Python Code Generation using Large Language Models

NLP with Deep Learning Project Report

Sashank Talakola
Jagath Kumar Reddy Katama Reddy
Junaid Ahmed Mohammed

Abstract

This project focuses on enhancing programmer productivity through Python code generation using lightweight Large Language Models (LLMs). The primary objective is to develop compact LLMs capable of running locally on personal hardware, thereby addressing privacy concerns associated with cloud-based solutions. Two approaches were explored: (1) building models from scratch via pretraining and subsequent fine-tuning, and (2) fine-tuning existing pre-trained models. A range of architectures, including GPT-2, Gemma 3, LLaMA 3.2, and Qwen 2.5 series, were evaluated for their effectiveness in code generation tasks. Experimental results indicate that the LLaMA 3.2 3B model consistently outperformed other candidates. This work demonstrates the viability of small, local LLMs as practical coding assistants, paving the way for privacy-preserving AI tools in software development.

1 Introduction

The advent of Large Language Models (LLMs) has significantly transformed the landscape of software development, particularly in the realm of automated code generation. These models, trained on vast corpora of natural language and source code, possess the capability to translate human-readable prompts into syntactically correct and semantically meaningful code snippets. This evolution has not only streamlined coding processes but also democratized programming by lowering the entry barrier for non-experts. LLMs like OpenAI's Codex, which powers GitHub Copilot, exemplify the practical applications of these models in real-world coding environments. Codex has demonstrated proficiency in generating code across multiple programming languages, with a notable emphasis on Python. Such tools have been instrumental in enhancing developer productivity by automating routine coding tasks and providing intelligent code suggestions.

Recent surveys have explored the capabilities, challenges, and future directions of LLMs in code generation. For instance, Jiang et al.[9] provide a comprehensive taxonomy of LLMs tailored for code generation, discussing aspects like data curation, performance evaluation, and ethical considerations. Similarly, Huynh et al.[8] explore the challenges and techniques associated with fine-tuning LLMs for code generation tasks, emphasizing the importance of model adaptability and evaluation metrics. Despite these advancements, several challenges persist. Ensuring the semantic correctness of generated code and maintaining code security are areas that require ongoing research and development.. Moreover, the integration of LLMs into existing development workflows necessitates careful consideration of user experience and trust in AI-generated code.

In this project, we explore the development and fine-tuning of compact LLMs for Python code generation, aiming to create models that can operate efficiently on personal hardware. By focusing on smaller models, we address privacy concerns and reduce computational overhead, making AI-assisted coding more accessible. Our experiments involve training models from scratch and fine-tuning pre-trained models, including GPT-2, Gemma 3, LLaMA 3.2, and Qwen 2.5 series. Among these, the LLaMA 3.2 3B model exhibited superior performance, balancing efficiency and accuracy.

```
Split the dataframe into features and target variable.
```

Listing 1: User Prompt

```
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    'age': [25, 32, 47],
    'salary': [50000, 60000, 80000],
    'purchased': [0, 1, 1]  # Target variable
})

# Define target column
target_column = 'purchased'

# Split into features and target
X = df.drop(target_column, axis=1)
y = df[target_column]
```

Listing 2: ChatGPT Output

```
target_variable = 'purchased'

X = df.drop(target_variable, axis=1)
y = df[target_variable]
```

Listing 3: GitHub Copilot Output

Although both ChatGPT and Copilot produced semantically correct and equivalent outputs for the given query, the code generated by ChatGPT often requires some manual editing before it can be used. In contrast, Copilot tends to generate cleaner code that aligns more strictly with the user's request, requiring less post-processing. For instance, ChatGPT may include unnecessary elements such as import statements or redundant dataframe declarations, which the user must manually remove, thereby ultimately reducing developer productivity.

For use cases like this, where concise code snippets are preferred, smaller LLMs could be a better fit. Their limited context length naturally encourages more focused and lightweight code generation, making them well-suited for such scenarios, which is exactly why, for this project, we have chosen to work with small LLMs.

2 Related Work

The advent of LLMs has significantly advanced the field of code generation. Models such as OpenAI's Codex, which is based on GPT-3 and trained on a vast corpus of code, have demonstrated the ability to translate natural language prompts into executable code across various programming languages. These models leverage the transformer architecture to capture complex patterns in code syntax and semantics, enabling them to generate syntactically correct and functionally relevant code snippets.

Recent surveys[8][9] have categorized the developments in LLMs for code generation, discussing aspects such as data curation, performance evaluation, and real-world applications. These surveys highlight the rapid progress in the field and the increasing capabilities of LLMs to assist in software development tasks.

2.1 Language-Independent Code Generation

While many LLMs are trained on code from multiple programming languages, achieving truly language-independent code generation remains a challenge. Efforts have been made to create models that can generalize across languages, enabling code translation and generation in less-represented languages. For instance, Giagnorio et al.[4] have investigated the effectiveness of various approaches for boosting LLMs' performance on low-resource languages, including fine-tuning and in-context learning.

