Aligning CodeLLMs with Direct Preference Optimization

Anonymous ACL submission

Abstract

The last year has witnessed the rapid progress of large language models (LLMs) across diverse domains. Among them, CodeLLMs have garnered particular attention because they can not only assist in completing various programming tasks but also represent the decision-making and logical reasoning capabilities of LLMs. However, current CodeLLMs mainly focus on pre-training and supervised fine-tuning scenarios, leaving the alignment stage, which is important for post-training LLMs, underexplored. This work first identifies that the commonly used PPO algorithm may be suboptimal for the alignment of CodeLLM because the involved reward rules are routinely coarse-grained and potentially flawed. We then advocate addressing this using the DPO algorithm. Based on only preference data pairs, DPO can render the model rank data automatically, giving rise to a fine-grained rewarding pattern more robust than human intervention. We also contribute a pipeline for collecting preference pairs for DPO on CodeLLMs. Studies show that our method significantly improves the performance of existing CodeLLMs on benchmarks such as MBPP and HumanEval.

1 Introduction

The past few years have witnessed the rapid development of large language models (LLMs) (Touvron et al., 2023; Chowdhery et al., 2023; Achiam et al., 2023). LLMs have quickly been used in specific domains like medicine (Thirunavukarasu et al., 2023), laws (Sun, 2023), finance (Yang et al., 2023), etc. LLMs designed for solving coding tasks, referred to as CodeLLMs, are particularly noteworthy due to their potential to automate and streamline programming, including bug detection and code generation, thereby enhancing productivity (Li et al., 2023; Wei et al., 2023; Guo et al., 2024).

Current research on CodeLLMs primarily focuses on the accumulation of extensive code-related corpora for pre-training, as well as the collection of instruction-following datasets for supervised fine-tuning (Roziere et al., 2023; Li et al., 2023; Hui et al., 2024). Given that the alignment plays an important role in improving the reasoning ability of LLMs (OpenAI, 2024b), seminal works (Le et al., 2022; Liu et al., 2023; Dou et al., 2024) have proposed to enhance CodeLLMs through Proximal Policy Optimization (PPO) (Schulman et al., 2017). However, we argue that they suffer from limitations in the definition of reward functions. For example, they commonly assign a fixed reward of -1 to any code snippet containing syntax errors that cannot be compiled, irrespective of the error count. As a result, code snippets with a single syntax error are treated no differently from those riddled with multiple errors. This makes the reward model fail to capture the nuanced preference distinctions among various code snippets and hence cannot efficiently guide the alignment of CodeLLMs.

This work proposes to align the CodeLLMs with the help of Direct Preference Optimization (DPO) (Rafailov et al., 2023). Unlike PPO, DPO does not explicitly define a reward model to capture preference, but alternatively uses model likelihood to represent that. By learning by comparing data pairs, DPO can automatically acquire the fine-grained differentiation between samples from coarse rewarding signals. Ideally, after training, a code snippet with few errors can be assigned a higher reward than that containing more errors. Compared to defining fine-grained, hand-crafted reward rules for better PPO, our DPO approach enjoys higher flexibility and can reduce the risk of reward hacking associated with flawed hand-crafted reward rules.

Given that using data pairs from other models for DPO can be sub-optimal and even lead to model degradation (Yan et al., 2024; Lai et al., 2024), we propose to construct on-policy preference data for DPO training, which distinguishes us from Weyssow et al. (2024). Specifically, we introduce external code executors to provide feedback for ranking code generations. An overview of this is depicted in Fig. 2. Empirically, our method has demonstrably increased the performance of CodeQwen1.5 7B on MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021), enhancing the scores from 0.783 to 0.804 and 0.829 to 0.878, respectively.

2 Methodology

2.1 Issues of PPO for CodeLLMs

Let $\pi_\theta(y|x)$ denote a large language model (LLM), which generates response y given the user instruction x. Demonstrated by Achiam et al. (2023), PPO (Schulman et al., 2017) algorithm has been the most powerful algorithm to align LLMs for performance enhancement.

