Fine-Tuning LLM using QLoRA for Software Bug Detection and Measuring the Degree of Improvement

Vamsi Dath Meka	Bhavyashree Putta	Tejaswi Tiyyagura	Teja Kavikondala	Satwik Pamulaparthy
Dept. of CS	*Dept. of CS*	*Dept. of CS*	*Dept. of CS*	*Dept. of CS*
vmeka@uic.edu	bputt@uic.edu	ttiyy@uic.edu	tkavik2@uic.edu	vpamu@uic.edu

1. Introduction

The rise of Large Language Models (LLMs) like OpenAI's GPT and Meta's LLaMA has transformed tasks in natural language and code generation. However, their effectiveness in detecting and correcting software bugs remains limited, as this task requires deep semantic understanding beyond surface-level syntax. Traditional bug detection methods are often manual, time-consuming, and difficult to scale. Given the increasing complexity of software systems, there is a growing need for intelligent, automated solutions that can assist in identifying and fixing bugs efficiently.

This project explores the use of QLoRA, a resource-efficient fine-tuning method, to adapt LLMs—specifically LLaMA 2 and CodeLLaMA—for bug detection and correction tasks. Using the PyTraceBugs dataset, which contains real-world buggy and corrected Python code along with tracebacks, we fine-tuned these models and evaluated their performance using BLEU, ROUGE-L, loss, and perplexity metrics. Our findings show that domain-specific pretraining and efficient fine-tuning can significantly enhance bug-fixing capabilities. This work demonstrates a practical approach to leveraging LLMs in software engineering, making automated debugging more accessible and scalable.

1.1. Original Goals of the Project

This project aimed to improve the performance of Large Language Models (LLMs) in software bug detection and automatic code correction, particularly for Python. While general-purpose models like **LLaMA 2** exhibit promise, their direct application to debugging tasks is limited. To address this, the following goals were set:

- **Fine-tune LLMs** on buggy Python code using the *PyTraceBugs* dataset to help the model learn bug patterns and fixes.
- **Use QLoRA** (Quantized Low-Rank Adapter) for parameter-efficient training, enabling large-model fine-tuning on constrained hardware.
- **Compare Fine-tuned model with base-line** using Evaluation Metrics to see if there is an improvement in performance.
- **Evaluate on PyTraceBugs**, leveraging its real-world buggy code and traceback logs for both training and testing.
- **Benchmark performance** using BLEU, ROUGE-L, loss, and perplexity metrics to compare pre-trained vs. fine-tuned models.

1.2. Modified Goals

During the implementation phase, several adjustments were made to refine the strategy and improve outcomes:

- **Experimented with CodeLLaMA:** In addition to LLaMA 2, CodeLLaMA was explored for its superior code comprehension capabilities.
- **Applied selective layer tuning:** Specifically tuned the `q_proj` and `v_proj` layers to minimize GPU memory usage while preserving performance.
- **Included perplexity as an evaluation metric:** This helped assess the fluency and confidence of the language model in generating code.
- **Replaced raw buggy code with AST-based input:** This shift provided richer contextual information, leading to better bug localization and correction performance.

These refinements enhanced model efficiency and improved output quality while remaining aligned with the project's original vision.

1.3. Assumptions

The following assumptions were made during the course of the project:

- **Fine-tuned models perform better** than baseline models in software bug detection and correction tasks.
- **Full fine-tuning was impractical** due to hardware constraints; hence, QLoRA was adopted for parameter-efficient fine-tuning.
- **Traceback logs provide useful context** that enhances the model's ability to identify and understand bugs.
- **The bugs in the dataset were assumed to be representative** of generalizable patterns in real-world Python code.
- **NLP evaluation metrics like BLEU and ROUGE-L** were considered appropriate for assessing code generation quality.

1.4. Definitions of Terms

To ensure clarity throughout this report, the following technical terms are defined as they relate to our project:

- **LLM (Large Language Model):** A neural network trained on vast amounts of text data capable of generating human-like responses and code. Examples include *LLaMA* and *GPT* models.
- **QLoRA (Quantized Low-Rank Adapter):** A fine-tuning technique that employs 4-bit quantization and low-rank adapters to train large models efficiently by updating only a small set of parameters.
- **CodeLLaMA:** A variant of Meta's LLaMA model that is pre-trained specifically on code-related data, making it more effective for programming tasks than general-purpose LLMs.
- **Buggy Code:** A segment of programming code that contains errors or malfunctions, which may lead to incorrect behavior or runtime failures.
- **Traceback:** A Python-specific error report that outlines the sequence of function calls that led to an exception, aiding in diagnosing bugs.
- **BLEU Score:** A metric used to evaluate the quality of generated sequences by comparing them with reference outputs. It is widely used in machine translation and code generation tasks.

