Improving Llama-2-7B Code Generation for Python Code Using LoRA

Yixiao Zeng
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
`yixiaz8@uci.edu`

Xiaofan Lu
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
`xiaofl14@uci.edu`

Zixu Yu
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
`zixuy@uci.edu`

Xucheng Zhao
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
`xuchez2@uci.edu`

Pengxuan Wu
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
`pengxuaw@uci.edu`

Abstract

Large Language Model (LLM) has demonstrated significant natural language understanding and generation capabilities. However, its performance in generating syntactically and semantically correct Python code can be further optimized. While traditional Supervised Full Fine-tuning (SFT) could enhance overall performance, it also comes with a high computational cost. LoRA, one of the Parameter Efficient Fine-Tuning (PEFT) method, offers a solution by allowing efficient fine-tuning with fewer parameters, thereby maintaining computational efficiency while improving model accuracy and reliability. We conducted extensive experiments to compare the performance of the PEFT-enhanced Llama-2-7B (fine-tuning based instruction-tuning on high quality programming question) with its baseline version across various LeetCode problems. The results show a marked improvement in code generation quality, including higher accuracy in syntax and increased success rates in automated code execution. We also analyzed the outputs of LoRA fine-tuning model and provided potentially future improvement directions.

1 Introduction

LLMs have revolutionary productivity in modern workflows, particularly in code generation. Both closed-source leading large models [1] and small models optimized for code generation, such as DeepSeeker Coder [4] and Code-Llama[13] have shown the ability to outperform human programmers in some domains. However, those more widely used open-source models still focus on traditional tasks. They do little training and optimization for code generation tasks. Furthermore, existing LLMs have excelled in evaluation metrics such as HumanEval[3] and MBPP[2]. There remains a significant gap in their support for generating code in more specialized and complex questions such as algorithms and data structure.

Preprint. Under review.

In this report, we focus on improving the performance of Llama-2-7B[16], a powerful open-source LLM, specifically for Python code generation. In addition, in order to reduce the huge amount of computation required for traditional fine-tuning, we chose to use the PEFT technique to reduce the amount of computation required for fine-tuning while guaranteeing the fine-tuning effect. After comparing several PEFT methods, such as P-Tuning[9], IA3[8], Prefix Tuning[6], and LoRA[5], we decide to go with LoRA, which offers relatively the best fine-tuning results and a smaller total size of weights based on previous research[19].

As for the choice of training dataset, distinguishing it from the traditional use of open source GitHub Repository or forum data such as Stackoverflow, our data is a mix of more challenging competition and straightforward traditional programming problems. In addition, we transformed them into the question-answer pair of instruction tuning[18], with appropriate prompting engineering, to maximize the fine-tuning effect as well as reduce the probability of the model generating redundant content (non-code content)

In addition, we introduce a new set of test methods to solve the status quo of simpler difficulty and single metrics of traditional tests. We randomly select some questions from Leetcode with three difficulty levels (easy, medium, hard). Also, we categorized the common types of problems, such as dynamic programming and graph theory, into nine distinct sections.

In the analysis section, we compare the original model with model after LoRA fine-tuning and analyze the difference between outcomes as well as the potential reasons behind it. Finally, we discuss some directions for future improvements.

2 Data

Our fine-tuning datasets include:

1. **CodeContests**: A dataset introduced by AlphaCode[7]. It comprises problems from well-known high-quality coding competition websites such as Codeforce and AtCoder. This dataset offers a rich source of complex algorithms and data structure problems. Among 13328 pairs, we filter 8139 pairs containing at least one Python 3 answer.
2. **Evol-Instruct-Code-80k-v1**: It is used by one of the State-Of-The-Art (SOTA) code generation models, WizardCoder[10]. Similar to the Alpaca dataset[15] generated by Self-Instruct technique[17], it builds on the idea of evolving existing instructional data to create a more complex and diverse dataset. We select 26588 input-output pairs whose answer of the code contains the tag "python." This dataset ensures the breadth of the entire fine-tuning data.

All of the selected dataset pairs are organized into a JSON file. Next, based on a series of prompt engineering skills such as Role-Play[14], as well as instruction tuning ideas, we construct the following question prompt:

```
prompt = "Below is an instruction that describes a task.\nYou are an
    AI program expert. Your task is to solve programming problems from
     interviews and coding contests only using Python in detail.\n###
    Instruction:" + example["question"] + "\n### Response:\n"
```

We also tokenize the target text (answer) and appends the tokenized target to the input IDs and attention mask.