Question: Given an integer array nums, return 0 if the sum of the digits of the minimum integer in nums is odd, or 1 otherwise.

Response 1:

```
class Solution:
    def sumOfDigits(self, nums: List[int]) -> int:
        return int(sum(map(int, str(min(nums)))) % 2 == 1)
```

Response 2:

```
class Solution:
    def sumOfDigits(self, nums: List[int]) -> int:
        return min(nums)%2==1
```

DPO reward difference:	**1.0469**	**Fine-grained reward**
Rule-based reward difference:	**0**	**Coarse reward**

Question: Reversing an integer means to reverse all its digits. Given an integer num, reverse num to get reversed1, then reverse reversed1 to get reversed2. Return true if reversed2 equals num. Otherwise return false.

Response 1:

```
class Solution:
    def isSameAfterReversals(self, num: int) -> bool:
        return not num or num % 10
```

Response 2:

```
class Solution:
    def isSameAfterReversals(self, num: int) -> bool:
        if num == 0: return True
        return num % 10
```

DPO reward difference:	**0.7376**	**Fine-grained reward**
Rule-based reward difference:	**0**	**Coarse reward**

Figure 1: Two cases for illustration of the reward difference of different responses given by the DPO reward and rule-based reward. When given a code question, response 1 and response 2 are two responses that have logic errors but the two responses are not the same. Reward difference means the reward of response 1 minus that of response 2. The rule-based reward assigns the same reward to different responses while DPO recognizes the reward difference between the different responses.

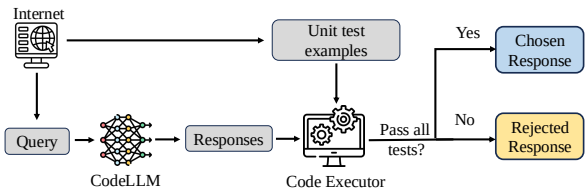

Figure 2: The pipeline of using execution feedback from the code executor to construct preference dataset.

Concretely, PPO maximizes the following objective:

$$\max_{\theta} J_r(\theta) = \max_{\theta} \sum_{x} \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \left[r(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_0(y|x)} \right], \tag{1}$$

where r is the reward function,and β is a hyperparameter to control the degree of deviation between the policy model π_θ and the reference model π_0.

The effectiveness of PPO is closely linked to the quality of the reward function r. For coding tasks, we can naturally utilize the execution feedback provided by an external code executor to characterize r, based on specific rules. For example, Le et al. (2022) define the following rewarding rule:

$$r(x,y) = \begin{cases} +1, & \text{if } y \text{ passed all unit tests} \\ -0.3, & \text{if } y \text{ failed any unit test} \\ -0.6, & \text{if } y \text{ happened runtime error} \\ -1, & \text{if } y \text{ happened compile error.} \end{cases} \tag{2}$$

Namely, the reward score is assigned based on the actual excitation state of the model response y.

Despite being widely adopted (Liu et al., 2023; Shojaee et al., 2023; Dou et al., 2024), the above rule can be too coarse and the reward space is very sparse. However, the exploration of complex tasks like coding in an environment characterized by sparse reward is particularly challenging. For instance, one response may contain plenty of syntax errors, while another may contain only one, yet both the two responses would receive identical rewards. This may cause considerable confusion for the model to align. Ideally, we expect the reward model to assign differentiated scores that accurately reflect the varying levels of quality among the responses.

2.2 DPO for CodeLLMs

Instead of exploring designing more fine-grained rewarding rules in a costly trial-and-error manner, this paper proposes to utilize DPO (Rafailov et al., 2023) to align CodeLLMs more efficiently and reliably. Technically, DPO operates on pre-collected preference data pairs and solves the following problem:

$$\mathcal{L} = -\sum_{(x,y^+,y^-)} \log \sigma \left[\beta \log \frac{\pi_\theta(y^+|x)}{\pi_0(y^+|x)} - \beta \log \frac{\pi_\theta(y^-|x)}{\pi_0(y^-|x)} \right], \tag{3}$$

where σ denotes the sigmoid function, and y^+ and y^- denote the preferred (i.e., chosen) response and the less preferred (i.e., rejected) one corresponding to the input x respectively. The term within the sigmoid function (i.e., subtraction between log-ratios) corresponds to an implicit reward difference learned by the model (Rafailov et al., 2023).