Additionally, benchmarks like HumanEval and MBPP have been used to assess LLMs' ability to generate code across different programming tasks. These benchmarks help in evaluating the generalization capabilities of LLMs and their proficiency in handling diverse coding problems.

2.2 Python Code Generation

Python, being one of the most popular programming languages, has been a primary focus for code generation research. Models like Codex have shown strong performance in generating Python code, often outperforming traditional code generation methods. In a research study by Levin et al.[11] have introduced frameworks such as Pythoness, which combines LLMs with test-driven development to generate Python code that passes specified tests.

Furthermore, research by Wang et al.[18] has explored the use of symbolic execution in Python code generation. For example, LLM-Sym is an agent that leverages LLMs to translate complex Python path constraints into code, enhancing the capability of symbolic execution engines

2.3 Model Architectures and Fine-Tuning Techniques

The architecture of LLMs and the techniques used for fine-tuning them play a crucial role in their code generation capabilities. Transformer-based architectures have been the foundation for many successful models. In the research work by Weyssow et al.[21] have explored Fine-tuning techniques, such as parameter-efficient fine-tuning (PEFT)[6], have been explored to adapt LLMs to specific code generation tasks without extensive computational resources.

Further, research wor by Tsai et al.[17] have examined data pruning methods to enhance the efficiency of model training for code generation. These methods aim to reduce the training data size while maintaining or improving model performance, making the training process more efficient.

2.4 Evaluation Frameworks

Traditional evaluation metrics like BLEU, initially developed for natural language generation tasks, are often inadequate for code generation because they primarily rely on surface-level token matching and fail to capture syntactic or semantic correctness. To address these shortcomings, specialized metrics such as CodeBLEU[15] have been introduced. CodeBLEU extends BLEU by incorporating code-specific structures, including syntax trees and semantic data flows, allowing for a more holistic assessment of generated code. This approach results in scores that better align with human judgments of code functionality and quality.

Further improvements have been proposed with CrystalBLEU[3], which refines the evaluation by accounting for trivially shared n-grams common across codebases. By removing these shared elements before computing similarity, CrystalBLEU provides a sharper distinction between genuinely similar and dissimilar code snippets, avoiding the inflation of scores seen with traditional BLEU. Studies have shown that CrystalBLEU can differentiate examples 1.9–4.5 times more effectively than BLEU, making it a valuable tool for evaluating modern code generation systems. These advanced metrics, by considering deeper structural properties of code, represent a significant step forward in accurately benchmarking the capabilities of LLMs for programming tasks.

3 Approach

We have experimented with various model ranging from 124M parameters to 2Billion parameters models. Following sections discuss more on these,

3.1 GPT-2 Pretraining & Finetuning

Our approach leverages continual pretraining of the GPT-2 124M model rather than training from scratch. This decision was informed by the work of Gururangan et al. (2020)[5], who demonstrated that domain-adaptive pretraining on target domain text before task-specific fine-tuning consistently improves performance. Their findings showed that continued pretraining on domain-specific corpora yields substantial gains in performance even when starting from already-trained language models. Continual pretraining aligned ideally with our use case for several reasons. First, it allowed us to build upon the foundational knowledge already encoded in GPT-2's weights while significantly reducing computational requirements compared to

training from scratch. As noted by Zhao et al. (2023)[22], continual pretraining can achieve comparable or superior results to scratch training while requiring only 10-20% of the computational resources.

The primary goal of our continual pretraining phase was to increase the model's exposure to Python code corpus, thereby enhancing its understanding of Python syntax, semantics, and common programming patterns. By pretraining on the CodeParrot dataset, which contains 50GB of cleaned Python code, we enabled the model to develop stronger representations of Python-specific constructs such as indentation rules, function definitions, class structures, and library usage patterns. This domain-specific knowledge is critical for generating syntactically valid and semantically meaningful code. The continual pretraining process involved adjusting the learning rate schedule to be appropriate for continued training rather than from-scratch training. Following recommendations from Howard and Ruder (2018)[7],

Upon completion of continual pretraining, we proceeded with fine-tuning the model on a Python instruction dataset containing 18K examples derived from Alpaca. Fine-tuning was essential to bridge the gap between the model's enhanced Python language understanding and its ability to follow specific instructions and generate contextually appropriate code solutions. During this phase, we applied instruction-tuning techniques similar to those described by Wei et al. (2022)[19] in their work on "Chain-of-Thought Prompting," adapting the methodology for code generation rather than reasoning tasks. The fine-tuning process incorporated specialized loss functions that placed greater emphasis on syntactic correctness, prioritizing the generation of compilable code over merely plausible text. Additionally, we employed gradient accumulation techniques to simulate larger batch sizes despite hardware constraints, following the methodology outlined by Ott et al. (2019)[13].

3.2 Other Models Finetuning

During the pre-training phase of the project, the results from the GPT-2 model did not meet our expectations. While the model demonstrated a general understanding of Python programming, its performance on instruction-specific generation was subpar. Although the initial few lines of generated code were acceptable, the quality significantly deteriorated after a couple of lines. We identified that this issue stemmed from the limited context length of 1,024 tokens, which restricted the model's ability to generate longer code sequences exceeding 10 lines. Recognizing the need for a larger context window, we decided to transition to models that offer greater out-of-the-box context lengths, such as Gemma 3, Llama 3.2, and Qwen 2.5 series. For these models, we have set the context size to 2,048 tokens.

