This is the Title

Your Name

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Your Name*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

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. There are three areas that can be considered the main drivers for the success of LLMs: (i) Model architectures: Transformers are capable yet simple models as they rely on the self-attention mechanism. (ii) Learning Algorithms: The Pre-training and Fine-tuning learning algorithms made models more fixable for handling a number of specialised tasks (iii) Scale: better performance can be expected by increasing the model parameters, the data, or the computation according to the scaling laws. These discoveries have resulted in excellent performance in program synthesis and understanding challenges and have been adapted to both commercial [11] and open-source [24] tools.

Accessibility to code language models is one of the important goals of AI development. The Democratization of AI is an important factor in the progress of the field, and AI companies such as Meta [2] and Stability-AI [3] are showing their commitment to it by publicly sharing their models and releasing their weights. However, this may not be enough, as the barrier to tuning and using LLM is higher than access to their weights. For the practitioner, the choices of model architecture and learning algorithm are not obvious, and exploring these options is costly due to the high computation costs. In order to achieve the benefits of democratization of AI use and development [36], these issues need to be resolved.

1.2 Problem Statement

The aim of this project is to address the accessibility gap and further efforts towards democratising the use and development of code LLMs. To make LLMs more usable, we need to first address the high computational cost of large language models. Using smaller models also comes with the cost of being limited to a small number of popular programming languages. Hence, We will train small code language models to perform code completion tasks [25] in a variety of programming languages. Additionally, to address the ambiguity in fine-tuning LLMs, we will provide empirical evidence on the effects different choices have on the training process in terms of time, cost, and performance.

1.3 Contribution

In this work, we fine-tuned a mono-lingual code LLM trained on Python to four new programming languages: Java, Rust, Ruby, and Swift. We used LoRa Parameter Efficient Fine-Tuning to train the four models, and they are all shared publicly along with their respective training datasets and evaluation results. We also did extensive ablation studies, trade-off computations, and error analysis to provide empirical evidence on the effects different choices in the training process have on the efficacy of the process. Finally, we discussed the current state of AI democratisation, the associated risks and benefits, current gaps, and possible ways forward.

1.4 Thesis Structure

This paper will be divided into the following five parts:

- The project's motivation, goals, and outcomes are primarily introduced in the first chapter, which also serves as an introduction. This chapter provides a general overview of the project, highlights its innovation and significance, and discusses its ultimate objective.
- The background primarily introduces prior research and work in the project's pertinent fields. In order for readers to become familiar with the project and better comprehend the works produced by this project.

- The methodology chapter provides a detailed introduction to each step of the project's implementation as well as the project's overall structure. Additionally, it goes into detail about how pipelines are built for data preprocessing, various methods of fine-tuning, and evaluation.
- The result and evaluation are covered in chapter four. The experiments conducted for the project will be covered in this chapter. This will include each experiment's goal, setup, and outcomes, as well as an evaluation of these outcomes.
- The fifth chapter is a summary, which will summarise the contributions of the projects in terms of AI democratization and discuss the benefits, risks, and ways forward for more accessible code language models.

Chapter 2

Background

This chapter is meant to present the reader with the necessary context for this project, which aims to democratize the use of code language models by efficiently training monolingual smaller models. We will begin by providing an overview of approaches for efficient training of large language models, as well as their impact on memory, run-time, and performance. Then we discuss the present state of the art in using multilingual code language models for code completion tasks. Finally, we review current trends and initiatives in the topic of AI democratization briefly.

2.1 Efficient Training Methods

Scaling is a critical component in reaching state-of-the-art performance in NLP, since recent research suggests that greater model size can result in predictable performance benefits [40]. Despite the benefits of scaling, it creates significant hurdles to making these advances available in resource-constrained situations [5]. To address these limitations, there has been a renewed emphasis on research aimed at improving model efficiency [39].

When implementing efficient training methods, there are many factors to consider. The work of [35] defines efficacy as the cost of producing a result (R) in terms of three factors: execution cost (E), dataset size (D), and the number of Hyperparameters (H): $Cost(R) \propto E \cdot D \cdot H$. On the other hand, [38] examines several efficiency strategies based on their effects on model precision, the number of computations (FLOPS), and total memory need. We will use a similar approach to Green and compare approaches based on the previous criteria and whether a method is for training, inference, or both, as shown in Table 4.1.