- **ROUGE-L Score:** A recall-oriented metric that evaluates the overlap of the longest common subsequences between generated and reference outputs.
- **Perplexity:** A measure of how well a language model predicts a given sequence. Lower perplexity values indicate better predictive performance.

1.5. Summary of Approach

The project followed a structured pipeline from data collection to evaluation, centered around adapting Large Language Models (LLMs) for the specific task of software bug detection and correction.The workflow included the following stages:

- **Data Preparation:** The PyTraceBugs dataset was cleaned and structured to make it compatible with transformer-based model inputs.
- **Model Selection:** Initially used *LLaMA 2*, and later switched to *CodeLLaMA* for better performance on code-related tasks.
- **Fine-Tuning:** Utilized *QLoRA* to perform parameter-efficient fine-tuning by updating a subset of adapter layers.
- **Evaluation:** Improvements were assessed using metrics such as *BLEU*, *ROUGE-L*, *loss*, and *perplexity*.
- **Comparison:** The fine-tuned models were benchmarked against their respective base models to quantify performance improvements.

1.6. Project Deliverables

The following key deliverables were produced as part of this project:

- **Dataset Preparation:** Extracted and processed buggy/fixed code pairs from the *PyTraceBugs* dataset, along with associated traceback logs, and converted them into a format compatible with transformer models.
- **Fine-Tuned Models:** Successfully fine-tuned *LLaMA 2* and *CodeLLaMA* models using **QLoRA**, optimized for detecting and correcting buggy Python code.
- **Training and Evaluation Pipeline:** Developed a complete training and evaluation pipeline utilizing *Hugging Face Transformers*, *PEFT (QLoRA)*, and *BitsAndBytes*, with evaluations conducted using BLEU, ROUGE-L, loss, and perplexity.
- **Result Analysis and Comparative Study:** Performed a comparative analysis between pre-trained and fine-tuned models, highlighting measurable improvements—particularly with CodeLLaMA on raw code inputs.

1.7. Contribution of Individual Team Members

- **Vamsi:** Took care of data preprocessing, converting the PyTraceBugs dataset into code and AST formats. Set up tokenizers and prompts, and managed model training on the A100 GPU.
- **Bhavya**: Fine-tuned the CodeLLaMA model using QLoRA for both code and AST formats. Handled training settings, validation, and calculated evaluation metrics like BLEU, ROUGE-L, loss, and perplexity.
- **Tejaswi**: Fine-tuned the LLaMA 2 model using QLoRA for both code and AST formats. Managed training, testing, and evaluation using the same metrics as above.
- **Teja:** Set up the model architecture and fine-tuning configuration. Built a modular training pipeline and helped with final testing and defining how model performance would be measured.
- **Satwik:** Compared model results, created visualizations like graphs and tables, and worked on organizing data splits. Also helped clean up code and format the final report.

2. Background Research

2.1. Primary Research

As part of our primary research, we extensively tested a range of tools, frameworks, and libraries essential for building and evaluating a large language model fine-tuning pipeline. Our hands-on exploration focused on Hugging Face's ecosystem, BitsAndBytes, QLoRA fine-tuning methods, and evaluation utilities. Additionally, we relied on peer collaboration and discussions with mentors and instructors to refine our strategy.

Tools and Software Tested:

- **Hugging Face Transformers and PEFT (Parameter-Efficient Fine-Tuning):** Used for model loading, LoRA adapter injection, and structured training workflows. Hugging Face's `Trainer` API was employed for consistent training and evaluation.
- **BitsAndBytes Library (by Tim Dettmers):** Enabled 4-bit quantization of large models, allowing us to fine-tune LLaMA 2 and CodeLLaMA on a single 24GB GPU (NVIDIA RTX 3090). This made the project feasible within hardware constraints.
- **QLoRA Implementation:** Implemented via the PEFT library, with low-rank adapters configured on the `q_proj` and `v_proj` layers. We conducted experiments to assess memory savings and performance impact.
- **PyTorch:** Served as the core deep learning framework for training, including custom loss function handling, validation routines, and checkpoint logic.

Evaluation Tools:

- `sacrebleu` for BLEU score evaluation.
- `rouge-score` for ROUGE-L metrics.
- `PyTorch` for computing loss and perplexity.

