reveals that DEED is not simply improved by selecting better problems. **3) DEED $\cup$ $\mathcal{D}_{train}$ is not as effective as DEED**, which shows that some data in $\mathcal{D}_{train}$ have negative effects with limited data. **4) The performance of DEED surpasses that of the Human-revised $\mathcal{D}_{train}$.** This finding may be attributed to a disconnect between the revision made by humans and the model's learning expectations. While human revisions are applied to all code data in $\mathcal{D}_{train}$, some data may inherently be challenging for the current model. As such, forced learning from these data may have counterproductive effects, highlighting a potential limitation in human-revised $\mathcal{D}_{train}$.

451 ### 3.4 The Effect of Iterations

452 **Baselines.** We study the effect of iterations on DEED. We analyze the progression of DEED's effectiveness across different iterations, starting from 0 iterations (i.e., generated directly with LLMs) and extending to one, and up to four iterations.

Table 4: Performance of DEED with the different number of iterations.

Iterations	Pass@1	Pass@5	Pass@10	Num. of Revised Code
0	15.6%	31.4%	40.2%	-
1	31.6%	46.3%	60.6%	31 (+31)
2	32.8%	46.8%	64.0%	41 (+10)
3	33.0%	46.7%	62.6%	43 (+2)
4	33.2%	47.1%	64.0%	44 (+1)

457 **Results.** We conduct this experiment on MBPP dataset and its results are displayed in Table 4. From the results we can observe a trend: as the number of iteration rounds increases, the performance of DEED in Pass@1 shows an increasing trend, and the improvement is significant in the first two iterations, achieving over 98% Pass@1 performance within this period. At the same time, the amount of revised code in each iteration is also increasing, indicating that errors are continuously discovered, corrected, and learned. Considering that Pass@10 has oscillations from the 2nd iteration to the 4th iteration, we choose to end after the second iteration as the final performance of DEED.

471 ### 3.5 The Effect of Revision with Other LLMs

472 **Baselines.** We evaluate the performance of automatic code revision and the impact on the final model $\mathcal{M}_{\theta^*}$ obtained through DEED when using alternative LLMs to substitute the base model as $\mathcal{M}_{\text{Revise}}$. The base model is set to **CodeGen-2B** and alternative LLMs containing **CodeGen-6B**, **Llama-7B**, **CodeLlama-7B**, and **ChatGPT**. In this experiment, we obtain $\mathcal{M}_{\text{Revise}}$ in both fine-tuning and few-shot prompting settings for comparison, and $\mathcal{M}_{\theta^*}$ is consistently fixed as the base model.

482 **Results.** Table 5 illustrates the experimental results of automatic code revision based on different models, and we can observe that: **1) SELF-REVISE (FT) employing the same model as the base model yields the best performance of $\mathcal{M}_{\theta^*}$.** For baselines using other LLMs in fine-tuning, CodeLlama exhibits superior performance in terms of Pass@k in $\mathcal{M}_{\text{Revise}}$, but its final effectiveness is somewhat compromised. This limitation is attributed to the divergence in training data and architectural frameworks between CodeLlama and

Table 5: Comparison of automatic code revision based on different LLMs in both fine-tuning and few-shot prompting settings, where $\mathcal{M}_{\text{Revise}}$ is reported the raw results on the $70\% * \mathcal{D}_{train}$ part and $\mathcal{M}_{\theta^*}$ is fine-tuned with filtered results as described in §2.2.

