Scaling Down Multi-Lingual Code Language Models

Ammar Khairi

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

The democratisation 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. Additionally, we explore the impact of different choices in fine-tuning LLMs, providing empirical evidence on training efficiency in terms of time, cost, and performance.

In pursuit of these goals, we fine-tune a Python-based mono-lingual code LLM for Java, Rust, Ruby, and Swift. Utilising the LoRa Parameter Efficient Fine-Tuning technique, we share these models, their training datasets, and evaluation results openly. Extensive hyperparameter tuning, trade-off analysis, and error analysis shed light on the effects of training process choices. This work contributes to the democratisation of AI by making Code LLMs more accessible and usable for practitioners. By addressing computational barriers and providing insights into training dynamics, we bridge the gap between AI development and practical application, fostering an environment where code intelligence tools can be readily adopted.

Keywords: ***Large Language Models, code intelligence, democratisation of AI, fine-tuning, programming languages***

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.

(*Ammar Khairi*)

Acknowledgements

First and foremost I would like thank my supervisors Prarit Agarwal and Camille Tiennot for their guidance, insightful conversations and helpful feedback through all the stages of this project.

A special thanks to Prof. Iain Murray for his great efforts in ensuring that I find all the resources required for the completion of this work.

Finally I want to express my gratitude for my Family for all their support throughout the year, without them I could not have done this.

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 [59]. (ii) Learning Algorithms: The Pre-training and Fine-tuning learning paradigm made models more flexible for handling a number of specialised tasks [23](iii) Scale: better performance can be expected by increasing the model parameters, the data, or the computation according to the scaling laws[35]. These discoveries resulted in excellent performance in program synthesis challenges and have been adapted to both commercial [18] and open-source [39] tools.

Accessibility to code LLMs 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 [3] and Stability [4] 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 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. On the other hand, users of these tools often have to choose between the high inference cost of running models locally or the monetary cost of access via hosted APIs [2]. In order to achieve the benefits of democratization of AI use and development[55], 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 explore the feasibility of using transfer learning on small pretrained models to new programming languages. We will train small code language models to perform code completion tasks [41] in a variety of programming languages. Additionally, to address the ambiguity in fine-tuning LLMs, we will provide empirical evidence on the effects of different choices on the training process in terms of time, cost, and performance.

1.3 Contribution

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. All the models are shared publicly along with their respective training datasets and evaluation results [1]. We also did extensive hyperparameter tuning, trade-off computations, and error analysis to provide empirical evidence on the effects of different choices on the efficiency of the training process have 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.

[1] https://github.com/ammarnasr/LLM-for-code-intelligence.git

- 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 analysis 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. 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 [62]. Despite the benefits of scaling, it creates significant hurdles to making these advances available in resource-constrained situations [7]. To address these limitations, there has been a renewed emphasis on research aimed at improving model efficiency [58].

When implementing efficient training methods, there are many factors to consider. The work of [54] defines efficiency 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, [57] 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 [57] and compare approaches based on the previous criteria and whether a method is for training, inference, or both, as shown in Table 2.1.

The methods in Table 2.1 cover a variety of techniques. Sparsity [44] is a technique

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

Table 2.1: Summary of Efficient Training Methods and their Effects on Efficiency & Performance. The ↓ indicates that a certain methods **decreases** either the memory usage, process run-time or fine-tuning performance based on the designated column, and the ↑ symbol indicates an **increase**. The **?** indicates different behaviours, while the 'Checkmark' and 'Dash' symbols show whether or not a method have an effect on a specific criteria.

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 [65]. 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 [34]. The work of [42] studies the trade-off between reduction in memory and reduction in parallelism when using smaller batch sizes (micro-batching). 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 [42], [19]. The work of [21] 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 [48]. Pruning and trained quantization [28] are model compression methods that can be used to enable inference on memory poor edge devices.

Parameter 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 [32] adds trainable parameters to each transformer block of the model. Another technique known as prompt-tuning [38] relates to techniques that modify the input prompt to improve modelling results. Finally, the Low-Rank Adaptation method (LoRa) [33] minimizes the number of trainable parameters by approximating the weight updates of the LLM through learned low-rank matrices inserted at different layers of the model.

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 [24], are used to provide representations for code sequences, which are useful in code comprehension tasks like clone detection and code classification. CodeT5 [61] is an encoder-decoder model that can be used for conditional generation tasks like code generation from natural languages. The work of [20] has demonstrated that left-to-right code (decoder only) language models, such as SantaCoder[9] and PolyCoder [63], 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 [53] and perplexity are simple to calculate and language agnostic. These measurements, however, have been demonstrated to have small correlation with code generation quality [18]. 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 [18] and the Mostly Basic Programming Problems (MBPP) [12] 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 [16] adds 18 new languages to HumanEval and MBPP, whereas [11] adds over 10 new programming languages to HumanEval, MBPP, and Math-QA [10]. Finally, the BabelCode and TP3 [49] benchmarks give fine-grained (unit test level) code completion evaluation in 14 different

programming languages. A recent approach to evaluating code generation is known as LLM Grading [27]. This method uses a very capable LLM (GPT-4 [48]) to grade candidate solutions. The main advantage of this method is that it gives a fine-grained signal of the model's coding capabilities and eliminates the need for writing unit tests manually, which can be very demanding.

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 agreement among experts when talking about the objectives, approaches, potential dangers, and advantages of initiatives aimed at achieving AI democratisation [55]. The work of [8] refers to AI democratisation solely on the basis of the decentralisation of knowledge production in the field. Whereas [55] 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 [13].

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 [14] 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 [8] and [31] advocates scaling down so that ideas are not pushed aside for lack of compatibility with available 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 [45]. Additionally, as we discussed in Section 2.1, 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 calls for national research cloud [5].

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 Figure 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 causal language modelling. However, we use the CodeGen-350M-mono [47] 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 [36], 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. 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 [43] and dataset cards [26] are automatically generated to be shared with the model on open source platforms such as HuggingFace [6] (Figure 3.2).