Table 1 provides information about the Model Name and No.of parameters in the model.

Model Name	**Parameters**
Gemma3	1B
Llama3.2	1B
Llama3.2	3B
Qwen2.5	1.5B

Table 1: Model names and their parameter sizes

Since these models were pre-trained on much more diverse datasets and had significantly larger parameter counts, we anticipated that they would perform better at code generation. Indeed, their out-of-the-box code generation capabilities were already noticeably superior to the GPT-2 model we had trained. However, given the much larger number of parameters, it was impractical to train these models using FP16 precision as we had with GPT-2. This led us to explore quantization techniques and parameter-efficient fine-tuning (PEFT) methods to reduce memory usage.

To address this, we chose the Unsloth fine-tuning framework, which provided quantized versions of the models and built-in support for LoRA — an essential feature for our PEFT approach. Specifically, we loaded the models in 4-bit precision, significantly reducing the memory footprint while maintaining model performance. Additionally, since each model required a custom prompt format, Unsloth's `get_chat_template` utility enabled us to build tailored templates for each one. Ultimately, we were able to fine-tune the models efficiently, typically requiring no more than 8GB of VRAM, making it feasible to train each model on a single GPU.

3.3 Evaluation Methodology

To evaluate the models, we primarily relied on cross-entropy loss during the pre-training phase. The GPT-2 model exhibited a generally decreasing loss curve, indicating consistent improvement. Since there was no ground truth available, we were unable to compute metrics like CodeBLEU. Although we attempted to calculate perplexity, the diversity of the generated code led to unstable results — in some epochs, perplexity values even became NaN — making it an unreliable metric for assessing generation quality.

Similarly, during the fine-tuning phase, we continued to use cross-entropy loss as the primary evaluation metric. As before, perplexity often resulted in NaN values, and CodeBLEU scores were inconsistent due to the diversity of generated outputs. For example, when the fine-tuned dataset used list comprehensions to sum elements, the model often generated equivalent code using traditional for-loops. Although functionally correct, such differences led to low CodeBLEU scores, highlighting its unreliability for our evaluation purposes. Initially, we assumed that fine-tuning would cause the model outputs to align more closely with the dataset; however, this did not occur consistently.

Additionally, we experimented with using BERT embeddings to compute similarity between generated code and ground truth. This method worked reasonably well for short code snippets but became unreliable for longer sequences, where embeddings for different outputs were often too similar. As a result, we had to abandon this approach as well. We also explored other evaluation frameworks, such as CodeJudge[16] and EvalPlus[12], but could not find suitable resources to effectively use them for model evaluation.

Therefore, the evaluation of our fine-tuned models ultimately had to rely on cross-entropy loss as the primary metric.

4 Data

4.1 Pre-Training Dataset

For the pretraining phase of the project, we initially opted to use the CodeParrot GitHub Dataset available on Hugging Face. This dataset is built from the public GitHub dataset hosted on Google BigQuery and comprises approximately 115 million code files across 32 programming languages, with around 60 different file extensions, totaling roughly 1TB of data. Each instance in the dataset contains the following attributes: `[code, repo_name, path, language, license, size]`. Since our project focused exclusively on Python, we filtered the dataset to include only instances where the `language` attribute was set to Python. This filtering resulted in a subset of about 7.2 million Python files. However, given the scale of this dataset and the limited computational resources available, it was not feasible to use the entire corpus. Therefore, we worked with a smaller chunk of the data.

Midway through the project, we encountered unexpected issues accessing the original dataset, specifically a "This dataset has 7 files scanned as unsafe" warning from Hugging Face, which rendered it temporarily unavailable. To address this, we transitioned to the CodeParrot Github Clean Dataset — a cleaned and safer version of the original. An additional benefit of the cleaned dataset was that it was already filtered for Python code, streamlining our preprocessing efforts. Moreover, this version included a `size` attribute representing the number of characters (not tokens) in each code sample. Although not a perfect measure, it allowed us to efficiently subset the data to smaller, more manageable pieces. After subsetting, we curated a final dataset containing approximately 200,000 Python code files.

Each instance of the dataset has the following structure -

Field	**Value**
repo_name	jdemel/gnuradio
path	gr-utils/modtool/templates/gr-newmod/docs/doxy...
size	1295

content	`from autobahn.asyncio.websocket import WebSocketClientProtocol, \`
	`  WebSocketClientFactory`
	
	`import asyncio`
	
	`class MyClientProtocol(WebSocketClientProtocol):`
	
	`  def onConnect(self, response):`
	`  print("Server connected: {0}".format(response.peer))`
	`  ...`

During pretraining, we used only the `content` attribute from the dataset. Although attributes such as `repo_name` and `path` could provide valuable additional context for more efficient and accurate code generation, they were not utilized. Incorporating this information would have required significant additional preprocessing to ensure these values were consistently present within the LLM's context window, which proved to be a complex and tedious task. Therefore, for practicality and efficiency, we chose to work solely with the `content` attribute.