Method	$\mathcal{M}_{\text{Revise}}$			$\mathcal{M}_{\theta^*}$	
	Pass@1	Pass@10	Pass@any	Pass@1	Pass@10
Few-shot Prompting					
CodeGen-6B	19.4%	60.1%	70.8%	26.8%	59.0%
Llama-7B	23.5%	67.7%	81.9%	20.8%	54.2%
CodeLlama-7B	20.2%	64.9%	75.0%	25.2%	59.6%
ChatGPT	61.4%	87.3%	92.1%	27.0%	62.4%
Base Model (SELF-REVISE (FSP))	18.9%	57.1%	69.4%	26.2%	58.2%
Fine-tuning					
CodeGen-6B	5.0%	20.3%	26.6%	29.4%	64.2%
Llama-7B	2.7%	8.5%	12.6%	23.2%	58.4%
CodeLlama-7B	5.1%	21.0%	34.6%	24.0%	60.2%
Base Model (SELF-REVISE (FT))	3.9%	18.9%	24.6%	**32.8%**	**64.0%**

493 the base model, leading to inconsistencies in the revised code with the base model's expectations. In contrast, CodeGen-6B, which is the same series of the base model with a large parameter, demonstrates slightly lower Pass@k in $\mathcal{M}_{\text{Revise}}$ than CodeLlama but still achieves commendable results for $\mathcal{M}_{\theta^*}$. **2) Although the Pass@k of SELF-REVISE (FSP) is higher than SELF-REVISE (FT) in $\mathcal{M}_{\text{Revise}}$, it does not perform as well on the ultimate $\mathcal{M}_{\theta^*}$.** We find this discrepancy may be due to the SELF-REVISE (FSP)'s tendency to learn superficial forms, i.e., it often resorts to copying code from the correct solution provided in the prompt, even when explicitly instructed not to in the prompt, as shown in Figure 5. Using ChatGPT as $\mathcal{M}_{\text{Revise}}$ results in substantially higher Pass@k compared to using the base model, does not significantly enhance the final model $\mathcal{M}_{\theta^*}$.

511 ### 3.6 Ablation Study on SELF-REVISE

512 **Baselines.** We further perform the ablation study to investigate the effectiveness of each input component in SELF-REVISE. Requirements and error codes are the indispensable basic inputs for performing automatic code revision. Therefore, we perform ablation experiments on the remaining three components, i.e., **correct solution**, **failed test cases**, and **error messages**. By removing these components individually, we observe their specific impact on the performance of automatic code revision and the final model, and thus evaluate the effectiveness of these components.

524 **Results.** We conduct the ablation study on MBPP dataset as shown in Table 6. First, we find that removing the failed test cases resulted in the largest

Table 6: Results of ablation study on SELF-REVISE.

Method	$\mathcal{M}_{\text{Revise}}$			$\mathcal{M}_{\theta^*}$	
	Pass@1	Pass@10	Pass@any	Pass@1	Pass@10
DEED	3.9%	18.9%	24.6%	**32.8%**	**64.0%**
- Correct Solution	3.4%	15.4%	19.8%	30.1%	61.9%
- Error Messages	3.1%	14.2%	17.3%	28.6%	58.7%
- Failed Test Cases	2.3%	5.1%	6.3%	26.1%	47.6%

drop in performance of all metrics. Failed test cases can demonstrate the inconsistency between the model-generated code output and the desired output, allowing LLMs to reason about and correct erroneous operations. Experimental results show that this point is most helpful for automatic code revision. Second, removing error messages or the correct code solution also results in a loss of performance. Error messages directly indicate surface errors in the generated code (such as syntax errors and runtime errors) and the location of the errors, which is also helpful for LLMs to revise the code. The correct code samples in the dataset can provide some reference for revising errors of LLMs, thus further reducing the difficulty of correction.

4 Related Work

Adaptation of LLMs. Numerous tasks rely on adapting LLMs to multiple downstream applications. Such adaptation is usually done via fine-tuning, which updates all the parameters of LLMs. Considering LLMs contain a large number of model parameters, performing full parameter tuning would be extremely expensive (Ding et al., 2023). Therefore, some parameter-efficient fine-tuning methods have been developed, including Adapter Tuning (Houlsby et al., 2019; Hu et al., 2023), Prompt Tuning (Lester et al., 2021; Liu et al., 2021b), Prefix Tuning (Li and Liang, 2021; Liu et al., 2021a), and Low-rank adaptation (Hu et al., 2022). They primarily optimize the efficiency of training model parameters but are not directly targeted at improving the efficiency of data usage. Another type of adaptation that does not require training is prompting (Liu et al., 2023), which depends on in-context learning (Dong et al., 2023a; Brown et al., 2020a). However, a limitation of them is that models often merely mimic the surface form of prompt, struggling to deeply understand or adapt to complex and abstract task requirements.