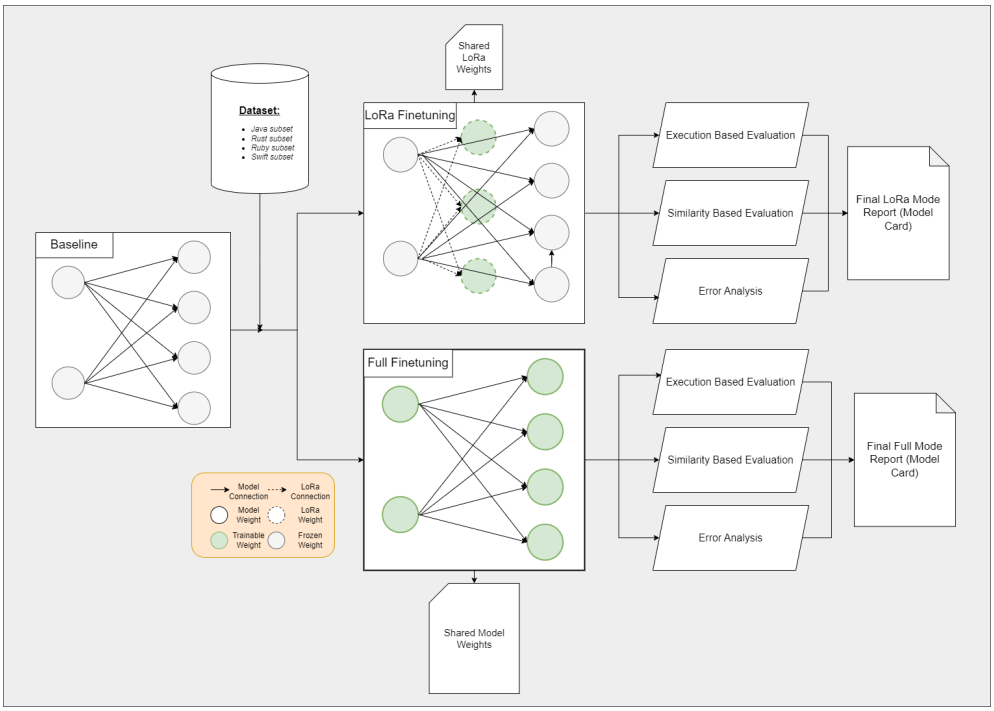
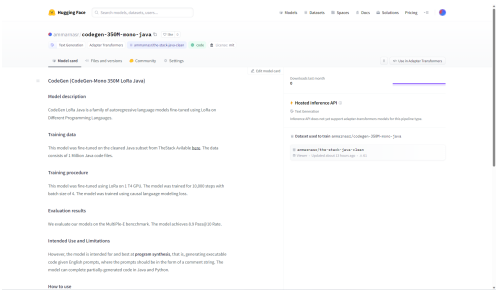
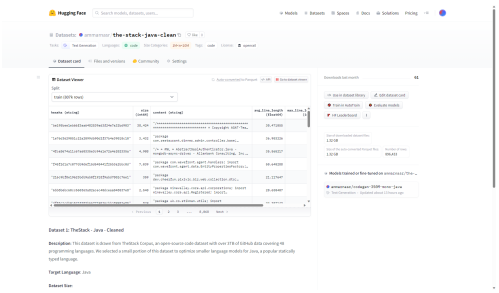


Figure 3.1: Project Diagram



(a) Model Card



(b) Dataset Card

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

3.2 Dataset

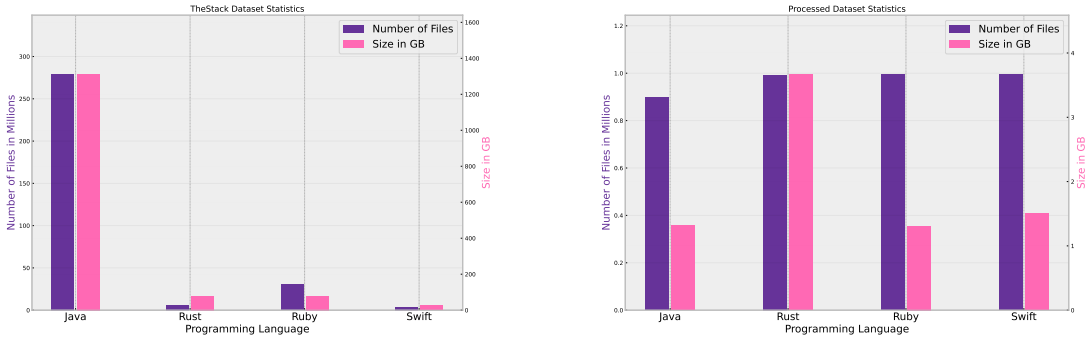
Our training datasets are primarily drawn from TheStack Corpus [36]. TheStack is an open-source code dataset with more than 3TB of GitHub data covering 48 different programming languages. We only use a small portion of this dataset because our experiments only consider a small subset of programming languages 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. These classifications are made on the basis of the quantity of files each programming language have in the training corpus as defined by [49]. 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 practices [63] and filtered out files with the following characteristics:

- An average line length of over 100 characters
- A maximum line length of over 1000 characters
- a ratio of alphabet to numeric characters 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 [47]. These Byte Pair Encoding (BPE) [56] tokenizers add special tab and white space tokens to the same GPT-2 vocabulary [52]. The training sequences are produced by first tokenizing the code in all the files in the dataset. Then these tokens are concatenated to form sequences of length 2048 tokens, with a special separator tokens to indicated different files. We also use sequences of different length for different experiments.



(a) TheStack Dataset [36] Statistics

(b) Processed Datasets Statistics

Figure 3.3: Comparison Of Datasets before and after Processing. The stats in (a) are taken from [36], while the numbers in (b) are the results of our data preprocessing pipeline.

3.3 Baseline Model

The careful selection of the baseline model is important in the context of this project, as it fundamentally underpins the subsequent phases of our fine-tuning methodology. In this section, we discuss our choice of the CodeGen-350M-Mono as our designated baseline model, offering a short analysis of its distinctive attributes and limitations.

The compactness of CodeGen-350M-Mono at 350 million parameters was a pivotal factor in the choice of the CodeGen model family as our baselines [47], [46]. The advantage of a compact model extends beyond the immediate computational efficiency gains when fine-tuning. Small models are more friendly in deployment and dissemination, enabling swifter accessibility and utilisation for a diverse array of practitioners.

Additionally, the CodeGen-350M model family includes equivalent multilingual models that share the same architecture and training procedures as the monolingual models. This inherently positions CodeGen-350M-Multi as a robust benchmark against which to measure the efficacy of our proposed approach. A notable drawback of the monolingual baseline is that its weights were initialised using a model trained on a corpus including a mix of programming languages and natural languages [25]. This limitation, however, applies to practically all code language models because it is considered standard practices in the area [63].

3.4 Training Objective

In this section, we will briefly discuss the training objective we used for fine-tuning the code language models to new programming languages. The self supervised auto-regressive language modelling approach [51] is used for both complete fine tuning and LoRa fine tuning. In this approach, we take a large corpus of unlabeled text and set up a task to predict the next word based on the words that came before. Mathematically, the training objective is expressed as the minimization of a loss function:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t|\mathbf{x}_{<t}) \tag{3.1}$$

In this equation, $\mathcal{L}$ represents the loss incurred during training, T signifies the sequence length, and x_t denotes the token at position t. The conditional probability $P(x_t|\mathbf{x}_{<t})$ captures the likelihood of the observed token given the preceding context, encapsulating the essence of the masked language modeling task. Notably, in the context of code generation, the token sequence $\mathbf{x}$ encompasses either code fragments to be completed or natural language instructions that require translation into code.

3.5 Full Fine-tuning

The concept of full fine-tuning is key to the pre-train and fine-tune paradigm, which has proven to be extremely effective in a variety of natural language processing tasks. In this section, we will go through the mechanics of Full fine-tuning and how we implemented it in the context of code language models.