The methods in Table 4.1 cover a variety of techniques. Sparsity [28] is a technique that aims to reduce the memory footprint of training models by dynamically adding sparse connections (zero parameters) through the training. Sparse matrix operations can be optimised to save memory, but their effect on the run-time is unclear [42]. Quantization techniques use the 16-bit representation of floats instead of the 32-bit representation to achieve a 2x reduction in memory requirements and speed up the computations [21]. The work of [26] studies the trade-off when using smaller batch sizes (micro-batching) between reduction in memory and reduction parallelism. The effect of micro-batching also varies, as smaller batches could improve generalisation, while larger batch sizes may result in more stable learning, both depending on other factors like the learning rate and normalisation techniques [26], [12]. The work of [14] takes a different approach by addressing self-attention quadratic time and memory complexity with sequence length. Flash-Attention is an optimised I/O aware (uses fewer accesses to GPU memory) that achieves up to 7.6x speedup and a linear memory to sequence length complexity. This method was the foundation that enabled the training of LLMs with very large context windows [31]. Pruning and trained quantization [17] are model compression methods that can be used to enable inference on memory poor edge devices.

Parameter's Efficient Fine-Tuning (PEFT) is central to the pre-training and fine-tuning paradigm for large language models, as fully fine-tuning LLMs requires significant time and resources. The goal of PEFT approaches is to optimise LLMs by modifying lightweight trainable parameters while leaving the majority of pretrained parameters unchanged. PEFT approaches vary; for example, the adapter method [19] adds trainable parameters to each transformer block of the model. Another technique known as prompt-tuning [23] relates to techniques that modify the input prompt to improve modelling results. Finally, LoRa [20] learns low-rank matrices to approximate the weight updates of the LLM.

2.2 Code Completion Using LLMs

Code LLMs Using language models for code generation and comprehension tasks has recently received a lot of attention. Large language models for code can be divided into three groups: masked language models, encoder-decoder models, and left-to-right language models. Masked LLMs, such as CodeBert [15],are used to provide representations for code sequences, which are useful in code comprehension tasks like

Method	Memory	Runtime	Performance	Training	Inference
Sparsity	↓	?	↓	✓	✓
Quantization	↓	↓	-	✓	✓
Micro-Batching	↓	↑	?	✓	-
Gradient checkpointing	↓	↑	-	✓	-
FlashAttention	↓	↓	-	✓	✓
Pruning	↓	↓	-	-	✓
Trained Quantization	↓	↓	-	-	✓
PEFT	↓	↓	↓	✓	-

Table 2.1: Summary of Efficient Training Methods

clone detection and code classification. CodeT5 is an encoder-decoder model that can be used for conditional generation tasks like code generation from natural languages. The work of [13] has demonstrated that left-to-right code (decoder only) language models, such as SantaCoder[6] and PolyCoder [41], perform exceptionally well on a wide range of tasks, including code generation and completion.

Evaluating Generations Code completion can be evaluated intrinsically through textual similarity metrics or extrinsically through execution-based metrics. Intrinsic evaluation measures such as CodeBLEU [34] and perplexity are simple to calculate and language agnostic. These measurements, however, have been demonstrated to have a small correlation with code generation quality [11]. Most execution-based metrics are composed of a set of code challenges, each with its own natural language description, function signature, and set of unit tests. HumanEval [11] and the Mostly Basic Programming Problems (MBPP) [9] are two of the most commonly used execution metrics, and both are Python-based. There are, however, a number of works that extend both metrics into additional programming languages. MultiPl-E [10] adds 18 new languages to HumanEval and MBPP, whereas [8] adds over 10 new programming languages to HumanEval, MBPP, and Math-QA [7]. Finally, the BabelCode and TP3 [32] benchmarks give fine-grained (unit test level) code completion evaluation in 14 different programming languages.

2.3 AI Democratisation

In recent years, there has been a significant increase in the conversation about ”AI democratisation.” However, the term itself is used in various ways, leading to a lack of clear communication among experts when talking about the objectives, approaches, potential dangers, and advantages of initiatives aimed at achieving AI democratisation. The work of Nur refers to AI democratisation solely on the basis of the decentralisation of knowledge production in the field. Whereas Elizabeth identifies four types of AI democratisation: democratisation of use, development, governance, and profits. The term participation is often used to highlight the importance of centering the values of all agents affected by the development of AI (ABEBA).

