Efficient Finetuning of LLMs for Domain-Specific Code Generation

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Barish Sarkhel

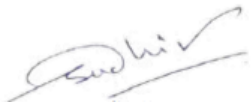
Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2025

Supervisor: Mr. Sudhir Kumar

Certificate

This is to certify that this dissertation entitled Efficient Finetuning of LLMs for Domain-Specific Code Generation towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Barish Sarkhel at Coriolis Technologies Pvt. Ltd. Pune under the supervision of Mr. Sudhir Kumar, Member of Technical Staff, Coriolis Technologies Pvt. Ltd. Pune , Department of ML-AI , during the academic year 2024-2025.



Mr. Sudhir Kumar

Committee:

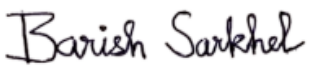
Mr. Sudhir Kumar

Dr. Leelavati Narlikar

This thesis is dedicated to my mother,
whose light still guides me beyond the stars.

Declaration

I hereby declare that the matter embodied in the report entitled Efficient Finetuning of LLMs for Domain-Specific Code Generation , are the results of the work carried out by me at the Department of ML-AI, Coriolis Technologies Pvt. Ltd., Pune, under the supervision of Mr. Sudhir Kumar, and the same has not been submitted elsewhere for any other degree.



Barish Sarkhel

Acknowledgments

I express my gratitude to my supervisor, Mr. Sudhir Kumar, for his invaluable guidance, encouragement, and unwavering support throughout this journey. His insights and expertise have been instrumental in shaping this thesis, and I am truly grateful for his patience and mentorship. I am grateful to him for showing confidence in me during challenging times. Additionally, I sincerely thank Mr. Rohan Nandode for diligently supervising the progress of this work and for providing valuable techniques and insightful ideas throughout the process.

I extend my sincere appreciation to Dr. Leelavati Narlikar, whose expertise and constructive feedback have greatly enhanced the depth and quality of my research. Her keen observations and scholarly advice have been a source of inspiration.

I sincerely thank the members of Coriolis Technologies Pvt. Ltd. for their unwavering support in every situation and for assisting me from the very beginning of this project. A special thanks to Mr. Sagar Jagatap, Mr. Shreyashkumar Sharma, Mr. Rajanish Khushawa, and Mr. Pranav Thube, for taking the time to review our work despite their busy schedules. I am also deeply grateful to Bhushan Deshpande, my friend and project partner, for his constant support throughout this journey. I extend my heartfelt appreciation to Dr. Basant Rajan, CEO of Coriolis Technologies Pvt. Ltd., for granting me the opportunity to work in such an inspiring and enriching environment.

My heartfelt gratitude goes to my parents—my mother, whose love and sacrifices will always guide me, and my father, whose unwavering support helped me navigate the toughest moments of life. I am also thankful to my elder sisters, whose constant encouragement and belief in me have been a pillar of strength throughout the journey. I am also deeply thankful to my friends, who have been my pillars of support during this journey. Their constant encouragement, thoughtful discussions, and moments of laughter have made this process

more enriching and enjoyable.

Finally, I extend my appreciation to everyone who has contributed to this work in any way. This thesis is a reflection of the collective support and kindness I have received, and I am truly grateful to all.

Abstract

This work investigates fine-tuning Large Language Models (LLMs) on domain-specific code generation, aiming to enhance their performance in test code generation. The work consists of both small size code dataset and large size code dataset.

It explores fine-tuning techniques for different open source large language models (e.g., Llama, Phi). The primary focus of this work was to meticulously prepare the dataset for the task and to explore efficient fine-tuning techniques that would enable the adaptation of our large dataset while operating within the constraints of limited computational resources. This involved curating and structuring the dataset to ensure its relevance and effectiveness, as well as identifying optimization strategies to enhance the fine-tuning process. By carefully balancing model performance with resource efficiency, this work aimed to achieve optimal results despite computational limitations. The primary focus of this work was to explore efficient fine-tuning techniques. Finding the best model and best set of hyperparameters for the datasets we were using was the main objective.

The project also focus on the impact of the size and complexity of the dataset in fine-tuning task. The project seeks to identify effective fine-tuning methods for both the cases. Ultimately, its goal is to deepen our understanding of fine-tuning LLMs and improve their applicability to specific tasks and domains.We also experiment with Retrieval-Augmented Generation (RAG) and Reinforcement Learning, to enhance model performance by improving response accuracy. By integrating all these techniques and methodologies, the project aims to contribute valuable insights into scalable and efficient model training, ultimately enhancing the robustness and practicality of LLMs for real-world applications.

Contents

Chapter 1

Introduction

Fine-tuning is a widely used technique for adapting open-source Large Language Models (LLMs) to specific use cases for personalized applications. Our work focused on test code generation by finetuning the base LLMs. We focused on the effectiveness of finetuning techniques in complex scenarios. It includes situations when large context length inputs need to be trained while operating under resource constraints. Along with that our works explores techniques which we can apply on top of the finetuned model to improve the quality of its output.

1.1 Objective of Our Research

The main objective of the work is to finetune open source Large Language Models for test code generation. The LLMs which are available as open source model are trained on various type of dataset. When we try to use them for any specific type of work, the performance of these models degrades significantly. Finetuning techniques help to train those models according to that specific domain. Through this technique the model parameters change and distribute in certain direction which is suitable to generate thar particular domain specific output.

This work required a significant amount of searching of base LLMs and finetune them to find out the desired model. Also we need to find out appropriate prompt and best set of

hyper-parameters for that particular model. Finding all of them and training the base LLM model based on them will lead us to the finetuned model which will give the desired output. The detailed of these techniques, dataset, base LLMs, hyperparameters etc. are described in subsequent chapters.

The test code that we want to generate by the LLMs are quite domain and product specific. Though they are written in Python programming language, they maintain a certain specific format and few specific libraries. The base models don't have these specific things as it is trained on a very generic dataset. The models need to learn complex logic to write these codes. Also one of the two dataset we used contain very large size codes. On an average 250 to 300 lines of codes were present for each cases in that dataset. The challenge was to make the model understand those logic properly and along with that generate such large code maintaining the proper formatting.

The work also focused on memory efficient finetuning techniques. We used parameter efficient finetuning techniques. To perform computations within our limited resources, we employed various strategies, including low-bit quantization and advanced optimization techniques.

After extensive search of base model and hyperparameters we can achieve success up to a certain accuracy. To make our output better and increase accuracy we combine our finetuned model with more advanced techniques like Retrieval Augmented Generation (RAG) or Reinforcement learning (RL). In this part our target was to achieve the best output for these complex test case datasets using different techniques over the finetuned model.

1.2 Existing Work in This Domain

This section describes the work done so far that is related to our study. This will give the reader an idea of the development happened in the domain of Large Language Models and where the research is standing currently.

- One of the most important objective of the Large Language Model is to predict the next token based on the context provided. The pioneer works in this field are Recurrent Neural Network (RNN) and Long Short Term Memory Models (LSTM) [1]. These

models theoretically can generate next token up to whatever length we want. But these models and the modified form of these models (For example Gated Recurrent Neural Networks on Sequence Modeling [2]) accuracy were not satisfactory when context length were large.

- One of the most notable invention in the Large Language Models domain was the transformer architecture. The idea of attention was introduced in 2014 [3]. In the work [4], they propose the Transformer model. The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality after being trained for very small time with limited computation cost. This invention become the root of today's modern LLM development.

- After that sequence of development happen in this domain based on that transformer architecture. Bidirectional Encoder Representations from Transformers (BERT) introduced next year which used masked language modeling (MLM), allowing models to learn contextual representations more effectively [5]. BERT model focused more on the attention by training it from both the direction. In that same year OpenAI introduced Generative Pre-trained Transformer (GPT-1), the first large-scale auto regressive model. [6]. They trained using a left-to-right causal language modeling (CLM) objective. The basic transformer architecture of decoder model is described more in detail in Chapter 2.

- The next major improvement in this direction was scaling up the model. In the paper [7] they showed that scaling up the model help in task-agnostic, few-shot performance, sometimes even reaching better then the state of art finetuning approach. They used 175 billion parameters, 10x more than any previous non-sparse language model.

- The subsequent improvement was the introduction of multi modal architecture model which incorporate both text and image dataset and improved in logical reasoning. [8]

- Another set of notable work was done by organizations like Meta and Microsoft. Their Llama and Phi series models showed impressive performance with very small size architecture. [9, 10]. As they made these models open source, these can be use in research and personal work. We extensively used these models in our work. You can find more details about these models in Chapter 4.

- Along with these development, parallel development was happening in the domain of finetuning to use the open source LLMs for domain specific works. BERT was one of

the first Transformer models where fine-tuning was standardized [5]. GPT-1 and GPT-2 models were further trained on domain-specific data [6]. These finetuning techniques were costly as all the parameters were updated for the downs-trim tasks.

- In subsequent years these expensive finetuning techniques were improved by the invention of parameter efficient finetuning techniques. The earlier parameter efficient finetuning techniques includes BitFit [12], Prefix-Tuning [13] etc.

- Advanced finetuning techniques like LoRA introduced in 2021 [14]. To make it more efficient and to make the finetuning possible with limited GPU resources QLoRA technique introduced in 2023 [15]. These techniques discussed in detail in Chapter 3.

- All of the finetuning techniques described above are supervised finetuning techniques. The training happened in those cases using the actual output. In last few years techniques were introduced where Reinforcement learning is done on top of the supervised finetuned model. Some of the techniques are Reinforcement learning through human feedback[RLHF] [28] and Reinforcement learning through AI feedback[RLAIF]. [29].

1.3 Significance and Advantages of Our Work

The work contributes with many crucial observations. The significant advantages achieved through our work are outlined here.

- Test code is a crucial component in the development of any website, ensuring functionality, reliability, and performance. Writing test code is time taking process. The test code which we tried to generate through LLM takes from 4 hour to 2 days while writing manually depending on the size and complexity. Our work significantly lower this time requirements by automating the code generation.

- The technique will generate test code outputs that are either completely correct or correct up to a certain percentage, requiring only minor modifications. It gives the exact framework in which the code need to be written. That is why through this technique a person with very limited knowledge about the product can work on test code generation.

- The work also focused on the situations where we try to finetune model using large context length. The challenges and trade-offs involved in training large-context code models are also discussed here. Also it depicts how the performance of the finetuned model changes with increasing context length.
- Our observations also shed light on the impact of different type of model choices and data pre-processing techniques.
- The work also focused on the techniques which can be implemented on top of finetuned model. It identifies the applicability of those techniques, their effects and the problem associated with them.

Throughout our work, we have made several significant findings, key improvements, and valuable observations that have contributed to a deeper understanding of the subject.

Chapter 2

Basic of Large Language Models (LLMs)

Large Language Models (LLMs) are the most revolutionary development in the field of data science and artificial intelligence. As the name suggests, the primary goal of these types of model is language processing and prediction of the next token of a sequence. These models use neural network along with techniques like attention with an extensive number of parameters to be able to predict the next token. These models can be used for variety of tasks like text generation, summarization of text, image generation from text, code generation, chatbots, etc. Our work focused on the code generation part. Generating code is a complex task because it involves many logical steps connected to each other. The code must be created based on a specific setup and follow a particular format. Due to the development of the LLMs in the last one decade, we can think of generating such complex tasks, as it evolved to remember long contexts.

LLMs are broadly classified into two types which are open-source models and close-source models. The open source models are those models whose weight, architecture, and some cases training data are publicly available. We can use these models for research and development purposes. We have done our finetuning task using some of these model (example: Llama, Phi etc). On the other hand closed source models are those models whose weights, architecture, data etc are not publicly available and these models mostly used in private purpose

(example: GPT model)

2.1 Architectural Framework

The transformer architecture which was proposed by [4] is the heart of Large Language Models. The idea of that work used in developing decoder only model [6]. In this section that architecture is explained. This architecture is used in almost all the base LLMs which are developed in recent time.

The architecture combines few important components. They are described here:

- **Embedding-** The embeddings are the vector representation of tokens. Each token corresponding to one vector representation.

  When we feed one input in LLM, it converted into a sequence of tokens. Each token then converted into vector and this combine create the input X matrix whose size is the size of *input token length* $\times$ *embedding size*.

- **Positional Embedding** In transformer-based models, Positional Embeddings are used to encode the order of tokens in a input sequence since transformers don't have the idea of inherent positional awareness of the tokens. The positional embedding and input token embeddings are added before passing it into the self-attention mechanism.

  A common approach to compute the positional embedding is to use sine and cosine functions. The equations for that is as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

  where:

  – pos represents the position in the sequence,

- i is the dimension index,
- d is the total embedding dimension.

These functions ensure that similar positions have embeddings that are close in vector space while allowing extrapolation to longer sequences.

- **Key, Query and Value-** Key(K), query(Q) and value(V) are three matrices which are used while training a LLM to improve the vector representation of the input. The tokens try to learn and modify the representations based on the other tokens representation. Also it try to understand the relation of that token with other tokens. The technique is called self-attention and it is described next. These matrices are the heart of that process.

- **Self-attention-**Self-attention allows a model to weigh the importance of different tokens when processing an input sequence. This process computes the attention scores of the inputs based on the query Q, key K, and value V matrices which are derived from the input embeddings.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the key vectors, ensuring numerical stability.

- **Multi-head attention-** Multi-head attention is the combination multiple self-attention layers. Different self attention layer learn about different characteristics of the dataset. At the end all the matrix are concatenated.

  Let, Z be the output of this layer. Here Z=[z_1, z_2, z_3,...] where z_i is the output matrix of i th self attention layer. The size of the Z matrix will be $(size\ of\ input\ sequence) \times (number\ of\ self\ attention\ layer \times embedding\ size)$.