Expert Discussions and Guidance:

- We consulted our course instructor for feedback on model selection (e.g., LLaMA vs. CodeLLaMA) and dataset formatting (e.g., raw code vs. AST-based representations).
- Internal peer discussions within the team were instrumental in troubleshooting training errors such as GPU memory overflows and token truncation, and in refining our evaluation strategies.

2.2. Secondary Research

We referred to a variety of academic papers, datasets, and technical blogs that significantly influenced our approach to model selection, evaluation design, and implementation.

Papers:

- T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "*QLoRA: Efficient Finetuning of Quantized LLMs*," *arXiv preprint arXiv:2305.14314*, 2023.
- M. Tufano, C. Watson, G. Bavota, and D. Poshyvanyk, "*An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation*," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 1–29, 2019.
- K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "*BLEU: a method for automatic evaluation of machine translation*," in *Proc. 40th Annu. Meet. Assoc. Comput. Linguist.*, 2002, pp. 311–318.
- C.-Y. Lin, "*ROUGE: A Package for Automatic Evaluation of Summaries*," in *Text Summarization Branches Out: Proc. ACL Workshop*, 2004, pp. 74–81.

- E. J. Hu et al., "*LoRA: Low-Rank Adaptation of Large Language Models*," *arXiv preprint arXiv:2106.09685*, 2021.
- M. Chen et al., "*Evaluating Large Language Models Trained on Code*," *arXiv preprint arXiv:2107.03374*, 2021.
- W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "*Unified Pre-training for Program Understanding and Generation*," *arXiv preprint arXiv:2103.06333*, 2021.
- T. Wolf et al., "*Transformers: State-of-the-Art Natural Language Processing*," in *Proc. 2020 Conf. Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45.

Dataset:

- E. N. Akimova et al., "*PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction*," in *Proc. 2021 Asia-Pacific Software Engineering Conference (APSEC)*, Taipei, Taiwan, 2021, pp. 141–145.

Blogs and Developer Guides:

- *Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA*, Hugging Face Blog (Published May 24, 2023): `https://huggingface.co/blog/4bit-transformers-bitsandbytes`
- *Hugging Face PEFT Developer Guide – LoRA*: `https://huggingface.co/docs/peft/main/en/developer_guides/lora`
- PEFT GitHub LoRA Guide: `https://github.com/huggingface/peft/blob/main/docs/source/developer_guides/lora.md`

2.3. Most Helpful Resources

Our project was guided by several foundational research papers, datasets, and developer resources that directly influenced the design, implementation, and evaluation of our fine-tuning pipeline.

- **T. Dettmers et al., "QLoRA: Efficient Finetuning of Quantized LLMs," arXiv:2305.14314 (2023)**
  This paper introduced the QLoRA technique, which was central to our project. It enabled parameter-efficient fine-tuning of large language models using 4-bit quantization and low-rank adapters. We adopted its quantization methods, adapter configuration, and training strategy to make large model fine-tuning feasible on limited hardware.
- **M. Tufano et al., "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," ACM TSE, 2019**
  This work explored using neural translation techniques to learn bug-fix patterns from real-world data. It inspired our supervised training approach by validating the feasibility of learning corrective edits from buggy code datasets. It also informed our evaluation methods using sequence-level similarity metrics like BLEU.
- **M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374 (2021)**
  This paper benchmarked LLMs such as CodeX and GPT-style models on code-related tasks. It helped us understand the limitations and expectations of LLMs in handling structured code inputs and informed our selection of models like CodeLLaMA for better performance in code correction.
- **E. N. Akimova et al., "PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction," APSEC 2021**
  This dataset served as the foundation for our experiments. Its structure—offering buggy and corrected code, along with traceback logs—allowed us to simulate realistic bug-fix learning scenarios. We used both the raw code and AST-based variants of this dataset for training and evaluation.

- **Hugging Face Blog: "Making LLMs Even More Accessible with bitsandbytes, 4-bit Quantization and QLoRA" (May 24, 2023)**
  This blog provided practical guidance on integrating 4-bit quantization using BitsAndBytes with Hugging Face's QLoRA framework. It was critical in setting up the hardware-efficient training pipeline used in our project.

- **Hugging Face PEFT Developer Guide – LoRA**
  This documentation explained the configuration of LoRA adapters, especially how to apply them selectively to attention components like `q_proj` and `v_proj`. We referenced this while implementing our PEFT configuration using the Hugging Face PEFT library.

3. Approach to Your Goals

3.1. Strategy Followed

Below is the detailed strategy followed for implementation.