These different explanations of the term are partly due to the different motivating factors for pursuing AI democratisation. The uneven distribution of AI harm due to inherent biases is one of the most important reasons behind the increased interest in inclusive AI development. Another reason is that increased contributions from around the world are essential to the progress of the field. Finally, The work of ’Compute Divide’ and ’Hardware Lottery’ advocates scaling down so that ideas are not pushed aside for lack of compatibility with compute.

Current trends towards AI democratisation also vary greatly. Open-source models and datasets are the most efficient way of facilitating wide and diverse inclusion in AI development. Additionally, as we discussed in Section 1, scaling down and lowering the barrier to entry are also being heavily researched. Finally, some works also propose solutions based on policies that enforce and encourage democratisation. such as (national research cloud).

Chapter 3

Methodology

3.1 Project Methodology Overview

The main objective of this project is to make access to code intelligence models easier in terms of use and development. Because of this, the project's overall structure is designed to be straightforward and modular, allowing practitioners to easily follow and reproduce any part of it. The diagram in 3.1 demonstrates how the project's steps progress from choosing a baseline model and processing the training dataset to evaluating and disseminating the final model weights and card. While the sections that follow in this chapter will go into more detail about particular parts, this section will provide an overview of how these parts relate to one another.

The first step in the pipeline in Figure 3.1 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 masked language modelling. However, we only use the CodeGen-350M-mono [30] 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 [22], file filtering, and generation of the training and validation splits were all standardised in this step. Any programming language chosen for this step will be subject to all of these operations.

The fine-tuning process can start once the training dataset and baseline model are available. This stage has the largest number of possible parameter configurations, but the main options are between full fine-tuning and LoRa fine-tuning. The fine-tuning can be completed on a free Google Colab T4 GPU [1] in less than 10 hours, depending on the method and parameters chosen (more details in Appendix). After completion, the weights and training metrics are all saved for use in the evaluation stage. The evaluation

is also divided into two stages. The first stage is the generation stage, where model outputs are produced according to specific prompts and inference parameters. The second stage is using these generations to create an evaluation report with different results and analyses. Finally , when all the outputs are available, model cards [27] and dataset cards [16] are automatically generated to be shared with the model on open source platforms such as HuggingFace [4] (Figure 3.2).

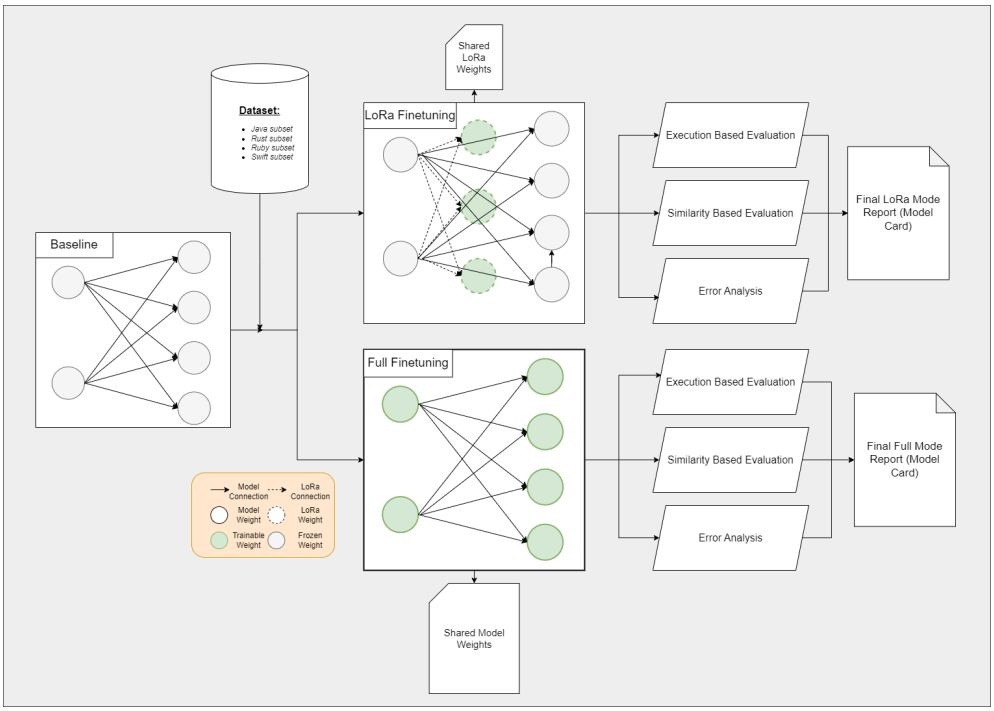
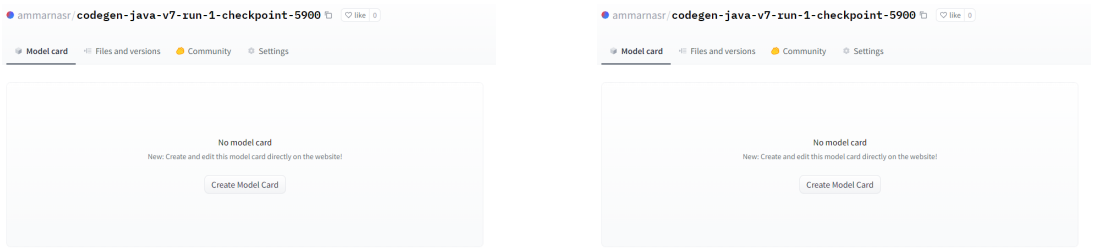


