Fine-tuning CodeLlama-7B on Synthetic Training Data for Fortran Code Generation using PEFT

Stanford CS224N Custom Project

Soham Govande
Department of Computer Science
Stanford University
govande@stanford.edu

Taeuk Kang
Department of Computer Science
Stanford University
taeuk@stanford.edu

Andrew Shi
Department of Computer Science
Stanford University
acshi@stanford.edu

Abstract

Zero-shot Large Language Models (LLMs) have exhibited remarkable capabilities for code generation. While fine-tuning models like GPT-4 can enhance their accuracy and output specificity, it introduces significant computational costs especially for models with large parameter sizes (Weyssow et al. (2024)). This study deploys Parameter-Efficient Fine-Tuning (PEFT) techniques to implement a lightweight LLM for Fortran code generation. Our model is capable of generating Fortran solutions for common coding exercises, using LeetCode problems as a benchmark. We demonstrate PEFT's utility by only fine-tuning a small subset of parameters while still achieving a highly accurate generative process. Our novel research extends state-of-the-art code generation models to legacy programming languages that have often been overlooked by the AI-driven evolution of software development tools.

1 Key Information

- Mentor: Bessie Zhang
- Team contributions:
  - Soham Govande: Set up fine-tuning parameters, oversaw fine-tuning process, extracted baseline metrics, collected public Fortran dataset
  - Taeuk Kang: Developed synthetic training data generator, built Fortran compiler, tested and evaluated prompts for training data generation, collected public LeetCode dataset
  - Andrew Shi: Evaluated generated Fortran code, conducted literature review and established research techniques, analyzed code quality and model results

2 Interactive Project Website

An interactive project showcase is available online, which demonstrates our methodology and results. The code outputs from our model and CodeLlama-7B-Instruct is compared, along with the original LeetCode problems and solutions in Python and C++ for reference.

The web resource is available at https://projects.taeuk.net/fortran-llm/.

3 Introduction

LLMs have played a critical role in increasing productivity in today's work life. In particular, code generation and completion—generating code snippets and functions across numerous programming languages—is a task many developers have incorporated into their workflow. Despite the demonstrated success of LLMs for this task, current models rely heavily on massive datasets to achieve zero-shot capabilities. Furthermore, while current LLMs have performed well on evaluation metrics like HumanEval (Chen et al. (2021)) and MBPP (Austin et al. (2021)) for common programming languages, few models support code generation for legacy languages like Fortran.

In this paper, we explore the more challenging task of language-specific code generation. In contrast to the large datasets many multi-purpose LLMs were trained on to learn programming languages like Python and C++, there is a scarcity of publicly-available Fortran code. The limited availability of code corpora poses unique challenges in learning language-specific libraries and syntax. While models like GPT-4 are capable of generating Fortran code, there are significant limitations to this approach in resource-scarce environments due to the cost barriers in fine-tuning large-scale models for few-shot settings (Brown et al. (2020)). Recent work suggests that In-Context Learning (ICL) is a viable solution to this problem by leveraging model-inherent capabilities to adapt to new tasks without requiring retraining (Wang et al. (2022)). However, the need for carefully-designed, contextually-relevant prompt examples creates challenges for tasks with limited precedent.

We aim to answer two guiding research questions: (1) How effective are current LLMs at generating Fortran code? (2) Can we fine-tune an existing lightweight LLM to learn the common patterns of Fortran code and solve basic programming exercises? Using open-source repositories and well-documented software projects, we use PEFT through low-rank adaptation (LoRA) to teach our model Fortran syntax in the learning stage. In the inference stage, we compare the outputs of our fine tuned model and the vanilla CodeLlama-7B-Instruct model on 540 LeetCode problems across three categories: incorrect, partially correct, and correct. We find that our fine-tuned model produces significantly better results when measured on metrics for code functionality and style.