Model	MBPP	MBPP+	HumanEval	HumanEval+
CodeQwen1.5 7B	0.783	0.667	0.829	0.774
+ RFT	0.767	0.667	0.793	0.738
+ DPO	**0.804**	**0.688**	**0.878**	**0.829**
DeepSeek-Coder 6.7B	0.743	0.651	0.768	0.707
+ RFT	0.717	0.635	0.768	0.701
+ DPO	**0.765**	**0.677**	**0.787**	**0.732**
CodeLlama-Instruct 7B	0.548	0.458	0.390	0.323
+ RFT	0.556	0.455	0.360	0.305
+ DPO	**0.571**	**0.466**	**0.427**	**0.348**

Table 1: The pass@1 perfomance of the three CodeLLMs on MBPP, MBPP+, HumanEval and HumanEval+ benchmarks. +RFT/+DPO means the CodeLLM is trained using the RFT/DPO algorithm. The **bold** indicates the optimal value.

Model	MBPP	MBPP+	HumanEval	HumanEval+
CodeQwen1.5 7B	0.783	0.667	0.829	0.774
on-policy DPO	**0.804**	**0.688**	**0.878**	**0.829**
off-policy DPO	0.799	0.683	0.860	0.799
DeepSeek-Coder 6.7B	0.743	0.651	0.768	0.707
on-policy DPO	**0.765**	**0.677**	**0.787**	**0.732**
off-policy DPO	0.735	0.640	0.774	0.695

Table 2: The pass@1 performance of CodeLLMs after on-policy and off-policy DPO training on several benchmarks. "on-policy DPO" means the data used for training is generated by the policy model itself. "off-policy DPO" means the data used for training is generated by another model. The **bold** indicates the optimal value.

We can observe that the learning of DPO does not hinge on the exact values of the reward function r but only needs the preference data pairs, which can be easily constructed given a coarse rewarding rule. Besides, thanks to the learning-to-rank pattern, DPO can form a fine-grained characterization of the preference difference after witnessing adequate, various data pairs. Fig. 1 provides an example of this. In particular, we collected two incorrect responses for a certain coding problem and we expect the different responses can be assigned different rewards. As shown, DPO can differentiate between the two responses while the hand-crafted rules assign the same reward to them.

Details for preference pair construction. We construct a set of (chosen, rejected) pairs for the training of DPO on CodeLLMs. As illustrated in Fig. 2, we first aggregate a substantial number of queries (i.e., x) and unit test examples from the Internet, including competitive programming platforms like LeetCode[1] and Codeforces[2]. Subsequently, each query is input into the CodeLLM to generate eight distinct responses, from which we extract code snippets. The code and the unit test examples are then submitted to the code executor for evaluation. We assess whether the output of the code executor aligns with the ground truth delineated in the unit test examples. If the output is consistent with the ground truth, the response is classified as chosen; otherwise, it is deemed rejected. For each query, we randomly select a single pair of chosen and rejected responses, yielding roughly 3,000 triples (x, y^+, y^-) to support the training of DPO.

[1] https://leetcode.com
[2] https://codeforces.com

3 Experiments

3.1 Experimental Details

CodeLLMs of concern. We selected CodeLLMs that have outstanding performance and have garnered wide attention from the research community, as the SFT model for further alignment. Specifically, we choose CodeQwen1.5-Instruct 7B (Bai et al., 2023), CodeLlama-Instruct 7B (Roziere et al., 2023), and DeepSeek-Coder-Instrcut 6.7B (Guo et al., 2024) as the target models to conduct our experiments.