Our method is orthogonal to the aforementioned adaptation techniques, allowing for its concurrent application with these methods to enhance overall effectiveness.

Code Generation with LLM. The rise of pre-training techniques has brought new momentum to the field of code generation. Against this backdrop, LLMs such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022), AlphaCode (Li et al., 2022), CodeGeeX (Zheng et al., 2023) and CodeLlama (Rozière et al., 2023) have emerged, greatly enhancing the performance of code generation.

For LLMs-based code generation, there are some methods to refine the outputs produced by LLMs. Self-refine (Madaan et al., 2023) enables LLMs to provide feedback on and correct their own generated content. Self-debug (Chen et al., 2023e) allows the LLMs to explain and refine their generated code based on execution results. They belong to prompting methods that are constrained by input length and highly sensitive to prompts (Zhao et al., 2021). Moreover, Self-edit (Zhang et al., 2023a) involves training an additional editor. This category of methods treats refinement as a post-processing step after code generation, whereas we utilize a self-revise to assist model in efficient training and thereby enhance the model itself. Compared to these post-processing methods, DEED only requires test cases during training. When training is complete, DEED can be directly used without incurring any additional resource or time costs.

Recently, Chen et al. (Chen et al., 2023a) propose an ILF method focused on using human feedback to refine model results. However, it necessitates continuous human involvement and the provision of feedback throughout the model's training phase, which incurs significant costs in practical applications. Further, Chen et al. (Chen et al., 2023c) propose a distillation method that employs ChatGPT (OpenAI, 2022) to generate a large amount of refinement to train small models. However, this method presents two primary limitations. Firstly, it necessitates a highly performant "teacher" model, significantly surpassing the capabilities of the "student" model. Secondly, commercial constraints and other factors likely prohibit its implementation.

5 Conclusion

In this work, we have proposed DEED, a Data-Efficient adaptation with Error-Driven learning for code generation, substantially improving the code generation performance of LLMs in specific scenarios with limited data. We reveal that LLMs are more efficient in learning from the revisions of their errors than the original code samples in datasets.

6 Limitations

Our work has several limitations, which we aim to address in our future work:

First, Due to the constraints in computational resources, our experiments were merely conducted on LLMs with parameters less than 7B. In the future, we plan to extend our research to larger LLMs as more resources become available.

Second, considering that no public dataset is entirely unfamiliar to LLMs and sourcing high-quality data for such a scenario is challenging, we employ public benchmarks to simulate specific code generation scenarios. However, the adaptations of LLMs to these scenarios still achieve significant improvement.

Third, our method introduces additional overhead by collecting erroneous outputs and their revisions compared to using original training data, but it does not impact the efficiency of the actual inference process. Moreover, compared to the huge overhead of training LLM, this additional overhead is acceptable.

References

Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2002. Agile software development methods: Review and analysis.

David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. 1985. A learning algorithm for boltzmann machines. *Cogn. Sci.*, 9(1):147–169.

Armen Aghajanyan, Akshat Shrivastava, Anchit Gupta, Naman Goyal, Luke Zettlemoyer, and Sonal Gupta. 2021. Better fine-tuning by reducing representational collapse. In *ICLR*. OpenReview.net.

Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. 2024. Studying llm performance on closed-and open-source data. *arXiv preprint arXiv:2402.15100*.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020a. Language models are few-shot learners. In *NeurIPS*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020b. Language models are few-shot learners. In *NeurIPS*.

