

REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING



**CODE GENERATION USING LARGE LANGUAGE
MODELS**

19011915 – Parsa Kazerooni
19011081 – Muhammed Hakan Kılıç

SENIOR PROJECT

Advisor
Asst. Prof. Göksel BİRİCİK

December, 2023

ACKNOWLEDGEMENTS

We would like to thank our advisor Asst. Prof. Göksel BİRİCİK, who proposed the project and helped us implement with his experience and knowledge in the field of natural language processing.

Parsa Kazerooni
Muhammed Hakan Kılıç

TABLE OF CONTENTS

LIST OF ABBREVIATIONS	v
LIST OF FIGURES	vi
LIST OF TABLES	vii
ABSTRACT	viii
ÖZET	ix
1 Introduction	1
2 Related Work	3
3 Feasibility	5
3.1 Technical Feasibility	5
3.1.1 Hardware Feasibility	5
3.1.2 Software Feasibility	5
3.2 Workforce and Time Planning	5
3.3 Legal Feasibility	6
3.4 Economic Feasibility	6
4 System Analysis	7
5 System Design	8
5.1 Software Design	8
5.1.1 Pre-trained Models	8
5.1.2 Dataset and Preprocessing The Data	9
5.1.3 Fine-tuning	9
5.1.4 Evaluation	10
5.2 Database Design	11
5.3 Input-Output Design	11
6 Implementation	13

7 Experimental Results	14
7.1 QLoRA adaptation	14
7.1.1 Quantization	14
7.1.2 LoRA	14
7.2 Training	15
8 Performance Analysis	19
8.1 HumanEval Evaluation	19
9 Result	23
References	24
Curriculum Vitae	26

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
LLM	Large Language Model
LoRA	Low-Rank Adaptation
CNN	Convolutional Neural Network
FT	Fine-Tune

LIST OF FIGURES

Figure 3.1	Workforce and Time Planning of The Project	6
Figure 5.1	Block Diagram of The System	8
Figure 5.2	The Distribution of token lengths of the dataset	10
Figure 5.3	Example of the dataset	11
Figure 6.1	User Interface	13
Figure 6.2	Example prompt	13
Figure 7.1	CodeGen-350m's training loss and two trial runs	17
Figure 7.2	Mistral-7b's training loss	17
Figure 7.3	Phi-2's training loss	18
Figure 7.4	Phi-2's training loss next to the learning rate's value	18
Figure 8.1	Prompt of Task HumanEval/117	19

LIST OF TABLES

Table 7.1	Changes of resource demand before and after quantization . . .	14
Table 7.2	LoRA configurations for each model	15
Table 7.3	The hyper-parameters used to finetune each model	16
Table 8.1	The evaluation results on $pass@1$ [%]	21

Code Generation Using Large Language Models

Parsa Kazerooni
Muhammed Hakan Kılıç

Department of Computer Engineering
Senior Project

Advisor: Asst. Prof. Göksel BİRİCİK

In this study, three different open-source large language models have been fine-tuned on prompt-code pairs dataset and their code generation capabilities were compared.

In today's fast paced age, reducing the time it takes for a project to hit the market has become more important than ever. Artificial intelligence developed for software developers to produce code has increased its popularity. With the aid of new state-of-the-art architecture called transformers, artificial intelligence, and more specifically, large language models, have garnered significant attention for their potential to expedite the software development process.

The scope encompasses three different open-source LLMs, each tailored to generate code effectively. By fine-tuning, enhancing the code generation of these models was aimed capabilities, aligning them with the needs of software development tasks.

By fine-tuning three models, CodeGen-350M, Mistral-7b and Phi-2 on a dataset consisting 20,000 instruction-code pairs, the models turned into a chat-based coding assistant, with proper python code generation that can handle simple programming problems. The accuracy metric that was used are functional correctness on HumanEval and MBPP problem sets and their extended test cases. Despite witnessing small change in the base model's logical thinking and coding abilities, the fine tuned models were able to generate expected results consistently by being asked in natural languages such as English, resulting in a user-friendly coding assistant model.

Keywords: Large Language Models, Code Generation, Transformers, HumanEval

Büyük Dil Modellerinden Yararlanarak Kod Geliştirme

Parsa Kazerooni
Muhammed Hakan Kılıç

Bilgisayar Mühendisliği Bölümü
Bitirme Projesi

Danışman: Dr. Öğr. Üyesi Göksel BİRİCİK

Bu çalışmada, 3 farklı açık kaynaklı büyük dil modeli, komut-kod çifti veri kümesi üzerinde ince ayardan geçirilmiş ve kod üretme yetenekleri karşılaştırılmıştır.

Günümüzün hızlı tempolu çağında, bir projenin piyasaya sürülmesi için gereken süreyi azaltmak her zamankinden daha önemli hale gelmiştir. Yazılım geliştiricilerin kod üretmesi için geliştirilen yapay zeka popülerliğini artırmıştır. Transformatör adı verilen son teknoloji ürünü yeni mimarilerin yardımıyla yapay zeka ve daha spesifik olarak büyük dil modelleri, yazılım geliştirme sürecini hızlandırma potansiyelleri nedeniyle büyük ilgi görmüştür.

Kapsam, her biri etkili bir şekilde kod üretmek için uyarlanmış üç farklı açık kaynaklı LLM'yi kapsamaktadır. İnce ayar yaparak, bu modellerin kod üretme yeteneklerini geliştirmeyi ve bunları yazılım geliştirme görevlerinin ihtiyaçlarıyla uyumlu hale getirme amaçlanmıştır.

CodeGen-350M, Mistral-7b ve Phi-2 olmak üzere üç modelin 20.000 komut-kod çiftinden oluşan bir veri kümesi üzerinde ince ayarlanmasıyla, modeller basit programlama problemlerinin üstesinden gelebilecek uygun python kodu üretimi ile sohbet tabanlı bir kodlama asistanına dönüştü. Kullanılan doğruluk ölçütü, HumanEval ve MBPP problem setleri ve bunların genişletilmiş test durumları üzerindeki işlevsel doğruluktur. Temel modelin mantıksal düşünme ve kodlama yeteneklerinde küçük değişiklikler görülmesine rağmen, ince ayarlı modeller İngilizce gibi doğal dillerde sorulduğunda tutarlı bir şekilde beklenen sonuçları üretebilmiş ve

kullanıcı dostu bir kodlama asistanı modeli ortaya çıkmıştır.

Anahtar Kelimeler: Büyük Dil Modelleri, Kod Üretme, HumanEval

1

Introduction

In today's fast paced age, reducing the time it takes for a project to hit the market has become more significant than ever. Code generating AIs can help developers automate repetitive and routine coding tasks, correct time consuming errors, write better and cleaner code, save time for other things. All these may lead to increased productivity and job satisfaction. Therefore, artificial intelligence developed for software developers to produce code has increased its popularity lately. With the aid of new state-of-the-art architecture called transformers, artificial intelligence, and more specifically, large language models, have garnered significant attention for their potential to expedite the software development process.