Full fine-tuning requires retraining all model parameters for a specific downstream task. The work of [23] demonstrated the applicability of this strategy to a wide range of NLP problems and introduced feature fine-tuning, which involves fine-tuning a specific subset of model layers. In this work, we study the efficiency and performance trade-off inherent in full fine-tuning compared to other efficient fine-tuning methods. Our implementation of full fine-tuning involves the re-training of the baseline 350 million parameters casual language modelling objective in a new programming language (Java). We use this primarily as:

- benchmark to the feasibility of our objective of extending monolingual language models to a new programming language on a small dataset

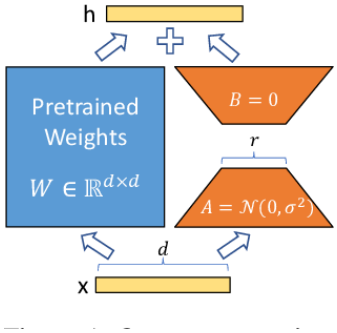


Figure 3.4: Figure adapted from [33]. Only The low rank matrices A and B are trained.

- A baseline for the resources required in order to achieve this objective and possible gains from using more efficient methods

3.6 Efficient Fine-tuning

Efficiency is a critical aspect in the training of large language models. This section covers the technical aspects of efficient fine-tuning, including parameter-efficient fine-tuning (PEFT) algorithms and training optimisation techniques. We describe the methods we used to optimise our training process, as well as their intended behaviour and implementation details.

The first aspect of efficient fine-tuning centres around parameter-efficient fine-tuning (PEFT). This direction relies on optimising the fine-tuning process by reducing the number of trainable parameters. A central framework of PEFT is The Low-Rank Adaptation Method (LoRa). This approach employs a method where it fixes the pre-trained model weights and introduces adaptable rank decomposition matrices into different layers of the Transformer architecture 3.4. Hence, it significantly minimises the number of trainable parameters. When compared to the fine-tuned GPT-3 175B using full fine tuning, LoRA achieves a 10,000x reduction in the number of trainable parameters and a 3x decrease in GPU memory requirements. Finally , LoRA demonstrates comparable performance to full fine-tuning on various large language models, with fewer parameters and without introducing inference latency like other PEFT methods such as Adaptors [32].

The second dimension of efficient fine-tuning addresses the optimisation of the training process. In Section 2, we show the variety of methods available; however, we will only use three of them: Quantization, Gradient Check-pointing, and Gradient Accumulation These techniques, while distinct in their mechanisms, converge in their

shared goal of enhancing training efficiency by minimising memory limitations. Quantization refers to the execution of some or all of the operations on tensors with reduced precision rather than full precision (floating point) values. This allows for a more compact model representation and the use of high performance vectorized operations on many hardware platforms. In the context of fine tuning code language models, this allows to halve the resource requirement (when 16-bit representation is used instead of 32-bit representation) without significantly affecting the precision of our model.

Most of the memory requirements when training code language models come from the optimizer saved states during back propagation. Gradient check-pointing allows us to minimise the memory footprint of the optimizer at the cost of an extra forward pass per batch, which significantly slows down training. Finally, we use Gradient accumulation to achieve a larger effective batch size without increasing the memory cost by accumulating smaller mini batches. This method also has a significant overhead in terms of computation. Throughout the project, we use these methods in various combinations based on the objective of the experiment and the other parameters.

3.7 Inference & Evaluation

The evaluation of code generation language models typically involves two interrelated phases: (i) Inference, wherein code is generated based on specific conditions; and (ii) Evaluation, which gauges the effectiveness of the generated code within a designated task. Both of these phases involve intricacies that can significantly impact the evaluation outcomes. In this section, we delve into the intricacies of these processes, explain their effects, and present our evaluation framework.

As described in Section 2, code language models create their results auto-regressively. The model begins with input text, which is then used to predict the next token, which is then used to produce the next. Decoding is the process of selecting output tokens to generate text, and there are various decoding strategies that differ in terms of efficiency and performance. The work of [18] showed that improved performance can be obtained through repeated sampling, whereas [17] improved efficiency through the use of speculative sampling. In this study, we use nucleus sampling with a *top-p* = 0.95, a *temperature* of 0.8, *10 samples* per prompt, each with a maximum length of *500 tokens.* Nucleus Sampling was proposed by [30], as an alternative to likelihood maximization strategies such as beam search which tends to generated repetitive and generic text when used as the decoding strategy. Nucleus sampling avoid this by limiting the search space

to words that accounts for the majority of the probability mass(the nucleus). The choice *top-p* effects the size of the nucleus; large values (close to 1) mean smaller nucleus and hence higher quality generation, smaller value lead to larger nucleus to sample from giving more diverse generations.We select a relatively high *top-p* value to ensure high quality code, while the high *temperature* skews the probability distribution to ensures our 10 samples are sufficiently different.

Evaluation paradigms include match-based and execution-based paradigms. Match-based evaluation involves calculating the degree of similarity between a generated code candidate and a reference code. This metric makes use of n-gram-based measurements such as Perplexity (PPL). Perplexity can be useful in informing us about how familiar our model is with programming language syntax, but it is limited in terms of capturing semantic aspects specific to code [50]. The Execution-based evaluation approach, on the other hand, comprises executing the generated code candidate through a series of test cases to determine its success rate. The work of [37] proposed assessing functional correctness using the pass@k measure, which generates k code samples per problem, considers a problem solved if any sample passes the unit tests, and reports the total fraction of problems solved.

In this work, we use both evaluation frameworks. Perplexity is simple to calculate and is evaluated directly based on the test splits on our cleaned datasets (Section 3.2). As for execution-based benchmarks, we use the MultiPL-E benchmark [16], which includes prompts and test cases in 18 programming languages. We set $k = 10$ and report an adjusted version of the *pass@k* metric that calculates an unbiased estimator to account for the high variance of the metric [18]. Equation 3.2 shows the unbiased estimator, where k is the pass rate, c is the number of correct samples, and n is the number of total samples.

$$pass@k := \mathop{\mathbb{E}}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{3.2}$$

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 hyperparameter tuning on the parameters of LoRa fine-tuning to understand the contribution of each factor to the process. In Section 4.3 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 [11] 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 [60], 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 [18], 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 [47]. 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 [47]), and the multi version is trained on a mix of four languages (Python, Java, JavaScript, and C) from TheStack dataset [36]. Then, we fine-tuned the mono model on the Java subset of the preprocessed dataset discussed earlier. We employed Full and LoRa fine-tuning using the same casual language modeling objective [51], 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 hyperparameter tuning Section 4.2.

4.1.2 Analysis

The results in Figure 4.1 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 4.2 , 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 fine-tuning, respectively. The training was done on a Google Colab T4 15GB GPU and lasted for around 12 hours.

The Loss curves in figure 4.1 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 4.2, 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.

We can also see from Figure 4.2 the effects of catastrophic forgetting, as the Python score for full-fine-tuning is 0.0 while the LoRa score is 3.75, which is also a significant drop from the 22.36 in the monolingual 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 [11].