- **Output matrix-** The size of the matrix need to same as the input matrix before putting into feed-forward neural network. A learnable matrix O which is called output matrix do this task.

$$O = Z \times W_O \tag{2.1}$$

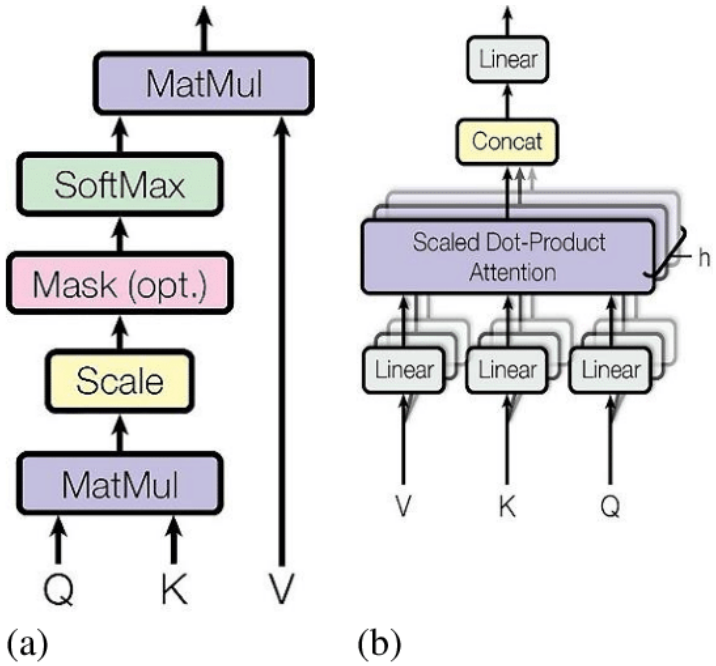


Figure 2.1: Self-attention and Multi head attention

- **Feed-forward neural layer-** A fully connected layer applied to each token independently, transforming representations through nonlinear activation. Usually SiLU function is used as activation function here.

- **Layer Normalization-** Layer Normalization is a technique that normalizes the inputs across the features for each training example, improving stability and convergence in deep learning models. It is defined as:

$$\hat{x}_i = \frac{x_i - \mu}{\sigma + \epsilon}$$

where: - x_i is the input feature, - $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$ is the mean of the inputs, - $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$ is the standard deviation, - ϵ is a small constant which provide numerical stability when the denominator is 0.

The normalized value is then scaled and shifted using learnable parameters γ and β:

$$y_i = \gamma\hat{x}_i + \beta$$

where γ and β are trainable parameters that allow the model to learn optimal trans-

formations.

The process starts with converting the input into tokens. Each token which is a number represent one input text token. The inputs then converted into embeddings which is vector representation of that token. Therefore we got the input matrix X. Three weight matrix W_Q, W_K and W_V are multiplied with the X matrix to get the query matrix Q, key matrix K and value matrix V. Different set of these matrix are created and used in each self attention layer of a multi-head attention block. The output again a matrix of size same as X. Now the output feed into a feed-forward (FF) neural network to bring the non-linear behavior above the attention. Each row put as an input of the FF layer and it create the corresponding row of the output matrix. Softmax function applied in that matrix. This provides the output of the next token.

The training of the model is done by calculating loss using the matrix we got after softmax function. The weights of W_Q, W_k, W_V, W_O (Output matrix) and FF layer matrices are learned during training. These are the weights we updated using finetuning technique and the process is described in next chapter. This fundamental architectural framework is utilized in nearly all modern LLMs.

2.2 Training Process

Training of any LLM involves dataset preparation, tokenization, embeddings, and optimizing the weights of the model. Large amount of data set and huge computation capability used in training the base models. The detailed of the base models we used is described in Chapter 4.

The dataset used for training these large models involves publicly available internet data, programming competition datasets,text books, safety-reviewed web content etc. The datasets used for training LLMs undergo extensive pre-processing and filtering to ensure high-quality learning. These datasets are structured to optimize the models for different use cases.

The tokenization is the process of converting the input in tokens which is basically one number representing that word. After that the vector representation of that word is taken out and that representation is used for training. The inputs are given in batches and parameters

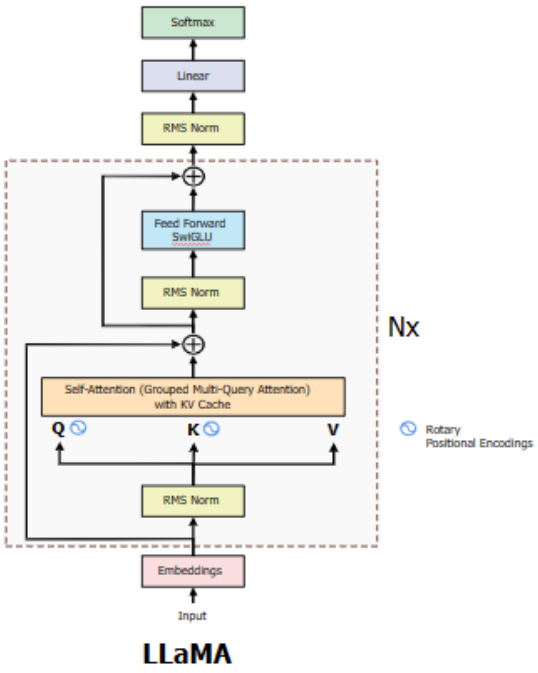


Figure 2.2: The-Decoder-Transformer-model-architecture

get learn based on them.

The training process mainly involved minimizing the loss by updating the weights. Optimizers like AdamW is used for effective training. To train on such huge dataset gradient accumulation and mixed-precision training (FP16/BF16) are used which help in reducing memory overhead while maintaining numerical stability. Many training process incorporate distributing the training process across multiple GPUs, using data parallelism and model parallelism to handle massive datasets efficiently. Also some regularization techniques like dropout and weight decay are used to prevent overfitting. More detailed of there things you can get in next chapter.

Overall these are the processes used while training the model. Every model tries something new in dataset preparation, model architecture or training method to outperform other models. All these processes are extreme costly and time consuming. Thanks to these model using which we can do the downstream tasks for out specific need.

Chapter 3

Finetuning

In the previous chapters, we discussed the basic architecture, training process, and development of base LLMs. These LLMs are developed on a large corpus of data from different fields. They perform quite well for any generic type questions. But when we try to use those models for some very specific tasks, we observe that the performances of these models are not up to the mark. Especially for those tasks where the output we need requires logic and specific knowledge of the product. For example, in our case, we wanted to generate test code those are written with some specific libraries and a specific type of logic. To solve this problem and make the LLMs understand these specific datasets, we use the fine-tuning technique. The basic idea of fine-tuning is to make the LLMs parameters optimize according to the specific dataset that we are using. Through the time it is understood that training all the parameters is quite a huge computation task. To solve this problem parameter efficient finetuning techniques are invented. In this chapter the fundamental developments in the domain of finetuning is discussed along with its implementation in our research.

3.1 Full Parameter Finetuning

As we discussed the main problem of base large language models are their incapability to do specified tasks. The direct approach of solving this problem is to train the base model with the specific dataset. In this case all the parameters of the base model is trained according to the dataset. This technique is known as Full Parameter Finetuning.

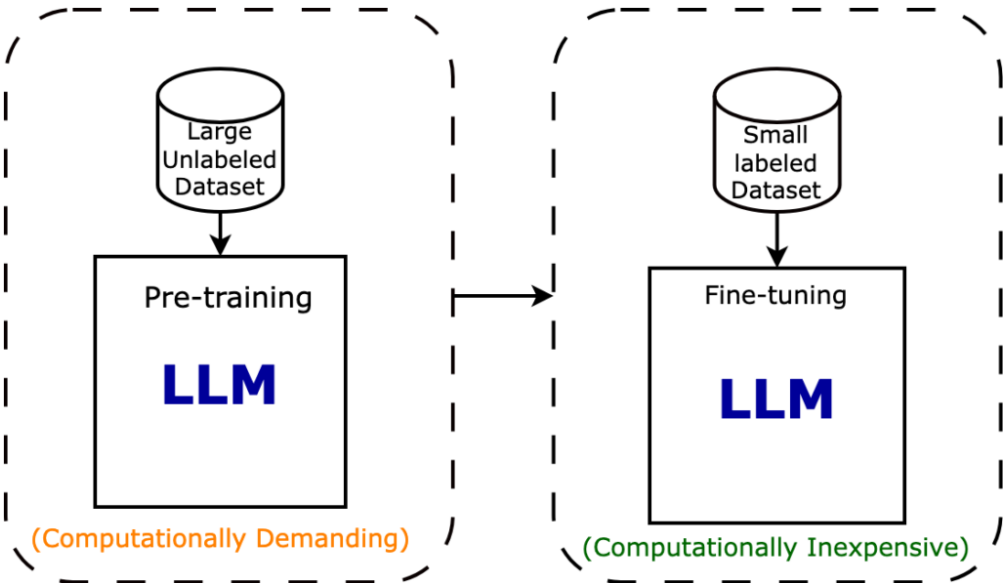


Figure 3.1: Finetune

This technique yields good accuracy if we train it with appropriate hyper-parameters and a large dataset. But this technique comes with its own problem. When we are taking about updating and optimizing the whole pre-trained model we are taking about updating billions of parameters. Huge amount of computation cost is associated with updating these high number of parameters. Also huge amount of domain specific data is needed to perform updating of all parameters.

Using LLMs to automate some specific task for some very big project can be done with this techniques. But most of the generative AI projects in research and industries can not use this technique due to its high computation requirements. To solve this problem different parameter efficient finetuning techniques are invented. We used such parameter efficient finetuning process in our work.

3.2 Parameter Efficient Finetuning Techniques

Parameter Efficient Finetuning Technique (PEFT) brings different methods to finetune a model without updating all the parameters. The core idea is to update a small section of parameters based on the specific dataset. Some most commonly used Parameter Efficient Finetuning Techniques are described here.

3.2.1 BitFit:

BitFit is a finetuning method where only the bias terms of the models or in some cases a subset of those bias terms are modified. The technique is useful as it changing very few parameters for fine-tuned task and require small computation costs. [12] As the amount of update of weights are low, the performance is not up to the expectation level for finetuning task on complex dataset.

3.2.2 Prefix tuning:

Prefix-tuning is also a parameter efficient fine-tuning technique. It freezes the weights of the actual matrices. The is done by adding some task specific matrix which are called prefix with the actual model. It takes inspiration from prompting and allowing subsequent tokens to attend to this prefix. [13]

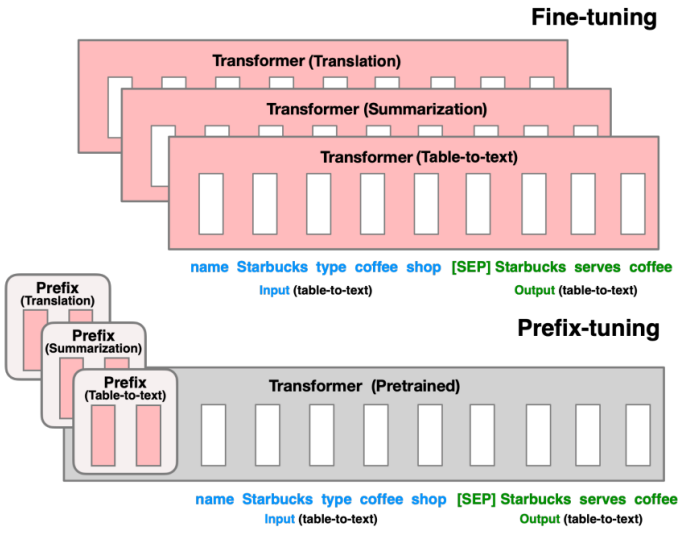


Figure 3.2: Prefix tuning

In our work we did not focus in these techniques as LoRA and QLoRA outperform all these techniques along with use lower GPU memory.

3.2.3 Low Rank Adaptation (LoRA):

Low-Rank Adaptation (LoRA) also freezes the weights of the actual matrices. It injects trainable low rank matrices into each layer of the Transformer architecture. These low rank matrices are trained and through this it reduces the number of trainable parameters for downstream tasks. [14]

Advantages of using LoRA

(a) The technique introduce two low rank matrix A and B for each of the weight matrix and trained them instead of training the complete matrix. It significantly reduces the computation costs and storage.

(b) LoRA technique reduces the computation up to three times compared full parameter update technique since we do not need to calculate the gradients or maintain the optimizer states for most parameters. Instead, we only optimize the injected, much smaller low-rank matrices.

(c) The simple linear design helps the model align with the actual matrix very easily. It does not make any inference with the actual matrix by construction.

Let, the actual matrix is of size $n \times m$ where n and m are very large number. These matrix made freeze. Here two matrices A and B are introduced with size $n \times l$ and $l \times m$, where l is a very small number compared to m and n. The A and B matrices are updated and while generating output A and B multiples and added with the freeze pretrained model matrix.

3.2.4 Quantized Low Rank Adaptation (QLoRA):

QLORA is an efficient finetuning approach build on the top of LoRA technique. This reduces memory usage enough while preserving full 16-bit finetuning task performance. It backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA).

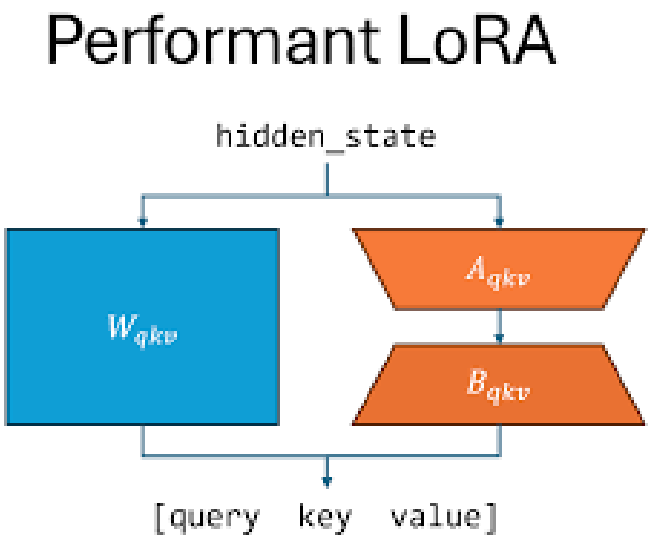


Figure 3.3: LoRA

QLORA introduces some innovative processes to save memory without any significant effect in performance:

(a) 4-bit NormalFloat (NF4) new data type is used. The information of this is theoretically optimal for normally distributed weights. This advanced technique uses specialized spacing based on normal distribution instead of uniform spacing to better preserve information in normally distributed weights. This helps in reducing the loss of information because of quantization.

(b) It uses double quantization to reduce the average memory footprint by quantization constants

(c) Paged Optimizers are used. It helps to manage memory spikes. [15]

As this technique provide one of the best quality of result with very limited resources, this technique is extensively used in our work.