4 Related Work

4.1 Code Generation

There are many existing LLMs that have displayed strong code generation capabilities. These models include general-purpose LLMs like GPT-4, GPT-Neo, and GPT-J. There are also LLMs designed specifically for code generation like CodeLlama, Codex, and CodeParrot. One study conducted by Chen et al. (2023) evaluated LLM performance on domain-specific tasks like web and game development. They investigated three representative models (ChatGPT, CodeGen, and PolyCoder) for code completion and found that all three models experienced a decline in performance on domain-specific datasets compared to general-purpose datasets.

In recent years, there has been more research around language-specific code generation. In a study conducted by Thakur et al. (2023), researchers fine-tuned three CodeGen models for Verilog code completion. The results showed that the fine-tuned CodeGen-16B model performed on-par with the much larger GPT-3.5-turbo and GPT-4 models across all prompting patterns. Furthermore, the fine-tuned model exhibited strong performance across problems of varying difficulty. It demonstrated almost a 27x increase in generating functionally-accurate completions post-tuning. This study underscores the viability of using a lightweight model to achieve comparable results to large-scale, general-purpose LLMs for language-specific code generation.

4.2 Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning is an established technique that has achieved good performance across multiple domains. However, one challenge with PEFT is selecting suitable parameters as it is an NP-hard problem (Fu et al. (2022)). There are multiple approaches in selecting parameters, including random approaches like the Mixout model (Lee et al. (2019)), rule-based approaches like LoRA (Hu et al. (2021)), and projection-based approaches like DiffPruning (Guo et al. (2020)).

LoRA is particularly applicable to domain or language-specific code generation for lightweight models due to its capabilities of reducing the number of trainable parameters by a factor 10,000 and

the GPU memory requirement by a factor of 3 (Hu et al. (2021)). It eliminates the need to maintain optimizer states for most parameters and only optimizes the injected low-rank matrices. Furthermore, it introduces no new inference latency as the trainable matrices are merged with the frozen original weights when deployed. When used to fine-tune GPT-2, LoRA techniques outperformed multiple baseline comparisons with fewer trainable parameters on all adaptation methods on the E2E NLG Challenge (Hu et al. (2021)).

4.3 Evaluation Framework

Prior research has utilized existing problem sets to perform inference on vanilla and fine-tuned models. These problem sets may originate from open-source datasets like HumanEval or are proprietorially generated. Thakur et al. (2023) performed inference on two separate datasets: a smaller Problem Set I, which contains 17 unique Verilog challenges and a larger Problem Set II, which consists of a more comprehensive 181 problems. In the inference stage, each problem was tested on three prompts of increasing detail, a sampling temperature, and the number of completions per prompt. Then, they compared each scenario using different metrics like completions vs. temperature, completions vs. prompts, and completions vs. # completions / prompt.

Other studies have used open-source datasets and correlation based metrics. Yin and Neubig (2017) uses the Hearthstone (HS), Django, and IFTTT datasets as a metric to evaluate Python code completion accuracy. Token-level BLEU-4, which is an average over all examples, was used as an evaluation metric.

5 Approach

5.1 Overview

Our research is divided into three distinct phases:

1. Synthetic instruction generation on Fortran code
2. Fine-tuning CodeLlama-7B-Instruct using PEFT and LoRA
3. Inference and evaluation on LeetCode problems

5.2 Instruction Generation

We collect 10,677 files of high-quality Fortran code from public GitHub repositories of the Fortran standard library, NASA codebase, and other well-documented software projects.

In the instruction generation phase, we prompt GPT-3.5-turbo with a temperature of 0.2 to generate a three to six sentence description of the Fortran code we scraped. We also provide sample LeetCode `problem description-solution code` pairs to better label our collected data in the style of common algorithm exercises. We perform this process on all 10,677 files and store the results in `problem description-Fortran code` pairs.