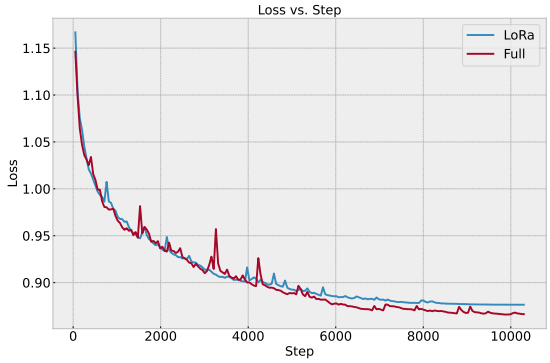


Figure 4.1: Comparison of Loss Curves for LoRa and Full Fine Tuning

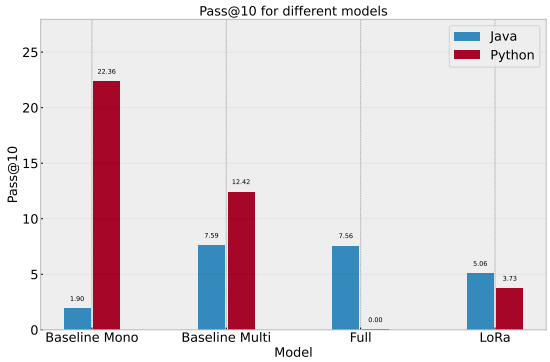


Figure 4.2: Comparison of Pass@10 for LoRa and Full Fine Tuning vs the Baselines

4.2 Hyperparameter Tuning

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 [46], [49]. 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 [29]. 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.

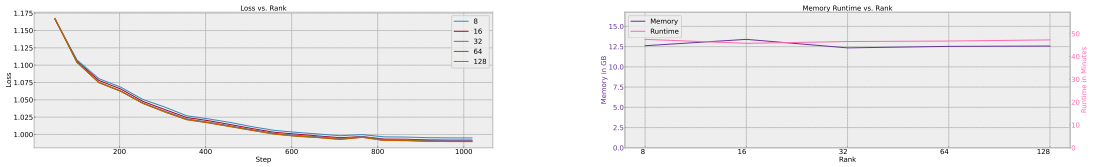
We conducted comprehensive hyperparameter tuning in this experiment to provide empirical evidence that practitioners can use to guide parameter selection. We study the effect a selected number of parameters have on the efficiency and performance of fine-tuning code language models with LoRa. The following 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 our first experiment in Section 4.1. Then, for each of the hyperparameter, 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

LoRa Rank This variable relates to the rank of the decomposed matrix that will be optimized rather than the high rank weight matrix of the model. The choice of this value immediately translates to the number of trainable parameters models in the model and, thus, the efficiency and quality of the fine-tuning process. According to [33] findings, even low rank values (2 or 3) can achieve good performance, even when the full rank is as high as 12,288 as in the case in the GPT-3 attention module [15]. We expected similar behaviour when fine-tuning our code language models. To test this hypothesis, we choose a range of ranks (8, 16, 32, 64, 128), then train and evaluate our baseline for 1000 steps on the Java dataset (as we will do in all the hyperparameter tuning). The values chosen for LoRa Rank range from %0.8 trainable parameters to 12% of the model's total parameters 4.1. Figure 4.3a shows that the evaluation loss remains consistent as the rank increases, against what we expected. This can be explained by the fact that beyond a certain rank size, the matrices injected as approximation to weights update

Rank	Trainable Parameters	Percentage
8	3214336	0.893%
16	6253568	1.724%
32	12332032	3.343%
64	24488960	6.427%
128	48802816	12.040%

Table 4.1: Different Lora Ranks Number of Parameters



(a) Evaluation Loss

(b) Memory and Run-time

Figure 4.3: Performance and Efficiency For Different LoRa Ranks

are effectively equivalent. Finally, Figure 4.3b indicates that the increased number of trainable parameters has no influence on efficiency; the difference in parameters is insignificant in terms of processing even up to almost 50 million parameters.

LoRa Target Modules These modules refer to specific components within the LoRa framework that are targeted for optimisation during the fine-tuning process. The selection of target modules plays a crucial role in determining the overall impact of optimisation on the model's performance. For example, focusing on modules related to the vocabulary distribution ("lm_head") might yield different results compared to targeting modules responsible for semantic understanding ("qkv_proj"). Additionally, different modules affect the efficiency of the training differently, as each module has a specific size and number of blocks in the model. Table 4.2 summarises the selection of modules for this experiment and the number of trainable parameters for each. The results in Figure 4.4 show that best performance is achieved when we insert LoRa parameters across all layers of our model (*Conf-7*). It can also be seen that the gains in performance from the inclusion of Fully connected with the attention modules (*Conf-4, Conf-5, Conf-6*) are significantly higher than including specific language specific layers (*Conf-1, Conf-2, Conf-3*). Regarding efficiency measures in Fig 4.5 we see the different modules choices have minimal effect on the run-time. However, In terms of memory usage , the addition of fully connected modules requires results in less memory usage than adding language feature layers, even though the latter have fewer parameters.

Index	Modules	Trainable Parameters	Percentage	Comments
0	[”qkv_proj”]	5417984	1.497%	Attention Modules (Query, Key and Value)
1	[”qkv_proj”, ”out_proj”]	8039424	2.205%	Attention + Output Projection
2	[”qkv_proj”, ”lm_head”]	8760320	2.398%	Attention + Language Modelling Head
3	[”qkv_proj”, ”out_proj”, ”lm_head”]	11381760	3.094%	Attention Modules with Language Features Modules
4	[”qkv_proj”, ”fc_in”]	11971584	3.249%	Attention + Input Linear Layers (1024 → 4096)
5	[”qkv_proj”, ”fc_out”]	11971584	3.249%	Attention + Output Liner Layers (4096 → 1024)
6	[”qkv_proj”, ”fc_in”, ”fc_out”]	18525184	4.939%	Attention with Fully Connected Modules
7	[”qkv_proj”, ”out_proj”, ”lm_head”, ”fc_in”, ”fc_out”]	24488960	6.427%	All Modules

Table 4.2: Different Possible Target Modules Configurations

We suspect this is primarily due to hardware specification , as GPUs can vectorize operations in Fully connected layers more effectively.