Figure 3.1: Project Diagram



(a) Model Card

(b) Dataset Card

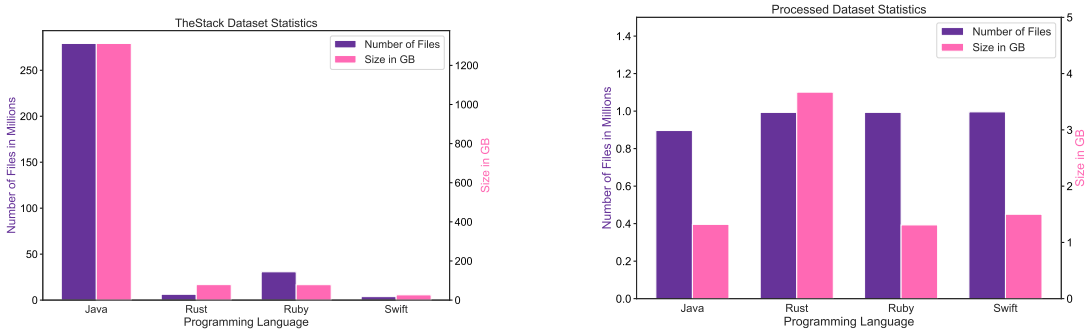
Figure 3.2: Example Of Generated Model and Dataset Cards Shared on HuggingFace

3.2 Dataset

Our training datasets are primarily drawn from TheStack Corpus [**?**]. TheStack is an open-source code dataset with more than 3TB of GitHub data covering 48 different programming languages. We only use a small portion of this dataset because our experiments only consider a small subset of programming languages and because optimising smaller language models only needs a small amount of data.

We had to perform several preprocessing steps in order to prepare our datasets. The first step is choosing our target programming languages. Ruby, Swift, Rust, and Java were ultimately chosen. We selected these languages because they represent the distribution of programming languages on GitHub. As seen in figure 3.3a, the most popular language is Java; Ruby and Rust are medium-resource languages, and Swift is a low-resource language. Additionally, Ruby was chosen because it shares syntax similarities with Python and is a dynamic language, whereas Java, Rust, and Swift are all statically typed. The next step is to select a sample of files from TheStack corpus and divide them into train, validation, and test splits. In order to train LLMs effectively, we sampled one million files from each of the four languages. To filter the files, we adhered to best practises [41] and filtered out files with the following characteristics: 1) an average line length of over 100 characters; 2) a maximum line length of over 1000 characters; and 3) a ratio of alphabet less than 25%. Then, we split the remaining files into train, validation, and test splits using ratios of 0.9, 0.05, and 0.05. Figure 3.3b shows the statistics for the final datasets.

The tokenizers used were the same as those for the baseline models [30]. These Byte Pair Encoding (BPE) [37] tokenizers add special tab and white space tokens to the same GPT-2 vocabulary [33]. The training sequences are produced by joining text from training data to fill the 2048 context length, or just 1024 tokens in the case of full fine-tuning.