4.2 Fine-Tuning Dataset

For the fine-tuning phase of the project, we selected the Python Code Instruction Dataset. This dataset is a filtered subset of Code Instructions Dataset, which contains instruction-code pairs across multiple programming languages. In the filtered version, only Python-specific instruction and code pairs were retained. The dataset includes the following attributes: `instruction`, `input`, `output`, and `prompt`.

Since the dataset comprised only around 18,000 rows, we were able to use the entire set for fine-tuning without significant computational burden. For our fine-tuning task, we ignored the `input` attribute, as it provided little to no additional useful information in most cases. Instead, we constructed prompts using only the `instruction` and `output` columns. Although the dataset provided a preformatted `prompt` field compatible with the Alpaca format—originally designed for the LLaMA series of models—we crafted custom prompt templates tailored to the specific models we fine-tuned. Additionally, because the original prompt incorporated the `input` attribute (which we did not use), we chose not to rely on the provided prompt and instead generated our own.

Each instance of the dataset has the following structure -

Field	**Value**
Instruction	Create a function to calculate the sum of a sequence of numbers.
Input	[1, 2, 3, 4, 5]

Output	`# Python code`
	`def sum_sequence(sequence):`
	`  return sum(sequence)`

Prompt	Below is an instruction that describes a task.

5 Experiments

5.1 GPT-2

5.1.1 Continual Pretraining

For the GPT-2 model, we conducted a series of experiments focused on continual pretraining followed by fine-tuning. Our implementation leveraged specific hyperparameters optimized for code generation tasks:

Training Configuration: We used the following key hyperparameters for GPT-2 continual pretraining:

Parameter	**Value**
Base model	GPT-2 (124M parameters)
Sequence length	1024 tokens
Training batch size	2 per device, gradient accumulation steps of 16 (effective batch size 32)
Learning rate	2e-4
Weight decay	0.1
Learning rate scheduler	Cosine scheduler with 750 warmup steps (approx. 1% of training data)
Maximum training steps	51,200

Table 4: Training Configuration

Learning Rate Selection: The learning rate of 2e-4 was selected based on scaling laws established in DeepMind's research, adjusting the standard 3e-4 rate by a factor of (47.5/40). Additionally, we considered that smaller models can generally accommodate higher learning rates without divergence issues. The VRAM constraints of our A16 GPUs also influenced this decision, as it balanced optimization speed with memory limitations.

Memory Optimization: We implemented gradient checkpointing to reduce memory footprint, allowing for longer sequence lengths while maintaining reasonable batch sizes. As noted by Chen et al. (2021)[1], gradient checkpointing trades compute for memory by recomputing activations during the backward pass instead of storing them, which was crucial for training with our 1024-token sequences.

Data Processing: We processed the CodeParrot dataset into manageable shards, implementing a shuffle buffer of 10,000 examples to ensure adequate stochasticity in training while maintaining memory efficiency. This approach follows best practices established by Raffel et al. (2020)[14] in their work on T5 models.

Validation Strategy: Regular evaluation was performed every 1024 forward passes using a held-out validation set, with a maximum of 512 evaluation steps to balance assessment accuracy with training throughput. Checkpointing Strategy: Model checkpoints were saved every 1024 forward passes, providing sufficient granularity for monitoring training progress while minimizing storage requirements.

Reproducibility: We fixed the random seed to 1 for all experiments to ensure reproducibility, following the methodology recommended by Dodge et al. (2020)[2].

Our pretraining implementation used a merged dataset in JSONL format, with each training instance formatted to expose the model to Python code structures and patterns.

5.1.2 Finetuning Phase

For the fine-tuning phase, we adapted the continually pretrained model checkpoint (saved at step 51,200) to follow instruction-based prompts using the "iamtarun/python-code-instructions-18k-alpaca dataset. The fine-tuning configuration incorporated the following parameters:

Parameter	**Value**
Pretrained model	Custom GPT-2 checkpoint from continual pretraining
Sequence length	1024 tokens
Maximum training steps	15,000

Table 5: Model Configuration

Parameter	Value
Learning rate	5e-5 (reduced from pretraining to prevent catastrophic forgetting)
Weight decay	0.1 (maintained from pretraining phase)
Learning rate scheduler	Linear scheduler (changed from cosine)
Warmup steps	None (compared to 750 in pretraining)
Per-device batch size	2
Gradient accumulation steps	8 (effective batch size of 16)
Mixed precision training	FP16 with optimization level O1

Table 6: Optimization Parameters

Due to computational infrastructure constraints, the fine-tuning process was completed for 14 epochs rather than the initially planned 100 epochs.

5.2 Reasoning Model - Qwen

In this experiment, we aimed to evaluate whether reasoning-focused models, such as the Qwen series, could outperform models like Llama and Gemma, which were not explicitly trained using chain-of-thought (CoT)[20] prompting strategies. Since programming tasks often involve complex reasoning, we initially hypothesized that reasoning-optimized models would demonstrate superior code generation capabilities compared to standard LLMs.