5.3 Parameter-Efficient Fine Tuning (PEFT)

In full fine-tuning, the model is initialized to pre-trained weights and are then consequently updated on each iteration in the gradient descent algorithm. It follows that the gradient descent algorithm can be used to minimize the following loss function:

$$\mathcal{L}(\Phi) = -\sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log(P_\Phi(y_t|x, y_{<t})) \tag{1}$$

However, when performed on a LLM like GPT-4, which has around 175 billion parameters, this process becomes computationally expensive. An alternative method is PEFT using LoRA, where the weights are updated in a more parameter-efficient method. On each update, the task-specific weight updates are performed with respect to a much smaller subset of parameters. Thus, the gradient

descent algorithm can be used to minimize the new loss function where $\Delta\Phi(\Theta) \ll \Phi_0$ (Hu et al. (2021)).

$$\mathcal{L}(\Phi) = -\sum_{(x,y)\in\mathcal{Z}}\sum_{t=1}^{|y|}\log(P_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x,y_{<t})) \tag{2}$$

The efficacy of LoRA lies in the hypothesis that pre-trained language models have low "intrisic dimension" which allows it to still learn efficiently with a random projection to a lower-dimension subspace (Aghajanyan et al. (2020)). Thus, for a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, in the forward-pass, we constrain the number of parameters that receive the update by performing low-rank decomposition where $W_0 + \Delta W = W_0 + BA$. Note that $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ and $r \ll min(d, k)$. For an input x, we see that the forward pass for $h = W_0x$ becomes $h = W_0x + \Delta Wx = W_0x + BAx$ (Hu et al. (2021)).

The matrix A is initialized as a random Gaussian distribution and B is initialized to be zero, so the matrix product $\Delta W = AB$ is initially set to zero. On each update, ΔWx is scaled by a factor of $\frac{\alpha}{r}$. In prior experiments where the Adam optimizer is used, tuning α is roughly the same as tuning the learning rate r, assuming that the initialization was scaled properly; thus, α is commonly set to the first value obtained for r (Hu et al. (2021)).

We now motivate the use case of LoRA in PEFT specifically for the Transformer model. In the Transformer model, there are four weight matrices in the multi-headed self-attention module: W_q, W_k, W_v, W_o (Vaswani et al. (2017)). In this implementation, we adapt only the attention weights to LoRA and freeze the MLP modules for downstream tasks. This allows for greater training efficiency due to the decreased numbers of parameters and a more simple architecture.

5.4 Inference and Evaluation

We perform inference on our fine-tuned model by inputting 540 LeetCode problems and evaluating its outputs. Our prompt format is as follows:

```
[INST]Write a full Fortran script with a valid main entry point that
    solves the following problem. The program should run with all
    provided examples and output to stdout. <LEETCODE_PROBLEM>[/INST]
    [FORTRAN]
```

For each example, we prompt the fine-tuned model and the vanilla model to generate code that solves the LeetCode problem, provided the original problem description. We use a max token length of 1024 to accommodate for our compute limitations. The model terminates with a `[/FORTRAN]` tag which we filter out when testing the code.

5.5 Baseline Benchmarks

We use the vanilla CodeLlama-7B model as a baseline benchmark comparison to our fine-tuned model. We feed the same input prompt to both models and categorize each output using the same prompt and problem statement. We also compare code length, separating code from comments as a metric for code conciseness and clear documentation.

5.6 Remarks on Novelty

One unique contribution of our research lies in the synthetic instruction generation of annotations and descriptions for Fortran code to perform more effective fine-tuning. Because Fortran is a legacy programming language, there are few "instruction-code" datasets on the public Internet. Furthermore, it is difficult to collect this real-world data due to the decreased usage of the language. However, our synthetic data generation method allows us to use retroactively published code from GitHub to generate useful training samples.

We prompted GPT-3.5-turbo to annotate the code we scraped, adding documentation and comments to provide a more interpretable dataset in the fine-tuning stage. In Appendix 1, we include an example

of a raw file scraped from GitHub, alongside the "instruction" annotation that was synthetically generated. We find that adding instructions to raw data greatly improves inference results because of the lack of existing knowledge in CodeLlama-7B on the Fortran programming language. In general, there have been few studies conducted on Fortran code generation, further motivating our unique method of constructing our fine-tuning dataset. It is worth noting that few literary studies have fine-tuned a code generation model for Fortran using our novel data collection method.