3.1.1 Data Formatting

We selected **PyTraceBugs**, a publicly available dataset that contains actual buggy Python code and their corresponding corrected versions. Each entry is enriched with traceback logs and exception types, making it a practical choice for training a model to detect and fix code issues.Below are the formats used:

- **Code-based:** This format allows the model to learn from human-readable syntax. It uses raw code text — the buggy version (`before_merge`) and the fixed version (`after_merge`).

- **AST-based:** This format helps the model understand deeper structural changes and semantics. Each code snippet is parsed into its Abstract Syntax Tree (AST) using Python's `ast` module. These ASTs represent the structure of the code in JSON format — the buggy version is `old_ast_json`, and the fixed version is `new_ast_json`.

Prompt Engineering:

- **For code:** Input = buggy code, Output = fixed code.

- **For AST:** Input = old AST JSON, Output = new AST JSON.

Tokenization:

- `LLaMA` tokenizer was used for LLaMA 2.

- `CodeLLaMA` tokenizer was used for CodeLLaMA.

- Both employed byte-level tokenization compatible with Hugging Face Transformers.

The data was tokenized using Hugging Face's `AutoTokenizer` to convert text into token IDs. Truncation, padding, and maximum sequence length were applied to ensure compatibility with model limits.

3.1.2 Model Architecture & Fine-Tuning Setup

We chose two large language models — **LLaMA 2** and **CodeLLaMA** — optimized for code understanding and generation.**QLoRA (Quantized Low-Rank Adapter)** was used to enable efficient fine-tuning on a single GPU.
Key Techniques:

- **4-bit Quantization:** Compressed model weights to fit large models into 6–8 GB VRAM.

- **LoRA Adapters:** Small learnable modules added only to the `q_proj` and `v_proj` layers in the attention mechanism.

- **Frozen Base Model:** Only the adapter layers were updated, preserving the original model weights.

LoRA Configuration:

```
LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.1,
    bias="none"
)
```

Quantization: 4-bit quantization was applied using `BitsAndBytes` to reduce memory usage significantly and support model fine-tuning in resource-constrained environments.

3.1.3 Fine-Tuning Strategy

Given the high resource demands of traditional fine-tuning, we used **QLoRA** to train models efficiently using 4-bit quantization and low-rank adapters. We fine-tuned both models independently on code-based and AST-based formats.
Training Configuration:

- **Learning Rate:** 2×10^{-4}
- **Batch Size:** 4 (with gradient accumulation of 8)
- **Epochs:** 3
- **Gradient Accumulation Steps:** 8
- **Gradient Checkpointing:** Enabled
- **Validation Checkpoints:** Every 500 steps
- **Model Saving:** Best model saved based on lowest validation loss
- **Logging:** Every 10 steps
- **Loss Function:** Cross-Entropy Loss (standard for token-level prediction)
- **Hardware:** A100 GPU.

Training was conducted using the Hugging Face `Trainer` API integrated with QLoRA modules. Tokenized datasets were passed to the trainer with proper data collators for efficient batch processing.

3.1.4 Modular Pipeline Design

The implementation followed a modular structure for maintainability and scalability:

- **Data Preprocessing:** Converted raw JSON/pickle files into tokenized sequences.
- **Model Loading:** Loaded pre-trained base models.
- **Fine-Tuning:** Applied QLoRA adapters to selected layers.
- **Training:** Executed memory-efficient training.
- **Evaluation:** Compared base and fine-tuned models using BLEU, ROUGE-L, loss, and perplexity.

3.1.5 Model Comparison and Interpretation

After fine-tuning, side-by-side comparisons were conducted to measure improvements.

- Metric tables and visualizations were used to assess how well the fine-tuned models performed, especially in bug detection and correction tasks.
- This helped quantify gains in model comprehension and error resolution capabilities.

3.1.6 Evaluation Metrics

We used four main metrics to evaluate model performance:

- **BLEU:** Measures n-gram overlap between predicted and reference sequences.
- **ROUGE-L:** Captures the longest common subsequence (LCS) overlap.
- **Loss:** Cross-entropy loss calculated from model logits.
- **Perplexity:** The exponential of loss; lower perplexity indicates better fluency and confidence.

Implementation:

- BLEU calculated using `sacrebleu`.
- ROUGE-L via `rouge_scorer`.
- Loss and perplexity computed on the tokenized test dataset using `PyTorch`.