However, contrary to our expectations, the Qwen 2.5 1.5B model did not outperform the other models, despite two key advantages: (1) it was trained using chain-of-thought prompting, and (2) it had a larger parameter size compared to the Gemma 3 and Llama 3.2 1B models. Even with these advantages, Qwen's performance was similar to — or in some cases worse than — the other models. One possible explanation is that effective reasoning may require significantly longer context windows or larger model sizes to be properly emulated. Similar patterns have been observed in previous chain-of-thought studies, where smaller models struggled to replicate coherent reasoning processes. The cross-entropy loss comparisons for these models are detailed in the results section.

Hyperparameter	Value
per_device_train_batch_size	4
gradient_accumulation_steps	4
warmup_steps	5
num_train_epochs	10
learning_rate	2e-4
logging_steps	1
optim	adamw_8bit
weight_decay	0.01
lr_scheduler_type	linear

Table 7: Qwen2.5 Hyperparameters

5.3 Model Size

In this experiment, we compared the code generation abilities of the models based on their number of parameters. Typically, we would expect models with a larger number of parameters to perform better. However, contrary to this expectation, we observed that the Gemma3 1B model — with roughly one-third the number of parameters — outperformed all other models, including the Qwen 2.5 (1.5B parameters) and Llama3.2 (3B parameters) models. It is important to note that Gemma3 was trained for only 5 epochs, whereas the other models were trained for 10 epochs. Despite the shorter training period, the cross-entropy loss for Gemma3 was consistently lower at every stage of training compared to the other models.

Although this experiment does not provide conclusive evidence that the number of parameters is irrelevant for better code generation, it clearly suggests that other factors — such as model architecture, pre-training quality, or data diversity — play a significant role in influencing performance.

Hyperparameter	Value
per_device_train_batch_size	4
gradient_accumulation_steps	4
warmup_steps	5
num_train_epochs	5
learning_rate	2e-4
logging_steps	1
optim	adamw_8bit
weight_decay	0.01
lr_scheduler_type	linear

Table 8: Gemma3 Hyperparameters

6 Results

In this section, we present the results of our experiments, using visual plots to illustrate key findings. A more detailed analysis of these results is provided in the subsequent Analysis section

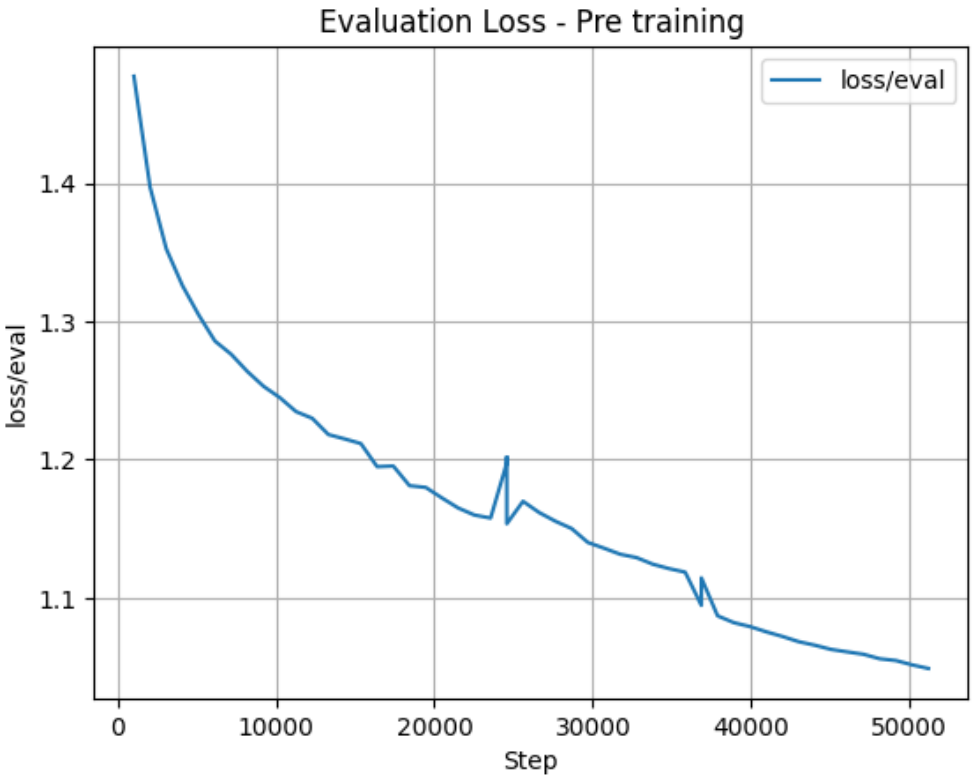

Figure 1: Evaluation cross entropy loss over steps during continual pretraining. The consistent downward trend indicates effective learning of Python semantics without catastrophic forgetting, suggesting the model remains in an underfitting regime and retains its foundational general language abilities..