(a) TheStack Dataset [22] Statistics

(b) Processed Datasets Statistics

Figure 3.3: Comparison Of Datasets before and after Processing

Chapter 4

Results and Evaluation

In this chapter, we will cover all the experiments in the project, discussing the objectives of each experiment , the experimental setup, and then presenting and analysing the outcomes. The chapter will be divided into four sections , where each section builds on the previous results and presents experiments with similar objectives. In the first section, we compare Full fine tuning methods vs. efficient fine-tuning methods and try to find out the difference in performance we can achieve in the Java programming language. Next, we perform ablation studies on the parameters of LoRa fine-tuning to understand the contribution of each factor to the process. In section three we study the trade-off between memory, time, and performance when fine-tuning code language models in an environment with limited resources. Finally, we use the knowledge obtained from previous results to train models on programming languages other than Java, and in the final section, we perform error analysis on the generations of these models.

4.1 Full Fine-Tuning vs. Efficient Fine-Tuning

4.1.1 Objective and Setup

We suggest that a possible solution to the high inference and training costs of Large Language Models is to use smaller, more compact architectures. However, this also comes at a cost, as smaller models are normally trained in only one programming language to achieve good performance with their limited parameters. In fact, work by [8] has empirically shown that training sufficiently large language models benefits the model's overall performance as knowledge can be transferred between different languages. On the other hand, training smaller models in an increasing number of

programming languages hinders learning in all of them.

However, we hypothesise that it is possible to fine-tune small mono-lingual language models to perform satisfactorily well in other programming languages. We argue that one can use full fine-tuning and utilise the trained model weights as initialization with knowledge about general programming knowledge and natural language. Full fine-tuning, however, suffers from the phenomenon known as catastrophic forgetting, which may cause the tuned model to forget the syntax of its original programming language. Using parameter-efficient techniques, we can mitigate the forgetting issue and reduce the computational cost. These gains come from only a small number of low rank matrices, which, on the other hand, may limit the learning ability of the model.

We used the task of code completion, particularly the one outlined by Humaneval, to test our hypothesis. We used the Codegen-350M-mono and Codegen-350M-multi auto-regressive code language models developed and released by Codeforce as baselines [30]. Both models share the same number of parameters (350 million) and structural elements. The difference is that the mono version is trained on Python only (BigPython [30]), and the multi version is trained on a mix of four languages (Python, Java, Javascript, and C) from TheStack dataset [22]. Then, we fine-tuned the mono model on the Java subset of the preprocessed dataset discussed earlier. We used Full and LoRa fine-tuning using the same masked prediction loss function, Adam optimizer, a 0.0005 Learning rate with cosine decay, and an effective batch size of 8. The maximum sequence length in full fine tuning was 1024, and in LoRa it was 2048 due to memory limitations. A detailed description of these parameters is in the ablation studies section, and the full list of parameters is in the appendix.

4.1.2 Analysis

The results in Figure 1 show that it is possible to fine-tune monolingual models for new programming languages in a reasonable amount of time with limited computational resources. As shown in figure 2 , the training lasted for 10,000 steps, or the equivalent of 80,000 different files, each with 1024 and 2048 Java tokens for Full and LoRa finetuning, respectively. The training was done on a Google Colab T4 15GB GPU and lasted for around 12 hours (details in the appendix).

The Loss curves in figure 1 show that the LoRa fine-tuned model evaluation loss flattened earlier than the full fine-tuned one. But the latter had a lower evaluation loss overall. In Figure 2, we see the differences more clearly The pass@10 rate of the full

fine-tuning is 50% higher than LoRa and, in fact, almost the same as the results from the multilingual baseline, which saw a much larger number of Java tokens (detailed breakdown of token count in the appendix).

We can also see from Figure 2 the effects of catastrophic forgetting, as the Python score for full-finetuning is 0.0 while the LoRa score is 3.75, which is also a significant drop from the 22.36 in the monolingul baseline. Additionally, we see that even though the monolingual baseline was only trained on Python, it still passes some of the Java problems. This is probably due to code spill (Java code inside Python files), and it has been reported in other works [**?**].