3.1.7 Benefits of Unified Training Design

- **Reusability:** The same training scripts could be reused with minimal modifications.
- **Scalability:** Pipeline supports different model sizes and code domains.
- **Efficiency:** Enabled fine-tuning of 7B parameter models on limited GPU memory.
- **Comparability:** Maintained consistent evaluation criteria across all models.

The only differences between training runs were the underlying model checkpoints and tokenizers. All other components — QLoRA setup, training logic, and evaluation pipeline — remained the same.

3.2. Motivation for Strategy

Fine-tuning LLMs for domain-specific tasks like bug detection and code correction presents a number of challenges. Our motivation for adopting this specific strategy centered around addressing these challenges effectively, while also ensuring robustness, scalability, and practical feasibility. The key motivations are as follows:

- **Resource Efficiency:** QLoRA enabled fine-tuning of large models on limited hardware (e.g., a single 8–24 GB GPU) by significantly reducing memory usage via 4-bit quantization.
- **Dual Input Perspective:** Leveraging both code-based and AST-based inputs allowed us to evaluate whether syntax-level or structure-level representations led to better learning and generalization.
- **Reusable Pipeline:** A modular and well-structured training pipeline ensured consistency, reusability, and scalability across multiple models and datasets.
- **Targeted Adaptation:** By updating only the LoRA adapter layers, we preserved the base model's general knowledge while efficiently tailoring it for domain-specific tasks like bug detection and correction.
- **Clear Evaluation Framework:** The use of standardized metrics and validation checkpoints facilitated meaningful performance comparisons and improved interpretability of results.

3.3. Division of Work

Below are the responsibilities that were distributed among team members.

- **Vamsi:** Handled data formatting, including preprocessing PyTraceBugs dataset into code-based and AST-based formats.Coordinated GPU resource utilization and managed training on A100.Implemented tokenizer setup and prompt engineering for both LLaMA 2 and CodeLLaMA.
- **Bhavya:** Led end-to-end fine-tuning of CodeLlama-7B using QLoRA across both code and AST formats; configured training hyperparameters, executed model training, and managed validation and testing cycles. Computed BLEU, ROUGE-L, loss, and perplexity to benchmark performance gains and ensure model reliability.
- **Tejaswi:** Led end-to-end fine-tuning of Llama2-7B using QLoRA across both code and AST formats; configured training hyperparameters, executed model training, and managed validation and testing cycles. Computed BLEU, ROUGE-L, loss, and perplexity to benchmark performance gains and ensure model reliability.
- **Teja:** Led the model architecture setup and fine-tuning configuration using QLoRA.Worked on modular pipeline structuring, including configuration file setup, modularization of scripts, logging utilities, and checkpoint monitoring;final testing.Focused on empirical study design,including defining evaluation metrics.
- **Satwik:** Led model comparison and visualization efforts,training/validation split logic, including result interpretation and graph/table generation.Contributed to code cleanup.Final report formatting.

4. Implementation

4.1. Software Design

The software design for this project was structured as a modular and scalable pipeline intended to support the full workflow — from dataset ingestion to model evaluation. Since the core of the project involved fine-tuning transformer-based language models, the software was primarily developed using Python and composed of modular scripts responsible for preprocessing, training, evaluation, and result analysis.The pipeline was organized into the following major components.

Data Processing Module - This module handled the initial preparation and structuring of the dataset for model training. Its responsibilities included:

- Loading and parsing the `.pkl` files from the *PyTraceBugs* dataset.
- Extracting relevant fields such as `before_merge` (buggy code), `after_merge` (corrected code), and `full_traceback`.
- Tokenizing code samples using tokenizers compatible with both `LLaMA` and `CodeLLaMA`.
- Formatting the input-output pairs for supervised fine-tuning tasks.

Model Fine-Tuning Module -

This module served as the core training engine and was responsible for:

- Loading pre-trained base models, including `LLaMA 2` and `CodeLLaMA`.
- Applying QLoRA adapters via the PEFT (Parameter-Efficient Fine-Tuning) library.
- Freezing the base model weights while enabling training on selected adapter layers (`q_proj` and `v_proj`).
- Executing the training loop with configurable batch sizes, learning rates, and epoch counts.

Evaluation Module - After training, this module was used to assess model performance. It provided:

- Inference capability to generate code predictions for unseen buggy inputs.
- Comparison of predictions against reference fixes using evaluation metrics: **BLEU**, **ROUGE-L**, **Loss**, and **Perplexity**.
- Output of evaluation results in both tabular and graphical formats for effective reporting.