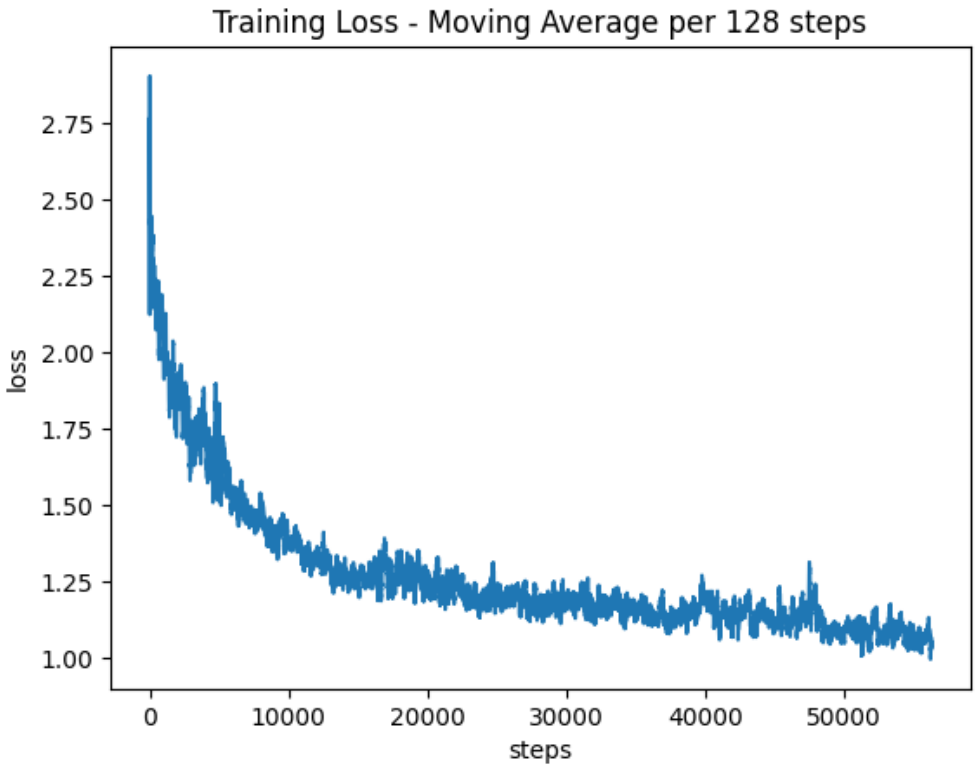

Figure 2: Moving average of training cross entropy loss over 128-steps during continual pretraining of GPT-2 on Python code. Despite fluctuations in short intervals, the overall decreasing trend indicates effective learning of code semantics, though at a slow rate, reflecting the inefficiency but success of continual pretraining.

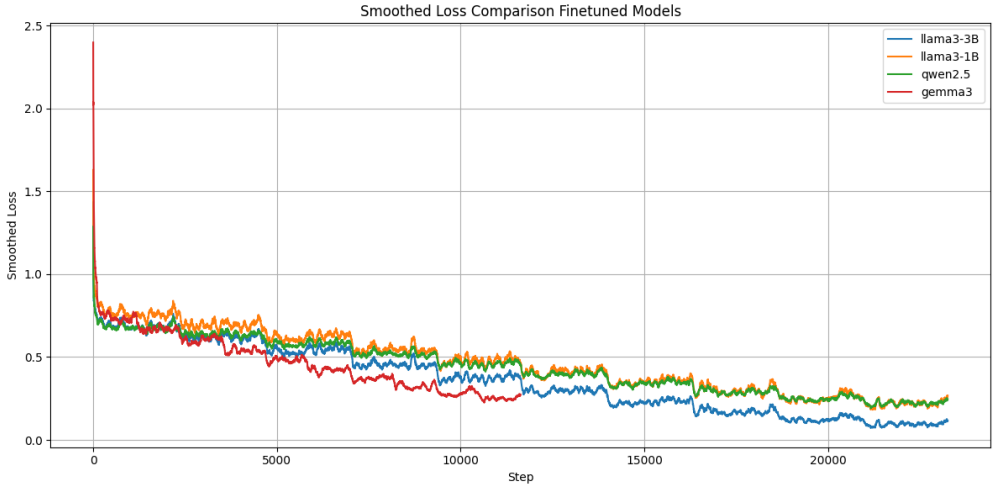

Figure 3: Comparison of large models: Training loss trajectories for Gemma 3B, LLaMA 3 1B, LLaMA 3 3B, and Qwen 2.5B during fine-tuning on Python Instruct 18K. Gemma 3B exhibited a steeper and more consistent loss decrease early on, with LLaMA 3 3B also showing stable convergence. LLaMA 3 1B and Qwen 2.5B lagged slightly behind, suggesting differences in optimization dynamics and fine-tuning efficiency across model scales.

6.1 Limited Evaluation

As detailed in the Approach section, several practical constraints impacted the evaluation process. The high diversity in generated code led to unstable perplexity values, often resulting in NaNs. Due to memory limitations, the maximum generation length was capped at 256 tokens, which occasionally caused empty or truncated code generations, further affecting the evaluation of metrics like CodeBLEU.