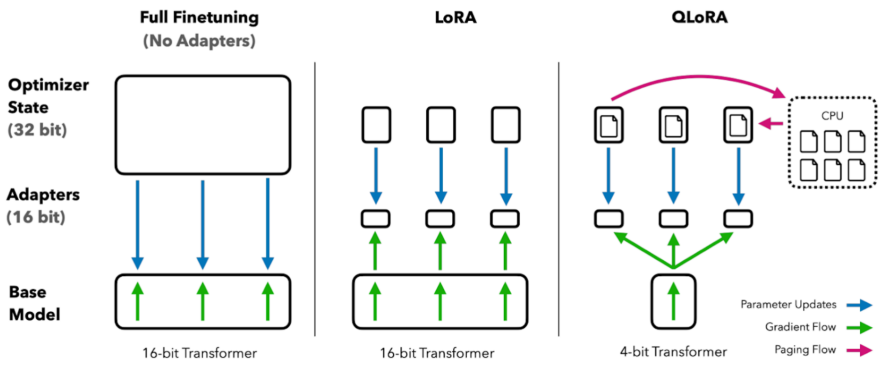


Figure 3.4: QLoRA

3.3 Hyperparameters in Finetuning

Two type of parameters are involved in any LLM training process. One type of parameters are those which got updated during the training process. Mainly weights and biases comes under this parameters. Another set of parameter are those which we fix before starting the training process. These parameters are known as hyperparameters.

Hyperparameters are one of the most important component in any finetuning task. These hyperparameters decides in which pathway the training process will happen. They help in generalizing the training process and make it computational efficient. They also help in perform the training process within the available computation capability. Regularization also done using hyperparameters like weight-decay, drop-out etc. Poor hyperparameters choices can lead to under fitting, over-fitting, or inefficient training. That is why choosing proper hyperparameters is very important. Here the important hyperparameters are defined along with its effects and importance.

3.3.1 Epochs:

Epochs counts the number of times an equivalent of the full training set has been processed. [16] In case of finetuning the specific task dataset process the number mentioned in the epoch. Increase this number helps the model to see the dataset again and again but it also increases the chances of overfitting.

3.3.2 Learning Rate

Let $R(\theta)$ be the objective function we aim to minimize, where θ represents model parameters. At some given step $\theta = \theta_m$, the equation used to compute gradient is as follows:

$$\nabla R(\theta_m) = \left. \frac{\partial R(\theta)}{\partial \theta} \right|_{\theta=\theta_m}$$

This gradient indicates the direction in which $R(\theta)$ changes. To minimize $R(\theta)$, we update θ in the opposite direction:

$$\theta_{m+1} = \theta_m - \rho \nabla R(\theta_m)$$

where ρ is the learning rate, controlling the step size. A properly chosen ρ ensures that each step moves towards a lower function value:

$$R(\theta_{m+1}) \leq R(\theta_m).$$

If the gradient becomes too large, weight updates may fluctuate significantly, disrupting smooth convergence. The learning rate ρ helps regulate these updates, ensuring stable training. [16]

3.3.3 Gradient accumulation steps:

Gradient Accumulation is a very useful technique when we have to train a large size model given limited available GPU memory. Here the gradients are accumulated over multiple batches before updating instead of updating the model parameters after each process. It does not update the parameters immediately after every batch. The gradients are summed up over multiple batches. When a certain n number of batches are passed, the accumulated gradients are used to update the model parameters. [17]

3.3.4 Batch size and Steps

Batch size refers to the number of training samples processed before updating the model's parameters. It plays a crucial role in training stability and computational efficiency.

Steps works as a supplement of epochs in the training process. It help in saving the checkpoints in between the training. The relation between them looks like this:

$$\text{Epochs} = \frac{\text{Steps} \times \text{Batch size} \times \text{Gradient accumulation steps}}{\text{Length of dataset}}$$

Choosing an optimal batch size is essential for smooth computation. Higher batch size help the model train better but it increases the computation cost. That is why we need to find a batch size which is efficient and also trainable with our limited resources.

3.3.5 Optimizer:

An optimizer is an algorithm that updates model parameters based on gradients to minimize the loss function, improving model performance. We used AdamW optimizer, which is a variant of the Adam optimizer that decouples weight decay from gradient updates, leading to better generalization by applying L2 regularization correctly.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \theta_t &= \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \lambda \theta_{t-1} \end{aligned} \tag{3.1}$$

where:

- g_t is the gradient at time step t.
- m_t and v_t are the first and second moment estimates.
- β_1 and β_2 are decay rates for the moment estimates.
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates.

- η is the learning rate.
- λ is the weight decay coefficient.
- ϵ is a small constant to prevent division by zero.
- θ_t represents the updated parameters.

[18]

3.3.6 Rank of LoRA:

In the LoRA technique two small matrices are used to update the weights. The rank of these two matrix are also hyperparameter and we can change them to change the trajectory of our training. [19] Increasing them increases the accuracy of training but demand more computation cost.

3.3.7 LoRA alpha:

This is a hyperparameter which use to scale the values in LoRA matrices. This α value maintains the influence of new trainable parameters introduced through LoRA techniques.

3.3.8 Weight decay:

Weight decay introduce a penalty term to the loss function. This new introduced value encourages the model to learn smaller weights during training.Depending on the the size and complexity of the model, the amount of training data, and the learning rate, the value of the weight decay is decided. It helps to find the right balance between preventing overfitting and underfitting model performance. [20]

3.3.9 Warmup steps:

Warmup steps is part of initial phase of training. Due to this the learning rate is gradually increased from a small value to the target learning rate over a specified number of steps or epochs. [21]

These are the most important hyperparameters used in finetuning technique. In our work we experimented extensively with this hyperparameters.

Chapter 4

Set up and Pre-trained Models

The process of finetuning a model depends on the set up and the resources available for the work. Along with that the base pre-trained models we are used are also very important. The base pre-trained model must be aligned with our requirements to achieve optimal results. The overall set up of our work and the description of the pre-trained models we used in our work are discussed here.

4.1 Set up

The most important part of a finetuning work is the availability of the amount of computation capability i.e. GPU resources we have. The work was started with 20 GB GPU system. While working with the 20 GB GPU we realized that this amount is not sufficient for working with larger context length. So in the second part of our work 40 GB GPU system is used. The GPU type was A 100. Even with the 40 GB GPU system several problems were there, which are described in Chapter 7.

The models we used were taken from Huggingface and Ollama. Mainly models we fine-tuned were from Huggingface and the models we used for validation and other works were taken from Ollama.

The work is done in Python language with version 3.8.10. The most used python libraries

and their versions are given below.

Library used

- transformers (4.46.3)
- torch (2.4.1)
- pandas (2.0.3)
- peft (0.13.2)
- huggingface_hub (0.24.6)
- datasets (3.1.0)

The workflow of our work is given in the picture below.

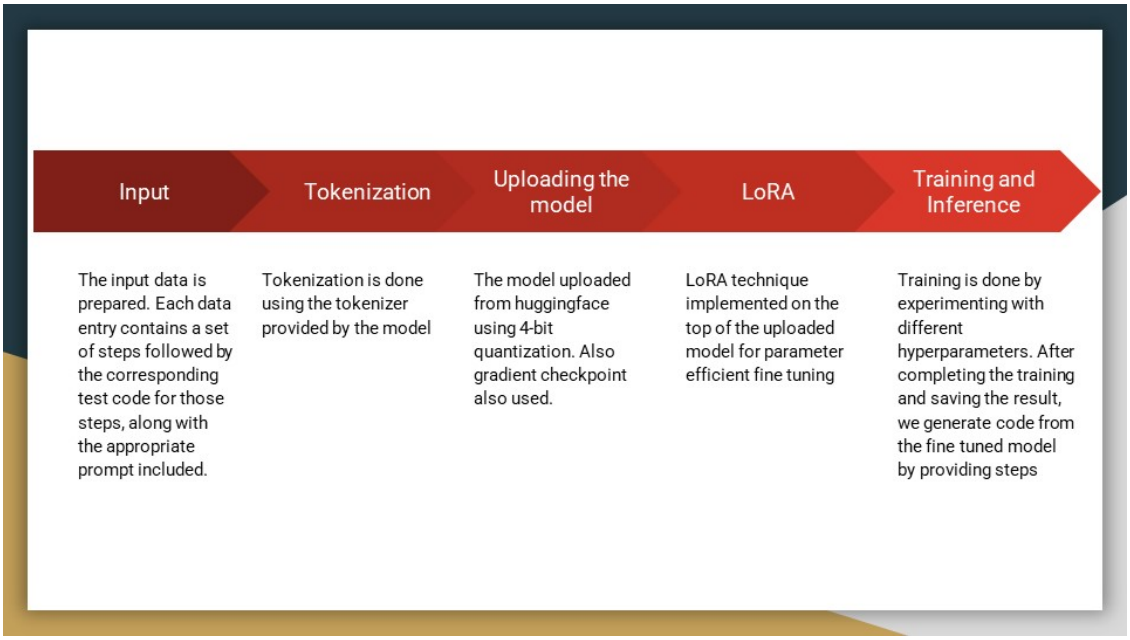


Figure 4.1: Workflow

4.2 Pre-trained Models

Pre-trained models are the backbone of any finetuning task. Open source models plays a vital role in this work. The models we collected from Huggingface. They provide the model along with weights, architecture and results which are helpful and necessary for finetuning. In our work we mainly used Llama series models released by Meta and Phi series models released by Microsoft. This section discusses the details of the models that are extensively used in our work.

4.2.1 Phi

Phi is a series of LLMs released by Microsoft. This models first time introduced in June, 2023. The models are highly efficient and we used the Phi models extensively in our work. We mainly used Phi-2, Phi-3.5 and Phi-4 models. The size of the models are not that large but the level of performance is quite impressive.

Phi-2

Phi-2 is a small-scale LLM but quite efficient and accurate. The model released in December, 2023. We started our finetuning work with small code dataset using this model. This model was pre-trained with 2.78 B parameters. Reinforcement learning from human feedback was not used to finetune the base model further. [22]

Phi-2 Model Specifications

Model Architecture: This is a Transformer-based model. The main objective was to predict next token.

Context Length: Maximum context length is 2048 tokens.

Dataset Size: The dataset consist of 250B tokens.It includes NLP synthetic data created by AOAI GPT-3.5 and filtered web data from Falcon RefinedWeb and SlimPajama, which was assessed by AOAI GPT-4.

Training Tokens: Number of training tokens were 1.4T tokens

GPUs Used: 96x A100-80G

Training Time: This base pre-trained model was trained for 14 days

The overall performance of the model was one of the best among all the small scale LLM models available at that time. But there are few limitations and problems associated with the model. The context length of the model is 2048. This size was sufficient for our small size code dataset, but not that helpful for large size code dataset. There is a chance of hallucination or unexpected behavior when we increase its context length beyond that size.

Phi-3.5-mini-instruct

Phi-3.5-mini-instruct is also a small-scale LLM but quite efficient and accurate on standard tasks. The model released in August, 2024. This model was pre-trained with 3.82 B parameters, little higher then Phi-2. The model was fine-tuned with supervised finetuning along with policy optimization. It was trained with dense reasoning dataset to improve its reasoning. The reasoning part especially focused on code generation. [23]

Phi-3.5-mini-instruct Model Summary

Model Architecture: It used a dense decoder-only Transformer model and used the same tokenization as Phi-3 Mini.

Inputs: Text. It is best suited for prompts using chat format.

Context Length: The maximum context length we can use for thus model is 128K tokens

GPUs Used: 512 H100-80G

Training Time: This base pre-trained model was trained for 10 days

Training Data: It was trained on 3.4T tokens

Release Date: August 2024

This model provide a longer context length without any significant increment in size of the model. This is quite helpful for tasks like finetuning as we need to train the parameters of the model in this case. There are drawbacks in this type models. The level of accuracy decreases and the output quality decreases. So this tread of is quite significant in pre-trained models.

Phi-4

Phi-4 is the most recent among all the Phi series models described here. The size of Phi-4 is larger then the above two, but it has shown significant improvement in its performance. The model released in December, 2024. This model was pre-trained with 14.7 B parameters, quite high then above two models. Supervised fine-tuning technique and direct preference optimization technique was used while training the model.. [24]

Phi-4 Model Summary

Architecture: It used a dense decoder-only Transformer model.

Inputs: Text, best suited for prompts in the chat format

Context Length: The maximum context length we can use for thus model is 16K tokens

GPUs Used: 1920 H100-80G

Training Time: This base pre-trained model was trained for 21 days

Training Data: The training data consist of 9.8T tokens

Release Date: December 12, 2024

This models takes 16 k context length which is lower then the above model but it is sufficient for our work. Also the performance of the model is significantly good then the above models. The tread of we were taking above managed beautifully here.

4.2.2 Llama

Llama is another open source model released by Meta. These models also shown promising results on standard benchmarks. They are openly used in research and finetuning for their open availability.

Llama-3.1-8B-Instruct

Llama-3.1-8B-Instruct is released by Meta in 2024. The model was also available in 70 B and 405 B but 8 B is suitable for local uses. They have demonstrated strong performance, outperforming many available open-source and closed chat models on common industry benchmarks. The details of the model are given below. [25]

Llama 3.1-8b-instruct Model Summary

Architecture: It used a dense decoder-only Transformer model.
Training Data: A new mix of publicly available online data.
Params: 8B
Input Modalities: Multilingual Text
Output Modalities: Multilingual Text and Code
Context Length: The maximum context length we can use for thus model is 128K
Token Count: This base pre-trained model was trained for 15T+
Knowledge Cutoff: December 2023

Llama-3.2-3b

This is an updated version of Llama-3.1 model. It provides large context length with very small overall size. Based on the situations model may perform not as expected because of the size. This model includes muultimodal characteristic in the model. The details of the model are given below. [26]

Llama-3.2-3b Model Summary

Architecture: It used a dense decoder-only Transformer model.
Training Data: A new mix of publicly available online data.
Params: 3B (3.21B)
Input Modalities: Multilingual Text
Output Modalities: Multilingual Text and Code
Context Length: The maximum context length we can use for thus model is 128K
Token Count: This base pre-trained model was trained for up to 9T tokens
Knowledge Cutoff: December 2023

The detailed of the results we got using these models are described in Chapter 6 and Chapter 7.

Chapter 5

Data Set and Validation Techniques

Data set that we are using plays a crucial role in the finetuning work. The size of the each data point, the complexity of the output that we want to generate, the total number of available data point for training, everything plays a significant role. In this chapter the description of our dataset is given along with dataset preparation techniques. Also the validation method we used to judge our trained models are also described here.

5.1 Data sets

The work is focused to automate the test code generation by finetuning large language models. We used two datasets which had test cases of two different domains. One of them had test cases with small size test code and the number of data points were also high. The other dataset consist of complex and very large size code. Also the number of data points in this dataset was lower then the previous one. We called the first dataset as 'Small code dataset' and the second dataset as 'Large code dataset' throughout the thesis.

