Task Specific Adaptation of Small Language Models

Sanjay Shrestha (THA079BCT039)

Deep Shrestha (THA079BCT000)

Rohan Dhakal (THA079BCT000)

Pradeep Pokhrel (THA079BCT000)

December 2025

INTRODUCTION

1.1 Background

Advent of LLM have fundamentally changed the software development and code writing process. LLM have become integral part of the workflow for developers and students. Even with this much of success, LLMs rely on massive datasets and big GPUs for training and running these models which makes them impossible for student and general people to run and train locally.

Fine-tuning is the process of further training a pre-trained LLM on a smaller, task-specific dataset. While the initial pre-training gives universal linguistic knowledge, fine-tuning shapes this generalized competence into specialized expertise.

The OPT-350M (Open Pre-trained Transformer) model, developed by Meta AI, is a decoder-only LLM. With its 350 million parameters, it gives an important balance. It is large enough to possess meaningful generative capacity, yet small enough to be computationally efficient for research, development, and fine-tuning on consumer-grade or limited-resource hardware. This makes it an ideal candidate for demonstrating efficient specialization techniques.

1.2 Objectives

A. Making better code generator than existing OPT-350M by finetuning it on python code datasets.
B. Using PEFT through QLoRA to finetune base model i.e OPT-350M.

1.3 Methodology

This project aims to fine-tune the OPT-350M large language model for Python code generation. The methodology is structured in three phases: dataset preparation, fine-tuning, and evaluation. Figure 1.1 shows the flowchart of the fine-tuning procedure.

Figure 1.1: Flowchart of the fine-tuning process

1.3.1 Dataset Description and Preparation

Fine-tuning OPT-350M for Python code generation requires a high-quality, domain-specific dataset containing diverse programming tasks and their corresponding solutions. We will construct a composite dataset from multiple reputable open-source sources.

Dataset Sources and Composition:

(1) FlyTech Python Code Dataset (2025)

- **Size:** ∼42,000 instruction–code pairs
- **Content:** Python scripts with comments, docstrings, and structured tasks
- **Reason for Selection:**
  - Curated specifically for code generation
  - High-quality, well-commented Python code
  - Diverse mix of algorithmic and real-world examples
- **License:** MIT License (per Hugging Face repository)

(2) Python Code Instructions 18k (Alpaca-style)

- **Size:** ∼18,000 instruction–response pairs
- **Content:** Instruction prompts and Python code responses
- **Reason for Selection:**
  - Matches the instruction-tuning structure required by OPT
  - Provides human-readable prompts with explanations
  - Complements FlyTech by increasing prompt diversity
- **License:** CC BY-NC 4.0 (non-commercial research use)

(3) LeetCode/HackerRank Programming Problems (Evaluation Only)

- **Size:** ∼200–300 problems (not used for training)
- **Content:** Problem descriptions + reference solutions written by us
- **Reason for Selection:**
  - Realistic competitive programming tasks
  - Enables functional correctness testing
- **License:**
  - Direct redistribution of LeetCode content is not allowed
  - Only prompts used for evaluation (fair use), with our own solutions

We will standardize the problem set used from LeetCode/HackerRank and not directly extract their internal datasets.

Dataset Cleaning

Collected datasets will go through the following processing steps:

- Removal of duplicate examples
- Filtering of incomplete or non-functional Python code
- Normalization of formatting and consistent indentation

Final Dataset Size (Expected)

- FlyTech: ∼40,000
- Alpaca: ∼17,000
- **Total usable examples:** ∼57,000

1.3.2 Train/Validation Split:

We will divide the cleaned dataset into:

- 90% training set
- 5% validation set
- 5% evaluation set

The evaluation set (LeetCode/HackerRank) remains entirely separate.

Instruction Formatting and Tokenization

Each training example will be converted into an instruction-style structure:

```
Instruction: Write a Python code for printing Hello World.
Response: <Python code with docstrings or comments>
```

Tokenization will be performed using the official OPT tokenizer to ensure compatibility with the pretrained model.

1.3.3 Model Fine-Tuning

We will fine-tune the OPT-350M model using QLoRA to enable efficient code generation while minimizing computational cost. The base model will be loaded in quantized form and kept frozen.

Fine-Tuning Approach

Fine-tuning follows an instruction-tuning paradigm: given a text prompt x, the model predicts the next token in the Python code sequence y.

$$\text{loss} = -\sum_{t=1}^{T} \log P_\theta(y_t^{\text{true}} \mid y_{<t}, x) \tag{1.1}$$

Model parameters are updated using gradient descent:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \frac{\partial(\text{loss})}{\partial \theta} \tag{1.2}$$

QLoRA for Efficient Training

QLoRA enables parameter-efficient fine-tuning by:

- Keeping the pretrained model weights frozen

- Loading weights in 4-bit quantization
- Adding trainable low-rank adapter matrices inside transformer blocks

Only the adapter parameters are optimized, reducing VRAM usage and making training feasible on Google Colab.

Training Setup

- **Platform:** Google Colab
- **Optimizer:** AdamW
- **Hyperparameters:** learning rate, batch size, and epochs tuned using validation loss
- **Regularization:** Dropout applied to adapter layers
- **Early stopping:** Enabled based on validation loss
- **Batching:** Token-based batching targeting $\sim$0.5M tokens per batch

If a single Colab session is insufficient for training the full dataset, the dataset will be divided into multiple segments and trained incrementally, with checkpoints saved between sessions.

Monitoring and Checkpointing

- Training monitored with Weights & Biases (WandB)
- Periodic checkpoints stored locally and on Google Drive
- Final model uploaded to Hugging Face Hub

1.3.4 Evaluation

The fine-tuned model will be compared against the base OPT-350M model using three evaluation strategies.

1. Functional Correctness (Primary Evaluation)

Generated solutions will be executed in a secure Docker-based sandbox. Each result is classified as:

- **Right:** Correct output for all test cases
- **Partial:** Runs but fails some test cases
- **Wrong:** Crashes or produces incorrect output

This directly measures true problem-solving ability.

2. Automatic Similarity Metrics (BLEU, CodeBLEU)

- **BLEU:** Evaluates token-level similarity to reference solutions
- **CodeBLEU:** Incorporates syntax, data-flow, and semantic structure; more suitable for code generation

These are used as supporting quantitative metrics alongside functional evaluation.

3. Combined Evaluation Insight

By combining functional execution with similarity-based metrics, we ensure that the generated Python code is not only syntactically valid but also meaningful and

semantically aligned with correct solutions.