George Casella, Christian P Robert, and Martin T Wells. 2004. Generalized accept-reject sampling schemes. *Lecture Notes-Monograph Series*, pages 342–347.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *ESEC/SIGSOFT FSE*, pages 18–30. ACM.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2023a. Improving code generation by training with natural language feedback. *CoRR*, abs/2303.16749.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023b. CodeT: Code generation with generated tests. In *ICLR*.

Hailin Chen, Amrita Saha, Steven C. H. Hoi, and Shafiq Joty. 2023c. Personalised distillation: Empowering open-sourced llms with adaptive learning for code generation. *CoRR*, abs/2310.18628.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR*.

Meng Chen, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, Juhong Wang, and Xiaodong Gu.

2023d. On the effectiveness of large language models in domain-specific code generation. *CoRR*, abs/2312.01639.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023e. Teaching large language models to self-debug. *CoRR*, abs/2304.05128.

Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An empirical study on the usage of transformer models for code completion. *IEEE Trans. Software Eng.*, 48(12):4818–4837.

Jeffrey Dean and Sanjay Ghemawat. 2008. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nat. Mac. Intell.*, 5(3):220–235.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023a. A survey for in-context learning. *CoRR*, abs/2301.00234.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt. *CoRR*, abs/2304.07590.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *ICLR*. OpenReview.net.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *ICLR*. OpenReview.net.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. In *EMNLP*, pages 5254–5276. Association for Computational Linguistics.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *CoRR*, abs/2303.06689.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *EMNLP (1)*, pages 3045–3059. Association for Computational Linguistics.

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *ACL/IJCNLP (1)*, pages 4582–4597. Association for Computational Linguistics.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9):195:1–195:35.

Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. 2024. Delving into parameter-efficient fine-tuning in code change learning: An empirical study. *CoRR*, abs/2402.06247.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021a. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021b. GPT understands, too. *CoRR*, abs/2103.10385.

Ilya Loshchilov and Frank Hutter. 2017. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101.

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. In *NeurIPS*.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*. OpenReview.net.

OpenAI. 2022. ChatGPT.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Nayan B. Ruparelia. 2010. Software development lifecycle models. *ACM SIGSOFT Softw. Eng. years*, 35(3):8–13.

Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end javascript code generation. In *ESEC/SIGSOFT FSE*, pages 1533–1543. ACM.

Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In *ISSTA*, pages 39–51. ACM.

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, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Haoran Xu, Seth Ebner, Mahsa Yarmohammadi, Aaron Steven White, Benjamin Van Durme, and Kenton Murray. 2021a. Gradual fine-tuning for low-resource domain adaptation. In *Proceedings of the Second Workshop on Domain Adaptation for NLP*, pages 214–221.

Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang, and Fei Huang. 2021b. Raise a child in large language model: Towards effective and generalizable fine-tuning. In *EMNLP (1)*, pages 9514–9528. Association for Computational Linguistics.

Haojie Zhang, Ge Li, Jia Li, Zhongjin Zhang, Yuqi Zhu, and Zhi Jin. 2022. Fine-tuning pre-trained language models effectively by optimizing subnetworks adaptively. In *NeurIPS*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *ACL (1)*, pages 769–787. Association for Computational Linguistics.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pages 12697–12706. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

A Algorithm of DEED

The complete process of DEED is listed in Algorithm 1.

Algorithm 1 Pseudocode of DEED.

Require: Dataset $\mathcal{D}_{train} = \{(r, c)\}$, initial LLM $\mathcal{M}_{\theta}$.
Ensure: LLM $\mathcal{M}_{\theta^*}$.