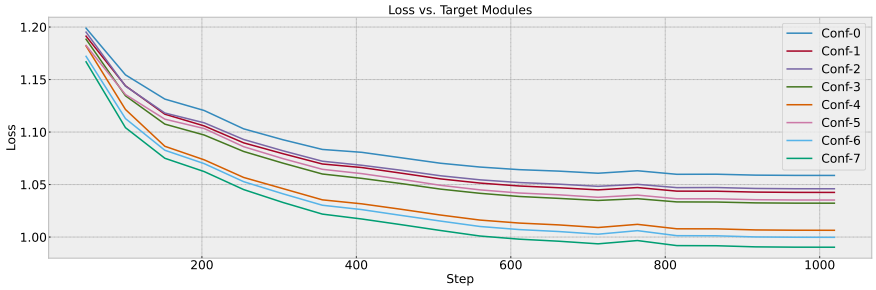


Figure 4.4: Evaluation Loss For Different Target Modules Configurations

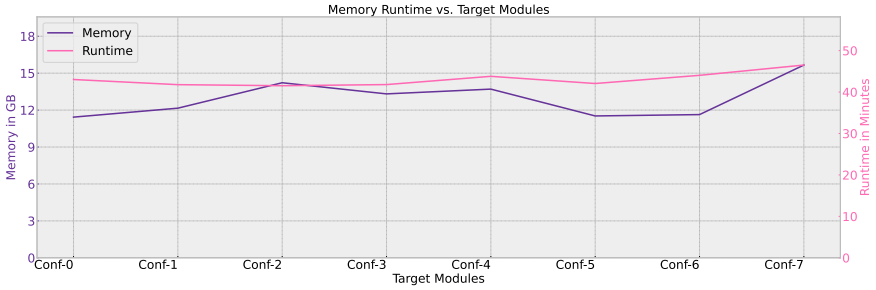
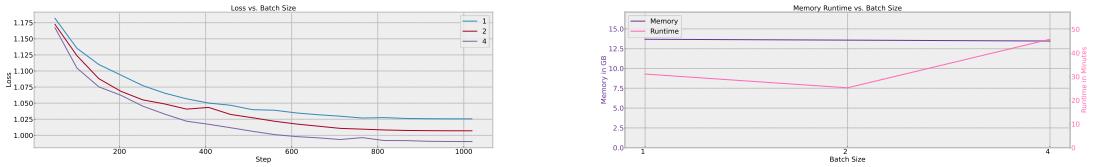


Figure 4.5: Memory and Run-time For Different Target Modules Configurations

Batch Size Batch size is a fundamental hyper-parameter that affects the training process by determining the number of samples processed before updating the model's weights. A larger batch size can lead to more stable gradients and potentially faster convergence, but it also requires more memory for processing. We select batch sizes of 1, 2 4, due to resource constraints. Figure 4.6a illustrates the relationship between batch size and evaluation loss. We can observe that smaller batch sizes convergence faster, but larger batch sizes catch up and eventually surpass the smaller ones in terms of performance. On the efficiency side, Figure 4.6b shows that all batch sizes allocate

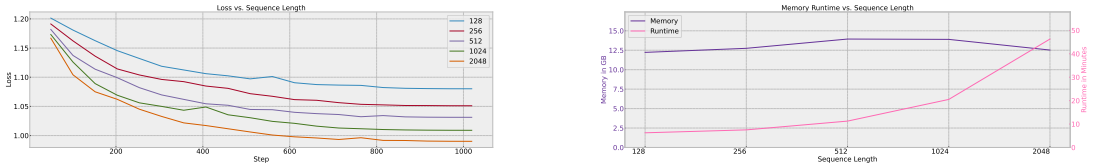
a maximum of about 14GB during training, this also can be attributed to the fact that GPU memory is allocated in powers of 2. That's way any batch higher than 4 needs to allocate extra blocks of memory and thus exceeding our 15GM limit. The run-time increases proportionally with the batch size.



(a) Evaluation Loss (b) Memory and Run-time

Figure 4.6: Performance and Efficiency For Different Batch sizes

Sequence Length refers to the length of input sequences used during training. In the context of code language models, longer sequences might capture more contextual information but also demand more computational resources. To investigate the impact of sequence length on fine-tuning, we experiment with sequence lengths of 128, 256, 512, 1024 and 2048 tokens. As shown in Figure 4.7a, longer sequences lead to improved performance, as they capture more context. . Efficiency-wise (Figure 4.7b), shorter sequence lengths are more memory-efficient and require fewer computations, but they sacrifice some performance.



(a) Evaluation Loss (b) Memory and Run-time

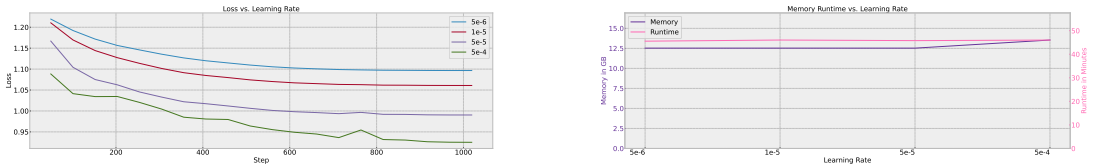
Figure 4.7: Performance and Efficiency For Different Sequence Lengths

Learning Rate is a key hyper-parameter that controls the step size during gradient descent. It greatly influences the convergence speed and stability of the training process. We explore learning rates of 5e-6, 1e-5, 5e-5, and 5e-4. Figure 4.8a demonstrates the effect of learning rates on evaluation loss. A higher learning rate (5e-4) initially leads to faster convergence, but we speculate it might become unstable and result in fluctuating performance, as we can see from Figure 4.7a it has the least smooth line of the the four. On the other hand, lower learning rates (5e-6 and 1e-5) exhibit slower but steadier convergence. The optimal learning rate seems to be around 5e-5 for this experiment.

Variable	Initial Value	Range	Optimal Value (based on loss)
LoRa Rank	64	Table: 4.1	128
Target Modules	Conf-6	Table: 4.2	Conf-7
Batch Size	2	[1, 2, 4]	4
Sequence Length	1024	[128, 256, 512, 1024, 2048]	2048
Learning Rate	5e-5	[5e-6, 1e-6, 5e-5, 5e-4]	5e-5

Table 4.3: Hyperparameter Tuning Summary

As for efficiency (Figure 4.8b), the impact of learning rate on memory and time is not substantial compared to other parameters.



(a) Evaluation Loss

(b) Memory and Run-time

Figure 4.8: Performance and Efficiency For Different Learning Rates

Summary The results of this hyperparameter tuning confirm that to large extent that better performance and increased computation cost are closely related. However, the gain in performance, increase memory requirement and longer run-time are all dependant on the parameters choice which results in different behaviour. Table 4.3 summarize this section and report the optimal value for each parameter, which we will use to train the final models in 4.4, while the next section explores the trade-off between efficacy and efficiency for different parameters in more details.

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

of time (more steps and epochs) may be advantageous when pre-training large language models[29].

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 [32], 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 performed 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. These patterns can then be utilised in resource constrained environments to find a suitable balance for the specific use case. To imitate real-world conditions, we confine our analysis to a specified memory size and time window (15GB of memory and 60 minutes of run-time). The performance of our baseline model was then measured using the evaluation loss on about 8 millions 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

To explore the connections among memory usage, run-time, and performance, we conducted fine-tuning tests on Java code, running for 1000 steps based on the initial parameters setup in Table 4.3. In order to assess the impact of various hyperparameters, we employed the same configuration as detailed in the hyperparameter tuning section. However, our focus was directed towards four parameters out of the available five, as adjusting the learning rate exhibited negligible influence on memory consumption and run-time. Thus, we maintained the learning rate at a constant value that yielded optimal results in Section 4.2.2, specifically 5e-4. Regarding the remaining four parameters (batch size, sequence length, Rank, and Modules), we executed experiments spanning the ranges outlined in Table 4.3. Subsequently, we extracted maximum memory allocation, run-time duration, and memory loss from each trial to formulate the data for Figure 4.9.