5.1.1 Small code dataset

This dataset consist of test cases where the test code are simple and small in size. In terms of line it had codes from around 10 lines to around 25 lines. These are test code to check

the quality of the product in different situations.

In this case we provide the LLM endpoint, test case description, input and output with appropriate prompts and the LLM generated test code according to that. The description describe the test case in one or two line. Here in the time of training we provide the endpoint, description, input, output and the actual code as input and while inference we provide the endpoint, description, input and output and asked it to generate the complete code. The details of the prompts and hyperparameters we used and the results we got for this dataset is described in Chapter 6.

The total number of training data points we used for this case was 943. 26 data points were used for validation purpose.

The size of the input context for these test code are described below in terms of tokens in Table 5.1. Also their distribution is shown in Figure 5.1 and 5.2.

Token Range	**Training Count**	**Validation Count**
50-100	0	0
100-150	38	0
150-200	161	1
200-250	159	6
250-300	235	2
300-350	111	3
350-400	67	3
400-450	51	4
450-500	33	2
500-550	33	1
550-600	9	1
600-650	7	3
650+	39	0

Table 5.1: Training and Validation Dataset Token Range Counts

When we are working on finetuning a LLM with one dataset, the similarity of the data points i.e. test code in our case is very important. If we work on some dataset where similarity is very less among them, the finetuning work would not happen properly. The similarity plots of the test code for this dataset is shown in Figure 5.3. We can confirm form the plot there there were enough similarity among the data points in the dateset.

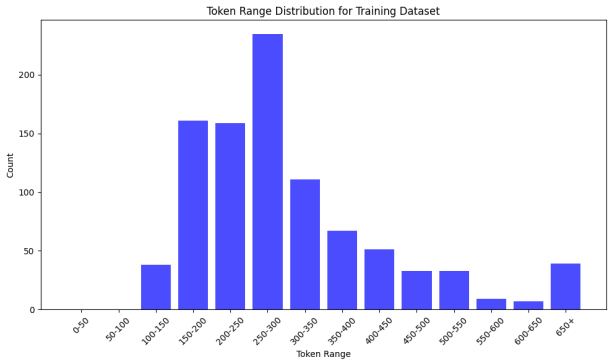


Figure 5.1: Training Dataset Token Range Counts

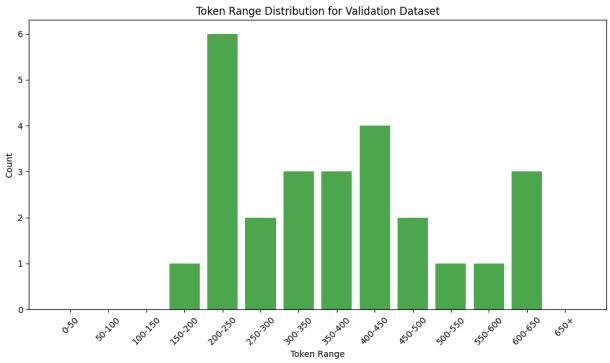


Figure 5.2: Validation Dataset Token Range Counts

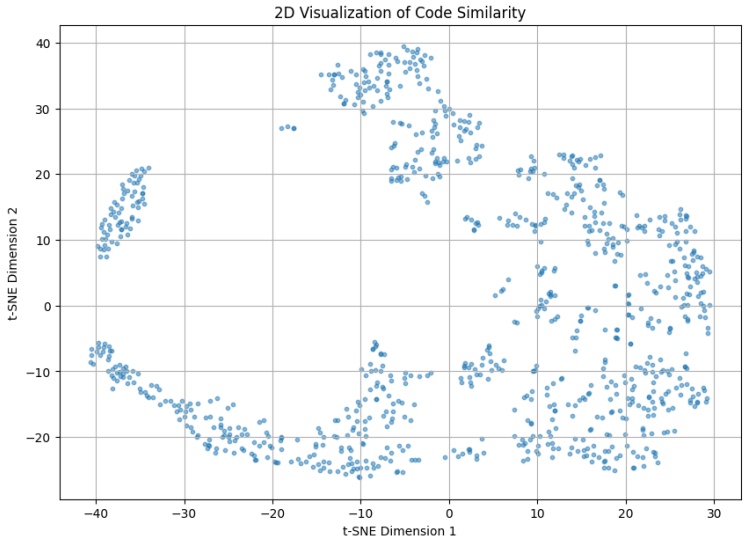


Figure 5.3: Similarity between test code

5.1.2 Large code dataset

This dataset consist of test cases where the test code are quite long. In terms of line it had codes from around 100 lines to around 450 lines. These were also test code to check the quality of the product in different situations.

In this case we provided the LLM the test case summary and test steps with appropriate prompts and the LLM generated test code according to that. The test case summary is usually one to two line description of the test code we want to generate and the test steps describes what are the steps we need to follow to write that given test codes In the time

of training we provided the test case summary, test steps and the actual code as an input and while inference we provide the finetuned LLM the test case summary and test steps and asked it to generate the complete code. The details of the prompts and hyperparameters we used and the results we got for this dataset is described in Chapter 7.

The total number of training data points used for this case was 421. 10 data points were used for validation purpose.

The size of the input context for these test code are described below in term of tokens in Table 5.2. Also their distribution is shown in Figure 5.4 and 5.5.

Token Range	Training Count	Validation Count
0-500	1	0
500-1000	27	0
1000-1500	63	1
1500-2000	74	5
2000-2500	91	0
2500-3000	42	0
3000-3500	30	0
3500-4000	23	1
4000-4500	17	0
4500-5000	9	0
5000-5500	12	2
5500-6000	15	1
6000-6500	17	0
6500+	0	0

Table 5.2: Training and Validation Dataset Token Range Counts

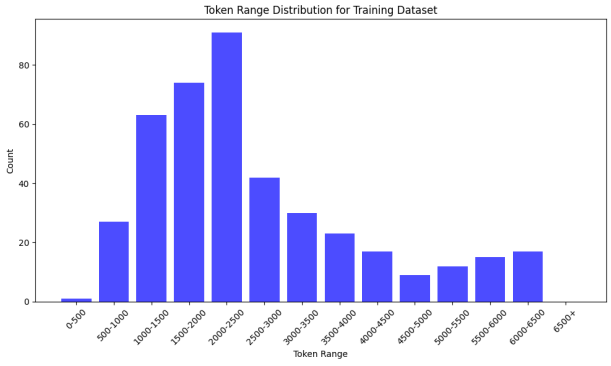


Figure 5.4: Training Dataset Token Range Counts

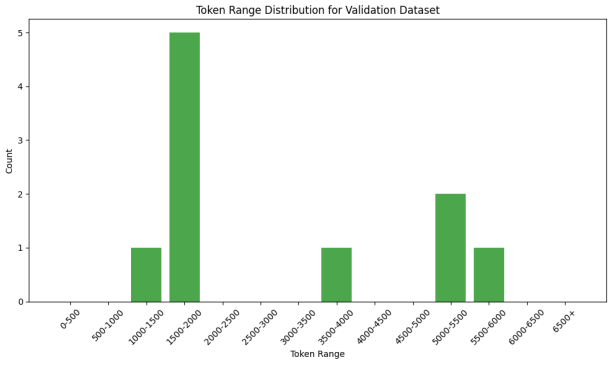


Figure 5.5: Validation Dataset Token Range Counts

The similarity plots of the test code for this dataset is shown in Figure 5.6.

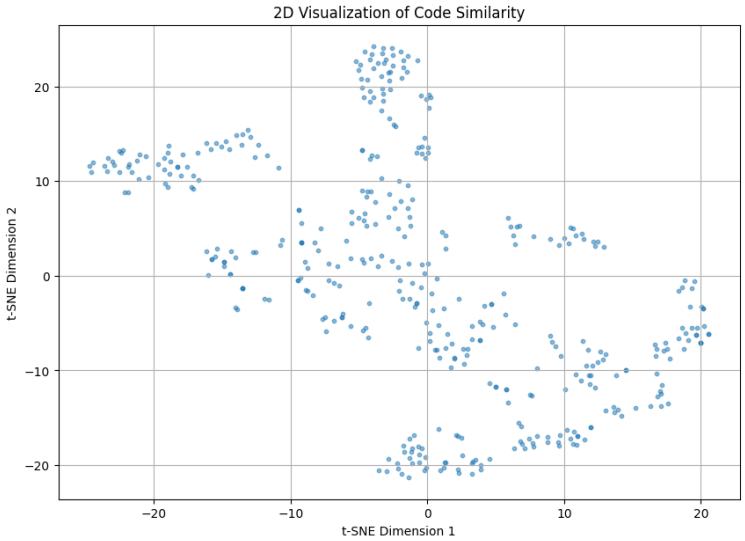


Figure 5.6: Similarity between test code

In both the cases the test code descriptions were stored in json format. In certain cases, data points were not in a readable format. They were cleaned and transformed into a human-readable format using LLMs and carefully designed prompts. We went through each data point one by one before start the training process.

5.2 Validation Techniques

Validating the performance of the finetuned model is one of the important aspects. As we discussed above our dataset contains very specific type of test code. That is why most of the time of our validation is done by human verification. The loss values helped as a primary guide about the state of the model. All these techniques are given here.

5.2.1 Loss Values

Loss values are numerical numbers which shows how much the generated data (in our case test code) is different from the actual data. These values helps as a primary guide about the finetuned model. As the number of labels are more then one, both Phi and Llama models

by default uses categorical cross-entropy loss. The equation for categorical cross-entropy function is given below:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{j=1}^{V} y_{ij}\log(\hat{y}_{ij})$$

where:

- N is the number of tokens in the sequence.
- V is the vocabulary size.
- y_{ij} is a one-hot encoded target, where $y_{ij} = 1$ if the i-th token corresponds to the j-th word in the vocabulary, otherwise 0.
- $\hat{y}_{ij}$ is the predicted probability of the i-th token being the j-th word, obtained from the softmax function.

Using the training and validation loss values we got the idea about how far the model is trained. Also it indicates about overfitting if the training loss goes significant lower but validation loss still stays in some high value. In our work we played with different hyper-parameters and found the set of hyperparameters which brings the loss values significant lower. We used those models which are trained on those selected set of hyperparameters. We generated results using those models.

5.2.2 Human Verification

After generating the results from the finetuned model for our validation dataset, we did human verification. First we checked for few different "temperature" values and find out the best temperature. It came out that the results are good when temperature is 0.1 or 0.2 in almost all the cases. We send the test code to the developers who are involved in writing them. They gave score to the generate code in the scale of 0 to 10 along with the issues those were present in that generated code. Over time, we also gained insights into the test code and evaluated them in the later stages of the project in that same way. The following scoring guide we used for scoring and it is given in Table 5.3.

Score	Descriptions
1-3	Very poor results. The code has major issues or hallucinations and requires significant modifications to run the test case.
4-5	Unusable code. It is somewhat related to the input steps but misses key objectives. Debugging would take more time than rewriting.
6-7	Adequate results. The code is usable but needs adjustments, such as fixing incomplete functions or incorrect function calls.
8	Good results. The code achieves the objective, with only minor issues (e.g., logging errors, variable name mismatches).
9-10	Very good results. The code aligns well with the steps, achieves the objective, and can be executed without issues.

Table 5.3: Scoring Guide for Code Quality

There are some limitation in validation using these techniques. The technique involves some level of subjectivity as a significant amount of human judgment involves here. The above table helps in reducing the subjectivity and make it standardize. While this method allows us to estimate a range of accuracy, it does not provide the exact accuracy. But the technique is absolutely reliable since it is conducted through human effort and without depending on any metric or automation technique.

The results of these techniques are described in detail in Chapter 6 and Chapter 7.

Chapter 6

Finetuning result with small code dataset

In Chapter 4, the set up and the models are described which we used. Also in Chapter 5, the small code dataset is described. Here the results are described which we got using those models and dataset. The prompt used, the hyperparameters set, the obtained loss values, and the achieved results are all explained in detail here. Training process were done using QLoRA techniques for all the cases.

6.1 Finetuned Phi-2 Results

The details of the works we have done using Phi-2 LLM model and small size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and at the end the results are described.

6.1.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and the selected ones are put in Table 6.1.

Prompt Name	Prompt
Prompt-1	###System Write down a python function that tests the API scenario described below. Make sure you use appropriate docstring while writing code. Below are the endpoint, inputs, and output of the function as well. ###Endpoint: {endpoint} ###Description: {question} ###Inputs: {inputs} ###Output: {output} ###Answer: {code}
Prompt-2	###System Write down a python function that tests the API scenario described below. Make sure you use appropriate docstring while writing code. Below are the endpoint, inputs, and output of the function as well. ###Endpoint: {endpoint} ###Description: {question} {desc} ###Inputs: {inputs} ###Output: {output} ###Answer: {code}

Table 6.1: Prompt Phi-2

6.1.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got are described in Table 6.2.

Table 6.2: Training Summary

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-05 Training batch size: 4 Evaluation batch size: 4 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 400 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	16	Final Training Loss: 0.2218 Final Validation Loss: 0.2694 Result_phi2_1	Losses did not go significantly lower.	Prompt-1
Learning rate: 5e-04 Training batch size: 5 Evaluation batch size: 5 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 1000 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	53	Final Training Loss: 0.0355 Final Validation Loss: 0.08976 Result_phi2_2	Loss values went lower but we tried few more set of hyperparameters before starting the result generation.	Prompt-1

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 5 Evaluation batch size: 5 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 1000 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	53	Final Training Loss: 0.0283 Final Validation Loss: 0.08941 Result_phi2_3	Results are good. Detailed results described in next subsection.	Prompt-2
Learning rate: 5e-04 Training batch size: 9 Evaluation batch size: 9 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 600 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	53	Final Training Loss: 0.0239 Final Validation Loss: 0.093 Result_phi2_4	Results are good. Detailed results described in next subsection.	Prompt-2

6.1.3 Generated results and analysis

We generated results using 3rd and 4th finetuned model i.e. Result_phi2_3 and Result_phi2_4 as both of them got significant lower training and validation loss. The first one is trained with 20 GB GPU and the second one is trained on 40 GB GPU. We used higher batch size to check if there is any significant improvement.

The result we got in both the cases were quite good. The results described below were generated with temperature 0.1 in both the cases. We scored each one of the validation data point in the scale of 0 to 10 as described in Chapter 5 and get the average over all of them. Both the cases we got score in between 8 and 8.5 on an average. We can conclude that the models performance was 80 to 85% correct. Also we can conclude that increasing the batch size from 5 to 9 does not give any significant improvement in results. The scores are described in Table 6.3 and Table 6.4.