Figure 4.1: Comparison of Loss Curves for LoRa and Full Fine Tuning

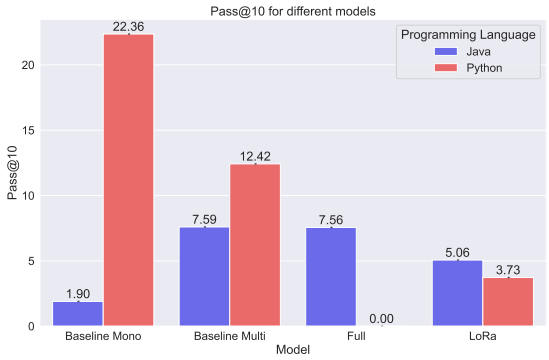


Figure 4.2: Comparison of Pass@10 for LoRa and Full Fine Tuning vs the Baselines

4.2 Ablation Studies

4.2.1 Objective and Setup

As we discussed in previous sections, fine-tuning entails selecting a large number of parameters. Choosing an acceptable parameter configuration might be a challenge for practitioners trying to fine-tune their own code language models. Even when employing smaller models and efficient fine-tuning approaches, the exploration cost remains significant due to the fact that knowledge about the effects of most of these parameters is still vague in the field [29], [32]. Furthermore, the effects of these parameters are subject to change due to changes in external variables such as model size, data quality, or downstream tasks [18]. Finally, In circumstances like ours, when we focus on efficiency measures (Memory and Time) as well as performance on a specific activity, parameter selection becomes more challenging.

We conducted comprehensive ablation studies in this experiment to provide empirical evidence that practitioners can use to guide parameter selection. We study the effect a selected number of parameters have on the efficacy and performance of fine-tuning code language models with LoRa. The following parameters have been chosen: LoRa Rank, LoRa layers, Batch Size, Sequence Length, and Learning Rate. In the following section, we will go over each parameter, justifying why we chose it, the expected behaviour, and our findings. We configured our experiments to use the same baseline model and fine-tuning settings as experiment 1. Then, for each of the ablation variables, we repeat the fine-tuning process across a fixed number of tokens. In each of these runs, we gradually adjust the variable and examine the effects of the fine-tuning process on performance and efficiency.

4.2.2 Analysis

4.3 Trade-off Analysis

4.3.1 Objective and Setup

The memory and time requirements of the fine-tuning process are two of the most significant drawbacks of fine-tuning code language models. To get good performance on a specific task, one may need to fine-tune larger models with higher GPU memory requirements. A number of studies have also indicated that training for longer periods

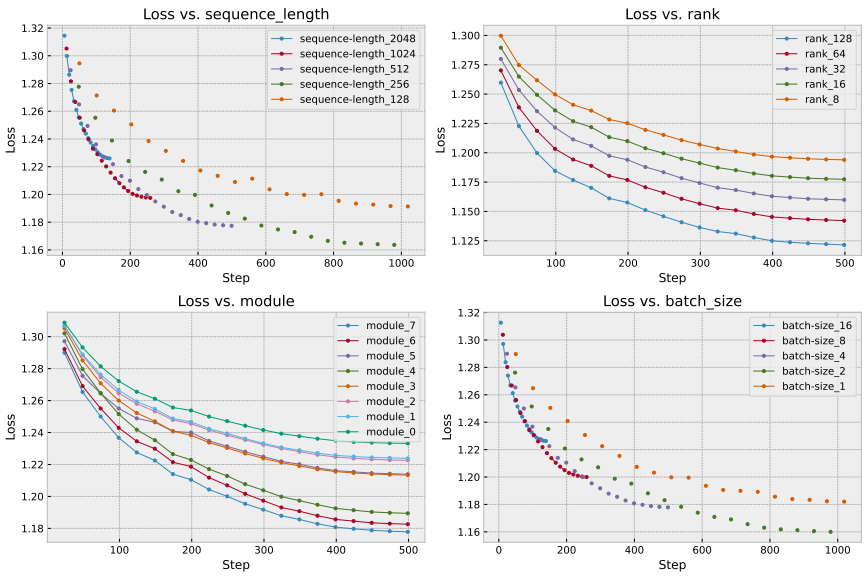


Figure 4.3: Ablation Studies

of time (more steps and epochs) may be advantageous when pre-training large language models[18]. These prerequisites must be met in order to make AI development accessible to everyone.