Despite these challenges, we relied on cross-entropy loss as the primary metric for assessing model performance, both during pre-training and fine-tuning. While this provided a stable measure of learning progress, the limitations discussed above impacted the ability to evaluate generated code quality comprehensively.

7 Analysis

Our comprehensive analysis of the experimental results revealed several key insights into the performance characteristics and practical applicability of lightweight LLMs for code generation.

The training dynamics of the GPT-2 continual pretraining phase deserve particular attention. Our training loss curves exhibited a characteristic pattern of local fluctuations superimposed on a gradually decreasing trend. This slow but steady decrease in cross-entropy loss indicates that the model was successfully learning Python code syntax and semantics, albeit at a relatively modest rate. While this gradual learning curve might appear computationally inefficient, it actually represents an important balance in the continual pretraining paradigm.

The deliberate pacing of the training loss decline suggests that the model was integrating Python-specific knowledge while preserving its general language capabilities acquired during its original pretraining. This preservation of general capabilities is crucial, as we wanted the model to retain its foundational English language understanding while gaining Python expertise. As noted by Howard and Ruder (2018), catastrophic forgetting is a common challenge in continual learning scenarios, and our gradual approach helped mitigate this risk.

Interestingly, the evaluation loss showed a more consistent downward trend than the training loss, suggesting the model was in an underfitting regime rather than approaching overfitting. This observation aligns with findings by [10], who demonstrated that smaller language models typically benefit from extended training periods before reaching their optimal performance plateau. Our training duration, while substantial, likely did not reach the model's full learning potential for the Python domain.

The underfitting observed in our evaluation metrics indicates that longer training could yield further improvements. However, we deliberately balanced domain specialization against maintaining general language capabilities. This approach is supported by Gururangan et al. (2020)[5], who noted that excessive domain-adaptive pretraining can lead to performance degradation on tasks requiring broader knowledge. Our training regime sought to optimize for both code generation capabilities and retention of general language understanding.

7.1 Cross-model comparison

We compared the training loss trajectories of the Gemma 3B, LLaMA 3 1B, LLaMA 3 3B, and Qwen 2.5B models during fine-tuning on the Python Instruct 18K dataset. Notably, the Gemma 3B model exhibited a steeper and more consistent downward trend in training loss compared to the other models during the initial training phases. This suggests that Gemma 3B may have benefited from better optimization dynamics or initial parameter configurations when adapting to our task.

The LLaMA 3 3B model also demonstrated a stable and steadily decreasing smoothed loss curve, suggesting effective fine-tuning behavior. In contrast, the LLaMA 3 1B and Qwen 2.5B models showed training loss curves that were closely aligned with each other but slightly lagged behind Gemma 3B and LLaMA 3 3B in terms of convergence rate.

These observations may be attributed to differences in pretraining scale, model architecture optimizations, or sensitivity to hyperparameter configurations during fine-tuning. Further ablation studies could clarify whether certain architectures exhibit more favorable fine-tuning characteristics for code generation tasks at moderate scales (1–3B parameters).

7.2 Llama3.2 3B Analysis

Overall, we observed that the Llama 3.2 3B model achieved the best results among the models evaluated. However, it is important to note that this comparison was made against the Gemma3 model, which was trained for only 5 epochs, unlike the others trained for longer durations. Despite this, Llama 3.2 consistently demonstrated superior performance based on cross-entropy loss and code generation quality.

At the end of this phase, we tested the final model across a variety of scenarios to assess its robustness and versatility. These tests helped us further validate its ability to handle different types of coding tasks, ranging from simple function implementations to moderately complex algorithmic problems.

7.2.1 Simple Prompt

```
Create a function to calculate the sum of a sequence of integers.
```

Listing 4: User Prompt

```
def sum_sequence(sequence):
    sum = 0
    for num in sequence:
        sum += num
    return sum


sequence = [1, 2, 3, 4]
print(sum_sequence(sequence)) # 10
```

Listing 5: Llama3.2 3B Output

In this scenario, we observed that the model generated perfect code. Given the simplicity of the query, the model was expected to perform exceptionally well on such tasks.

7.2.2 Simple Prompt with extra details

```
Create a BERT model using the huggingface module.
```

Listing 6: User Prompt

```
import numpy as np
import torch
from transformers import BertModel, BertConfig
from transformers import BertTokenizer

# Create the model
model = BertModel.from_pretrained('bert-base-uncased')

# Create the config
config = BertConfig.from_pretrained('bert-base-uncased')

# Create the tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Create the model
model = BertModel.from_pretrained('bert-base-uncased')

# Create the config
config = BertConfig.from_pretrained('bert-base-uncased')

# Create the tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

Listing 7: Llama3.2 3B Output

The model performed very well in this scenario as well. Even though the prompt explicitly required the use of the Hugging Face module, the model successfully generated a correct implementation of a BERT model. This outcome demonstrates that the model possesses a reasonable understanding of popular Python libraries and that its code generation ability extends beyond basic or generic Python programming tasks.

7.2.3 Prompt with lot of details