6 Experiments

6.1 Data

For our training data, we scraped public GitHub repositories of the Fortran standard library, NASA codebase, and other well-documented software projects. We chose these repositories by identifying the top 100 most popular repositories for the search query "fortran". In total, we retrieved 10,677 files of high-quality Fortran code. For our testing data, we scraped the first 540 LeetCode Easy problems from leetcode.com. Our dataset includes problems spanning the topics of string modification, hashing, sorting, dynamic programming, and mathematics. We associate each LeetCode problem with its entire problem description, example test cases, and Python, C++, Java, and JavaScript solution code.

6.2 Evaluation method

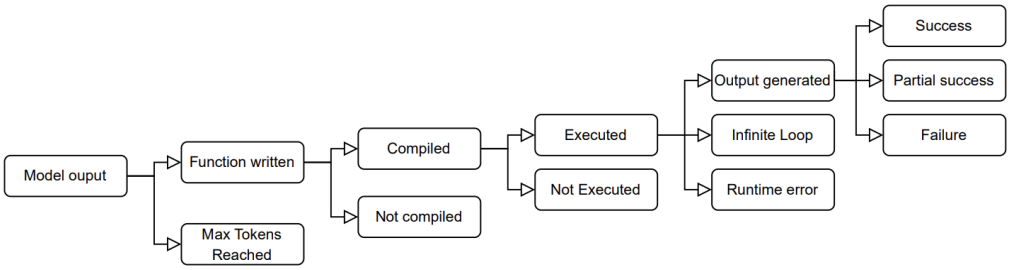


Figure 1: Evaluation framework for model comparison

We define a rigorous evaluation framework to compare the integrity of our fine-tuned model outputs and the vanilla CodeLlama-7B-Instruct hf outputs. We evaluate the code on both functionality and style.

For our functionality objective, we evaluate each of the 540 inference outputs using the evaluation framework defined in Figure 1. We used a max token length of 1024 due to GPU memory limitations. In the case that the inference output exceeded the max token limit, we conservatively assume that the code generation failed despite the probable root being the token length limitation. We then categorize outputs based on compile-time success and execution success. For compilation, we utilize the standard GNU Fortran compiler (GFortran) to serve as an unbiased compiler for this metric. If both metrics are met, we analyze the runtime behavior in further detail. We test the inference output on the example test cases and consequently label each function as incorrect, partially correct, or correct. The model outputs are compared directly to the answer key on LeetCode. We quantitatively compare the number of inference outputs in each of the categories. The compile-time and execution metrics are automatic while the code output and test case examination are human-assisted.

For our stylistic objective, we quantitatively measure the number of lines contained in inference output, separated into the number of lines of actual code and the number of lines of comments. We use this analysis as a qualitative measurement for code interpretability and potential to be robustly incorporated into a real-world codebase.

6.3 Experimental details

We used CodeLlama-7B-Instruct as our base model, which is specifically developed for code generation tasks. For tokenization, we use the CodeLlamaTokenizer, which is optimized for parsing and

understanding code syntax and semantics. 5% of our dataset was allocated for validation to test our model on unseen data.

In the model fine-tuning phase, we use PEFT and LoRA on all linear layers, as described in Section 4. We fine-tune our model for one epoch using a micro batch size of 32 and learning rate of 0.0001. We use the AdamW optimizer to solve the minimization problem, as implemented in Problem Set 5. For the LoRA hyperparameters, we use a rank parameter of 16 and an alpha parameter of 32, which we find to be the most optimal for our problem. Additionally, we use a dropout percentage of 5% to prevent overfitting the model to the training set, an assumption that is commonly used in code generation problems (Wang et al. (2024)).