3 Methods and Implementation

3.1 LoRA

LoRA (Low-Rank Adaptation) is a technique focused on optimizing the fine-tuning process of LLMs by only updating a small number of parameters α where Ω is the entire set of the parameters and $\alpha \subseteq \Omega$. Using low-rank matrices gives advantages to compact representations and reduces complexity. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$ LoRA construct a low-rank decomposition $W_0 + \Delta W = W_0 + BA\alpha$, where $B \in \mathbb{R}^{d \times r}$ has the size of $d \times r$ where d is the dimension of input.

$A \in \mathbb{R}^{r\times k}$ has the size of $r \times k$ where k is output dimension. Normally, matrix A has a Gaussian normalization, and the rank $r \ll \min(d, k)$.α scale the product AB, determine how much influence the low-rank adaptation in the original weight.

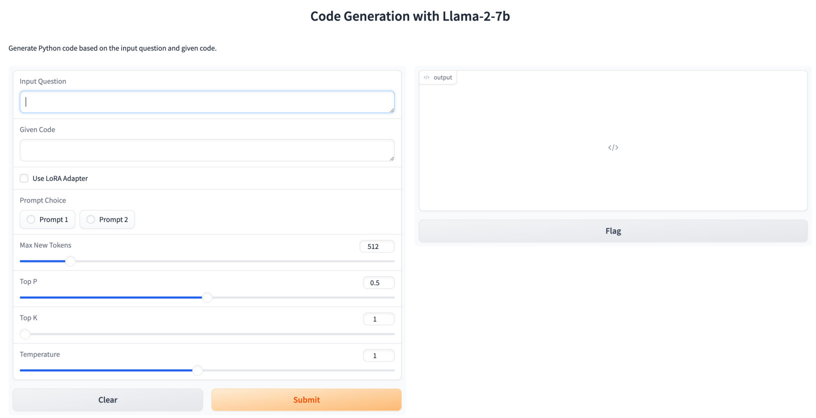
3.2 Fine-tuning and Inference

In our fine-tuning section, we set the LoRA r value to 8, the α value to 16, and the dropout value to 0.05. We also set number of epoch to 4, learning rate to 0.0001 and use Adam optimizer. Based on the calculations, our total training parameters (6.2M) are only 0.0933% of the model's total parameters (around 7B). This is undoubtedly a significant reduction in fine-tuning. We used an NVIDIA RTX 4090 24GB for fine-tuning. Since we adjusted the precision of the model to its original training precision of bfloat16 and implementing DeepSpeed ZeRO[12], we do not need any quantization to fine-tune Llama 2-7B. This also ensures that the results are not affected by the potentially inference degeneration of any quantization method.

During the inference stage, we incorporate the prompt engineering skills. We insert inputs and given LeetCode starting code into the following prompt:

```python
prompt = "You are an high-level AI program expert. Your task is to
    solve programming problems from interviews and coding contests
    only using Python in detail. Given INSTRUCTION, finish up the
    RESPONSE part in detail based on given code:\n###INSTRUCTION: {
    input_question}\n###RESPONSE:\n# Given Code\n{given_code}\n"
```

In the prompt, we setup some directives to limit the model's output ranges so that it doesn't generate extra contents beside the Python code. We also use some keywords such as "in detail" to improve inference quality. Finally, we develop a set of Gradio-based GUI interfaces, which improved the efficiency of our inference.



Code Generation with Llama-2-7b

Generate Python code based on the input question and given code.

Input Question

Given Code

Use LoRA Adapter

Prompt Choice: Prompt 1, Prompt 2

Max New Tokens 512

Top P 0.5

Top K 1

Temperature 1

Figure 1: Gradio Interface

4 Result and Analysis

4.1 Result

The benchmarks commonly used to evaluate large language models (LLMs) are HumanEval and MBPP. These datasets measure functional correctness in synthesizing programs from docstrings, assessing language comprehension, algorithms, and simple mathematics, with tasks comparable to basic software interview questions. However, these datasets do not categorize problems by difficulty, which limits their ability to demonstrate model performance across different complexity levels. Additionally, each problem in these datasets has a limited number of test cases, averaging around four per problem. This small number of test cases is insufficient for thoroughly testing code robustness and correctness.

To address these limitations, we propose a new evaluation method. We select 100 LeetCode questions, categorize them by difficulty, and test the model-generated code against the original LeetCode test cases, which average around 120 test cases per question. With the exponential increase in test cases, we also create an evaluation metric based on the number of test cases passed rather than the traditional pass-at-k tries approach.