Test Case	**Score**	**Test Case**	**Score**	**Test Case**	**Score**
Validation data 1	09	Validation data 2	10	Validation data 3	07
Validation data 4	10	Validation data 5	08	Validation data 6	08
Validation data 7	08	Validation data 8	08	Validation data 9	07
Validation data 10	06	Validation data 11	07	Validation data 12	10
Validation data 13	09	Validation data 14	10	Validation data 15	07
Validation data 16	07	Validation data 17	07	Validation data 18	08
Validation data 19	08	Validation data 20	08	Validation data 21	09
Validation data 22	09	Validation data 23	07	Validation data 24	07
Validation data 25	07	Validation data 26	08		
Average	8.01				

Table 6.3: Test Cases and Scores for Result_phi2_3

Test Case	Score	Test Case	Score	Test Case	Score
Validation data 1	09	Validation data 2	10	Validation data 3	08
Validation data 4	01	Validation data 5	08	Validation data 6	08
Validation data 7	09	Validation data 8	09	Validation data 9	06
Validation data 10	06	Validation data 11	08	Validation data 12	10
Validation data 13	10	Validation data 14	10	Validation data 15	08
Validation data 16	08	Validation data 17	08	Validation data 18	10
Validation data 19	09	Validation data 20	10	Validation data 21	08
Validation data 22	08	Validation data 23	06	Validation data 24	07
Validation data 25	04	Validation data 26	08		
Average	8.26				

Table 6.4: Test Cases and Scores for Result_phi2_4

In Table 6.1.3 few examples are shown. The actual code, code generated by base model and finetuned model for the same inputs is given here. The examples are shown to give some idea about our dataset and outputs.

Table 6.5: Comparison of Validation data and Generated Outputs (Phi-2)

Test Case	Original Code	Base Model Generated Code	Fine-Tuned Model Generated Code

Validation data 2	`def test_version_config_with_invalid_token(invalid_exec_api):`
	`  """ Fetching the information of Version and Build Number with invalid token """`
	`  r = invalid_exec_api.config_version()`
	`  result = r.json()`
	`  test_assert.status(r, 401)`
	`  assert result['detail'] == "Invalid token."`
		`def test_invalid_token(self):`
		`  """ Fetching the information when invalid token is provided`
		
		`  Fetching the information of Version and Build Number with invalid token`
		
		`  ### Inputs:`
		`  nan`
		
		`  ### Output:`
		`  { "status":401, "message":"invalid token" }`
		`  """`
		`  response = requests.get(self.endpoint)`
		`  self.assertEqual(response.status_code, 401)`
		`  self.assertEqual(response.json(), {"status":401, "message":"invalid token"})`
		
		`if __name__ == '__main__':`
		`  unittest.main()`
			`def test_config_version_with_invalid_token(invalid_exec_api):`
			`  """ Fetching the information of Version and Build Number with invalid token """`
			`  r = invalid_exec_api.config_version()`
			`  result = r.json()`
			`  test_assert.status(r, 401)`
			`  assert result['detail'] == 'Invalid token.'`

Validation data 18	def test_library_hvmtypes(run_api):
	  """
	  Getting the list of Hypervisor type
	  """
	  r = run_api.library_hvmtypes()
	  result = r.json()
	  test_assert.status(result, LIBRARY_HVM_TYPE, "library_hvmtypes")
	  test_assert.status(r, 200)
		def test_get_hypervisor_type():
		  """
		  This function tests the get_hypervisor_type API
		  """
		  # Test the API
		  response = requests.get("https://api.hvm.org/library/rest/hvmtypes/")
		  assert response.status_code == 200
		  assert isinstance(response.json(), list)
		  assert len(response.json()) > 0
		  assert isinstance(response.json()[0], dict)
		  assert "type" in response.json()[0]
		  assert "name" in response.json()[0]
		  assert "version" in response.json()[0]
		  assert "description" in response.json()[0]
		  assert "status" in response.json()[0]
		  assert "response" in response.json()[0]
		  assert
			def test_library_hvmtypes(run_api):
			  """
			  Getting the list of Hypervisor type
			  """
			  r = run_api.library_hvmtypes()
			  rjson = r.json()
			  test_assert.status(r, 200)
			  assert rjson['count'] == 1, "The error is %s" % rjson
			  assert rjson['name'] == "dtype", "The error is %s" % rjson

6.2 Finetuned Llama-3.1-8b-Instruct Results

The results we got using Llama-3.1_8b_instruct model is described in this section. As the model was showing promising results on standard dataset, we planned to use it in our case also.

6.2.1 Prompts

The idea about the prompt was clear from previous experiments. The idea we got about the system massage and the inputs we need to provide were clear. In this case we used a similar type of prompt along with appropriate tokens. The prompt is given in Table 6.6.

Prompt Name	Prompt
Prompt-1	<|begin_of_text|><|start_header_id|>system<|end_header_id|> Write down a python function that tests the API scenario described below. Make sure you use appropriate docstring while writing code. Below are the endpoint, inputs, and output of the function as well.<|eot_id|> <|start_header_id|>user<|end_header_id|> Endpoint: {endpoint} Description: {question} {desc} Inputs: {inputs} Output: {output} Answer: {code} <|eot_id|> <|start_header_id|>assistant<|end_header_id|>

Table 6.6: Prompt Llama-3.1_8b_instruct

6.2.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got for selected cases are described in Table 6.7. Also a short summary of the result for each cases are described.

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 3 Evaluation batch size: 3 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit lr_scheduler_type: linear Warmup steps: 100 Training Steps: 210 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	3	Final Training Loss: 0.03 Final Validation Loss: 0.082455 Result_llama 3.1_1	Results are not as good as the Phi-2 results. Detailed results described in next subsection.	Prompt-1
Learning rate: 5e-04 Training batch size: 8 Evaluation batch size: 8 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit lr_scheduler_type: linear Warmup steps: 100 Training Steps: 170 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	7	Final Training Loss: 0.041 Final Validation Loss: 0.09 Result_llama 3.1_2	The above Llama 3.1 result is better then this.	Prompt-1

Table 6.7: Training Summary Llama-3.1_8b_instruct

6.2.3 Generated results and analysis

We generated result using temperature 0.1, 0.2 and few higher temperatures. The code was coming better for temperature 0.1 compared to other cases. We generated results for these cases for temperature = 0.1. The best case results we got is summarized Table 6.8 and Table 6.9. Each of them are scored manually based on the guideline discussed in the previous chapter.

The outputs on an average 70 to 75 percent accurate in best case scenario.We observe that increasing the batch size does not showed any significant improvement in this case also. Though the pre-trained model was larger then Phi-2 with higher context length, Phi-2 outperform this model in this task.

Test Case	**Score**	**Test Case**	**Score**	**Test Case**	**Score**
Validation data 1	09	Validation data 2	10	Validation data 3	05
Validation data 4	06	Validation data 5	05	Validation data 6	05
Validation data 7	05	Validation data 8	04	Validation data 9	08
Validation data 10	09	Validation data 11	09	Validation data 12	08
Validation data 13	07	Validation data 14	07	Validation data 15	08
Validation data 16	08	Validation data 17	06	Validation data 18	07
Validation data 19	09	Validation data 20	10	Validation data 21	06
Validation data 22	10	Validation data 23	05	Validation data 24	09
Validation data 25	07	Validation data 26	07		
Average	7.3				

Table 6.8: Test Cases and Scores for Result_llama 3.1_1

Test Case	Score	Test Case	Score	Test Case	Score
Validation data 1	09	Validation data 2	10	Validation data 3	04
Validation data 4	05	Validation data 5	03	Validation data 6	05
Validation data 7	-	Validation data 8	-	Validation data 9	07
Validation data 10	-	Validation data 11	08	Validation data 12	10
Validation data 13	07	Validation data 14	07	Validation data 15	07
Validation data 16	05	Validation data 17	03	Validation data 18	05
Validation data 19	10	Validation data 20	10	Validation data 21	07
Validation data 22	09	Validation data 23	04	Validation data 24	09
Validation data 25	09	Validation data 26	08		
Average	7.3				

Table 6.9: Test Cases and Scores for Result_llama 3.1_2

6.3 Finetuned Phi-4 Results

The results we got using Phi-4 is described in this section. As the model was showing promising results on standard dataset, we planned to use it in our case also. The model released in December, 2024. So the work is done after significant work was completed on small as well as large code dataset.

6.3.1 Prompts

The idea about the prompt was already clear from previous experiments. In this case we used a similar type of prompt along with appropriate eos tokens. The prompt is given in Table 6.10.

Prompt Name	Prompt
Prompt-1	<|im_start|>system<|im_sep|> Write down a python function that tests the API scenario described below. Make sure you use appropriate docstring while writing code. Below are the endpoint, inputs, and output of the function as well.<|im_end|> <|im_start|>user<|im_sep|> Endpoint: {endpoint} Description: {question} Inputs: {inputs} Output: {output} Answer: {code} <<|im_end|> <|im_start|>assistant<|im_sep|>

Table 6.10: Prompt Phi-4

6.3.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got for selected cases are described in Table 6.12. Also a short summary of the result for each cases are described. The work is also done using 4-bit quantization and QLoRA. The first and third one used more then 20 GB GPU RAM.

Table 6.11: Training Summary Phi-4

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 7 Evaluation batch size: 7 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit lr_scheduler_type: linear Warmup steps: 100 Training Steps: 170 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	13	Final Training Loss: 0.018 Final Validation Loss: 0.09 Result_phi4_1	We expected to get better result here but we got only around 75 % accurate result on an average.	Prompt-1
Learning rate: 5e-04 Training batch size: 2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit lr_scheduler_type: linear Warmup steps: 100 Training Steps: 310 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	3	Final Training Loss: 0.03 Final Validation Loss: 0.08 Result_phi4_2	This gives the best result among all the experiments we did. We got around 85 % accuracy in this case.	Prompt-1

Table 6.12: Training Summary Phi-4

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 3 Evaluation batch size: 3 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit lr_scheduler_type: linear Warmup steps: 100 Training Steps: 230 Rank of LoRA matrix = 256 Value of LoRA alpha = 512 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=1024	4	Final Training Loss: 0.03 Final Validation Loss: 0.09 Result_phi4_3	Here also we tried to train with a situation somewhere in the middle of the above two with higher Lora rank but the result was till around 75 % accurate on an average.	Prompt-1

6.3.3 Generated results and analysis

We generated results for these cases for temperature = 0.1. The best case results we got is summarized Table 6.13, Table 6.14 and Table 6.15.

Here we got around 85 % accuracy in the best case scenario. The results we got using more then 20 GB GPU RAM and more resources like higher batch size and higher lora rank were expected to be good. But as you saw in the above tables the results are not as expected.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data 1	10	Validation data 2	10	Validation data 3	04
Validation data 4	04	Validation data 5	06	Validation data 6	05
Validation data 7	04	Validation data 8	05	Validation data 9	10
Validation data 10	07	Validation data 11	09	Validation data 12	10
Validation data 13	10	Validation data 14	04	Validation data 15	07
Validation data 16	10	Validation data 17	07	Validation data 18	05
Validation data 19	09	Validation data 20	10	Validation data 21	07
Validation data 22	09	Validation data 23	07	Validation data 24	08
Validation data 25	08	Validation data 26	08		
Average	7.4				

Table 6.13: Test Cases and Scores for Result_phi-4_1

Test Case	Score	Test Case	Score	Test Case	Score
Validation data 1	10	Validation data 2	10	Validation data 3	09
Validation data 4	10	Validation data 5	06	Validation data 6	09
Validation data 7	09	Validation data 8	10	Validation data 9	09
Validation data 10	08	Validation data 11	09	Validation data 12	10
Validation data 13	05	Validation data 14	06	Validation data 15	08
Validation data 16	08	Validation data 17	07	Validation data 18	10
Validation data 19	09	Validation data 20	10	Validation data 21	07
Validation data 22	09	Validation data 23	07	Validation data 24	08
Validation data 25	10	Validation data 26	08		
Average	8.5				

Table 6.14: Test Cases and Scores for Result_phi-4_2

Test Case	Score	Test Case	Score	Test Case	Score
Validation data 1	10	Validation data 2	10	Validation data 3	03
Validation data 4	04	Validation data 5	06	Validation data 6	05
Validation data 7	-	Validation data 8	06	Validation data 9	09
Validation data 10	08	Validation data 11	05	Validation data 12	09
Validation data 13	09	Validation data 14	04	Validation data 15	08
Validation data 16	08	Validation data 17	08	Validation data 18	05
Validation data 19	09	Validation data 20	10	Validation data 21	07
Validation data 22	09	Validation data 23	07	Validation data 24	07
Validation data 25	09	Validation data 26	07		
Average	7.3				

Table 6.15: Test Cases and Scores for Result_phi-4_3

6.4 Overall results

In this chapter the results of small code dataset is summarized. The results are descent for almost all the cases. For few cases it achieved around 80 % to 85 % accuracy. Based on the performance of Phi-4 throughout our work we expected it to give better results then what we achieved in certain cases. Overall the results are satisfying for this dataset.

Chapter 7

Finetuning results with large code dataset

In Chapter 4, the set up and the models are described which we used. Also in Chapter 5, the large code dataset is described. Here in this chapter the results are described which we achieved using those models and dataset in the same way as the small code dataset results are described in Chapter 6. The prompt we used, the hyperparameters we set, the loss values obtained, and the results achieved everything is described here. All the training process are done with 4-bit quantization with QLoRA techniques

7.1 Finetuned Phi-2 Results

The details of the works we have done using Phi-2 LLM model and large size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and at the end the results are described. As the size of the model was small we were able to use larger batch size for sufficient input context length. Also this model can be processed within 20 GB GPU limit.

7.1.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and the selected few are put in Table7.1.

Prompt Name	Prompt
Prompt-1	###System Generate a complete Python code based on the following test case summary and test steps. ###Test Case: <summary of test_case> ###Test Steps: <test_steps> ###Python Code: <code>
Prompt-2	###System Generate a complete Python code based on the following test case summary and test steps. ### Name: <test_case_name> ###Test Case: <summary of test_case> ###Test Steps: <test_steps> ###Python Code: <code>

Table 7.1: Prompt Phi-2

7.1.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got are described in Table 7.1.2. The results are summarized in each of the set of hyperparameters. If the results looked satisfactory, detailed results are given in next section. Only selected set of hyperparameters and results are described here to give the reader the idea of how we reach the best set of hyperparameters for this model and dataset combination.