In this project, three different open-source LLMs were tuned to generate code for a given prompt or to improve their accuracy on the specific task of code generation. Fine-tuning involves taking pre-trained models and further training them on smaller, specific datasets to refine their capabilities and improve performance in a particular task or domain. Fine-tuned models on code datasets outperform base models such as GPT and BART, which have not been specifically trained on code datasets.

There are three general methods for fine-tuning a model: self-supervised, supervised, and reinforcement learning. Supervised fine-tuning involves training the LLM on a labeled dataset, where input-output pairs are provided for a specific task. Self-supervised learning, on the other hand, involves fine-tuning the model without explicit labels or external supervision. Instead, it utilizes the inherent structure and patterns within the data to acquire meaningful representations. Fine-tuning through reinforcement learning involves training the LLM in an environment where it takes actions and receives rewards based on those actions. The model learns to maximize the cumulative reward over time. Most of the best-performing LLMs on the well-known benchmarks, were trained with a combination of self-supervised and reinforcement learning with human feedback.

Since the nature of the problem relies on an instruction-based conversation with the

model, our model must perform a sequence-to-sequence inference that includes a high-quality prompt with clear instructions, possible but not mandatory inputs, and formatting hints to generate the response code. To accomplish this task, transformers [1] are used as the architecture for our language model. To fine-tune a pre-trained transformer model, we need a dataset consisting of enough instruction-code pairs. Then, the prompt is constructed by merging instructions and codes with the aforementioned prompt style and fed to the training algorithm.

Our goal was to create a model that would accurately answer coding related problems by fine-tuning other LLMs, and present this model to the end user in a nice, user-friendly interface.

Our method differs from other fine-tuned LLMs in terms of dataset selection. Our model has been trained on high-quality, interview style coding problems which helped a lot to the success of our model when it comes to the hard coding problems.

Section number 2 lists the related work that has been done on the subject. The following section contains the feasibility work that was done before starting this project. Sections number 4 and 5 are dedicated to system analysis and design, respectively. Images taken from our program are in section number 6. Sections 7 and 8 were dedicated to the experiments that were done and their results. Finally, in section number 9, the results of this project were shared.

2 Related Work

After the significant successes of large language models (LLMs) such as BERT[2] and GPT[3], there has been a substantial increase in research efforts in the field of LLMs in recent years. Typically, LLMs can be categorized into three different architectures: encoder-only models, decoder-only models, and encoder-decoder models.

Encoder-only and decoder-only models are often ideal for understanding tasks like code retrieval or generation tasks like code synthesis, respectively. As for encoder-decoder models, they can be adapted to both code understanding and generation tasks, but they do not always outperform decoder-only or encoder-only models.

CodeX is one of the first LLM which has been tailored for code generation. It is a GPT-like LLM which means that it utilizes auto-regressive transformer model. Codex is finetuned on publicly available code from GitHub. Training dataset was collected in May 2020 from 54 million public software repositories hosted on GitHub, containing 179 GB of unique Python files under 1 MB. Dataset was filtered before training the model. files which were likely auto-generated, had average line length greater than 100, had maximum line length greater than 1000, or contained a small percentage of alphanumeric characters was discarded. After filtering, our final dataset totaled 159 GB [4].

CodeGEN is a GPT-like casual language model with the context length of 2,048 tokens with model sizes of 350M, 2.7B 6.1B, 16B billion parameters. CodeGEN was released as three models, trained on General NL, Multilingual Code, and Python-focused datasets. ThePile (ref) is a subset of github repositories with more than 100 stars, which the majority of the corpus is English text, resulting to CodeGEN-NL with Natural language capabilities. The BIGQUERY is a multi-lingual dataset extracted from Google's public dataset which contains open-source licensed codes in multiple programming languages. Also there is a 5.5TB subset derived from the BigQuery dataset only containing Python code, that was used to train CodeGEN-Mono [5].

CodeGeeX is a multilingual code generation model with 13 billion (13B) parameters, pre-trained on a large code corpus of 23 programming languages. As of June 22, 2022, CodeGeeX has been trained on more than 850 billion tokens on a cluster of 1,536 Ascend 910 AI Processors for over two months. The original GPT model uses a pooler function to obtain the final output. We use an extra query layer [6] on top of all other transformer layers to obtain the final embedding through attention. The training corpus contains two parts. The first part is from open source code datasets, the Pile and CodeParrot6 . The Pile contains a subset of public repositories with more than 100 stars on GitHub, from which we select files of 23 popular programming languages including C++, Python, Java, JavaScript, C, Go, and so on. We identify the programming language of each file based on its suffix and the major language of the repository it belongs to. CodeParrot is another public Python dataset from BigQuery. The second part is supplementary data of Python, Java, and C++ directly scraped from GitHub public repositories that do not appear in the first part. We choose repositories that have at least one star and a total size within 10MB, then we filter out files that: 1) have more than 100 characters per line on average, 2) are automatically generated, 3) have a ratio of alphabet less than 40%, 4) are bigger than 100KB or smaller than 1KB. We format Python code according to the PEP8 standards [7].

StarCoder is a fine-tuned model of StarCoderBase which has 15.5B parameter with 8K context length, infilling capabilities and fast large-batch inference enabled by multi-query attention. StarCoderBase has the same architecture as SantaCoder [8] It is a decoder-only Transformer with Fill-in-the-Middle [9], MultiQuery-Attention [10], and learned absolute positional embeddings. StarCoderBase is trained on 1 trillion tokens sourced from The Stack [11], a large collection of permissively licensed GitHub repositories with inspection tools and an opt-out process. StarCoderBase has been fine-tuned on 35B Python tokens, resulting in the creation of StarCoder .

Code Llama were trained on sequences of 16k tokens and show improvements on inputs with up to 100k tokens. 7B and 13B Code Llama and Code Llama - Instruct variants support infilling based on surrounding content. Code Llama reaches state-of-the-art performance among open models on several code benchmarks, with scores of up to 53% and 55% on HumanEval and MBPP, respectively. Code Llama were trained on 500B tokens during the initial phase, starting from the 7B, 13B, and 34B versions of Llama 2. Code Llama is trained predominantly on a near-deduplicated dataset of publicly available code. This dataset contains many discussions about code and code snippets included in natural language questions or answers. Data is tokenized via byte pair encoding [12], employing the same tokenizer as Llama and Llama 2 [13].