6.4 Results

Table 1: Code Quality Comparison on Fine-Tuned Model and CodeLlama-7B-Instruct

Model	Function Written	Compiled	Executed
Fine-Tuned Model	522	185	125
CodeLlama-7B-Instruct	448	131	50
	Output Generated	Timed Out	Runtime Error
Fine-Tuned Model	121	2	2
CodeLlama-7B-Instruct	49	0	1
	Incorrect	Partially Correct	Correct
Fine-Tuned Model	32	33	56
CodeLlama-7B-Instruct	4	13	32

Table 2: Style Quality Comparison on Fine-Tuned Model and CodeLlama-7B-Instruct

Model	Total Comment Lines	Total Code Lines
Fine-Tuned Model	1,482	25,044
CodeLlama-7B-Instruct	3,777	27,367

Using our inference results, we find that our fine-tuned model significantly outperforms the vanilla CodeLlama-7B-Instruct model on generating Fortran code for LeetCode problems. For each of the 540 LeetCode problems, we first evaluate the code snippet's ability to reach the execution stage in the evaluation framework. Our fine-tuned model outputted successfully-compiled code for 185 examples and successfully-executed code for 125 examples. The vanilla CodeLlama-7B-Instruct model outputted successfully-compiled code for 131 examples and successfully-executed code for 50 examples. Some examples were truncated midway during token generation due to the max token limit of 1024.

After the execution stage, we find that our model successfully generated outputs for 121 examples and the fine-tuned successfully generated outputs for 49 examples. Our model generated code that timed out for 2 examples and code with a runtime error for 2 examples. The vanilla model generated no examples that timed out during execution and 1 example that encountered a runtime error.

Finally, we test each code snippet on the example test cases provided on LeetCode. We define correct code as one that passes all tests for which it produced output for; we define partially correct code as one that passes at least one test for which it produced output for; we define incorrect code as one that passes none of the tests for which it produced output for or if the model produced a hallucination. Our fine-tuned model generates correct code for 32 examples, partially correct code for 33 examples, and incorrect code for 56 examples. The vanilla model generates correct code for 4 examples, partially correct code for 13 examples, and incorrect code for 32 examples.

Overall, we observe a 41% improvement in successful compilation, a 150% improvement in successful execution, and a 75% improvement in producing correct outputs when comparing our fine-tuned model to CodeLlama-7B-Instruct.

7 Analysis

We observe that our model outperforms the baseline CodeLlama-7B-Instruct. As noted by Rozière et al. (2024), CodeLlama-7B-Instruct is trained on "publicly available code" from popular languages, such as Python, C++, Java, and more and is likely that only a very small amount of Fortran code snippets has been included. While the CodeLlama-7B-Instruct model may possess general coding abilities across various languages, its exposure to Fortran is likely limited, leading to a performance gap compared to our specialized model. This gap is evident in the higher percentages of successfully compiled and executed functions, as well as the increased number of correct and partially correct solutions generated by our fine-tuned model.

When analyzing the first tokens of the model output, our model always attempted to write valid Fortran code, while the baseline model has demonstrated several instances where invalid tokens are generated. Our model, in all cases, started with `program`, indicating the start of a valid Fortran program. The baseline model had more variability in its initially generated tokens: `program` and `module`, both of which are valid Fortran code, `!`, indicating a start of a Fortran comment, ```, a Markdown syntax indicating the start of a code block, or `\n`, a line break.

There were two cases where the baseline model generated only a line break (examples 530 and 783), two cases where it generated non-ending copyright statements (examples 637 and 1869), and one case where a non-existing GitHub repository was cited and linked to (example 2269).

We believe that the cases of copyright statements and the repository links are both affected by the baseline model's tendency to write more comments than our own, as shown from Table 2. The baseline model generated more than twice the amount of comment lines. Some comments were valid descriptions of the code, but a lot of comments included separators (e.g., lines with `---` only). A lot of code includes a header comment with a separator and copyright information, which explains the baseline model's tendency to write code.