Table 7.2: Training Summary Phi-2

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-05 Training batch size: 6 Evaluation batch size: 6 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: paged_adamw_8bit Warmup steps: 100 Training Steps: 600 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=4096	47	Final Training Loss: 0.5463 Final Validation Loss: 0.5291 Result_phi2_1	Losses did not go significantly lower and saturated at this label. The results we generated are not good. Many libraries it called which were not present anywhere in the dataset. Some cases in small scale and some cases in large scale hallucination happened.	Prompt-1
Learning rate: 5e-03 Training batch size: 6 Evaluation batch size: 6 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: paged_adamw_8bit Warmup steps: 100 Training Steps: 100 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=4096	8	Final Training Loss: 0.91 Final Validation Loss: 0.7358 Result_phi2_2	Loss values are very high. Also the are scattering too much. Most probably that happened because of small learning rate. Overall the training process was not stable.	Prompt-1

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 6 Evaluation batch size: 6 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: paged_adamw_8bit Warmup steps: 100 Training Steps: 600 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=4096	47	Final Training Loss: 0.222 Final Validation Loss: 0.319 Result_phi2_3	The loss values are lower then the above two cases. The generated results were not hallucinating in large scale. But the results were not good. Many libraries it called which were not present anywhere in the dataset. Overall the accuracy was not more then 30 %.	Prompt-1
Learning rate: 5e-04 Training batch size: 3 Evaluation batch size: 3 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: paged_adamw_8bit Warmup steps: 100 Training Steps: 800 Rank of LoRA matrix = 36 Value of LoRA alpha = 72 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=8192	27	Final Training Loss: 0.089 Final Validation Loss: 0.187 Result_phi2_4	Results are better then the above ones. Losses also went significant low. Around 50 % accuracy is achieved with this set of hyperparameters. For this case detailed results are described in next section.	Prompt-2

7.1.3 Generated results and analysis

Result_phi2_4 model result was little better then the other cases. As described in Table 7.1.2, it achieved around 50 % accuracy. The score of each test cases given in Table 7.3. Description is given below the table.

Test Case	Score	Test Case	Score	Test Case	Score
Validation case no 1	06	Validation case no 2	04	Validation case no 3	05
Validation case no 4	04	Validation case no 5	04	Validation case no 6	04
Validation case no 7	02	Validation case no 8	06	Validation case no 9	06
Validation case no 10	07				
Average	4.8				

Table 7.3: Test Cases and Scores for Result_phi2_4

7.2 Finetuned Llama-3.1 Results

The details of the works we have done using Llama-3.1-8b-instruct LLM model and large size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and the results are described. The format is same as before. The model was large and we had to use 40 GB GPU system. We tried to use 20 GB GPU but we were unable to pass at least one batch size but that was not possible. We tried to use the lora technique in half of the layers, in this case we were able to pass one batch size but the results were too bad. Even using one batch size we were able to process 1 to 2 batch size only. We had to compromise with input context length to pass two batch size.

7.2.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and the

selected few are put in Table7.4.

Prompt Name	Prompt
Prompt-1	<|begin_of_text|><|start_header_id|>system<|end_header_id|> <steps> <|eot_id|> <|start_header_id|>user<|end_header_id|> Write a python code based on the steps given above.<|eot_id|> Python Code: <code> <|eot_id|> <|start_header_id|>assistant<|end_header_id|>
Prompt-2	<|begin_of_text|><|start_header_id|>system<|end_header_id|> <steps> <|eot_id|> <|start_header_id|>user<|end_header_id|> Write a python code based on the steps given above.<|eot_id|> Python Code: <code> <|eot_id|> <|start_header_id|>assistant<|end_header_id|>

Table 7.4: Prompt for Llama-3.1

7.2.2 Hyperparameters and loss values

The set of hyperparameters used are described here. Selected set of hyperparameters are given to show the trajectory through which we achieve lower loss values. But none of the case provide any satisfactory results. The problem of hallucination was there in almost all the cases. Other then hallucination, calling wrong libraries was a big problem. None of the results were described in detailed as none of them provide satisfactory results in validation dataset. Summary of the results are added in this section.

Table 7.5: Training Summary Llama-3.1

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-05 Training batch size: 2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: paged_adamw_8bit Warmup steps: 50 Training Steps: 200 Rank of LoRA matrix = 8 Value of LoRA alpha = 16 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', Weight Decay: 0.01 input_size=4096	25	Final Training Loss: 0.62 Final Validation Loss: 0.72 Result_Llama 3.1_1	Losses did not go significantly lower and saturated at this label. This is expected as this was the first try. Multiple attempts made by modifying the hyperparameters.	Prompt-1
Learning rate: 5e-04 Training batch size: 2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 20 Optimizer: paged_adamw_8bit Training Steps: 100 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', Weight Decay: 0.01 input_size=4096	7	Final Training Loss: 0.25 Final Validation Loss: 0.297 Result_Llama 3.1_2	Loss values are low compared to the previous cases. Results generated using the finetuned model but multiple problems were there. The code hallucinated many cases, where it is nor hallucinated it creates and called functions which were present nowhere in the dataset.	Prompt-1

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 50 Optimizer: paged_adamw_8bit Training Steps: 150 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=4096	10	Final Training Loss: 0.15 Final Validation Loss: 0.26 Result_phi2_3	Loss went below the previous cases. But the problems described in the previous cases still exist and results are not good enough to use.	Prompt-1
Learning rate: 5e-04 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: paged_adamw_8bit Warmup steps: 100 Training Steps:280 Rank of LoRA matrix = 72 Value of LoRA alpha = 144 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=6500	3	Final Training Loss: 0.035 Final Validation Loss: 0.18 Result_Llama 3.1_4	The loss values went down significantly compared to other cases. But there is no corresponding improvement in the quality of results. We experimented with wide range of temperature, repeat penalty and other hyperparameters but the hallucination and other problems still persist.	Prompt-2

7.3 Finetuned Phi-3.5 Results

The details of the works we have done using Phi-3.5_mini_instruct LLM model and large size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and at the results are described.

7.3.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and got the idea about them in the previous cases and we wrote a similar type prompt with appropriate eos tokens for this model and it is given in Table7.12.

Prompt Name	Prompt
Prompt-1	<|system|> Generate a complete Python code based on the following test case summary and test steps.<|end|> <|user|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|end|> <|assistant|>

Table 7.6: Prompt Phi-3.5

7.3.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got are described in Table 7.7. The results were not satisfactory in this case also and summary of the results are added here.

Table 7.7: Training Summary Phi-3.5

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size:1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 50 Training Steps: 200 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5000	2	Final Training Loss: 0.2481 Final Validation Loss: 0.2898 Result_phi-3.5_1	Losses did not go significantly lower and saturated at this label. We tried with different set of hyperparameters to make it low at least the level we got in previous cases.	Prompt-1
Learning rate: 5e-05 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 400 Rank of LoRA matrix = 36 Value of LoRA alpha = 72 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5500	7	Final Training Loss: 0.284 Final Validation Loss: 0.344 Result_phi-3.5_2	Loss values increased compared to the previous case. We tried with some more changes whatever possible with the limited resources. (Resources described in Chapter 4	Prompt-1

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-06 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 1200 Rank of LoRA matrix = 36 Value of LoRA alpha = 72 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5700	14	Final Training Loss: 0.478 Final Validation Loss: 0.517 Result_phi-3.5_3	The loss values went even higher then the previous cases. The results we generated and it was hallucinating.	Prompt-1
Learning rate: 5e-04 Training batch size: 2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 260 Rank of LoRA matrix = 72 Value of LoRA alpha = 144 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=8192	3	Final Training Loss: 0.136 Final Validation Loss: 0.258 Result_phi-3.5_4	The loss values are lower then the previous cases though it is not significant low. We can not achieve better loss values compared to this case for this model. We generated and reviewed results with this. In this case also it is hallucinating and the results were not at all close to actual result.	Prompt-1

7.4 Finetuned Llama-3.2 Results

The details of the works we have done using Meta Llama-3.2_3b LLM model and large size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and the results are described. The model was chosen as it provides larger context window with a very small model size. So we got more independence in changing the hyperparameters.

7.4.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and got the idea about them in the previous cases and we wrote a similar type prompt with appropriate eos tokens for this model and it is given in Table7.8.

Prompt Name	Prompt
Prompt-1	<|begin_of_text|><|start_header_id|>system<|end_header_id|> Generate a complete Python code based on the following test case summary and test steps. Use chain of thought while generating the output.<|eot_id|> <|start_header_id|>user<|end_header_id|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|eot_id|> <|start_header_id|>assistant<|end_header_id|>

Table 7.8: Prompt Llama-3.2

7.4.2 Hyperparameters and loss values

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size:2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 260 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=6500	6	Final Training Loss: 0.04 Final Validation Loss: 0.18 Result_Llama-3.2_1	Results were around 50 to 55 % correct. Detailed result is given in the next section for this case.	Prompt-1
Learning rate: 5e-04 Training batch size:2 Evaluation batch size: 2 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 180 Rank of LoRA matrix = 72 Value of LoRA alpha = 144 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=6500	4	Final Training Loss: 0.06 Final Validation Loss: 0.19 Result_Llama-3.2_2	Most of the results hallucinated. The above result looks better then this. The training stopped as the validation loss saturated at this range. Overall this model can not achieve more then 55% of accuracy.	Prompt-1

7.4.3 Generated results and analysis

Test Case	Score	Test Case	Score	Test Case	Score
Validation case no 1	06	Validation case no 2	04	Validation case no 3	05
Validation case no 4	04	Validation case no 5	04	Validation case no 6	06
Validation case no 7	06	Validation case no 8	06	Validation case no 9	07
Validation case no 10	04				
Average	5.2				

Table 7.9: Test Cases and Scores for Result_Llama-3.2_1

The accuracy of the results were between 50 to 55 % on an average. The hallucination problem was low compared with the Llama-3.1 model. A smaller model size offers greater flexibility in adjusting hyperparameters.

7.5 Finetuned Phi-4 Results

The details of the works we have done using Phi-4 LLM model and large size code dataset is summarized here. This section starts with the prompt we used, followed by the hyperparameters we used with that prompt and at the end the results are described. This model showed very impressive result in our small size code dataset case. We expected that it will perform better in this case also and eventually it produced best results among all the results generated across all the models.

7.5.1 Prompts

The prompts are described which we used for our work for this model. The model is trained and the results are generated based on this. We experimented with different prompt and got the idea about them in the previous cases and we wrote a similar type prompt with appropriate eos tokens for this model and it is given in Table7.10. As it was giving comparatively good result compared to other models, we tried with few more prompts for this model.

Prompt Name	Prompt
Prompt-1	<|im_start|>system<|im_sep|> Generate a complete Python code based on the following test case summary and test steps.<|im_end|> <|im_start|>user<|im_sep|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|im_end|> <|im_start|>assistant<|im_sep|>
Prompt-2	<|im_start|>system<|im_sep|> Generate a complete Python code based on the following test case summary and test steps. Use chain of thought while generating the output.<|im_end|> <|im_start|>user<|im_sep|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|im_end|> <|im_start|>assistant<|im_sep|>

Table 7.10: Prompt Phi-3.5

Prompt Name	Prompt
Prompt-3	<|im_start|>system<|im_sep|> Generate a complete Python code based on the following test case summary and test steps. Use chain of thought while generating the output.<|im_end|> <|im_start|>user<|im_sep|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|im_end|> <|im_start|>assistant<|im_sep|> <code> <|im_end|>
Prompt-4	<|im_start|>system<|im_sep|> Generate a complete Python code based on the following test case summary and test steps. Think step by step while generating the output. Put non-python text in comments.<|im_end|> <|im_start|>user<|im_sep|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <|im_end|> <|im_start|>assistant<|im_sep|> <code> <|im_end|>

Table 7.11: Prompt Phi-3.5

Prompt Name	Prompt
Prompt-5	<|im_start|>system<|im_sep|> Generate a complete Python code based on the following test case summary and test steps. Think step by step while generating the output. Put non-python text in comments.<|im_end|> <|im_start|>user<|im_sep|> Name: <test_case_name> Test Case: <summary of test_case> Test Steps: <test_steps> Python Code: <code> <|im_end|> <|im_start|>assistant<|im_sep|>

Table 7.12: Prompt Phi-3.5

7.5.2 Hyperparameters and loss values

The hyperparameters we used and the losses we got are described in Table 7.13. The results are summarized in each of the set of hyperparameters. Detailed result for each case is given in next section. The best result of each prompt are showed here. We were able to process only one batch size for this model in this case. As this is a 14B model, it was not possible to use more batch size then one. This might be a hindrance to generate the best result out of it.

Table 7.13: Training Summary Phi-4

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size:1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 420 Rank of LoRA matrix = 24 Value of LoRA alpha = 48 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5000	5	Final Training Loss: 0.098 Final Validation Loss: 0.208 Result_phi-4_1	The results were around 55 to 60 percent correct on an average. Detailed results are in next section.	Prompt-1
Learning rate: 5e-04 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 300 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5400	7	Final Training Loss: 0.0503 Final Validation Loss: 0.205 Result_phi-4_2	The results were around 60 to 65 percent correct on an average. Detailed results are in next section.	Prompt-2

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 10 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 220 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5400	5	Final Training Loss: 0.0855 Final Validation Loss: 0.206 Result_phi-3.5_3	The results were around 60 to 65 percent correct on an average. Detailed results are in next section.	Prompt-3
Learning rate: 5e-04 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 460 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5400	5.5	Final Training Loss: 0.05 Final Validation Loss: 0.18 Result_phi-4_4	The results were around 65 to 70 percent correct on an average. Detailed results are in next section.	Prompt-4

Training Parameters	Epochs	Results	Comments	Prompt
Learning rate: 5e-04 Training batch size: 1 Evaluation batch size: 1 Seed value: 42 Gradient Accumulation Steps: 5 Optimizer: adamw_8bit Warmup steps: 100 Training Steps: 460 Rank of LoRA matrix = 48 Value of LoRA alpha = 96 Matrices LoRA technique used: 'Query projection', 'Key projection', 'Value projection', 'Output projection', 'Gate projection', 'Up projection', 'Down projection' Weight Decay: 0.01 input_size=5400	5	Final Training Loss: 0.05 Final Validation Loss: 0.18 Result_phi-4_5	The results were around 65 to 70 percent correct on an average. Detailed results are in next section.	Prompt-5

7.5.3 Generated results and analysis

The score of each test cases given in Table 7.14, Table 7.15,Table 7.16, Table 7.17 and Table 7.18. Description is given below the tables. Results are generated for temperature 0.1. For multiple cases we tried to generate code using temperature 0.2 and most of the cases they are almost same as 0.1. Some times codes are not generated or hallucinated for one or two cases. For that case score is replaced by the score of temperature 0.2. If it hallucinated both the cases, we did not put any score for that.