3.1 Technical Feasibility

3.1.1 Hardware Feasibility

In the process of developing this project, we need computational power to train the model, storage space to save the model and its parameters. To meet these requirements, a CPU and disk storage is needed. A computer which meets these stated requirements should be as follows:

- Fast Internet Connection to use Cloud Computing
- 16 GB VRAM GPU such as V100 or A100
- CPU with 3GHz+ frequency
- 20+ GB SSD

3.1.2 Software Feasibility

As the project is implemented in Python, a cross-platform language, it is agnostic to any specific operating system requirements. A general purpose operating system such as Windows or Linux will be sufficient.

3.2 Workforce and Time Planning

Workforce and Time Planning of the project is shown on the Gantt diagram in Figure 3.1.

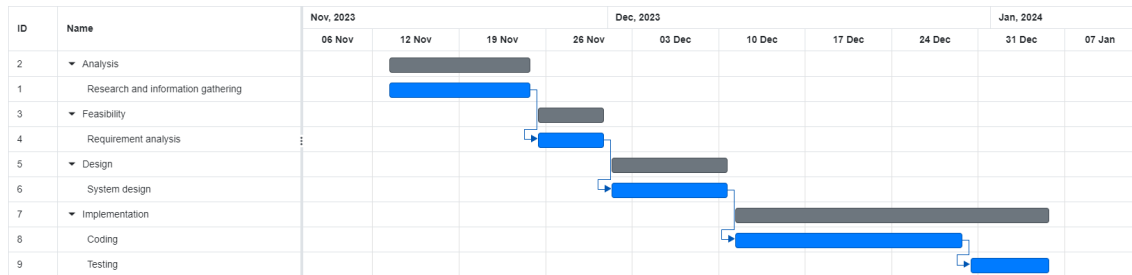


Figure 3.1 Workforce and Time Planning of The Project

3.3 Legal Feasibility

The project is non-commercial and does not violate anyone's protected rights. It complies with all laws and regulations. No special ethics committee approval is required for this project. The data set used is not confidential. All rights belong to Yıldız Technical University Computer Engineering Department.

3.4 Economic Feasibility

Approximately 5210 TL (1500 TL motherboard, 1300 TL LCD monitor, 1560 TL CPU, 620 TL RAM, 330 TL SSD) would be needed to meet the hardware requirements for the project. However, ColabPro has been used to satisfy the hardware requirements which costs 162tl each month for approximately 20 hours of V100 GPU computation. Since the programming language and environment to be used in the project are free of charge, no expenditure will be made on software. Since two employees need to be employed part-time for 5 months for the development period of the project, approximately 100.000 TL is required as it will be equal to 5 man-month salaries. In total, 100810tl is required for this project.

There will be no financial gain in the event of the successful conclusion of the project.

4

System Analysis

The goal of the project is to produce a model that is capable of answering code related questions accurately. Information that was needed to achieve this goal have been obtained with the help of web and our project advisor Asst. Prof. Göksel BİRİCİK. Our dataset consists of question-answer pairs related to the coding domain. It has been gathered from various sites on the web such as Github, Huggingface, Leetcode etc.

A popular destination of many researchers which who work with language models, CNNs, or other types of machine learning models is Huggingfaces and its Transformers library enables us to access a large library of pre-trained models and wide range of datasets. The library contains various auxiliary functions to manipulate, train, save and publish the available models on the hub. We grab most of our resources using this tool. We also need computational resources that are powerful enough to handle the intense training and large amount of data. We use Google Colab Pro to access high-performing GPUs and plenty of memory for our tasks.

As for benchmarking the performance and accuracy, HumanEval [4], MBPP [14] and custom test-cases were planned to be used to measure the success of the model. The model was given certain test-cases and its response was extracted in a runnable code, evaluating each test-case and passed or failed the test-cases. Minimizing the scoring loss that would occur by giving natural language instructions and generating consistently accurate solutions, is sufficient for the project to be considered successful.

5.1 Software Design

The block diagram of the system that we designed to generate code to the code related questions can be seen in figure 5.1

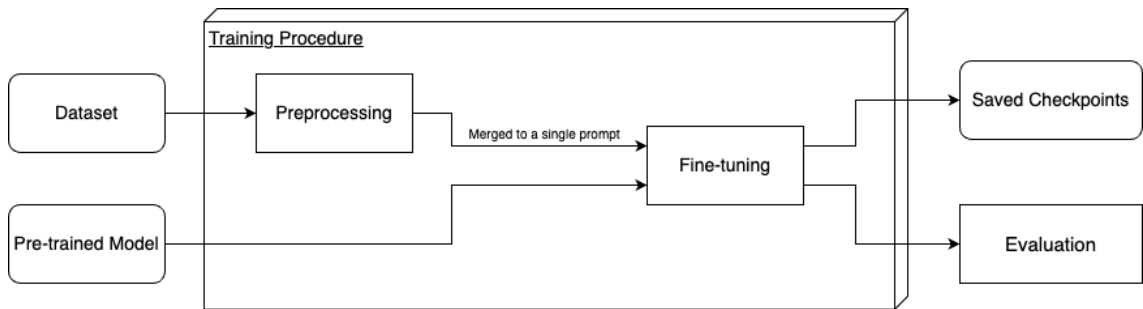


Figure 5.1 Block Diagram of The System

5.1.1 Pre-trained Models

As mentioned before, instead of training a language model from scratch, it is a common strategy to use a pre-trained model trained on a more general set of data, then to fine-tune it according to our needs with more specific data. A noticeable number of LLMs today are trained on the web data, such as blog posts, forums, and etc. It's certain that the general web corpus, contains text like source codes as well; Which can be found in developer blogs, Github, code tutorials and etc. That means that there are great quantities of LLMs that understand both English and source code, but it might not be accurate enough to be utilized as a code generation model. We used three different open source models to employ in our work:

- CodeGen-350m [5]
- Mistral-7b [15]
- Phi-2 [16]

These models served as our initial checkpoints, forming the foundation upon which we built our fine-tuned models. The choice of using the smaller versions of the models was based on our computational resources limit and budget.

5.1.2 Dataset and Preprocessing The Data

To prepare the data for the models CodeGen[5] and Phi-2[16], it was decided to use Alpaca style formatting for prompts demonstrated as the following:

Alpaca Style Formatting:

Prompt: Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

Instruction: {{Instruction/Description of the problem}}

Input: {{Input if necessary}}

Response: {{Model Response}}

The second model, Mistral[15], was recommended to be used with its own optimal prompt template

Mistral Style Formatting:

Prompt: <s> [INST] {{Instruction/Description of the problem}} [/INST]
{{Model Response}}

The generated prompts were stored in the dataset as a new column. To get an overview of the length of each prompt, the distribution of token sizes was calculated by tokenizing the prompts, as shown in Figure 5.2

Each model comes with its own tokenizer, So we didn't have to come up with our implementation of it.

After that, the tokenized dataset was split into train and test divisions, with 80% of the data to be dedicated to training. (around 16,017 train samples and 4005 test samples).

5.1.3 Fine-tuning

After preparing the data, the next step is to tokenize it properly and feed it into the training loop. Tokenizing strings with different lengths may result in some inconsistency that can be prevented by adding a padding token to the smaller strings. After that, the training process begins. Firstly, the tokenized dataset is fed to the model, applying a forward pass, and then the loss is calculated. Following that,

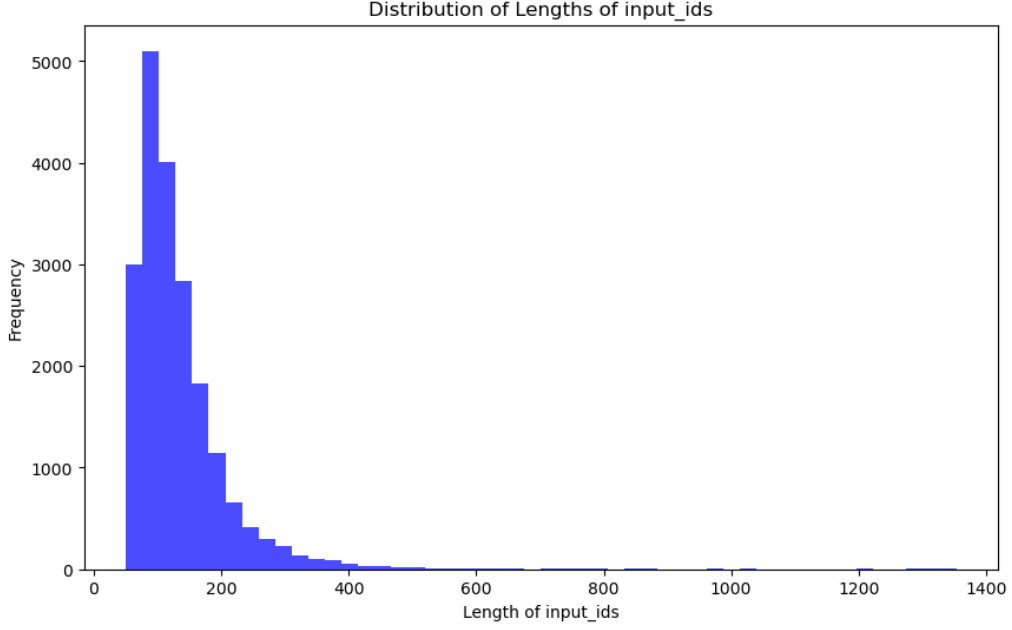


Figure 5.2 The Distribution of token lengths of the dataset

backpropagation is applied, and by using optimizing algorithms such as the Adam optimizer [17], we achieve slightly better parameters and start the loop over with the new parameters.

5.1.3.1 Fine-tuning with PEFT Techniques

To improve our pre-trained models' performance and suitability for our domain, we utilized Parameter Efficient Fine-tuning (PEFT) methods. PEFT offers a flexible strategy for refining models, enabling us to tailor the pre-trained models to match the unique traits of our target dataset. By undergoing multiple rounds of fine-tuning, we fine-tuned the models to better align with the intricacies of our data, striking a balance between generalization and specificity. We used LoRA [18] technique when fine-tuning our model which is one of the PEFT techniques. LoRA is a training technique that speeds up the training of expansive models with reduced memory usage. It involves incorporating pairs of rank-decomposition weight matrices, known as update matrices, into the current weights. Specifically, it focuses training efforts solely on these recently introduced weights.

5.1.4 Evaluation

The assessment of a Large Language Model trained on code involves utilizing various benchmarks, such as HumanEval and MBPP (Model-Based Probing Procedures).

HumanEval provides a structured evaluation framework that involves 164

hand-written programming problems with average of three tests per each problem, which are ran through the model-generated code outputs. This benchmark allows for the qualitative assessment of the model’s code generation in terms of correctness and accuracy. [4].

Additionally, the Mostly Basic Python Problems dataset (MBPP) [14] comprises a suite of probing tasks designed to systematically investigate the LM’s understanding and representation of code-related concepts, syntax, and programming logic. Through these probing procedures, the model’s internal mechanisms and its ability to comprehend, reason, and generate code in diverse scenarios are scrutinized, providing valuable insights into its strengths, weaknesses, and overall proficiency in handling coding tasks [14].

We utilized HumanEval and MBPP and their extended test sets introduced by EvalPlus[19] during evaluating our fine-tuned model for this report.

In the subsequent sections, we delve into the experimental setup, characteristics and results obtained through the implementation of our fine-tuned models.

5.2 Database Design

All of the models have been fine-tuned on the same dataset that’s been used for fine-tuning the Code Alpaca model. It contains 20K instruction-following data. Dataset is a list of dictionaries, each dictionary contains the following fields: instruction, input(optional) and output. A little glimpse at the dataset:




output string · lengths	instruction string · lengths	input string · lengths
		
arr = [2, 4, 6, 8, 10]	Create an array of length 5 which contains all even numbers between 1 and 10.	
Height of triangle = opposite side length * sin (angle) / side length	Formulate an equation to calculate the height of a triangle given the angle, side lengths and opposite..	
def replace(self, replace_with): new_string = "" for char in self: if char == " ": new_string +=_	Write a replace method for a string class which replaces the given string with a given set of..	string = "Hello World!" replace_with = "Greetings!"
arr = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45]	Create an array of length 15 containing numbers divisible by 3 up to 45.	
def find_num_distinct_states(matrix): states = set() for row in matrix: state = "".join([str(x) for x in_	Write a function to find the number of distinct states in a given matrix.	matrix = [[1, 0, 0], [1, 0, 1], [1, 1, 1]]

Figure 5.3 Example of the dataset

5.3 Input-Output Design

The input and output of the system is handled by Web page components. The user enters a prompt in a given text box and then clicks submit. The input is then put into the appropriate instruction format as mentioned in section 5.1.2. After it is put into

the right format, it is tokenized by model's tokenizer and fed to the model. The output generated by the model then gets untokenized and presented to the user within the same page. Picture of the interface can be seen in figure 6.1 and figure 6.2

6

Implementation

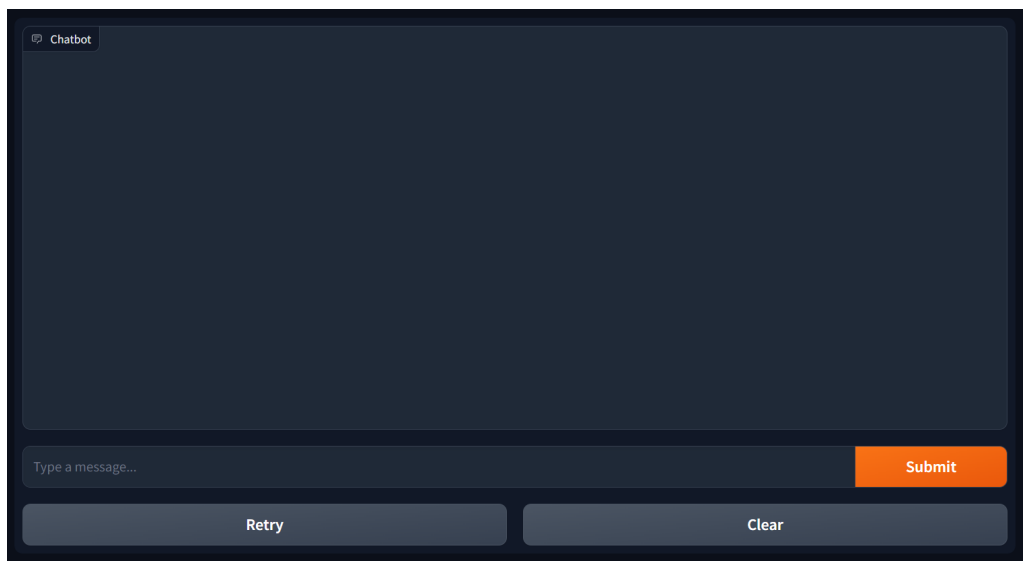


Figure 6.1 User Interface

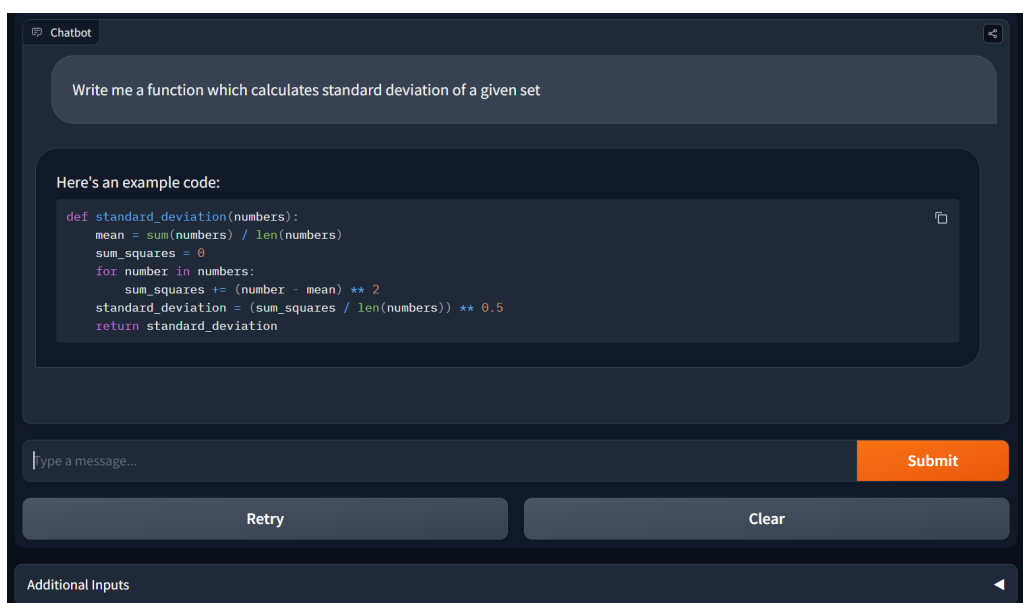


Figure 6.2 Example prompt

7

Experimental Results

7.1 QLoRA adaptation

Using QLoRA results in a trade-off between Memory usage and training time. Experiments show that while it conserves 33% memory, it increases the training time upto 39% [20]. According to the environment provided, Efficient memory utilization was a more important factor than training time. Because of that, QLoRA adaptation was decided to be used in our workflow.

7.1.1 Quantization

Based on our computational resources, the smaller CodeGen model was chosen to be trained locally without quantization, since it would entirely fit in the available memory. But the bigger model, Mistral, would require significantly longer training and inference time if it was loaded on the local machine because the memory usage was bottle-necked. So we decided to load it on a remote Colab environment with GPU and CUDA support, running on Tesla V100 GPU with 16 Gigabytes of Video RAM. the Mistral model was configured with 4-bit quantization, which after loading it to the GPU, it only took 5.5 Gigabytes of the resources. Table 7.1 compares the quantized model with the base model in terms of resource demand.

Table 7.1 Changes of resource demand before and after quantization

Model	VRAM Required	Minimum Available Hardware
Mistral-7b	12 GB + Inference	Tesla A100 (24Gb RAM)
4bit Quantized Mistral-7b	5.5 GB + Inference	Telsa V100 (16Gb RAM)

7.1.2 LoRA

The LoRA configuration that was used vary based on the model size. Table 7.2 demonstrates each model's configuration before the training.

To select the trainable parameters, the original QLoRA paper[21] recommends us to consider all the linear layers of the model to achieve the best results most of the time. Each model has a unique structure with different layers, table 7.2 lists the linear layers that LoRA adaptors were attached to in each model.

Table 7.2 LoRA configurations for each model

**There was also lm_head layer but it wasn't required*

Model	LoRA Configuration	Linear Layers*
CodeGen-350m	$r = 4$	qkv_proj
	$\alpha = 8$	out_proj
	dropout = 0.05	fc_in
		fc_out
Mistral-7b	$r = 16$	down_proj
	$\alpha = 32$	up_proj
	dropout = 0.05	{k,o,q,v}_proj
		gate_proj
Phi2-2.7b	$r = 16$	fc1
	$\alpha = 32$	fc2
	dropout = 0.05	dense
		{k,q,v}_proj

7.2 Training

After configuring Model with QLoRa adaptors and preparing the dataset, model is ready to be trained. As mentioned previously, we used different environments for each model, so some of the training arguments such as batch size, vary based on the computational power. Despite that, the training algorithm applied was Supervised Fine-tuning Using Transformer Reinforcement Learning library from Huggingfaces[22]. Table 7.3 denote the training arguments used in each model's training process.

Table 7.3 The hyper-parameters used to finetune each model

Hyperparameters	CodeGen-350m	Mistral-7b	Phi-2
Learning Rate	$2e - 4$	$2e - 5$	$2e - 5$
Scheduler Type	Constant	Constant	Cosine
Optimizer	NLLLoss	AdamW	AdamW
Steps	2000	200	500
Batch Size	4	1	8
Gradient accumulation	-	4	4
epochs	1	1	1

Loss function The Negative Log-likelihood loss function (NLLLoss) [23] was the default method to calculate loss in our setup. For LLM Finetuning, there's no specification of which loss function should be used, but the common one during QLoRA finetuning is paged version of AdamW method[24]. Since we only applied Quantization on the Mistral and Phi-2 models, we chose to use 32-bit paged AdamW optimizer for that and don't change smaller model's optimization function.

CodeGen Training's Convergence Before training the model, we tried out different hyperparameters for CodeGen's model, as the training library was relatively new for us, we couldn't resume the training after interruption or use the checkpoints properly, it resulted into three different runs that are demonstrated in figure 7.1. After the second run, we were mostly happy we decided to let it train on approximately 40% of the dataset (8000 samples with batch size 4, making up 2000 steps) But we noticed convergence after 180th step, what we learned from that run was that it would be better if we used a linear or cosine learning rate scheduler, since the learning rate was fairly large. Additionally, handling checkpoints and re-entrance properly, would be beneficial, since that after 180th step, it looked like the model was ready but interrupting the process would result in losing all the process, meaning that we had to wait until 2000th step to be able to save the final model successfully. After that run, while achieving a good quality output, we weren't happy with the events during training, Thus, we focused on implementing a proper training system for the next trainings.

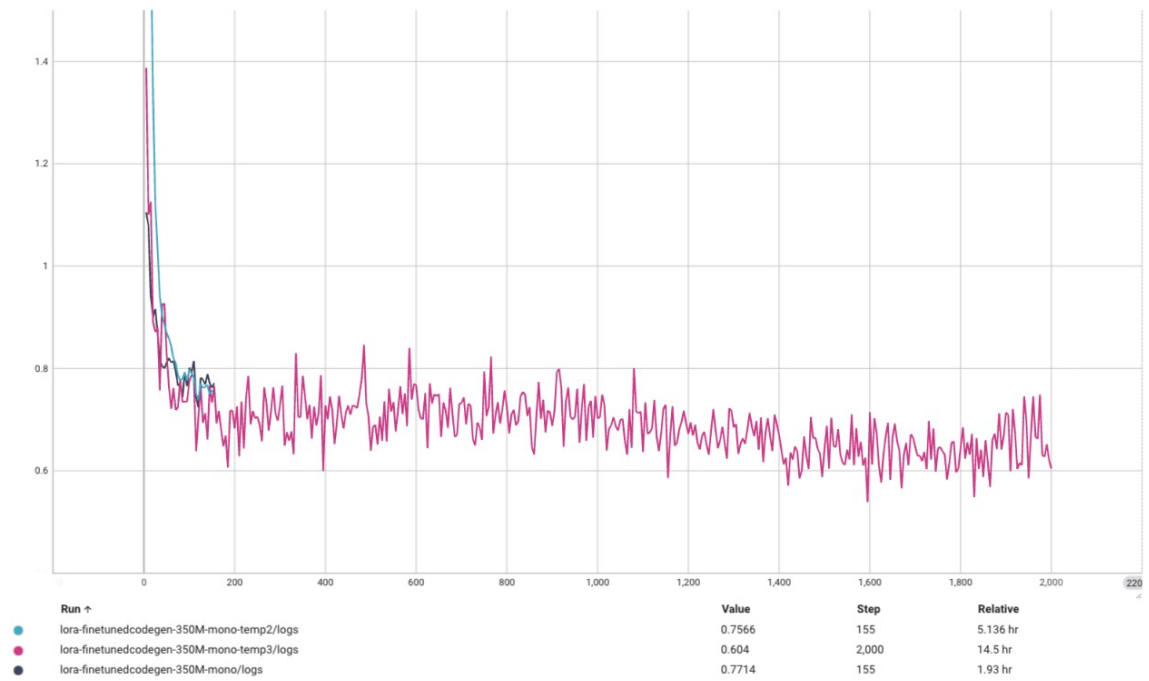


Figure 7.1 CodeGen-350m's training loss and two trial runs

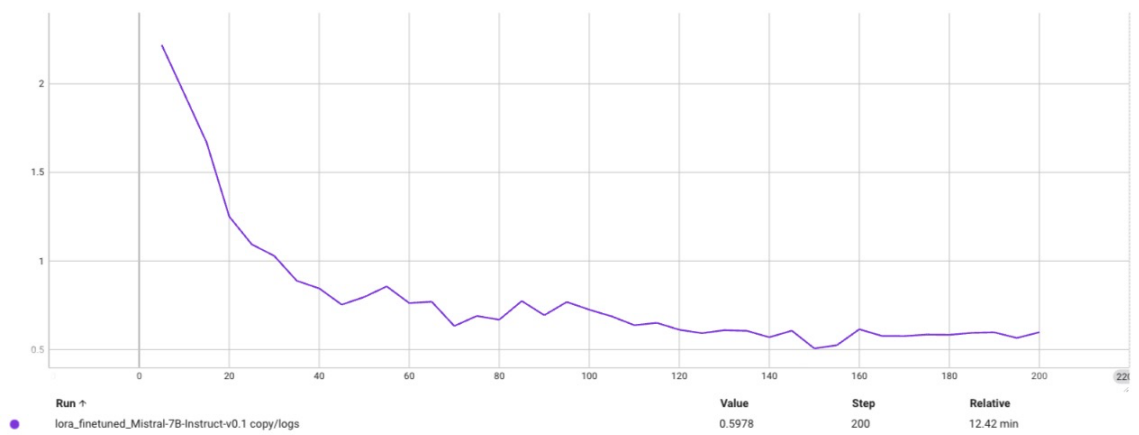


Figure 7.2 Mistral-7b's training loss

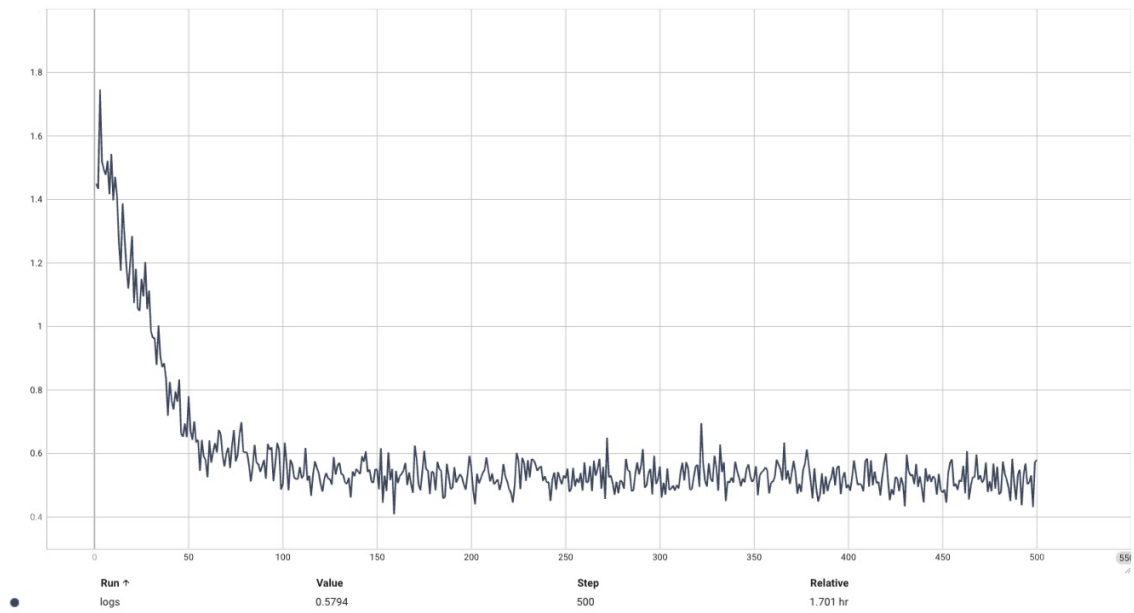


Figure 7.3 Phi-2's training loss

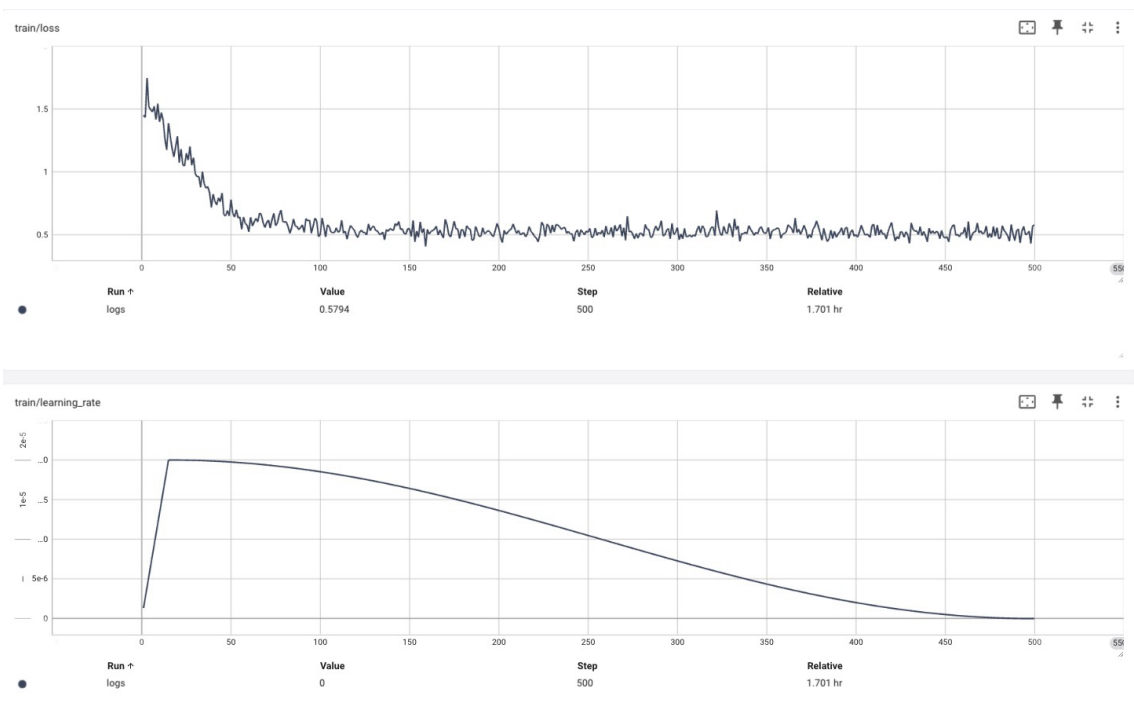


Figure 7.4 Phi-2's training loss next to the learning rate's value

8.1 HumanEval Evaluation

The original HumanEval test set, was proposed to assess completion models. The input template of each task is as follows:

HumanEval Task format:

Template:

```
{{imports}}
def {{function_signature}}:
    """
    {{doc_string}}
    """
```

Example: HumanEval/117

```
def select_words(s, n):
    """Given a string s and a natural number n, you have been tasked to implement
    a function that returns a list of all words from string s that contain exactly
    n consonants, in order these words appear in the string s.
    If the string s is empty then the function should return an empty list.
    Note: you may assume the input string contains only letters and spaces.
    Examples:
    select_words("Mary had a little lamb", 4) ==> ["little"]
    select_words("Mary had a little lamb", 3) ==> ["Mary", "lamb"]
    select_words("simple white space", 2) ==> []
    select_words("Hello world", 4) ==> ["world"]
    select_words("Uncle sam", 3) ==> ["Uncle"]
    """
```

Figure 8.1 Prompt of Task HumanEval/117

By feeding the prompt to the completion model, which works by predicting next token in each step, the prompt remains unchanged and thus, the prompt already is runnable in python. However, feeding that to a chat-based model with natural

language prompts, doesn't guarantee that the imports and function signature would be the same, resulting in a redundant code that is not even callable by the test cases.

The main challenge that we solved in this chapter, was to come up with a solution to be able to generate a code that is properly implemented in a way that the original function signature and required imports are generated by the instruction model.

First, the prompt is processed to separate imports, function signature and problem description into different variables, then the instruction is written as follows:

Instruction for HumanEval Task format:

Template: Alpaca

Prompt: Below is an instruction that describes a programming task. Write a response code that appropriately completes the request.

Please complete the following Python code without providing any additional tasks such as testing or explanations.

Instruction: Write a Python function '{{function_signature}}' to solve the following problem:

{{description}}

Input: Start the code with '{{imports}}'

Output:

Instruction for HumanEval Task format:

Template: Mistral

Prompt: <s> [INST] Below is an instruction that describes a programming task. Write a response code that appropriately completes the request.

Please complete the following Python code without providing any additional tasks such as testing or explanations.

Write a Python function '{{function_signature}}' to solve the following problem:

{{description}}

INPUTS: Start the code with '{{imports}}'

[/INST]

By inputting this formatted prompt, the generated responses were all formatted correctly except one case. This meant that we could evaluate the model's generated code by functional correctness, without the failing cause of different function names.

But the code was still not runnable because it was wrapped around other tokens, e.g. [CODE] ... [/CODE] for the finetuned Mistral model. We applied code filtering based

on each model’s output format. and extracted the pure code for each task.

We evaluated the finetuned models in two different settings, one sample per task, and 10 samples per task. $\text{pass}@1$, $\text{pass}@10$, $\text{pass}@100$ are the suggested metrics by the paper[4]. But because of our computational limitations, we could only generate one samples per task. We tried generating 10 samples per HumanEval task using our Mistral model, but its GPU time costed us more than our expectations.

Table 8.1 demonstrates the evaluation results:

Table 8.1 The evaluation results on $\text{pass}@1$ [%]

Model	HumanEval	HumanEval+	MBPP	MBPP+
Finetuned CodeGen-350m	14.6	-	-	-
Finetuned Mistral-7b-Instruct	26.8	20.7	41.4	33.6
Finetuned Phi-2 2.7b	54.9	47.6	63.4	51.6

It is important to mention that since the original base models were completion-based, comparing the results directly to them seems unfair, because evaluation of the chat-based models on a testset designed for completion is already in a disadvantage. Therefore, a new set of utility functions had to be implemented to sanitize the generated samples.