This excessive commenting behavior exhibited by the baseline model can be attributed to its training data, which likely included a diverse range of coding styles and practices from various sources. While comments are generally considered good practice for code readability and documentation, an excessive amount of comments, especially those with little informative value, can clutter the code and lead to hallucinations where valid code is not generated at all.

The cases where the baseline model generated non-ending copyright statements or cited non-existing GitHub repositories further exemplify this tendency to incorporate extraneous comments. It is possible that the model encountered such examples in its training data and generalized this behavior to inappropriate contexts.

In contrast, our fine-tuned model, having been exposed to a curated dataset of Fortran code snippets with ample amount of code, may have learned more concise and focused commenting styles specific to the Fortran programming language and its conventions. This could explain the lower number of comment lines generated by our model, as well as the absence of such anomalies like non-ending copyright statements or fictitious repository links.

The superior performance of our fine-tuned model aligns with the findings of previous studies that have demonstrated the benefits of domain-specific fine-tuning for language models as demonstrated in Chen et al. (2023). By exposing the model to a concentrated dataset of Fortran code examples, it can better capture the syntax, idioms, and coding patterns specific to the Fortran programming language.

8 Limitations

First, the largest limitation of our data collection method is that the training data, real-world GitHub repositories, does not accurately reflect the evaluation metric, which is LeetCode problems. A large part of our training set includes boilerplate code for things like matrix mathematics, standard library subroutines, and template-style macros. In contrast, most LeetCode problems rely on algorithmic expertise. Future research may consider using a dataset more similar to the evaluation metric, although it may be more difficult to obtain.

Future research might consider re-running our evaluation metric on more powerful GPUs. Because of our GPU memory constraints, we were forced to stop at a maximum of 1024 tokens. As a result, roughly 100 out of the 1,000 inferences terminated without the Fortran code being finished. To remain unbiased, we mark these as 'failures' for both our model and the baseline model, but it would be more accurate to allow the inference to finish the programs' completion.

Finally, one promising direction for future research is Reinforcement Learning from Human Feedback (RLHF). RLHF can enable a model to iteratively get better over time by incorporating prior results. For example, if a model generated invalid code that didn't compile, we could penalize it with an external negative stimulus, and if it generated valid code that accurately solved the problem, then we could reward it with a positive stimulus.

9 Conclusion

In this paper, we implemented fine-tuning of CodeLlama-7B-Instruct for Fortran code generation using PEFT and LoRA techniques. We used a novel method of synthetic data generation by prompting GPT-3.5-turbo to annotate and describe the code we scraped in the format of `problem description-Fortran code` pairs, similar to that of LeetCode problems. Our fine-tuned model exhibited significant improvements in generating Fortran code that compiles and executes successfully. Furthermore, our fine-tuned model outperformed the vanilla model in generating accurate Fortran code that successfully passes the example test cases provided for each problem.

Our results show a promising future for code generation research, even for domain or language-specific tasks. Implementing RLHF on our fine-tuned model will likely improve results. Furthermore, increasing CPU memory allows for longer max token lengths, a feature we believe will contribute to better results. A common end goal in the field of code generation is to generate functional and deployable end-to-end systems that can be used in real-world settings. Techniques like those pioneered by Chang et al. (2023) and new models like Cognition AI's Devin have significantly changed perceptions in research and academic communities of the possibilities of code generation.

References

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. 2020. Intrinsic dimensionality explains the effectiveness of language model fine-tuning.

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

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. 2020. Language models are few-shot learners.

Jonathan D. Chang, Kiante Brantley, Rajkumar Ramamurthy, Dipendra Misra, and Wen Sun. 2023. Learning to generate better than your llm.

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

Meng Chen, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, Juhong Wang, and Xiaodong Gu. 2023. On the effectiveness of large language models in domain-specific code generation.

Zihao Fu, Haoran Yang, Anthony Man-Cho So, Wai Lam, Lidong Bing, and Nigel Collier. 2022. On the effectiveness of parameter-efficient fine-tuning.