In this case the accuracy was 55 to 60 % at the beginning which reached up to 65 to 70 % at the end. The results were impressive then any of the previous results.

Test Case	Score	Test Case	Score	Test Case	Score
Phi-4 Val data 1	04	Phi-4 Val data 2	06	Phi-4 Val data 3	0 6
Phi-4 Val data 4	05	Phi-4 Val data 5	04	Phi-4 Val data 6	05
Phi-4 Val data 7	07	Phi-4 Val data 8	08	Phi-4 Val data 9	06
Phi-4 Val data 10	06				
Average	5.7				

Table 7.14: Test Cases and Scores for Result_Phi-4_1

Test Case	Score	Test Case	Score	Test Case	Score
Phi-4 Val data 1	08	Phi-4 Val data 2	05	Phi-4 Val data 3	08
Phi-4 Val data 4	06	Phi-4 Val data 5	04	Phi-4 Val data 6	05
Phi-4 Val data 7	-	Phi-4 Val data 8	06	Phi-4 Val data 9	08
Phi-4 Val data 10	05 (0.2)				
Average	6.2				

Table 7.15: Test Cases and Scores for Result_Phi-4_2

Test Case	Score	Test Case	Score	Test Case	Score
Phi-4 Val data 1	07	Phi-4 Val data 2	06	Phi-4 Val data 3	07
Phi-4 Val data 4	-	Phi-4 Val data 5	05	Phi-4 Val data 6	-
Phi-4 Val data 7	05	Phi-4 Val data 8	05	Phi-4 Val data 9	08
Phi-4 Val data 10	07				
Average	6.3				

Table 7.16: Test Cases and Scores for Result_Phi-4_3

For the last two models temperature=0.2 showing little better results, so those are put here.

Test Case	Score	Test Case	Score	Test Case	Score
Phi-4 Val data 1	08	Phi-4 Val data 2	05	Phi-4 Val data 3	08
Phi-4 Val data 4	06	Phi-4 Val data 5	-	Phi-4 Val data 6	05
Phi-4 Val data 7	05	Phi-4 Val data 8	06	Phi-4 Val data 9	08
Phi-4 Val data 10	08				
Average	6.6				

Table 7.17: Test Cases and Scores for Result_Phi-4_4

Test Case	Score	Test Case	Score	Test Case	Score
Phi-4 Val data 1	07	Phi-4 Val data 2	08	Phi-4 Val data 3	08
Phi-4 Val data 4	05	Phi-4 Val data 5	06	Phi-4 Val data 6	04
Phi-4 Val data 7	05	Phi-4 Val data 8	07	Phi-4 Val data 9	10
Phi-4 Val data 10	08				
Average	6.8				

Table 7.18: Test Cases and Scores for Result_Phi-4_5

7.6 Overall Results

The large size code dataset required an extensive search of base models, hyperparameters to achieve some good result. This part was the most significant of the whole thesis. Achieving an accuracy of around 70 % with this large size complex code is satisfying. We observed that large models with small context length performance were better then the models which provide large context length with large model size or small model size. The Phi-4 is a model which involves small context length with larger model size. That small context length was sufficient for our work and it gives very impressive results.

Chapter 8

Enhancing Fine-Tuned Model Performance

The small size code dataset and large size code dataset are finetuned using different base models and different set of hyperparameters. The detailed description of our experiments with them are summarized in Chapter 6 and Chapter 7. In this chapter the techniques are described which applied on top of the finetuned model to enhance their performance. The techniques explored in this chapter are combining finetuned model with RAG and using Reinforcement training on the finetuned model. We used the best finetuned model we got in this techniques.

8.1 Combining Finetuning and Rag

Retrieval-Augmented Generation (RAG) is a framework that integrates pre-trained sequence-to-sequence (seq2seq) language models with a non-parametric memory component, such as a dense vector index of Wikipedia, vector database of pdfs or for our case vector database of test codes, accessed via a neural retriever. This combination allows the model to retrieve relevant information from external sources and generate responses conditioned on both the input query and the retrieved documents. The RAG approach enhances the model's ability to access and manipulate knowledge, improving performance on knowledge-intensive tasks and addressing limitations associated with relying solely on parametric memory. [27]

8.1.1 Pipeline to combine RAG with finetuned model

The input provided for both the dataset contains a description of the test codes. The goal of this technique is to identify similar input descriptions from the dataset, present them to the fine-tuned model along with their corresponding code, and then provide the input description for which we seek to generate code. The following steps need to be followed to streamline the process:

- **Step 1:** We have the description for each of the data points. Now we have to make a vector database using one model to store the vector distribution of each of the data points.
- **Step 2:** We need to find out the most similar type n number of input descriptions. There are techniques like cosine similarity we can use to do this.
- **Step 3:** After getting the most similar n number of test case input description, we need to attach the corresponding test codes.
- **Step 4:** After step 3, we need to attach the input description for which we want to generate test code with the similar test cases.
- **Step 5:** The input should be put inside the finetuned model along with appropriate prompt.
- **Step 6:** After the above 5 steps we prepared our input which need to be put inside the finetuned LLM to generate output.

We followed the above 6 steps to combine the finetuned model with RAG. The details of the inputs prompts, models and results are described in the next two sub sections.

8.1.2 Results achieved for small code dataset

The input descriptions, which contains endpoint, description, inputs and output, for the small code dataset also converted in vector database using the model BAAI/bge-base-en. .

Prompt-1

```
###System:
You are an automation engineer, good at writing test codes
Generate comprehensive test code for the user query, drawing upon
the knowledge and functionality provided within a given
context.
Understand the context and generate code for the API
specifications mentioned in the query.
The provided context is the test code for an already existing
API specification or it may be the content of some library
file.
Query is the specific REST API for which a test function is to
be written.

###Follow the below guidelines:
1. Do not return results that are not in the context.
2. The library files are already written; call the library
functions where needed instead of rewriting them.
3. Use docstrings to specify the function's functionality.
4. Use appropriate function names.
5. Assert correct status code and message as given in the query.
6. Use decorators where necessary.
Think step by step and use the same coding style as that of the context.

## Context:
{context which contains description of similar test cases and
their codes}

Write down a python function that tests the API scenario
described below. Make sure you use appropriate docstring while
writing code. Below are the endpoint, inputs, and output of
the function as well.
{question which contains the input description of the test case
for which we want to generate test code.}
```

In this case we retrieve the similar input descriptions using cosine similarities. The best finetuned Phi-4 and Phi-2 model named Result phi-4_2 and Result phi2_4 respectively are used here. We used Result phi2_3 instead of phi-2_4 as the first one showed quite good result with very limited resources.

We provided three similar test cases for both the models. The prompt, parameters and corresponding results are described below.

Parameters	**Values**
Temperature	0.1
Seed value	42
Number of similar case	3

Table 8.1: Parameters Phi-2 Small Dataset

Using the finetuned model Result-phi2_3 Prompt-1 8.1.2 and the Given parameters in Table 8.1 we got quite good result. The accuracy was around 84 %. Results for temperature 0.1 is given in Table 8.2.

Test Case	**Score**	**Test Case**	**Score**	**Test Case**	**Score**
Validation data no 1	09	Validation data no 2	10	Validation data no 3	06
Validation data no 4	09	Validation data no 5	08	Validation data no 6	08
Validation data no 7	08	Validation data no 8	08	Validation data no 9	09
Validation data no 10	04	Validation data no 11	09	Validation data no 12	10
Validation data no 13	09	Validation data no 14	08	Validation data no 15	08
Validation data no 16	07	Validation data no 17	09	Validation data no 18	08
Validation data no 19	09	Validation data no 20	10	Validation data no 21	08
Validation data no 22	10	Validation data no 23	06	Validation data no 24	10
Validation data no 25	10	Validation data no 26	08		
Average	8.4				

Table 8.2: Test Cases and Scores for Result RAG + Finetuning_Phi-2_3

Prompt-2

```
<|im_start|>system<|im_sep|>
You are an automation engineer, good at writing test codes
Generate comprehensive test code for the user query, drawing upon
the knowledge and functionality provided within a given
context.
Understand the context and generate code for the API
specifications mentioned in the query.
The provided context is the test code for an already existing
API specification or it may be the content of some library
file.
Query is the specific REST API for which a test function is to
be written.

###Follow the below guidelines:
1. Do not return results that are not in the context.
2. The library files are already written; call the library
functions where needed instead of rewriting them.
3. Use docstrings to specify the function's functionality.
4. Use appropriate function names.
5. Assert correct status code and message as given in the query.
6. Use decorators where necessary.
Think step by step and use the same coding style as that of the context.

## Context:
{context which contains description of similar test cases and
their codes}

Write down a python function that tests the API scenario
described below. Make sure you use appropriate docstring while
writing code. Below are the endpoint, inputs, and output of
the function as well.<|im_end|>
<|im_start|>user<|im_sep|>
{question which contains the input description of the test case
for which we want to generate test code}
```

Parameters	Values
Temperature	0.1
Seed value	42
Number of similar case	3

Table 8.3: Parameters Phi-4 Small Dataset

Using the finetuned model Result-phi4_2 Prompt-2 8.1.2 and the Given parameters in Table 8.3 we got quite good result. The accuracy was more then 90 %. Results for temperature 0.1 is given in Table 8.2.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	10	Validation data no 2	10	Validation data no 3	07
Validation data no 4	08	Validation data no 5	08	Validation data no 6	08
Validation data no 7	08	Validation data no 8	08	Validation data no 9	10
Validation data no 10	08	Validation data no 11	10	Validation data no 12	09
Validation data no 13	10	Validation data no 14	08	Validation data no 15	09
Validation data no 16	09	Validation data no 17	09	Validation data no 18	10
Validation data no 19	09	Validation data no 20	10	Validation data no 21	08
Validation data no 22	10	Validation data no 23	10	Validation data no 24	10
Validation data no 25	10	Validation data no 26	10		
Average	9.1				

Table 8.4: Test Cases and Scores for Result RAG + Finetuning_Phi-4_2

8.1.3 Results achieved for large code dataset

The input descriptions, which contains test case name, test case summary and test case steps, for the large code dataset also converted in vector database using the model BAAI/bge-base-en . In this case also we retrieve the similar input descriptions using cosine similarities. The

best Phi-4 finetuned model named Result Phi-4_5 is used here.

Prompt-1

```
<|im_start|>system<|im_sep|>
You are an AI assistant fine-tuned to generate test case code. Your job
is to create code based on your understanding of specialized test
cases and by using a similar type of test case as a reference.<|im_end|>
<|im_start|>user<|im_sep|>
Generate a full Python script to automate the given test case. Use
your understanding of specialized test cases and follow the pattern from
the provided test case example.

**Write each and every function of your code completely with exactly
as it is done in Test Case examples and do not pass any function.
Do not directly use the functions and methods provided in the
context. Instead, understand how they are used in the test cases
and write all the required functions and methods explicitly to
complete the steps in the test case.Think step by step while
generating the output. Put non-python text in comments.
Adheres to PEP 8 coding guidelines and uses snake_case for
variable, function, and method names.**

**Test Case 1: Generate a Python script based on the provided
testing steps to verify test mention in the summary.**
Python Code 1:
###--Start of python code--
<python code>
###--End of python code--

Generate a complete Python code based on the following test case summary
and test steps. Think step by step while generating the output. Put
non-python text in comments.
<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

In this we were unable to provide three similar context due to the limitation in GPU

resource. Within the 40 GB GPU RAM we can provide only one test case for all the cases. For those cases which were small we can provide up to two similar test case. The prompt, parameters and corresponding results are described below.

Parameters	Values	Parameters	Values	Parameters	Values
Temperature	0.1 and 0.2	Seed value	42	Number of similar case	1

Table 8.5: Parameters Phi-4 large Dataset Prompt-1

Using the Prompt-1 8.1.3 and the Given parameters in Table 8.5 we got quite good result. The accuracy was more then 70 %. Results for temperature 0.1 is given in Table 8.6 and results for temperature 0.2 is given in Table 8.7.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	07	Validation data no 2	06	Validation data no 3	08
Validation data no 4	05	Validation data no 5	04	Validation data no 6	07
Validation data no 7	05	Validation data no 8	09	Validation data no 9	10
Validation data no 10	10				
Average	7.1				

Table 8.6: Test Cases and Scores for Finetuning + RAG Result_Phi-4_5(temp=0.1)

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	07	Validation data no 2	04	Validation data no 3	08
Validation data no 4	05	Validation data no 5	04	Validation data no 6	07
Validation data no 7	09	Validation data no 8	05	Validation data no 9	10
Validation data no 10	10				
Average	6.9				

Table 8.7: Test Cases and Scores for Finetuning + RAG Result_Phi-4_5(temp=0.2)

Prompt-2

```
<|im_start|>system<|im_sep|>
Generate a complete Python script to automate the given test case
using the test case summary, test steps, and a similar test case as
a reference. Think step by step while generating the output.
Put non-python text in comments.<|im_end|>
<|im_start|>user<|im_sep|>Generate a full Python script to automate
the given test case. Use your understanding of specialized test
cases codes while generating the output. One similar type test
case example is provided in the begining. You can use that for
your referrence.
**Think step by step while generating the output. Put non-python
text in comments.
Adheres to PEP 8 coding guidelines and uses snake_case for
variable, function, and method names.**

**Test Case 1: Generate a Python script based on the provided
testing steps to verify test mention in the summary.**
Python Code 1:
###--Start of python code--
<python code>
###--End of python code--

**Test Case to be automated- Generate a Python script based
on the provided testing steps to verify test mention in the summary.**
<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Parameters	Values
Temperature	0.1
Repeat Penalty	00
Seed value	42
Number of similar case	1

Table 8.8: Parameters Phi-4 Small Dataset Prompt-2

Using the Prompt-2 8.1.3 and the Given parameters in Table 8.8 we got quite good result. The accuracy was more then 70 %. Results are given in Table 8.9.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	07	Validation data no 2	06	Validation data no 3	08
Validation data no 4	06	Validation data no 5	04	Validation data no 6	07
Validation data no 7	05	Validation data no 8	09	Validation data no 9	10
Validation data no 10	10				
Average	7.2				

Table 8.9: Test Cases and Scores for Finetuning + RAG Result_Phi-4_5 Prompt-2 (temp=0.1)