There are even more intricate variations in the relationship between time, memory, and performance. For example, fine-tuning only a subset of the model's parameters may deliver the same performance as full fine-tuning in a given amount of time. This could be because of the choice of parameters to be fine-tuned [**?**]or because the smaller model saw a greater number of training tokens in the same time span. On the other hand, slowing the iterations by increasing the batch size could increase the performance, as larger batch sizes are associated with more precise weight updates and better guided learning curves.

In this experiment, we run a trade-off analysis to see if there are any patterns that can help us achieve our goal of making fine-tuning and using code language models more accessible. To imitate real-world conditions, we confine our experiment to a specified memory size and time window. The performance of our baseline model was then measured using the evaluation loss on 50,000 tokens of Java code. We chose a new set of parameters for each fine-tuning run to change the memory allocation and duration of the process. We will discuss these criteria and provide insights into the obtained outcomes in our analysis.

4.3.2 Analysis

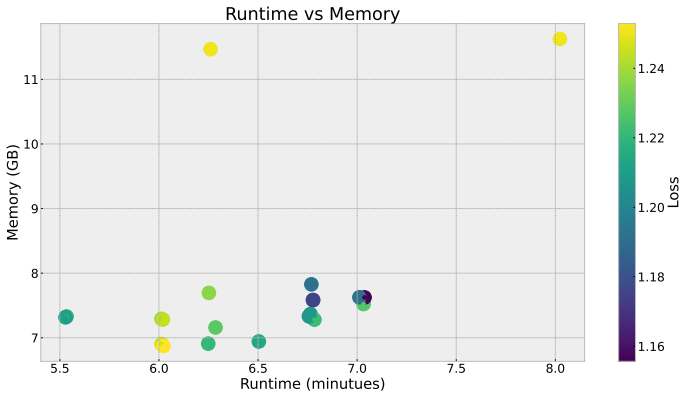


Figure 4.4: Trade Off Analysis

4.4 Fine-tuning on Low resource languages

4.4.1 Objective and Setup

Large corpora of code crawled from online sources such as GitHub and Stack Overflow are frequently used to train code language models. These datasets typically include a significant number of programming languages, such as TheStack [22], which includes 30 programming languages, and the dataset collected by [41], which includes 12 languages in addition to natural languages. These programming languages, however, are not uniformly distributed in the dataset, as popular languages such as Java and Python typically account for a considerable amount of the data. These variations are typically mirrored in models that have been pre-trained on such datasets, resulting in better performance in popular programming languages and worse performance in low-resource ones.

To overcome this issue, a variety of methods have been offered, including an equal sampling of different programming languages [32]. Our methods, on the other hand, take a different approach to overcoming this challenge. We remove the biases in the datasets by utilising monolingual baselines and then training separate LoRa adapters for each programming language. This also benefits code language model users, as they can simply use an adapter in their unique programming language or easily train and share it. These adapters can also be fine-tuned on a more specialised dataset, such as a Python for web development dataset.

To test our approach, we fine-tuned our baseline model in four different programming languages: Java, Ruby, Rust, and Swift. As demonstrated in Section 3.2 and Figure 3.3a, 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. The following section contains details on the various training parameters as well as a comparison of the results of the evaluation.