Benchmarks and evaluation. In order to accurately assess the coding capabilities of CodeLLMs, we selected the most commonly used MBPP (Mostly Basic Programming Problems) (Austin et al., 2021) and HumanEval (Chen et al., 2021) benchmarks. HumanEval comprises 164 Python problems, validated through test cases to evaluate the code produced by CodeLLMs in a zero-shot setting. Similarly, MBPP features 500 problems assessed in a few-shot setting. Further, MBPP+ and HumanEval+ by Liu et al. (2024) added dozens of times more unit test cases to the original MBPP and HumanEval benchmarks, which better reflects the ability of the CodeLLMs to truly understand and solve the coding questions. We also report the performance of CodeLLMs on these two benchmarks. To ensure stable and fair comparisons, for each benchmark, we utilize

the greedy decoding strategy, letting the CodeLLMs generate responses to the input code questions. We report the pass@1 performance across the benchmarks to illustrate the coding capabilities of the CodeLLMs.

3.2 Main Results

To validate the effect of DPO training, we compare the performance of CodeLLMs that use rejection sampling fine-tuning (RFT) (Zelikman et al., 2022; Yuan et al., 2023) and DPO for training, respectively. Specifically, we use the chosen responses from the constructed preference dataset for RFT training and both the chosen and rejected responses from the constructed preference dataset for DPO training. All training hyperparameters can be acquired from the Appendix B. As illustrated in Table 1, while the RFT algorithm does enhance the model's performance, its improvements are not as significant as those realized through the DPO algorithm. This discrepancy arises because, during the optimization process with the DPO algorithm, the model is able to learn from rejected responses, enabling it to avoid generating undesirable patterns during inference and ultimately reduce errors in the generated code snippets.

3.3 On-policy DPO vs. Off-policy DPO

Recently, there have been some researchers who also applied DPO in the field of CodeLLM. Weyssow et al. (2024) leverage the Magicoder Evol-Instruct dataset (Wei et al., 2023) as the source of queries. For each query, they randomly select four LLMs from fourteen LLMs to answer the coding problem and leverage ChatGPT to rate the four responses. Then they can get the preference dataset according to the rate from ChatGPT and the constructed dataset will be used for DPO training. However, using data generated by other models for DPO training is an off-policy mode. We want to emphasize that whether the preference data for DPO training is on-policy or not is very important. As analyzed by (Lai et al., 2024; Yan et al., 2024), off-policy training may lead to the degradation of the model to be optimized. Therefore, directly using the responses generated by other models for DPO may not be a good choice. The recommended choice is to construct the on-policy dataset using the policy model.

To validate the argument, we set up our experiments as follows: We first used CodeQwen1.5 7B and DeepSeek-Coder 6.7B to construct preference datasets, respectively. In the DPO training stage, we exchange the training data between Deepseek-Coder and CodeQwen1.5. In other words, we use data generated by one model to train another model. The performances of CodeLLMs after DPO training are shown in Table 2. As shown, using off-policy data leads to sub-optimal results. Notably, when we use preference data generated by CodeQwen1.5 to train the DeepSeek-Coder, the performance after training is even worse than the original instruct model, which indicates that using off-policy data for DPO training might be harmful to the model.

4 Related Works

4.1 CodeLLMs

With the rapid development of LLMs, domain models specifically designed for the field of coding are continuously emerging, providing great convenience for humans. Several pre-trained LLMs have demonstrated significant potential for code generation, including Santacoder (Allal et al., 2023), CodeGPT (Lu et al., 2021), etc. Moreover, CodeLLMs that underwent fine-tuning demonstrated more competitive performance, with standout models including Starcoder (Li et al., 2023), CodeLlama (Roziere et al., 2023), Wizardcoder (Luo et al., 2023), DeepSeek-Coder (Guo et al., 2024), and Qwen-Coder (Hui et al., 2024). However, when compared with the state-of-the-art CodeLLMs such as Claude-3.5-Sonnet (Anthropic, 2024) and GPT-4o (OpenAI, 2024a), the capabilities of these models lag significantly behind. A reason is that these CodeLLMs heavily rely on pre-train and SFT, either lacking alignment or not performing well.