Demi Guo, Alexander M. Rush, and Yoon Kim. 2020. Parameter-efficient transfer learning with diff pruning.

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

Cheolhyoung Lee, Kyunghyun Cho, and Wanmo Kang. 2019. Mixout: Effective regularization to finetune large-scale pretrained language models.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, 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. 2024. Code llama: Open foundation models for code.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. Verigen: A large language model for verilog code generation.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Sheng Wang, Liheng Chen, Jiyue Jiang, Boyang Xue, Lingpeng Kong, and Chuan Wu. 2024. Lora meets dropout under a unified framework.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation.

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2024. Exploring parameter-efficient fine-tuning techniques for code generation with large language models.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation.

A Appendix

A.1 Instruction Annotation for GitHub file

We include an example of a raw file scraped from GitHub, alongside the instruction annotation labeled as `problem` that was synthetically generated.

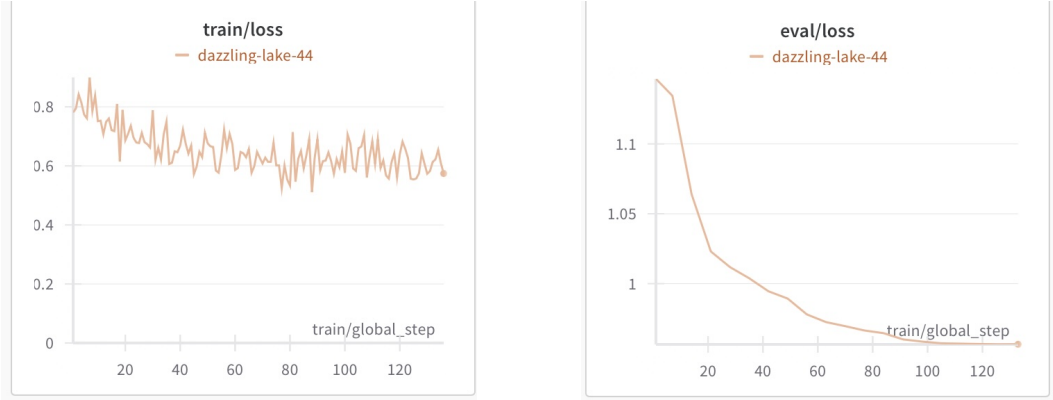

```
{"problem": "Given the iteration number, squared distance, energy
threshold, and an array of electronic energies, this code snippet saves
the information in a file named 'restart_scf'. It first writes a message
to the standard output indicating the iteration number at which the
calculation stopped in the self-consistent field (scf) loop. Then, it
checks if the file 'restart_scf' already exists. Next, it opens the file
in formatted mode and writes the iteration number, squared distance, energy
threshold, and the electronic energies array to the file. Finally, it
closes the file, ensuring the information is saved for future reference.",

"solution": "SUBROUTINE save_in_electrons (iter, dr2, ethr,
et)!-----------------------------------USE kinds, ONLY: dpUSE io_global,
ONLY: stdoutUSE io_files, ONLY: iunres, seqopnUSE klist, ONLY: nksUSE wvfct,
ONLY: nbnd!IMPLICIT NONE!INTEGER, INTENT (in) :: iterREAL(dp), INTENT(in)
:: dr2, ethr, et(nbnd,nks)!LOGICAL :: exst!WRITE(stdout,'(5x,C̈alculation
stopped in scf loop at iteration ;i6)') iterCALL seqopn (iunres,
```

'restart_scf', 'formatted', exst)WRITE (iunres, *) iter, dr2, ethrWRITE (iunres, *) et(1:nbnd,1:nks)CLOSE (unit=iunres, status='keep')!SUBROUTINE save_in_electrons"}

A.2 Training and Evaluation Loss

We include graphs for the training loss and evaluation loss from the training phase, both of which are decreasing at increasing time steps.



(a) Training loss graph

(b) Evaluation loss graph

Figure 2: Training and evaluation loss graphs