Configuration and Utility Scripts -

To ensure ease of experimentation and reproducibility, configuration and utility tools were separated out:

- A centralized `JSON` or `YAML` configuration file was used to manage model hyperparameters, file paths, and training settings.
- Utility scripts supported features like logging, checkpointing, error handling, and runtime monitoring.

This modular design ensured high flexibility and ease of use across various training scenarios, allowing seamless switching between model variants and data formats.

4.2. External Components

The following external components and libraries were utilized to support various stages of model training, fine-tuning, evaluation, and experimentation:

- **Hugging Face Transformers:** Used for loading pre-trained LLaMA 2 and CodeLLaMA models, managing tokenizers, and integrating with the training pipeline.
- **PEFT (Parameter-Efficient Fine-Tuning):** Enabled the application of QLoRA (Quantized LoRA) adapters for efficient fine-tuning of large language models with minimal GPU memory usage.
- **BitsAndBytes:** Provided 4-bit quantization functionality, significantly reducing the memory footprint during training while preserving model performance.
- **PyTorch:** Served as the core deep learning framework for defining, training, and evaluating models.
- **PyTraceBugs Dataset:** A public dataset containing buggy Python code, fixes, and traceback logs.Facilitated dataset loading, preprocessing, and batching compatible with Transformers-based pipelines. It was used for both training and evaluation. Available at: `https://github.com/acheshkov/pytracebugs?tab=readme-ov-file`

  Dataset Overview

  - **Origin:** Collected from public GitHub repositories under permissive licenses such as MIT, Apache, and BSD.
  - **Granularity:** Focuses on function-level and method-level code snippets for better contextual understanding during training.
  - **Structure:**
    - **Buggy Dataset:** Contains code snippets with known bugs, extracted from bug-fix commits and pull requests linked to GitHub issues.
    - **Stable Dataset:** Includes correct code snippets from well-maintained and stable codebases.
  - **Data Splits:** Each dataset is divided into training, validation, and test sets to support reproducible evaluation and model tuning.
- **SacreBLEU and ROUGE-Scorer:** Used to compute BLEU and ROUGE-L scores for evaluating the similarity between generated and reference code fixes.
- **Python `ast` Module:** Utilized to convert Python code into Abstract Syntax Trees (ASTs) for structured representation and training on AST-based inputs.
- **NVIDIA CUDA and PyTorch GPU Libraries:** Supported accelerated training on GPU-enabled hardware environments such as the RTX 3090.

These components collectively enabled the creation of a scalable, efficient, and reproducible framework for fine-tuning LLMs on code correction tasks.

4.3. Design of Empirical Studies

To evaluate the effectiveness of fine-tuning Large Language Models (LLMs) for software bug correction, we designed empirical studies comparing base and fine-tuned versions of LLaMA 2 and CodeLLaMA using both code-based and AST-based datasets. Our goal was to measure performance improvements using multiple evaluation metrics and identify which input representation and model architecture offered the most accurate bug fixes.

- **Controlled Comparison:** Both LLaMA 2 and CodeLLaMA were fine-tuned independently using identical training configurations (QLoRA, learning rate, and batch size) to ensure a fair comparison across models.
- **Dataset Variants:** Two versions of the dataset were used—raw buggy code and Abstract Syntax Trees (ASTs). Each model was trained on both formats to assess syntactic vs. structural input effectiveness.
- **Evaluation Metrics:** BLEU score, ROUGE-L, cross-entropy loss, and perplexity were used to evaluate syntactic correctness, structural similarity, model learning efficiency, and generation fluency.
- **Validation and Checkpointing:** Fine-tuned models were evaluated on a held-out test set. Validation was performed at regular intervals, and the best-performing model checkpoint was selected based on the lowest validation loss.
- **Comparative Analysis:** Results were compared between fine-tuned and base models, as well as across LLaMA 2 and CodeLLaMA, to understand model effectiveness and the impact of fine-tuning strategies.

This experimental design ensured consistent and reproducible results, allowing us to draw meaningful conclusions about the performance gains achieved through QLoRA-based fine-tuning.

5. Results

5.1. Empirical Results

In this section we describe the results for both the LLMs we used by each metric and describes what can be inferred from those values. This section summarizes the evaluation outcomes of fine-tuning LLaMA 2 using both AST-based and code-based inputs. The results are analyzed using BLEU, ROUGE-L, loss, and perplexity metrics.