4.4.2 Analysis

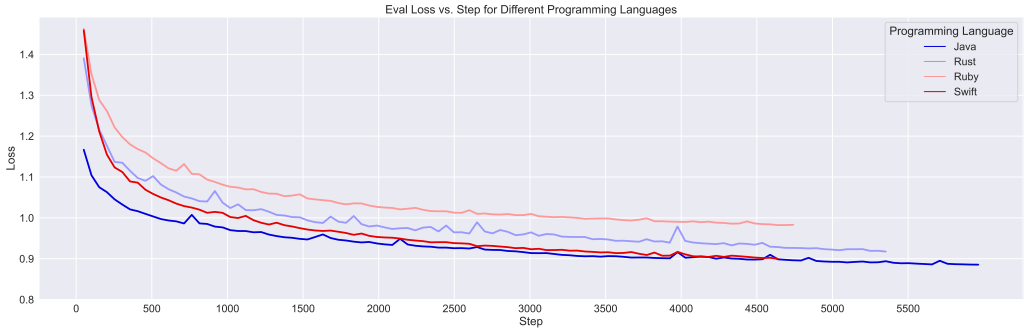


Figure 4.5: Evaluation Loss For Target Programming Languages

4.5 Error Analysis

4.5.1 Objective and Setup

In our final experiment, we conducted error analysis on our fine-tuned models in order to better understand and obtain insights into their performance. We investigate the code produced by efficiently fine-tuned models, fully fine-tuned models, and both monolingual and multilingual baseline models. The code is generated as responses to prompts in different programming languages, taken from our evaluation dataset [10]. We make an effort to identify any systematic errors exhibited by the models that reveal the shortcomings of our approach. We additionally follow [9]'s work and categorise some of the prompts depending on their type and level of difficulty. This knowledge will enable us to develop the models in the future and identify the most effective uses for them now.

4.5.2 Analysis

Category	Theme	Example
Highest Performing Problems	Single operations	def sum_to_n(n: int) ->int: """sum_to_n is a function that sums numbers from 1 to n. 3 """
	Common type questions	def fib(n: int) ->int: """Return n-th Fibonacci number. """
Lowest Performing Problems	Problems with multiple subproblems	def separate_paren_groups(paren_string: str) ->List[str]: """ Input to this function is a string containing multiple groups of nested parentheses. Your goal is to separate those group into separate strings and return the list of those. Separate groups are balanced (each open brace is properly closed) and not nested within each other Ignore any spaces in the input string. """ (Sub-Problems: Remove Spaces, Identify Parentheses Groups, Balanced Parentheses Checking, Extract Substrings, Combine Results)
	Solving a more common version of the problem	def intersperse(numbers: List[int], delimeter: int) ->List[int]: """ Insert a number 'delimeter' between every two consecutive elements of input list 'numbers' """ return list(map(lambda x: x if x % 2 == 0 else x + delimeter, numbers)) (the model try to insert the delimiter between even elements instead of every two consecutive elements)
	Miscellaneous errors	def parse_music(music_string: str) ->List[int]: """ Input to this function is a string representing musical notes in a special ASCII format. Your task is to parse this string and return list of integers corresponding to how many beats does each not last. Here is a legend: 'o' - whole note, lasts four beats 'o—' - half note, lasts two beats '.—' - quater note, lasts one beat parse_music('o o— .— o— o— .— .— .— .— o o') >>>[4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4] """ (Specialized Problem)

Table 4.1

Chapter 5

Conclusions

Bibliography

[1] colab.google.

[2] Meta AI is sharing OPT-175b, the first 175-billion-parameter language model to be made available to the broader AI research community.

[3] Scale virtual events.

[4] Hugging Face – The AI community building the future., August 2023.

[5] Nur Ahmed and Muntasir Wahed. The de-democratization of ai: Deep learning and the compute divide in artificial intelligence research, 2020.

[6] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars!, 2023.

[7] Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms, 2019.

[8] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash

Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023.

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

[10] 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.

[11] 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.

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[13] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. Pangu-coder: Program synthesis with function-level language modeling, 2022.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

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

[17] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[18] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[19] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, 2019.

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

[21] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of bfloat16 for deep learning training, 2019.

[22] 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.

[23] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning, 2021.

[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023.

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[26] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.

[27] 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.

[28] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization, 2019.

[29] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.

[30] 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.

[31] OpenAI. Gpt-4 technical report, 2023.

[32] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. Measuring the impact of programming language distribution, 2023.

[33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[34] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[35] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai, 2019.

[36] Elizabeth Seger, Aviv Ovadya, Ben Garfinkel, Divya Siddarth, and Allan Dafoe. Democratising ai: Multiple meanings, goals, and methods, 2023.

[37] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[38] Nimit S. Sohoni, Christopher R. Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report, 2022.

[39] Marcos Treviso, Ji-Ung Lee, Tianchu Ji, Betty van Aken, Qingqing Cao, Manuel R. Ciosici, Michael Hassid, Kenneth Heafield, Sara Hooker, Colin Raffel, Pedro H. Martins, André F. T. Martins, Jessica Zosa Forde, Peter Milder, Edwin Simpson,

Noam Slonim, Jesse Dodge, Emma Strubell, Niranjan Balasubramanian, Leon Derczynski, Iryna Gurevych, and Roy Schwartz. Efficient methods for natural language processing: A survey, 2023.

[40] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022.

[41] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

[42] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration.