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

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

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

LIST OF FIGURES

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 the scope of this project, three different open-source LLM's were fine-tuned to generate code for a given prompt or improve their accuracy on the specific task of code generation. Fine-tuning is the process of 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. For that reason, fine-tuned models on code datasets outperform base models (e.g., GPT, BART, etc.), which haven't been trained on code datasets specifically. There are three generic ways which can be used to fine-tune a model: self-supervised, supervised, and reinforcement learning. Supervised fine-tuning is a method where the LLM is trained on a labeled dataset, meaning that it is provided with input-output pairs for a specific task. Self-supervised learning is a method where the model is fine-tuned without the need for explicit labels or external supervision. Instead, it leverages inherent structure and patterns within the data to learn meaningful representations. Reinforcement learning fine-tuning 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 our problem relies on an instruction-based conversation with the

model, our model needs to perform a sequence-to-sequence inference, which contains a high-quality prompt with clear instruction, possible but not compulsory inputs, and formatting hint to generate the response code. To perform 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 mentioned prompt style and fed to the training algorithm.

Our goal was to produce a model that answers to coding related problems accurately by fine-tuning other LLMs and present this model to the end user via 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.

In the section number 2, we presented the related works that has been done on this subject. The section after that contains our feasibility work that we have done before starting on this project. Section number 4 and 5 are dedicated to system analysis and design respectively. Last, we demonstrated our program in the section number 6.

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 GPTLike 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
Feasibility

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 it's parameters. To meet this requirements, a CPU and disk storage is needed. A computer which meets these stated requirements should be as follow:

- 15 GB RAM
- CPU with 3Ghz+ frequency
- 128 GB SSD
- Motherboard
- LCD Monitor

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 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.

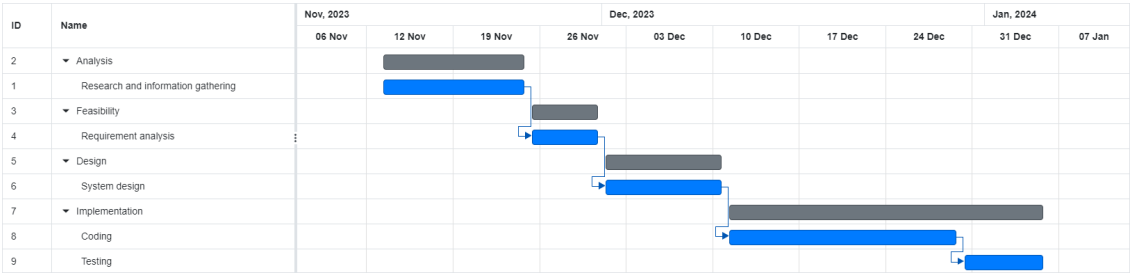

Figure 3.1 Workforce and Time Planning of The Project

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) is needed to meet the hardware requirements for the project. 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.

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 used to measure the success of the model. The model was given certain test-cases and its response to those were compared to the given output in other words the ground truth. The improvement on the scoring results compared to the score that the base models achieved, is sufficient for the project to be considered successful.

5
System 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

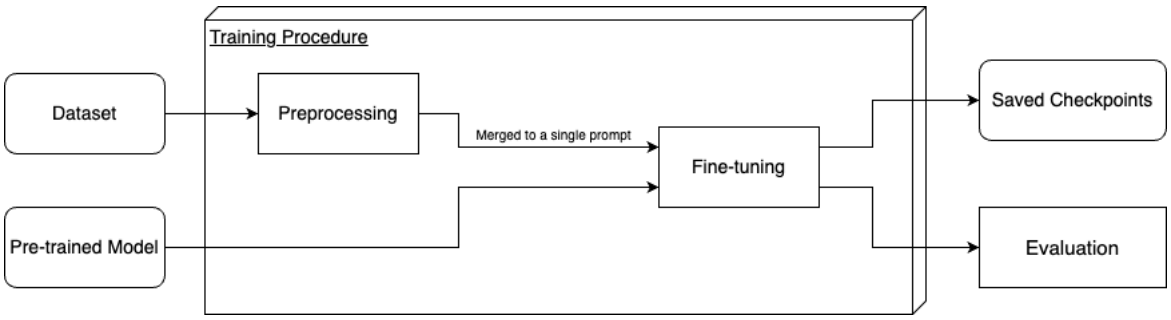

Figure 5.1 Block Diagram of The System

5.1 Dataset and Preprocessing The Data

The dataset was gathered from the web from various sources such as Github, Leetcode and datasets like CodeContest [15], Stack [11] etc. Every instance of the dataset was put in Alpaca style format [16] which goes like this:

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:

5.2 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:

- Falcon-7b [17]
- Mistral-7b [18]
- Code Llama-7b [13]

These models served as our initial checkpoints, forming the foundation upon which we built our fine-tuned models. The choice of using the smaller seven billion parameters versions of the models was based on our computational resources limit and budget. Aditionally, it is notable to mention that even though Code Llama model is already a fine-tuned version of Llama 2 [19] but it can be trained further to achieve a higher quality code.

5.3 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, backpropagation is applied, and by using optimizing algorithms such as the Adam optimizer [20], we achieve slightly better parameters and start the loop over with the new parameters.

5.3.1 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 [21] 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.

5.4 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 human assessors who interact with the model-generated code outputs. This benchmark allows for the qualitative assessment of the model's code generation in terms of correctness, readability, and syntactic and semantic accuracy [4].

Additionally, 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 both HumanEval and MBPP when evaluating our fine-tuned model.

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

6
Implementation

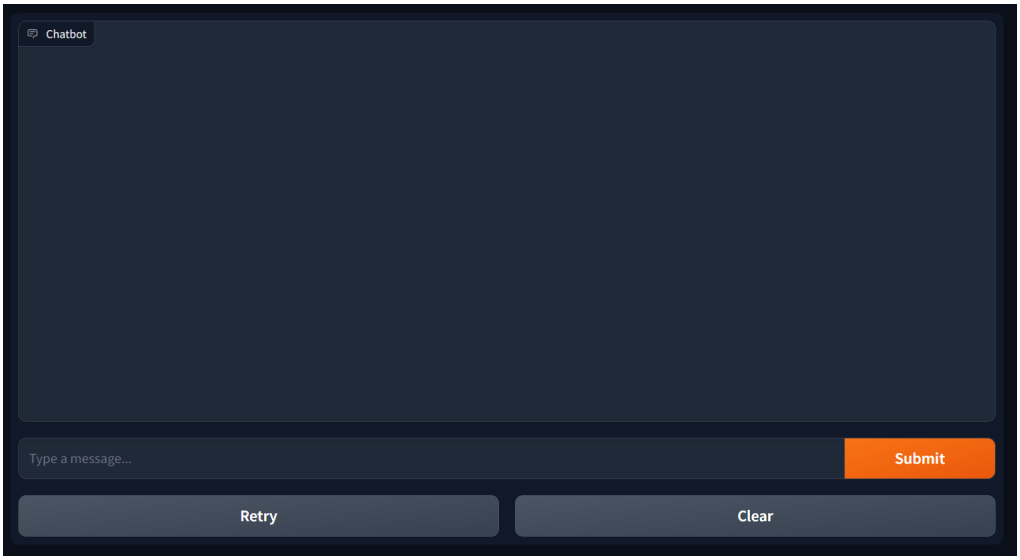

Figure 6.1 User Interface

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] Y. Li *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. DOI: 10.1126/science.abq1158. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158.

[16] R. Taori *et al.*, *Stanford alpaca: An instruction-following llama model*, https://github.com/tatsu-lab/stanford_alpaca, 2023.

[17] E. Almazrouei *et al.*, "Falcon-40B: An open large language model with state-of-the-art performance," 2023.

[18] A. Q. Jiang *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].

[19] H. Touvron *et al.*, *Llama 2: Open foundation and fine-tuned chat models*, 2023. arXiv: 2307.09288 [cs.CL].

[20] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[21] E. J. Hu *et al.*, *Lora: Low-rank adaptation of large language models*, 2021. arXiv: 2106.09685 [cs.CL].