Figure 4.9 depicts a scatter plot of the experiments based on their memory usage and run-time, with the upper-right corner indicating setups that demand the most resources. Markers are colour-coded to reflect the parameter being modified among the four (as shown in the legend), while the marker size represents the loss observed in the experiments(bigger circles corresponds to larger loss). The value the parameter takes is shown by the index inside the circle, which references the ranges outlined in Table 4.3 for clarity.

In terms of the run-time axis, a notable observation is that the majority of experiments fall on the right side, surpassing 40 minutes of run-time. Experiments featuring shorter sequence lengths and smaller batch sizes tend to exhibit lower run-times. That because we measure the run-time as time to complete 100 steps, shorter sequences and smaller batches take less time per step due to the lower number of computation. Interestingly, extremely short sequence lengths notably degrade performance, whereas reductions in batch size yield more modest effects on performance. However, opting for shorter sequence lengths leads to reduced memory allocation, while the memory allocation for different batch size values remains constant. Experiments involving changes in rank display similar loss values and resource requirements as observed in experiment 4.2.2. Lastly, it's evident that altering target modules solely affects memory allocation while maintaining constant run-time. The minimum loss occurs when both run-time and memory are at their limits, reinforcing the direct relationship between scaling and performance.

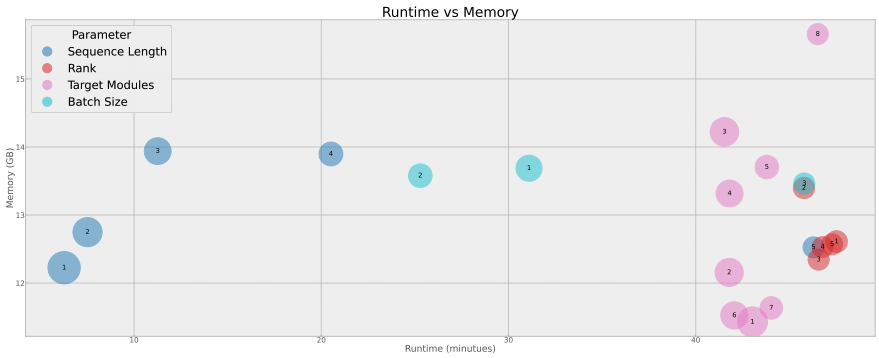


Figure 4.9: Trade Off Analysis between Memory, Run-time and Performance for Different Parameters

4.4 Fine-tuning on Low Resource Ranguages

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 [36], which includes 30 programming languages, and the PolyCoder dataset [63], 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 temperature based sampling of different programming languages [49]. 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 Competitive Programming dataset [40].

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

In this experiment, we fine-tuned our baseline using the same hyper-parameter setup, LoRa configuration, and number of tokens for our four programming languages. Similar to experiments in Section 4.1, we used the CodeGen-Mono baseline with its GPT-2 tokenizer and fine-tuned the model on the processed datasets from Section 3.2. 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. Optimisation 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. These parameters were selected based on the findings from the hyperparameter tuning results in Table 4.3, and training optimization methods (Gradient Check-pointing & FP16) were selected based on the review from Section 2.1 to enhance fine-tuning process without affecting the performance.

The results in Figure 4.10 show the loss on the evaluation split every 50 steps for each model. We notice that all the models follow a similar trajectory. The loss falls sharply in the early steps , and as the training progresses, the graphs flatten, indicating convergence. However, there are a few noteworthy artefacts upon closer examination of the graphs and the training data. The first observation is that the Java loss curve starts significantly lower when compared to all other models. We hypothesise that this is related to the fact that the Baseline model has already seen Java code in its pretraining due to the data spill described in experiment 4.1. The second observation is concerning the Ruby model. We notice that the loss converges at a higher value compared to the other programming languages. That is due to the fact that Ruby shares a lot of similarities with Python as they both use indentation instead of braces, are dynamically typed, and share a number of keywords. These similarities confuse the model as it frequently generates Python code instead of Ruby, as we will show in the error analysis section (Figure 4.15). The work of [11] reported similar observations in the task of code translation.

The Pass@10 rates are illustrated in Figure 4.11, 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, 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. It's noteworthy that the application of parameters recommended from the hyperparameter tuning results yields a 37% enhancement in performance compared to models fine-tuned in experiment

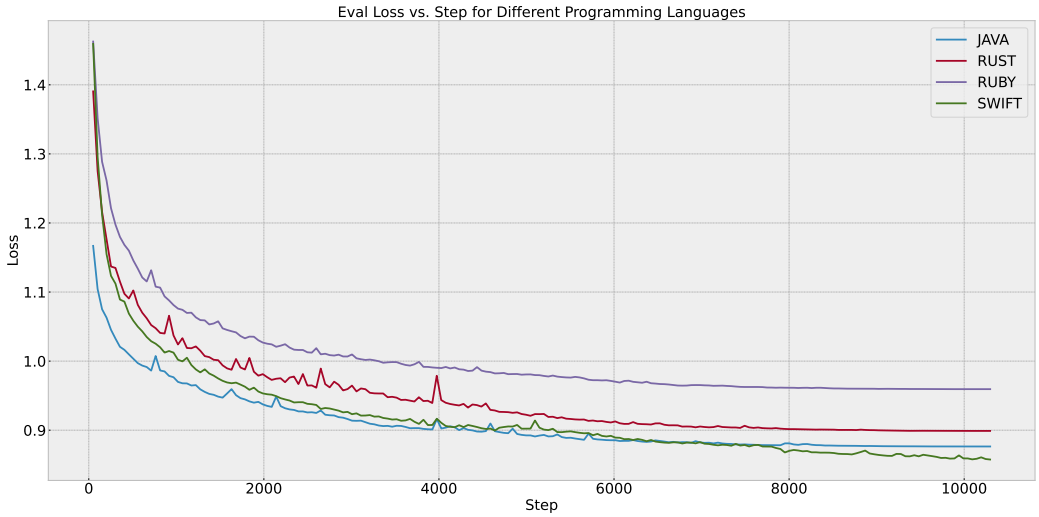


Figure 4.10: Evaluation Loss For Target Programming Languages

4.1. The remaining models for Ruby, Rust, and Swift exhibit improvements in their respective programming languages with negligible results in all other languages.

In the calculation of Perplexity, our focus was exclusively on the four target programming languages: Java, Ruby, Rust, and Swift. The test splits from the processed datasets were used, and perplexity was calculated over 100 files using a stride of 1024 tokens. The overall average score for each of the six models was reported across the four target languages and show in Figure 4.12. Perplexity serves as a similarity-based metric, and therefore, these results primarily reflect the models' grasp of syntax rather than their reasoning and logic, a distinction from the Pass@10 score. However, these results do offer insightful observations. Firstly, the notably higher perplexity (worse score) observed for the Ruby language across all models (excluding the Ruby model) can be attributed to its similarity with Python, as discussed earlier. Secondly, it's evident that fine-tuning on one programming language consistently leads to worse perplexity scores in other languages compared to the monolingual baseline. This finding aligns with [11]'s discovery that small language models struggle to effectively incorporate multilingual knowledge across different languages, as is often the case with larger language models.