1: Initial iteration index $l = 0$ and $\mathcal{M}_{\theta_{l+1}} = \mathcal{M}_{\theta}$.
2: *# Iterative Adaptation*
3: **repeat**
4: Update $l = l + 1$.
5: *# Error Code Collection*
6: Perform rejection sampling to collect error codes $\{c'\}_l$ based on $\mathcal{M}_{\theta_l}$ via Eq. (1) and (2).
7: *# Automatic Code Revision*
8: Perform acceptance sampling to collect revised codes $\{c^*\}_l$ based on $\mathcal{M}_{\theta_l}$ and SELF-REVISE via Eq. (2), (3), and (5).
9: Calculate the union of $\{(r, c^*)\}_{1:l}$ via Eq. (10).
10: *# Model Optimization*
11: Fine-tune $\mathcal{M}_{\theta_l}$ to yield $\mathcal{M}^*_{\theta_l}$ via Eq. (6) if the computational resources are sufficient, otherwise via Eq. (7), (8), and (9).
12: Update $\mathcal{M}_{\theta_{l+1}} = \mathcal{M}_{\theta^*_l}$.
13: **until** End condition is satisfied
14: **return** $\mathcal{M}_{\theta^*_l}$

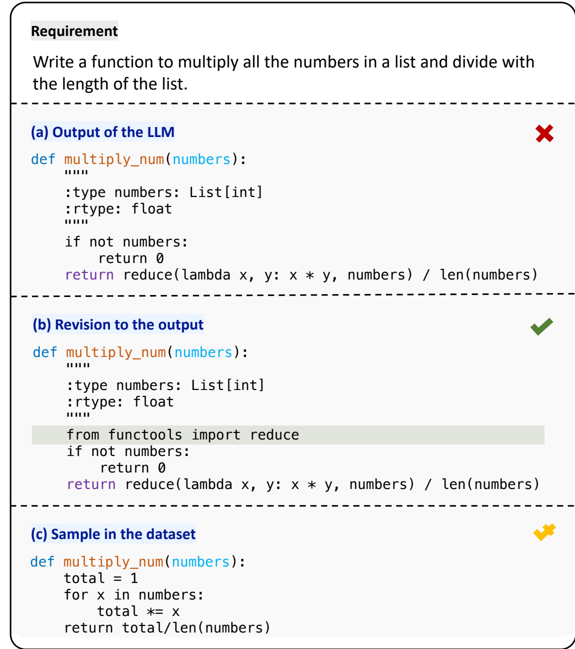

Figure 4: A motivation example of DEED.

B Motivation Example

Aligning LLMs with specific scenarios and addressing their unique challenges by learning samples in the dataset is difficult, especially when training data are limited. We present a motivation example in Figure 4 to clarify the advantages of using error-driven learning in the LLMs adaptation process of code generation.

By observing the output (a) generated by LLMs, we can find that LLMs generate a basically correct code that adopts a commonly-used function 'reduce' (Dean and Ghemawat, 2008). However, this code still fails due to a critical error: it does not take dependencies into account, i.e., without importing the correct libraries and functions. This observation demonstrates that LLMs have most of the capabilities to solve the problem, but also reveals a shortcoming in dealing with dependencies, which is related to the code generation scenario[3]. This shortcoming can be overcome just by bootstrapping LLMs to import the correct dependencies, as shown in revision (b). However, in traditional fine-tuning methods, it is challenging to overcome the shortcoming through learning samples in the dataset. Because sample (c) in the dataset proposed a different solution from output (a), it did not use the 'reduce' function. LLM needs to put in more effort to learn the new solution from scratch and also misses the opportunity to overcome its shortcomings. Furthermore, there is a potential risk when training LLMs with sample (c): LLMs may incorrectly infer that sample (c) is the optimal solution for this requirement, resulting in the omission of the Guard Clause "`if not numbers\n return 0`" in output (a). Omitting the Guard Clause is an inadvisable programming pattern, which is undesirable to learn. Due to the absence of the Guard Clause as a safeguard for handling edge cases, an error could occur in the edge case where the input list is empty. **Therefore, using revision (b) to train LLMs is a better choice, which allows LLMs to focus on and learn to solve the critical error, while simultaneously avoiding the inherent disadvantages of original data.**

Further, we explore the effectiveness of adopting error-driven learning from the perspective of model optimization. We consider the potential significant discrepancy between the model-generated output and the sample in the dataset. By learning the revisions of the model's erroneous outputs, we can find more effective navigation in the optimization process. This might provide a shorter, smoother

[3]Making LLMs generate code with dependencies that match the development environment can be viewed as a code generation scenario. The required dependencies are usually different in different development environments. For example, if the development environment is Python2, "reduce" is a built-in function, but if it is Python3, it must be imported from the standard library "functools" in order to be used.