5.1.1 Emperical results for LLaMA2

Approach	Model	Loss	BLEU	ROUGE-L	Perplexity
Code Based Fine Tuning	Base	1.2472	54.68	0.556	3.48
	Fine-tuned	1.1060	54.73	0.557	3.02
AST Based Fine Tuning	Base	1.1613	7.68	0.252	3.19
	Fine-tuned	6.7875	7.67	0.252	2.26

Table 1: Evaluation Results for LLaMA 2 (Code-Based vs AST-Based Fine Tuning)

Code-Based Metrics (Raw Code Differences)

- **BLEU Score:** *Base Model:* 54.68, *Fine-Tuned Model:* 54.73
  Slight increase — indicates marginal syntactic improvement.
- **ROUGE-L Score:** *Base Model:* 0.556, *Fine-Tuned Model:* 0.557
  Slight improvement — better sequence-level overlap.

- **Loss:** *Base Model:* 1.2472, *Fine-Tuned Model:* 1.1060
  Decrease — suggests better learning of bug-fix patterns.
- **Perplexity:** *Base Model:* 3.48, *Fine-Tuned Model:* 3.02
  Decrease — indicates improved fluency and coherence.

Key Conclusions:

- **Consistent Metric Improvements Across the Board:** Slight improvements across BLEU, ROUGE-L, loss, and perplexity indicate effective learning and alignment with the bug-fix task.
- **Effective Generalization and Fluency:** The decrease in loss and perplexity without any increase in instability suggests code-based fine-tuning supports better generalization and more fluent outputs than AST-based input for LLaMA 2.

AST-Based Metrics (Abstract Syntax Tree Differences)

- **BLEU Score:** *Base Model:* 7.68, *Fine-Tuned Model:* 7.67
  Slight decrease — fine-tuning did not improve syntactic similarity.
- **ROUGE-L Score:** *Base Model:* 0.252, *Fine-Tuned Model:* 0.252
  No change — structural overlap remains identical.
- **Loss:** *Base Model:* 1.1613, *Fine-Tuned Model:* 6.7875
  Significant increase — indicates potential overfitting or poor adaptation to AST format.
- **Perplexity:** *Base Model:* 3.19, *Fine-Tuned Model:* 2.26
  Decrease — indicates improved fluency and model confidence.

Key Conclusions:

- **Inconsistent Improvement Despite Lower Perplexity:** While perplexity decreased, BLEU and ROUGE-L showed no gain, suggesting fluency improved without enhancing syntactic or structural accuracy.
- **Overfitting or Misalignment on ASTs:** The sharp increase in loss suggests overfitting or a challenge in learning meaningful representations from ASTs due to limited semantic context.

5.1.2 Emperical results for CodeLLaMA

This section presents the evaluation metrics for CodeLLaMA under both code-based and AST-based fine-tuning regimes. The performance is measured using BLEU, ROUGE-L, loss, and perplexity.

Approach	Model	Loss	BLEU	ROUGE-L	Perplexity
Code Based Fine Tuning	Base	0.9878	55.24	0.560	2.69
	Fine-tuned	0.8636	55.22	0.559	2.37
AST Based Fine Tuning	Base	0.7002	8.82	0.255	2.01
	Fine-tuned	0.3938	7.75	0.255	1.48

Table 2: Evaluation Results for CodeLLaMA (Code-Based vs AST-Based Fine Tuning)

Code-Based Metrics (Raw Code Differences)

- **BLEU Score:** *Base Model:* 55.24, *Fine-Tuned Model:* 55.22
  Negligible drop — BLEU score remained stable, suggesting that fine-tuning had minimal impact on syntactic overlap, possibly due to CodeLLaMA already being optimized for code.
- **ROUGE-L Score:** *Base Model:* 0.560, *Fine-Tuned Model:* 0.559
  Almost unchanged — sequence-level similarity between predicted and reference outputs was retained post fine-tuning.

- **Loss:** *Base Model:* 0.9878, *Fine-Tuned Model:* 0.8636
  Significant reduction — indicates improved learning of bug-fix patterns and better generalization.
- **Perplexity:** *Base Model:* 2.69, *Fine-Tuned Model:* 2.37
  Notable decrease — improved confidence and fluency in generating corrected code snippets.

Key Conclusions:

- **Stable Output Quality with Improved Learning:** While BLEU and ROUGE-L scores remained stable, the notable improvement in loss and perplexity suggests better internal model understanding and consistency.
- **Effective Fine-Tuning without Sacrificing Syntactic Integrity:** CodeLLaMA maintained output quality while becoming more efficient and confident in bug-fix generation.