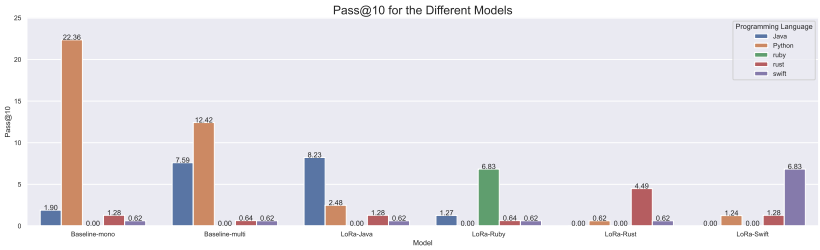


Figure 4.11: Pass@10 Rates Across Different Programming Languages by Fine-tuned models and Baselines

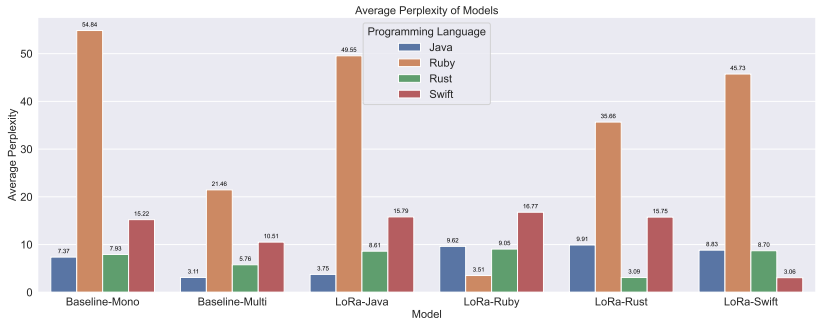


Figure 4.12: Perplexity Across Different Programming Languages by Fine-tuned models and Baselines

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 the fine-tuned models from Section 4.4. The code is generated as responses to prompts in different programming languages, taken from our evaluation dataset [16]. We make an effort to identify any systematic errors exhibited by the models that reveal the shortcomings of our approach. We additionally follow [12]'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

The outcomes presented in Section 4.4 underscored the enhanced performance of all four of our fine-tuned models in comparison to both the monolingual and multilingual

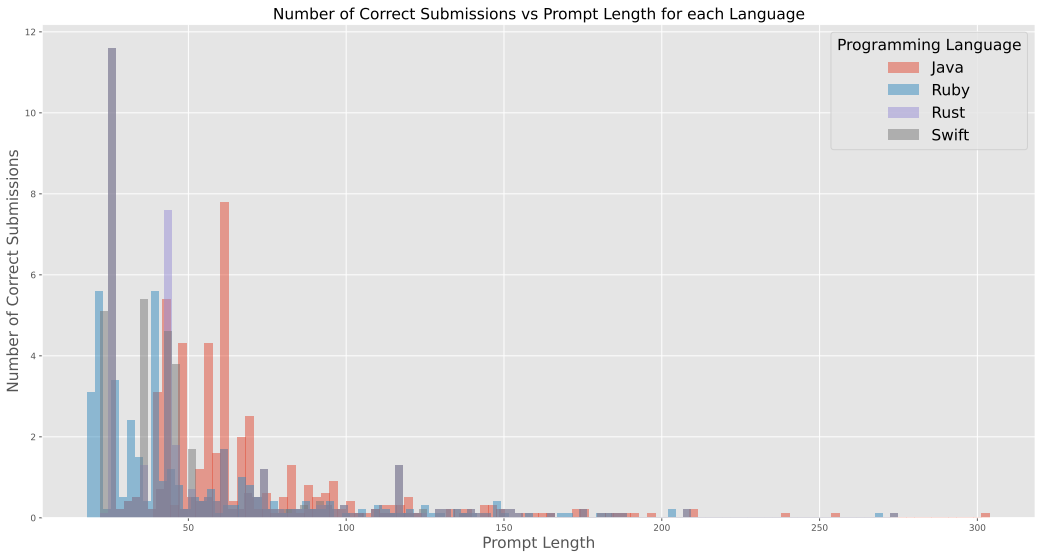


Figure 4.13: Distribution of Number Correct Solutions vs The Length of the Problem of the for Different Programming Languages

```python
from typing import List

def median(l: List[int]) -> float:
    """Return median of elements in the list l.
    >>> median([3, 1, 2, 4, 5])
    3
    >>> median([-10, 4, 6, 1000, 10, 20])
    15.0
    """


def check(candidate):
    assert candidate([3, 1, 2, 4, 5]) == 3
    assert candidate([-10, 4, 6, 1000, 10, 20]) == 8.0
    assert candidate([5]) == 5
    assert candidate([6, 5]) == 5.5
    assert candidate([8, 1, 3, 9, 9, 2, 7]) == 7

def test_check():
    check(median)

test_check()
```

Figure 4.14: Example Problem in Python from MultiPl-E; The import line and the function signature with the docstring are the problem prompt. The *check()* function is then executed on the generated code to verify it.

baselines on the MultiPl-E benchmark. In this experiment, we delve deeper into these findings by scrutinising the code generated by the models for specific prompts. Our initial focus involves analysing the distinct challenges posed by the problems in the MultiPl-E benchmark. A representative problem from the dataset is illustrated in Figure 4.14. The prompt encompasses the code required to link necessary modules, the function signature, and the accompanying docstring. Subsequently, unit tests are employed to evaluate the code generated by the code language model in response to the prompt. The model generates 10 solutions for each prompt, and if any of these solutions passes all unit tests, it is considered solved, aligning with the Pass@10 metric formulation we discussed in Section 2.1.

In order to assess our models' performance concerning specific prompts, we construct a histogram that plots the number of correct solutions (up to 10) against the length of the prompts. Notably, Figure 4.13 illustrates that all solutions pertain to shorter problems with prompts containing fewer than 100 words, despite the fact that 23% (37 out of 161) of the prompts exceed this word count. However, the challenge does not stem solely from prompt length, as our models were trained on sequences up to 2048 tokens in length and achieved favourable perplexity outcomes even with the use of large strides values (longer context).

This observation leads us to believe that prompt length primarily serves as a proxy for problem difficulty. Further examination reveals that certain shorter problems are consistently unsolvable by all models, as demonstrated in the example in Figure 4.16. Given these insights and drawing from the work conducted by [12], we categorize problem difficulty into three distinct classes, as outlined in the accompanying Table 4.4:

- Complex problems: Typically longer in nature, these problems entail intricate logic that necessitates a series of steps to reach a solution.
- Misleading Problems: Characterised by ambiguous phrasing, these problems possess simple solutions; however, the wording often misleads the model into addressing a different problem or producing code in an unintended language
- Hard problems: Unlike complex problems, these challenges are rendered difficult due to factors other than their size. This category encompasses problems that are inherently challenging, possibly due to their unique nature.