4.2 Execution Feedback from Code Executor

Reinforcement learning from human feedback has proven to be effective (Achiam et al., 2023), but it is highly labor-intensive. Fortunately, the unique nature of coding tasks allows us to leverage execution feedback from a code executor to evaluate whether the code generated by CodeLLMs meets the problem's requirements. For example, some works use execution feedback from the code executor in inference time, for instance, Zhong et al. (2024) leverage the feedback signal from the code executor to help locate the bug in the code snippet, and iteratively refine the generated code until it passes all test examples. Some works leverage the execution feedback from the code executor to provide a reward signal according to certain rules (Liu et al., 2023; Dou et al., 2024) and then PPO is applied to align the CodeLLMs for better performance. In this paper, we propose an alternative approach to leveraging execution feedback: utilizing it to construct data for DPO training.

5 Conclusion

In this paper, we highlight that current CodeLLMs primarily focus on the pre-training and supervised fine-tuning stages, while neglecting the potential of the alignment stage. The existing works on using PPO to align CodeLLMs may suffer from the issue of coarse reward definition. Therefore, we propose an approach to further enhance the ability of CodeLLMs by leveraging the execution feedback from the code executor to construct a preference dataset for DPO training. Moreover, we have also demonstrated that, for coding tasks, compared to off-policy DPO, it is more beneficial to adopt on-policy DPO. In conclusion, this work proposes a practical method that can be directly used to improve the coding performance of models.

6 Limitations

One limitation of this work is that the relatively small number of coding problems available on the internet restricts us to constructing a limited set of preference data pairs for DPO training. Due to this constraint, we were unable to investigate the impact of the size of the training data on the model's final performance during the alignment phase. Future work can be done by exploring synthesizing coding questions for data augmentation.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Anthropic. 2024. Introducing claude. https://www.anthropic.com/claude.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv:2406.18629*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

OpenAI. 2024a. Hello gpt-4o. https://openai.com/index/hello-gpt-4o.

OpenAI. 2024b. Introducing openai o1. https://openai.com/o1.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Zhongxiang Sun. 2023. A short survey of viewing large language models in legal aspect. *arXiv preprint arXiv:2303.09136*.

Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine*, 29(8):1930–1940.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Martin Weyssow, Aton Kamanda, and Houari Sahraoui. 2024. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032*.

Yuzi Yan, Yibo Miao, Jialian Li, Yipin Zhang, Jian Xie, Zhijie Deng, and Dong Yan. 2024. 3d-properties: Identifying challenges in dpo and charting a path forward. *arXiv preprint arXiv:2406.07327*.

Hongyang Yang, Xiao-Yang Liu, and Christina Dan Wang. 2023. Fingpt: Open-source financial large language models. *arXiv preprint arXiv:2306.06031*.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Chuanqi Tan, and Chang Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D Goodman. 2022. Star: Self-taught reasoner bootstrapping reasoning with reasoning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pages 15476–15488.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 851–870.

Hyperparameter	Value
Training epochs	1
Learning rate	5e-6
Learning rate schedule	cosine
warmup ratio	0.05
batch size	16

Table 3: Hyperparameters of RFT.

Hyperparameter	Value
Training epochs	1
Learning rate	5e-6
Learning rate schedule	cosine
warmup ratio	0.05
batch size	16
beta	0.1

Table 4: Hyperparameters of DPO.

A The Underlying Reward of DPO

The DPO algorithm drives that if π_θ can maximize the Eq. (1), then the underlying reward can be given by:

$$r(x, y) = \beta \log \frac{\pi_\theta(y \mid x)}{\pi_0(y \mid x)} + C(x), \tag{4}$$

where $C : \mathcal{X} \to \mathbb{R}$ is a scalar function. π_θ is the policy model and π_0 is the reference model.

B Hyperparameter Setting

The hyperparameters used for RFT and DPO are shown in Table 3 and Table 4, respectively.