```
Create a BERT model using huggingface module and write the backpropagation
    code using accelerator for multi-GPU training.
```

Listing 8: User Prompt

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import BertModel, BertConfig

class BertModelWithAccelerator(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.bert = BertModel(config)
        self.out_layer = nn.Linear(config.hidden_size, config.hidden_size)

    def forward(self, x):
        out = self.bert(x)
        return self.out_layer(out)

bert_config = BertConfig.from_pretrained('bert-base-uncased')
bert_model = BertModelWithAccelerator(bert_config)

# Training
for epoch in range(10):
    train_loss = 0
    train_acc = 0
    for i, (input, target) in enumerate(train_dataloader):
        output = bert_model(input)
        loss = F.cross_entropy(output, target)
        train_loss += loss.item()
        train_acc += F.sigmoid(output).max(1)[1]

    print('Epoch␣%d:␣loss:␣%.3f,␣accuracy:␣%.3f' % (epoch, train_loss / len(train_dataloader)
        , train_acc / len(train_dataloader)))
```

Listing 9: Llama3.2 3B Output

In this scenario, the model performed poorly and produced a hallucinated output. Although the prompt explicitly instructed the use of the accelerate module, the model mistakenly interpreted this as a model name rather than a library. Additionally, it generated a standard PyTorch training loop, completely ignoring the intended usage of the accelerate framework. This example highlights a key limitation: the model tends to struggle when instructions become highly specific or require familiarity with less commonly used libraries or tools.

8 Limitations

One of the primary limitations of our work is that we focused exclusively on small-scale models. While these models are easier to train and deploy, their limited parameter count restricts their ability to generate highly complex or large-scale code structures. This limitation will be clearly demonstrated in the project demo, where the generated code may lack depth, optimization, or the handling of edge cases typically expected in more sophisticated programming tasks.

Another important limitation is that these models have been specifically fine-tuned for Python code generation. While Python is a widely used and versatile language, its programming paradigms differ significantly

from those in other languages such as Java, C++, or Rust. As a result, these models may struggle to adapt to the syntax, structure, and idiomatic patterns of languages outside the Python ecosystem, limiting their generalizability across different programming contexts.

Additionally, the models are not designed to function as conversational agents or chat tools. They are capable only of generating a single block of code based on the given input query. Unlike more advanced coding assistants, these models do not have the capability to engage in multi-turn interactions, refine previous responses, or regenerate code based on iterative user feedback. This one-shot generation approach restricts their usability in real-world coding workflows where iterative refinement and dialogue are often crucial.

9 Conclusion

In this project, we successfully pre-trained and fine-tuned small LLMs for Python code generation, with model sizes ranging from 124 million to 3 billion parameters. However, the evaluation phase did not proceed as smoothly as anticipated. Code generation tasks inherently allow for multiple valid outputs, making the evaluation of these models particularly challenging. As a result, traditional evaluation methods, such as CodeBLEU, proved inadequate for capturing the true complexity and variability of the generated code.

This highlights the need for alternative evaluation strategies that are better suited to handle the diversity inherent in code generation tasks. Furthermore, our experiments revealed that model size alone is not the sole determinant of performance. Other design choices — including model architecture, quality of pre-training, and diversity of the training data — played significant roles in influencing the effectiveness of code generation. These findings suggest that future work should place greater emphasis on these factors rather than focusing exclusively on scaling model size.

References

[1] M. Chen et al. Evaluating large language models trained on code. *Proceedings of NeurIPS 2021*, 34:15012–15028, 2021.

[2] J. Dodge et al. Evaluating pre-trained transformers for text generation. In *Proceedings of NeurIPS 2020*, pages 8193–8206, 2020.

[3] Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. Enhancing code generation for low-resource languages: No silver bullet, 2025.

[5] S. Gururangan et al. Don't stop pretraining: Adapt language models to domains and tasks. *Proceedings of ACL 2020*, 58(1):1250–1263, 2020.

[6] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, 09–15 Jun 2019.

[7] J. Howard and S. Ruder. Universal language model fine-tuning for text classification. In *Proceedings of ACL 2018*, pages 1–9, 2018.

[8] Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications, 2025.

[9] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[10] J. Kaplan, S. McCandlish, D. Amodei, et al. Scaling laws for neural language models. *Proceedings of NeurIPS 2020*, 33:4798–4809, 2020.

[11] Kyla H. Levin, Kyle Gwilt, Emery D. Berger, and Stephen N. Freund. Effective llm-driven code generation with pythoness, 2025.

[12] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.

[13] M. Ott et al. Scaling neural machine translation. *Proceedings of NAACL 2019*, 1(1):299–310, 2019.

[14] C. Raffel et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

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

[16] Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models, 2024.

[17] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Code less, align more: Efficient llm fine-tuning for code generation with data pruning, 2024.

[18] Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. Python symbolic execution with llm-powered code generation, 2024.

[19] J. Wei et al. Emerging properties of large language models. *Nature Communications*, 13(1):563–575, 2022.

[20] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[21] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models, 2024.

[22] W. Zhao et al. Exploring continual pretraining of large language models. *Proceedings of NeurIPS 2023*, 36:10012–10025, 2023.