Self-Revise (FSP)

Case I

Requirement

Write a function to perform index wise addition of tuple elements in the given two nested tuples.

Revision of error code

```
def add_nested_tuples(test_tup1, test_tup2):
-    return tuple(map(lambda x, y: (x[0] + y[0], x[1] + y[1]), test_tup1,
     test_tup2))
-
-if __name__ == "__main__":
-    import doctest
-    doctest.testmod()
+    return tuple(tuple(map(sum, zip(tup1, tup2))) for tup1, tup2 in
     zip(test_tup1, test_tup2))
```

Sample code in dataset

```
def add_nested_tuples(test_tup1, test_tup2):
    res = tuple(tuple(a + b for a, b in zip(tup1, tup2)) for
tup1, tup2 in zip(test_tup1, test_tup2))
    return (res)
```

Case II

Requirement

Write a function to check if the common elements between two given lists are in the same order or not.

Revision of error code

```
def same_order(l1, l2):
-    if len(l1) != len(l2):
-        return False
-    for i in range(len(l1)):
-        if l1[i] != l2[i]:
-            return False
-    return True
+    common_elements = set(l1) & set(l2)
+    l1 = [e for e in l1 if e in common_elements]
+    l2 = [e for e in l2 if e in common_elements]
+    return l1 == l2
```

COPY

Sample code indataset

```
def same_order(l1, l2):
    common_elements = set(l1) & set(l2)
    l1 = [e for e in l1 if e in common_elements]
    l2 = [e for e in l2 if e in common_elements]
    return l1 == l2
```

Self-Revise (FT)

Case I

Requirement

Write a function to extract all the pairs which are symmetric in the given tuple list.

Revision of error code

```
def extract_symmetric(test_list):
    symmetric_pairs = set()
    for i in range(len(test_list)):
-        for j in range(i+1, len(test_list)):
-            if test_list[i] == test_list[j]:
-                symmetric_pairs.add((test_list[i], test_list[j]))
+        if test_list[i][::-1] in test_list:
+            symmetric_pairs.add(tuple(sorted(test_list[i])))
    return symmetric_pairs
```

Sample code in dataset

```
def extract_symmetric(test_list):
    temp = set(test_list) & {(b, a) for a, b in test_list}
    res = {(a, b) for a, b in temp if a < b}
    return (res)
```

Case II

Requirement

Write a function to remove odd characters in a string.

Revision of error code

```
def remove_odd(str1):
    new_str = ""
    for i in range(len(str1)):
-        if i % 2 == 0:
+        if not i % 2 == 0:
            new_str += str1[i]
    return new_str
```

Sample code in dataset

```
def remove_odd(str1):
    str2 = ''
    for i in range(1, len(str1) + 1):
        if(i % 2 == 0):
            str2 = str2 + str1[i - 1]
    return str2
```

Figure 5: Cases for two settings of self-revise, where "-" and "+" respectively indicate lines of code before and after revision.

path to a good local minimum compared to learning from samples in the dataset, rather than attempting to direct it toward a distant area that may not align well with its existing knowledge or biases. We conduct the statistical analysis of the discrepancies in the model's latent representations[4]. The findings reveal that the average distance between the model's erroneous outputs and the dataset's samples is 12.35, whereas the average distance between the erroneous outputs and their revisions is significantly lower, at 6.39. **These experimental results suggest that within the model's representation space, revised codes are closer and similar to the erroneous output codes than the original code samples. This evidence lends support to our hypothesis of why the error-driven learning method is more efficient.**

Therefore, our work is determined to explore the use of error-driven learning to achieve a data-efficient adaptation method, aimed at enhancing the performance of LLMs in specific code generation scenarios.