Illustrative examples and generated samples for each of these problem categories are showcased in Figures 4.17, 4.15, and 4.16, respectively.

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 4.4: Examples Problems taken from MultiPL-E [16] covering different categories and themes.

```ruby
# Filter an input list of strings only for ones that contain given substring
# >>> filter_by_substring([], 'a')
# []
# >>> filter_by_substring(['abc', 'bacd', 'cde', 'array'], 'a')
# ['abc', 'bacd', 'array']
def filter_by_substring(strings, substring)
  return [s for s in strings if s.find(substring)!= -1]
```

Figure 4.15: An example of a misleading problem in Ruby. The prompt code is similar to the of Python as both use the Keyword (*def*). The model is confused by that and uses Python inline for syntax to complete the code; resulting in a syntax error.

```swift
/// For a given list of input numbers, calculate Mean Absolute Deviation
/// around the mean of this dataset.
/// Mean Absolute Deviation is the average absolute difference between each
/// element and a centerpoint (mean in this case):
/// MAD = average | x - x_mean |
/// >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
/// 1.0
func mean_absolute_deviation(numbers: [Double]) -> Double {
    var sum = 0.0
    var count = 0
    for number in numbers {
        sum += abs(number)
        count += 1
    }
    return sum / Double(count)
```

Figure 4.16: An example of a specialized math problem in Swift. Even though the problem is not necessarily complex and the steps are explicitly shown, this proved a tricky problem for all the models. Here the swift model closest attempt end up calculating the mean of absolute values instead of the mean absolute deviation.

```java
import java.util.*;
import java.lang.reflect.*;
import org.javatuples.*;
import java.security.*;
import java.math.*;
import java.io.*;
import java.util.stream.*;
class Problem {
    // 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 array 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.
    // >>> separateParenGroups(("( ) (( )) (( )( ))"))
    // (new ArrayList<String>(Arrays.asList((String)"()", (String)"(())", (String)"((())()")))
    public static ArrayList<String> separateParenGroups(String paren_string) {
        ArrayList<String> arr = new ArrayList<String>(Arrays.asList(paren_string));
        while (arr.size() > 1) {
            int[] i = new int[arr.size()];
            for (int i_ = 0; i_ < arr.size(); i_++) {
                int[] i_2 = new int[arr.size() - 1];
                for (int i_3 = 0; i_3 < arr.size() - 1; i_3++) {
                    int[] i_4 = new int[arr.size() - 1];
                    for (int i_5 = 0; i_5 < arr.size() - 1; i_5++) {
                        int[] i_6 = new int[arr.size() - 1];
                        for (int i_7 = 0; i_7 < arr.size() - 1; i_7++) {
                            int[] i_8 = new int[arr.size() - 1];
                            for (int i_9 = 0; i_9 < arr.size() - 1; i_9++)
```

Figure 4.17: An example of a complex problem in Java. The problem require a number of sub-problems to be solved first show in Table 4.4. Here the model just try to brute force a solution and trail off until the maximum number of tokens is reached.

Chapter 5

Conclusions

The journey towards democratising 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 democratisation in AI. Through the extension of a mono-lingual code LLM to encompass multiple programming languages, we have demonstrated the potential of these models to empower developers across diverse landscapes, while staying computationally efficient. The empirical insights gained through extensive experimentation shed light on the intricacies of fine-tuning code LLMs. These insights equip practitioners with valuable knowledge to navigate the complexities of model training, save resources, and ultimately drive innovation more effectively.

All the undertakings in this project are driven by the fundamental aim of democratizing code language models and ensuring seamless access for all to these effective and valuable tools. Nevertheless, the aspiration of AI democratisation must not be pursued thoughtlessly, as there are several issues that need to be addressed, as highlighted in other research works [55]. Within the context of code LLMs, one of the most prominent concerns is the use of unlicensed code for training these models. As our contributions involve sharing training datasets and models, we have made a deliberate effort to exclude code from GitHub repositories that is not properly licenced. We acknowledge that language models could potentially replicate code verbatim or be employed for unintended purposes, including IP infringement[64], [22]. Thus, we recommend that future directions in this domain strike a balance between advancing the democratisation of AI and addressing the potential consequences that may arise as a result.

Bibliography

[1] colab.google. https://colab.google/. [Online; accessed 2023-08-10].

[2] GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot. [Online; accessed 2023-08-17].

[3] Meta AI is sharing OPT-175b, the first 175-billion-parameter language model to be made available to the broader AI research community. https://ai.facebook.com/blog/democratizing-access-to-large-scale-language-models-with-opt-175b/. [Online; accessed 2023-08-08].

[4] Scale virtual events. https://exchange.scale.com/public/videos/emad-mostaque-stability-ai-stable-diffusion-open-source. [Online; accessed 2023-08-08].

[5] Stanford experts call for national resource for ai research. https://www.axios.com/2020/04/01/ai-research-stanford. [Online; accessed 2023-08-15].

[6] Hugging Face – The AI community building the future. https://huggingface.co/, August 2023. [Online; accessed 2023-08-10].

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Abeba Birhane, William Isaac, Vinodkumar Prabhakaran, Mark Diaz, Madeleine Clare Elish, Iason Gabriel, and Shakir Mohamed. Power to the people? opportunities and challenges for participatory AI. In *Equity and Access in Algorithms, Mechanisms, and Optimization*. ACM, oct 2022.

[14] Abeba Birhane, Pratyusha Kalluri, Dallas Card, William Agnew, Ravit Dotan, and Michelle Bao. The values encoded in machine learning research, 2022.

[15] 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. Language models are few-shot learners, 2020.

[16] 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.

[17] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.

[18] 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.

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

[20] 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.

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

[22] Tim Davis. Tim Davis on X. https://twitter.com/DocSparse/status/1581461734665367554, October 2022. [Online; accessed 2023-08-16].

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[24] 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.

[25] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

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

[27] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.

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

[29] 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.

[30] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020.

[31] Sara Hooker. The hardware lottery, 2020.

[32] 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.

[33] 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.

[34] 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.

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[36] 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.

[37] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code, 2019.

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

[39] 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.

[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi 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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, dec 2022.

[41] 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.

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

[43] 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.

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

[45] Fiona Murray, Philippe Aghion, Mathias Dewatripont, Julian Kolev, and Scott Stern. Of mice and academics: Examining the effect of openness on innovation. *American Economic Journal: Economic Policy*, 8(1):212–252, 2016.

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

[47] 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.

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

[49] 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.

[50] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. On the difficulty of benchmarking inductive program synthesis methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1589–1596, 2017.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

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

[53] 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.

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

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

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

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

[58] 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.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[60] Tu Vu, Aditya Barua, Brian Lester, Daniel Cer, Mohit Iyyer, and Noah Constant. Overcoming catastrophic forgetting in zero-shot cross-lingual generation. *arXiv preprint arXiv:2205.12647*, 2022.

[61] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859, 2021.

[62] 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.

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

[64] Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. Codeipprompt: Intellectual property infringement assessment of code language models. 2023.

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