Parameters	Values
Temperature	0.1 and 0.2
Repeat Penalty	00
Seed value	42
Number of similar case	2

Table 8.10: Parameters Phi-4 Small Dataset Prompt-3

Prompt- 3

```
<|im_start|>system<|im_sep|>
Generate a complete Python code based on the following test case summary
and test steps. Think step by step while generating the
output. Put non-python text in comments.<|im_end|>
<|im_start|>user<|im_sep|>Generate a full Python script to automate
the given test case. The top two test cases that are most similar
in the dataset given at the beginning. You can use that as a
reference if the test cases are similar and helpful, or else
generate it based on your own understanding.
**Think step by step while generating the output. Put non-python
text in comments.
Adheres to PEP 8 coding guidelines and uses snake_case for
variable, function, and method names.**

**Test Case 1: Generate a Python script based on the provided
testing steps to verify test mention in the summary.**
Python Code 1:
###--Start of python code--
<python code>
###--End of python code--

**Test Case 2: Generate a Python script based on the provided
testing steps to verify test mention in the summary.**
Python Code 1:
###--Start of python code--
<python code>
###--End of python code--

**Test Case to be automated- Generate a Python script based on
the provided testing steps to verify test mention in the summary.**
```

Using the Prompt-3 8.1.3 and the Given parameters in Table 8.10 we got quite good result. The accuracy was around 70 %. As described above we got out of memory error

for the first three cases where the codes were large when we gave two similar test cases. Results for temperature 0.1 is given in Table 8.11 and results for temperature 0.2 is given in Table 8.12.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	-	Validation data no 2	-	Validation data no 3	-
Validation data no 4	05	Validation data no 5	04	Validation data no 6	07
Validation data no 7	05	Validation data no 8	09	Validation data no 9	09
Validation data no 10	08				
Average	6.8				

Table 8.11: Test Cases and Scores for Finetuning + RAG Result_Phi-4_5 Prompt-3 (temp=0.1)

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	-	Validation data no 2	-	Validation data no 3	-
Validation data no 4	05	Validation data no 5	04	Validation data no 6	07
Validation data no 7	05	Validation data no 8	09	Validation data no 9	09
Validation data no 10	08				
Average	6.8				

Table 8.12: Test Cases and Scores for Finetuning + RAG Result_Phi-4_5 Prompt-3 (temp=0.2)

Overall we can conclude that we achieved around 2 to 4 percent increment in this technique for this dataset.

8.2 Reinforcement Learning

The second technique we tried on top of our finetuned model was Reinforcement learning. We focused on Proximal Policy Optimization (PPO) [31] which is a reinforcement learning

algorithm that improves upon Trust Region Policy Optimization (TRPO) [30] by simplifying the implementation while maintaining performance. It is a policy-gradient method that optimizes a clipped surrogate objective to ensure stable and reliable updates.

8.2.1 PPO Objective Function

PPO optimizes the following objective function:

$$J(\theta) = \mathbb{E}_t \left[\min(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t) \right] \tag{8.1}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policy.
- A_t is the advantage function, measuring how good an action is compared to the expected return.
- ϵ is a small positive constant (e.g., 0.1 or 0.2) that defines the clipping range.
- The clipping function ensures that the policy update is within a small range, preventing drastic changes that could destabilize training.

8.2.2 PPO Training Process

PPO follows these steps:

- Collect trajectories using the current policy π_θ.
- Compute the advantage estimates A_t using Generalized Advantage Estimation (GAE).
- Optimize the clipped objective function using stochastic gradient descent (SGD) or Adam optimizer.
- Update the policy iteratively while ensuring that the updates are constrained via clipping.

8.2.3 Implementation Pipeline for Applying The Technique in Our Case

The implementation is done in our set up using the following steps

- The code generated by providing the inputs for every batch using the finetuned model.
- Compared it with the actual code and score it between 0 and 1 by Phi-4 model.
- The input tensor, output tensor and score is used to train the model batch wise.

The process is time taking as multiple tasks are done in the training loop. For the same cost the computation cost is also high. Along with that ppo training process itself requires good amount of computation cost.

8.2.4 Results for small and large size code dataset

For the small code dataset we used the Phi-2 finetuned model.

Test Case	Score	Test Case	Score	Test Case	Score
Validation data no 1	09	Validation data no 2	10	Validation data no 3	07
Validation data no 4	10	Validation data no 5	08	Validation data no 6	08
Validation data no 7	08	Validation data no 8	08	Validation data no 9	08
Validation data no 10	06	Validation data no 11	08	Validation data no 12	10
Validation data no 13	09	Validation data no 14	10	Validation data no 15	07
Validation data no 16	07	Validation data no 17	08	Validation data no 18	08
Validation data no 19	08	Validation data no 20	08	Validation data no 21	09
Validation data no 22	09	Validation data no 23	07	Validation data no 24	07
Validation data no 25	07	Validation data no 26	08		
Average	8.2				

Table 8.13: Test Cases and Scores

The model achieved around 80% with finetuning. Around 2 to 3 % of improvement we observed here. The results are given in Table 8.13.

The technique took more then 35 GB GPU RAM while training the phi-2 model. Which shows the amount of computation burden comes with this techniques. As a result the large size code dataset was impossible within our limited resources. Along with that large size code dataset only performed good with phi-4 model and that is 4 to 5 times larger then phi-2. As a result it would require more computation costs. For all these reasons unfortunately we can not implement the process in large size code dataset.

Chapter 9

Conclusion

The work comprehensively explored the fine-tuning techniques, optimization strategies, and resource-efficient methods to enhance the performance of large language models for test code generation. Test codes are one of the most important part in web development and automating them using large language models reduces huge amount of time and work power for that. Our work include some notable and essential development and observations in that direction.

The initial part was started with dataset preparation. It is very important to prepare a quality data to generate the expected output. The test step descriptions and test steps prepared and every data points reviewed before starting the training. Extensive test with prompts done in each steps of our work to extract the best output from the models.

Training with smaller input length is little easier as it provides more options to play with the hyperparameters. As each line of the code need to be logically connected with each other, if the size of the code is small, it become little easy to maintain the logical connections between each other. Also having higher number of data points helps in better training. We observed that due to all these reasons our small size code dataset performed well in the finetuning work and we achieved around 85% accuracy with that dataset which we saw in Chapter 6.

The level of challenge increased with the large size code dataset. As the size of the code increased more then 10 times compared with the small code dataset, we observed that

the difficulty to generate good quality code increases. Extensive experiment was done on that dataset to achieve results as good as possible and it is described in Chapter 7. After thorough experiment with different set of hyperparameters, base LLMs and prompts we achieved around 70% accuracy.

After conducting a broad set of experiments, we observed that the results reached a stabilization point for both the datasets. In the next step, focus was given on the techniques which we can use on the top of the finetuned model to improve its accuracy.

The first technique tried in that direction was Retrieval-Augmented Generation (RAG). Through this technique some test cases of that same context is also provided to the model along with the steps for the test case for which we want to generate code. Through this technique the performance of the finetuned model for small code dataset increased for around 5 % and 2 to 4 % for large code dataset. The results showed the effectiveness of the technique.

The second technique used was Reinforcement learning. The proximal policy optimization technique we tried. The result for small code dataset improved around 2 to 3 %. But the process is quite computation expensive. We were unable to implement the technique for our large size code dataset.

After a wide range of experiments we achieved some significant result and notable observations which is described in the thesis in the direction of automating our tasks through the recent development in Large Language Models. The work contributed significantly in reducing the time of the writing test codes. It shows a pathway of generating code of a specific domain. Also it gives a description of how the results one can expect depending on the kind of dataset they are having. The importance of datasets are analyzed in the work. The work depicts that automating some task through LLM depends on quality data and as well as meticulous experiment with each model and parameter to achieve the ultimate goal. In the overall representation of our work in the thesis, prompts are given along with the dataset, model and the hyperparameters to give the complete picture. Based on our extensive work on the prompting, we realized that the art of using a good dataset and a good model involves a proper efficient prompt to bridge both of them. The model need to understand different parts of the dataset (for example test case summary, test steps and test codes are different part of the dataset for large size code dataset) and the relation between them. Different parts, their relation and the task the LLM need to do must be describe properly through prompting. This helps the LLMs to learn properly while training and generating efficient

output while using them. We observed that we can make our life more efficient by using the LLMs with some needful actions. It has the ability to perform in complex situation if we use them properly. Another important aspect of the work was focusing on efficient training with limited GPU RAM. The problem becomes crucial when we try to make LLM learn with longer context length. We tried with different techniques and it shows the efficiency of those techniques also in this type of work. I hope the observations and techniques will help for the further work in this direction.

There are several potential improvements that can be made, either by building on top of the fine-tuned model or by enhancing the base model, to improve test code generation and, more broadly, any type of logical task. The Reinforcement learning techniques can be used further with different ways to make the model learn better. In every type of tasks where further finetuning is done with RL, reward model plays a crucial role there. Based on another AI model can save time but it requires more computation cost. Along with the computation cost, it includes a certain level of uncertainty in the process. Wrong reward can impose more focus on wrong direction or make the model less focus on right direction. Efficiently rewarding can make the results better.

Enhancing base models within a specific domain will enable them to learn domain-specific knowledge more effectively, leading to better performance when fine-tuned. So the improvement in base model preparation and sophisticated finetuning process can lead the performance in new hight.

Bibliography

[1] Hochreiter, S., and Schmidhuber, J., "Long Short-Term Memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997. Available at: https://www.bioinf.jku.at/publications/older/2604.pdf

[2] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y., "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," CoRR, abs/1412.3555, 2014. Available at: https://arxiv.org/pdf/1412.3555

[3] Sutskever I., Vinyals O., and Le Q. V., "Sequence to Sequence Learning with Neural Networks," arXiv preprint arXiv:1409.0473, 2014. Available at: https://arxiv.org/pdf/1409.0473

[4] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser Ł., and Polosukhin I., "Attention Is All You Need," Advances in Neural Information Processing Systems (NeurIPS), 2017. Available at: https://arxiv.org/pdf/1706.03762

[5] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K.,"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," arXiv preprint arXiv:1810.04805, 2018. Available at: https://arxiv.org/pdf/1810.04805

[6] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. "Improving Language Understanding by Generative Pre-Training," OpenAI, 2018. Available at: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[7] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. "Language Models are Few-Shot Learners," Advances in Neural Information Processing Systems (NeurIPS), 2020. Available at: https://arxiv.org/pdf/2005.14165

[8] OpenAI."GPT-4 Technical Report," arXiv preprint arXiv:2303.08774, 2023.Available at: https://arxiv.org/pdf/2303.08774

[9] Microsoft. "Phi-3: Efficient Large-Scale Language Model for Edge Devices," Microsoft Research, 2024. Available at: https://www.microsoft.com/en-us/research/publication/phi-3-efficient-large-scale-language-model-for-edge-devices/

[10] Meta AI. "Llama 3: Open and Efficient Foundation Language Models," Meta AI Blog, 2024.Available at: https://ai.meta.com/blog/llama-3-open-and-efficient-foundation-language-models/

[11] Fan L., Bai Z., Zhang L., and Li J., "A Bibliometric Review of Large Language Models Research from 2017 to 2023," arXiv preprint arXiv:2304.02020, 2023.

[12] Zaken, E. B., Ravfogel, S., Goldberg, Y., "BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models," arXiv preprint arXiv:2106.10199, 2021. Available at: https://arxiv.org/pdf/2106.10199

[13] Li, X. L., Liang, P., "Prefix-Tuning: Optimizing Continuous Prompts for Generation," arXiv preprint arXiv:2101.00190, 2021. Available at: https://arxiv.org/pdf/2101.00190

[14] Hu, E., Shen, Y., Wallis, P., et al., "LoRA: Low-Rank Adaptation of Large Language Models," arXiv preprint arXiv:2106.09685, 2021. Available at: https://arxiv.org/pdf/2106.09685

[15] Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L., "QLoRA: Efficient Finetuning of Quantized LLMs," arXiv preprint arXiv:2305.14314, 2023. Available at: https://arxiv.org/pdf/2305.14314

[16] Hastie, T., Tibshirani, R., and Friedman, J., *An Introduction to Statistical Learning*, 2nd ed., 2023.

[17] Hopsworks, "Gradient Accumulation," Available at: https://www.hopsworks.ai/dictionary/gradient-accumulation

[18] Loshchilov, I., & Hutter, F. "Decoupled Weight Decay Regularization," arXiv preprint arXiv:1711.05101, 2017. Available at: https://arxiv.org/pdf/1711.05101

[19] Raffel, C., Shazeer, N., Roberts, A., et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," arXiv preprint arXiv:2106.09685, 2021.

[20] Mudadla, S., "WeMudadlaight Decay in Deep Learning," Available at: https://medium.com/@sujathamudadla1213/weight-decay-in-deep-learning-8fb8b5dd825c

[21] GeeksforGeeks, "In the Context of Deep Learning, What Is Training Warmup Steps?," Available at: https://www.geeksforgeeks.org/in-the-context-of-deep-learning-what-is-training-warmup-steps/

[22] Microsoft, "Phi-2" 2023. Available at: https://huggingface.co/microsoft/phi-2

[23] Microsoft, "Phi-3.5-mini-instruct" 2024. Available at: https://huggingface.co/microsoft/Phi-3.5-mini-instruct

[24] Microsoft, "Phi-4" 2024. Available at: https://huggingface.co/microsoft/phi-4

[25] Microsoft, "Llama-3.1-8B-Instruct" 2024. Available at: https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[26] Microsoft, "Llama-3.2-3B" 2024. Available at: https://huggingface.co/meta-llama/Llama-3.2-3B

[27] Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Küttler H., Lewis M., Yih W., Rocktäschel T., Riedel S., and Kiela D., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," arXiv preprint arXiv:2005.11401, 2020. Available at: https://arxiv.org/pdf/2005.11401

[28] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L. Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. "Training language models to follow instructions with human feedback," arXiv preprint arXiv:2203.02155, 2022. Available at: https://arxiv.org/pdf/2203.02155

[29] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard. "RLAIF: Scaling Reinforcement Learning from Human Feedback with AI Feedback," International Conference on Learning Representations (ICLR), 2023. Available at: https://arxiv.org/abs/2303.12345

[30] Schulman, J., Levine, S., Moritz, P., Jordan, M., and Abbeel, P., "Trust Region Policy Optimization," *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015. Available at: https://jmlr.org/proceedings/papers/v37/schulman15.pdf

[31] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347, 2017. Available at: https://arxiv.org/abs/1707.06347