C Detailed Datasets

MBPP (Austin et al., 2021) contains crowd-sourced Python programming problems, covering programming fundamentals. We selected the version in the work (Chen et al., 2023a), which consists of 276 problems and some generated error codes alongside their human-revised counterparts, thus facilitating subsequent experiments.

HumanEval (Chen et al., 2021) is a widely-used code generation benchmark, containing 164 handwritten programming problems, proposed by OpenAI. Each programming problem includes a function signature, a NL description, use cases, a correct solution in Python, and several test tests.

DS-pandas (Lai et al., 2023) comprises 291

[4]Specifically, on MBPP dataset, we obtain erroneous outputs of CodeGen-2B, revisions of the outputs, and samples in MBPP. We concatenate the requirements with their code, input them into CodeGen-2B, and extract the hidden representations from the model's final layer. Then, we compute the Euclidean distances within the model's representational space to quantify the disparities between these three elements.

data science problems utilizing Pandas libraries, sourced from real-world problems posted by developers on StackOverflow. This dataset can evaluate the ability of LLMs to utilize specific data-analysis libraries for code generation.

D Detailed Implementation Details

For full parameter fine-tuning, i.e., Fine-tuning (Full) (Devlin et al., 2019), we use the AdamW optimizer (Loshchilov and Hutter, 2017), with hyperparameters $\beta_1 = 0.9$ and $\beta_2 = 0.9$, accompanied by a linear learning rate schedule. The initial learning rate is set to 5e-6, with a batch size of 1 and gradient accumulation of 32 steps for training across 10 epochs. For parameter-efficient fine-tuning, i.e., Fine-tuning (LoRA) (Hu et al., 2022), the learning rate is set to 2e-4. Additionally, the rank r is adjusted to 128, and the scaling factor α is set at 8. All other hyperparameters remain aligned with Fine-tuning (Full). For few-shot prompting (Brown et al., 2020b), we set the number of examples in prompt to 4. All baselines in the experiments use consistent settings.

In the error code collection step (§2.1) and the automatic code revision step (§2.2), we use temperature (Holtzman et al., 2020; Ackley et al., 1985) sampling to generate multiple samples: 5 samples in the former and 30 in the latter, with the temperature set to 0.8. To obtain the final revised code in the automatic code revision step, we choose the one of revised code exhibiting the minimum Levenshtein distance (Levenshtein et al., 1966) to the error code. The number of iterations is set to 2.

E Detailed Metrics

Following the practice of real software development which utilizes testing for evaluation (Ruparelia, 2010; Abrahamsson et al., 2002), we employ the Pass@k (Li et al., 2022) metric to measure the functional correctness of the generated code by executing test cases. We use the unbiased version (Chen et al., 2021) of Pass@k, where $n >= k$ samples are generated for each problem, count the number of correct samples $c <= n$ which pass test cases and calculate the following estimator,

$$\text{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \quad (11)$$

For automatic code revision, we add the pass@any metric which refers to the percentage of tasks for which the model generates at least one correct code that passed all test cases.

F Case Study

We use the case study to qualitatively assess the effectiveness of automatic code revision (§2.2), i.e., SELF-REVISE (FSP) and SELF-REVISE (FT) employed by DEED, examples of which are presented in Figure 5. Upon manual inspection of the outcomes produced by SELF-REVISE (FSP), two prevalent modification patterns are identified. First, the removal of redundant code is a common alteration. This includes the deletion of unnecessary blocks such as “if name == ‘main’ ” and other test codes, which are often extraneous in the context of the desired output. Second, SELF-REVISE (FSP) exhibits a tendency to directly copy correct code samples from the prompt. In contrast, SELF-REVISE (FT) is capable of making minimal yet effective modifications to the model’s initial error code outputs, thereby generating the correct code. Based on the observations, SELF-REVISE (FT) is recommended as the more preferable method for automatic code revision within DEED.