Scaling Down Multilingual Language Models of Code

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

The democratization of AI and access to code language models have become pivotal goals in the field of artificial intelligence. Large Language Models (LLMs) have shown exceptional capabilities in code intelligence tasks, but their accessibility remains a challenge due to computational costs and training complexities. This paper addresses these challenges by presenting a comprehensive approach to scaling down Code Intelligence LLMs. To enhance usability, we focus on training smaller code language models, which lowers the computation cost of inference and training. We extend these models to diverse programming languages, enabling code completion tasks across various domains.

1 Introduction

In recent years, Large Language Models (LLMs) have emerged as powerful tools with applications spanning various domains. Notably, their effectiveness in code intelligence tasks has been remarkable, leveraging the structured and rule-based nature of programming languages. Accessibility to code LLMs is one of the important goals of AI development as seen by the increasing amount of open-source models and datasets. However, this may not be enough, as the barrier to tuning and using LLM is higher than just access to their weights. For the practitioner, the choices of model architecture and learning algorithms are not obvious, and exploring these options is costly due to the high computation costs. The aim of this project is to address the accessibility gap and further efforts towards democratizing the use and development of code LLMs. In this work, We extend the capabilities of a mono-lingual code LLM originally trained on Python to encompass a diverse range of programming languages including Java, Rust, Ruby, and Swift. This expansion is achieved efficiently by employing Parameter Efficient training techniques and various training optimization methods.

2 Project Methodology

The first step in the pipeline in Figure 1 is the selection of the baseline model that will be fine-tuned. Any auto-regressive language model can be chosen at this stage because the fine-tuning task is causal language modelling. However, we use the CodeGen-350M-mono (6) because it is a small and monolingual code language model trained only on Python. Processing the training dataset is the next step. The sampling of training data from the TheStack corpus (4), file filtering, and generation of the training and validation splits were all standardised in this step. The fine-tuning stage in which we compare between full fine-tuning and LoRa fine-tuning. The fine-tuning can be completed on a free Google Colab T4 GPU in less than 10 hours, depending on the method and parameters chosen. After completion, the weights and training metrics are all saved for use in the evaluation stage. Finally, we create an evaluation report with different results and analyses, then share the generated model cards (5) and dataset cards (3) along with the trained weights.

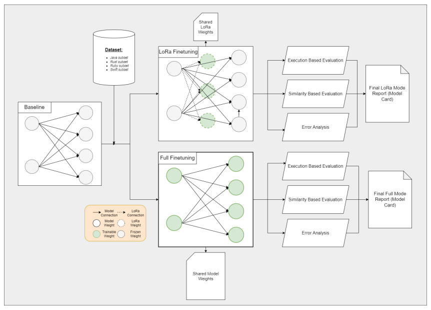

Figure 1: Project Diagram

3 Fine-tuning Small code language models on Low Resource Languages

To test our approach, we fine-tuned our baseline model in four different programming languages: Java, Ruby, Rust, and Swift. These languages were mostly chosen because they represent varying levels of availability in the Stack dataset. We train, test, and share four distinct LoRa adapters using our preprocessed datasets. We used the CodeGen-Mono baseline (6) with its GPT-2 tokenizer and fine-tuned the model on the processed datasets from Section **??**. Each model was trained for 10,000 steps, and the training sequence length was 2084 tokens. For the LoRa configuration, we used a rank of 128 with a dropout of 0.05 on the attention, fully connected, and language model head modules. Optimization was done using the Adam optimizer with a 5e-4 learning rate, 100-step warm-up, and cosine weight decay. We used a batch size of 4 with no gradient accumulation. Gradient check-pointing was used for efficiency along with 16-bit training. The Pass@10 rates (obtained according to (2) formulation) are illustrated in Figure 2, which depicts the evaluation of the four models along with the monolingual and multilingual baselines. The evaluation was carried out across 161 code completion problems from MultiPl-E (1), spanning five different programming languages. As anticipated, the monolingual baseline achieves the highest Pass@10 rate in Python. Conversely, the multilingual baseline, trained on a vast corpus of 119.2 billion tokens encompassing C, C++, Go, Java, JavaScript, and Python, scores notably lower in Python (12.42) and Java (7.59). Better performance in Java can be seen in our Java model, which achieves a Pass@10 rate of 8.23 despite being fine-tuned on a relatively smaller dataset of 100 million Java tokens. The Ruby, Rust, and Swift models all have the best performances in their respective languages.

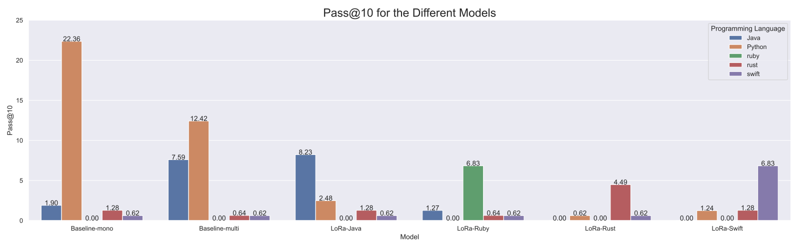

Figure 2: Pass@10 Rates by Fine-tuned models and Baselines

4 Conclusions

The journey towards democratizing the field of artificial intelligence and code language models is a broad mission that requires addressing challenges on various fronts. This work has presented a concerted effort to bridge the gap between advanced AI technologies and their practical usability, particularly in the domain of code intelligence. By focusing on accessibility, usability, and empirical understanding, we have contributed to the ongoing narrative of democratization in AI. Nevertheless, the aspiration of AI democratization must not be pursued thoughtlessly, as there are several issues that need to be addressed, as highlighted in other research works.

References

[1] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7): 3675–3691, 2023. doi: 10.1109/TSE.2023.3267446.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021. URL `https://api.semanticscholar.org/CorpusID:235755472`.

[3] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III au2, and Kate Crawford. Datasheets for datasets, 2021.

[4] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022.

[5] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. ACM, jan 2019. doi: 10.1145/3287560.3287596. URL `https://doi.org/10.1145%2F3287560.3287596`.

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