First we needed to filter out the natural language part of the generated response and filter out the code inside the code block indicators, the filtering method varies based on the model. Then we needed to correct the indentation of the generated code, since Python is a sensitive language on indentation unlike other languages.

Additionally, After analysing the evaluation results data for each task, we found that 13 out of 399 tasks in evaluation process of Mistral model and 15 out of 399 tasks for Phi-2 model on MBPP problems, were failed because the name of the function was generated differently and the test suite couldn’t call the function as its entry point. It is because MBPP prompts don’t specify the function signature in the prompt. Thus, we developed scripts to post-process the samples and rename the incorrect function names, and ran the evaluation again, focusing on the functional correctness and eliminating other failure points. The performance went from 38.8% to 41.4% on MBPP and 31.8% to 33.6% on MBPP+ for Mistral model; from 60.9% to 63.4% on MBPP and 49.6% to 51.6% on MBPP+ for Phi-2 model’s $\text{pass}@1$ results on the benchmarks. In summary, the sanitizing process increased the benchmark results around 2.2%.

In conclusion, except the mistral model which under-performed after finetuning,

CodeGen model achieved 1.8% better score on HumanEval pass@1 than the original score reported on its paper [5]. As for Phi-2, the reported score of the base Phi-2 model on MBPP for pass@1 is 59.1%, and our model got the score of 63.4% resulting it to be our best performing model.

To address the problem that has been stated in the abstraction section, we developed a code generator LLM to help developers code fast. Since it would be impossible to train a model from scratch, widely used successful models such as CodeGEN, Mistral and CodeLlama has been used. These models were fine-tuned with one of the PEFT techniques called LoRA(Low Rank Adaptation).

CodeGen-350m model was LoRA finetuned with 8000 instruction-code pair samples, and achieved pass@1 accuracy of 14.6% on the HumanEval test-set.

Mistral-7B-Instruct-v0.2 was Quantized in 4-bits format and QLoRA finetuned with only 200 data samples, and achieved pass@1 scores of 26.8% and 20.7% on the HumanEval, HumanEval+ test-sets. It also achieved scores of 38.8% and 31.8% on the MBPP, MBPP+ test-sets.

Phi-2 2.7b was Quantized in 4-bits format and QLoRA finetuned and achieved pass@1 scores of 54.9% and 47.6% on the HumanEval, HumanEval+ test-sets respectively. It also achieved scores of 60.9 % and 49.6 % on the MBPP, MBPP+ test-sets respectively.

Possible room for improvement we thought could be using better computational resources and bigger models. Maybe with better computational resources than we had, more computationally demanding models with 40 or 70 billion parameters would be able to achieve much higher scores since the models we used are fairly small to have reasoning and cognition in their embedding. Also it is known that training with more data often leads to better results so maybe with better and more data, results might get better.

References

- [1] A. Vaswani *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL].
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, cite arxiv:1810.04805Comment: 13 pages, 2018. [Online]. Available: <http://arxiv.org/abs/1810.04805>.
- [3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:160025533>.
- [4] M. Chen *et al.*, *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374 [cs.LG].
- [5] E. Nijkamp *et al.*, *Codegen: An open large language model for code with multi-turn program synthesis*, 2023. arXiv: 2203.13474 [cs.LG].
- [6] A. Khan, “Zeng et al 2022,” *Forests*, vol. 13, Dec. 2022. DOI: 10.3390/f13122168.
- [7] Q. Zheng *et al.*, *Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x*, 2023. arXiv: 2303.17568 [cs.LG].
- [8] L. B. Allal *et al.*, *Santacoder: Don’t reach for the stars!* 2023. arXiv: 2301.03988 [cs.SE].
- [9] M. Bavarian *et al.*, *Efficient training of language models to fill in the middle*, 2022. arXiv: 2207.14255 [cs.CL].
- [10] N. Shazeer, *Fast transformer decoding: One write-head is all you need*, 2019. arXiv: 1911.02150 [cs.NE].
- [11] D. Kocetkov *et al.*, *The stack: 3 tb of permissively licensed source code*, 2022. arXiv: 2211.15533 [cs.CL].
- [12] R. Sennrich, B. Haddow, and A. Birch, *Neural machine translation of rare words with subword units*, 2016. arXiv: 1508.07909 [cs.CL].
- [13] B. Rozière *et al.*, *Code llama: Open foundation models for code*, 2023. arXiv: 2308.12950 [cs.CL].
- [14] *Mbpp*, <https://github.com/google-research/google-research/tree/master/mbpp>, Accessed: 2023-11-25.
- [15] A. Q. Jiang *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].
- [16] S. Gunasekar *et al.*, *Textbooks are all you need*, 2023. arXiv: 2306.11644 [cs.CL].
- [17] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

- [18] E. J. Hu *et al.*, *Lora: Low-rank adaptation of large language models*, 2021. arXiv: 2106.09685 [cs.CL].
- [19] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation,” in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: <https://openreview.net/forum?id=1qvx610Cu7>.
- [20] S. Raschka, *Practical tips for finetuning llms using lora (low-rank adaptation)*, Nov. 2023. [Online]. Available: <https://magazine.sebastianraschka.com/p/practical-tips-for-finetuning-llms>.
- [21] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, *Qlora: Efficient fine-tuning of quantized llms*, 2023. arXiv: 2305.14314 [cs.LG].
- [22] Huggingfaces, *Trl documentation*. [Online]. Available: https://huggingface.co/docs/trl/%20-%20https://huggingface.co/docs/trl/sft_trainer.
- [23] Pytorch, *Nllloss*. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html>.
- [24] I. Loshchilov and F. Hutter, *Decoupled weight decay regularization*, 2019. arXiv: 1711.05101 [cs.LG].

Curriculum Vitae

FIRST MEMBER

Name-Surname: Parsa Kazerooni

Birthdate and Place of Birth: 22.06.2001, Tahrán

E-mail: parsa.kazerooni@std.yildiz.edu.tr

Phone: 0531 400 83 10

Practical Training: No experience yet

SECOND MEMBER

Name-Surname: Muhammed Hakan Kılıç

Birthdate and Place of Birth: 15.10.2000, Mersin

E-mail: hakan.kilic1@std.yildiz.edu.tr

Phone: 0535 300 65 67

Practical Training:

Garanti Technology Cyber security intern

YOMLAB summer intern

TUBİTAK part-time developer

Project System Informations

System and Software: Linux Operating System, CoLab environment, Python

Required RAM: 16GB VRAM

Required Disk: 20GB