To evaluate the performance of our fine-tuned model, we compared its code generation capabilities with the original Llama 2 7B model on a set of LeetCode problems. Specifically, we focused on the number of test cases passed by each generated solution. Our evaluation metric P_{model} is defined as follows:

$$P_{\text{model}} = \frac{1}{Q} \sum_{i=1}^{Q} \frac{C_i}{T_i} \tag{1}$$

where Q is the total number of LeetCode questions evaluated. C_i is the number of test cases the model passes on the i-th question. T_i is the total number of test cases for the i-th question.For this evaluation, we generated 100 inference outputs for each LeetCode question using the fine-tuned and original Llama 7B models. We then calculated the average number of test cases passed by each model using the metric defined above.

Table 1: Comparison of different techniques in pretrained and finetuned models

	Array	Binary	DP	Graph	String	Matrix	Linked List	Tree	String
Pretrained	0.7%	29%	10%	0%	5%	0%	0%	22%	22%
Finetuned	36%	33%	2%	0.7%	11%	7%	3.8%	23%	23%

As shown in Table 1, our fine-tuned model demonstrates a significant improvement in passing test cases compared to the original Llama 7B model, highlighting the effectiveness of our fine-tuning approach.

The evaluation results indicate that the fine-tuning process has successfully enhanced the model's ability to generate correct Python code. The fine-tuned model's higher average test case pass rate suggests improved accuracy and reliability in code generation, making it a more effective tool for automated coding tasks.

4.2 Analysis

Besides testing our model on LeetCode, we also ran both the original and fine-tuned models on single-pass HumanEval, and due to Python's indentation and syntax errors, the fine-tuned model failed all HumanEval test cases. In contrast, the original model passed one out of 164 test cases. After we examined the output of HumanEval from both models, we figured out that the fine-tuned model was more likely to produce extra test cases, causing indentation or syntax errors. We tried different prompt engineering techniques to prevent the model from generating extra test cases. Still, due to the 8k examples in the training data having both code solutions and test cases, we couldn't find a prompt to prevent the model from generating error-prone test cases.

When comparing the code generated by the original and the LoRA-finetuned LLaMA-2-7B models, we observed that for problems where both models failed to produce correct solutions, the finetuned LLaMA-2-7B model was more likely to generate code that compiles and appears reasonable. In contrast, the original LLaMA-2-7B model often produced simplistic solutions, such as merely returning a statement without meaningful implementation. If we had access to ground truth output sequences for each coding problem, we would likely conclude that the LoRA-finetuned LLaMA-2-7B model shows a significant reduction in the edit distance between the model's output and the ground truth compared to the original LLaMA-2-7B model. This reduction in edit distance can be attributed to the next-token prediction objective used during finetuning. However, token-wise edit distance is not an effective metric for evaluating code generation quality. Given this limitation, we believe that to achieve better executable code generation, a different objective function might be necessary to guide language models more effectively.

For example, Direct Preference Optimization (DPO) [11], a direct alternative of RLHF but without a reward model, could be an objective function that further improves the model's code generation ability by allowing the policy model to capture the difference between rejected and chosen samples.

Currently, no published work focuses on using DPO finetune LLM for code generation; However, we could generate chosen and rejected training pairs by using target LLM we want to finetune to solve problems on platforms like Leetcode and Codeforce when LLM fails to solve the question we will record the LLM answer as rejected sample, and solution to corresponding question as chosen sample, then finetuned target LLM on the generated rejected and chosen pairs with DPO. However, the process described above is beyond this work's scope because we will need to build an automatic reject and chosen pair collection pipeline to use DPO fine LLM on coding generation and additional computation resources to deploy policy and original model during DPO training.

5 Conclusion and Discussion

In this work, we finetuned LLaMA-2-7B using LoRA on a dataset of approximately 30,000 Python code question-answer pairs. To comprehensively evaluate the improvement in code generation attributable to LoRA finetuning, we tested the original LLaMA-2-7B and the LoRA-finetuned LLaMA-2-7B on 100 LeetCode problems. Additionally, we proposed a novel evaluation metric that more accurately reflects the model's code generation quality by focusing on the number of passed test cases rather than traditional pass or non-pass evaluation metrics.

In the end we also discussed about how to use DPO to improve further supervise finetuned LLM's code generation ability. We are aware our current code generation tasks are limited to single-function generation with short descriptions. In contrast, real-life programming tasks usually involve long context requirements and complex code bases, and we will leave that to future research.

References

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

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

[3] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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

[5] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

[7] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[8] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022.

[9] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 61–68, Dublin, Ireland, May 2022. Association for Computational Linguistics.

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

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

[12] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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

[14] Murray Shanahan, Kyle McDonell, and Laria Reynolds. Role play with large language models. *Nature*, 623(7987):493–498, 2023.

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

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

[17] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[18] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[19] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462*, 2023.