AST-Based Metrics (Abstract Syntax Tree Differences)

- **BLEU Score:** *Base Model:* 8.82, *Fine-Tuned Model:* 7.75
  Decline observed — suggests CodeLLaMA struggled with syntactic improvement from AST representations, likely due to structural mismatch with its training data.
- **ROUGE-L Score:** *Base Model:* 0.255, *Fine-Tuned Model:* 0.255
  No change — structural similarity stayed constant, showing limited impact from fine-tuning on ASTs.
- **Loss:** *Base Model:* 0.7002, *Fine-Tuned Model:* 0.3938
  Large decrease — suggests successful learning of AST-level transformation patterns during training.
- **Perplexity:** *Base Model:* 2.01, *Fine-Tuned Model:* 1.48
  Substantial drop — indicates greater internal confidence despite weak performance on surface-level metrics.

Key Conclusions:

- **Improved Semantic Understanding in Structured Input:** A 44% reduction in validation loss and 26% drop in perplexity indicate CodeLLaMA learned meaningful semantic transformations from AST inputs, even if syntactic metrics like BLEU and ROUGE-L did not improve.
- **Mismatch Between Input Type and Output Metrics:** Although the model gained internal efficiency and structural comprehension, the rigid format of ASTs may have limited its ability to express syntactic improvements detectable by BLEU and ROUGE metrics.

The following is a comparison of evaluation metrics for both LLMs after fine-tuning.

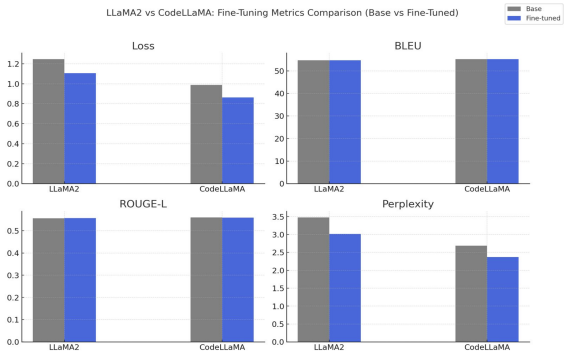

Figure 1: Comparison of LLaMA2 Vs CodeLLaMa

Conclusion Highlights:

- **Fine-tuning improved performance** – Reduced loss/perplexity and increased BLEU/ROUGE-L scores.
- **CodeLLaMA outperformed LLaMA2** – Likely due to its pretraining on source code.
- **Models handled buggy code better than AST** – Buggy code retains familiar patterns for LLMs.
- **QLoRA + CodeLLaMA is effective** – Works well for fine-tuning on code tasks.
- **Approach is efficient and accurate** – Achieves strong results with low resources.

5.2. Software Deliverables

The project produced several software components and artifacts that together form a complete and reproducible pipeline for fine-tuning large language models on code correction tasks. The following deliverables were created:

- **Preprocessing Scripts:** Python scripts to parse, clean, and convert the PyTraceBugs dataset into both code-based and AST-based formats suitable for model training.
- **Tokenization Pipelines:** Configured tokenizers using Hugging Face's `AutoTokenizer` for both LLaMA 2 and CodeLLaMA, including padding, truncation, and formatting utilities.
- **QLoRA Fine-Tuning Scripts:** Training scripts built on Hugging Face's `Trainer` API with support for QLoRA adapters, gradient accumulation, and quantized model loading via BitsAndBytes.
- **Configuration Files:** YAML/JSON-based config files to store hyperparameters, LoRA adapter settings, and training checkpoint options for both LLMs.
- **Evaluation Modules:** Standalone scripts for calculating BLEU, ROUGE-L, loss, and perplexity over model predictions using `sacrebleu`, `rouge_scorer`, and PyTorch.
- **Model Checkpoints:** Saved versions of the fine-tuned LLaMA 2 and CodeLLaMA models (including adapter weights) for both code and AST input types.
- **Results and Visualization Scripts:** Scripts to generate tabular summaries and plots of model performance, enabling comparison between base and fine-tuned versions.

These deliverables collectively support reproducibility, extendability, and further experimentation in neural code correction using QLoRA.
Below are the links for finetuning and evaluating LLMs for bug detection and correction.
For CodeLLaMA-`https://github.com/Vamsi-Dath/QLoRA_for_Software_Bug_Detection`
For LLaMA2 finetuning-`https://colab.research.google.com/github/tejaswi693/llama2-finetune/blob/main/final-model/llama2_7b_hf_fine_tune_model.ipynb?authuser=2#scrollTo=syD8_R0f0TrC`
For LLaMA2 Metrics evaluation-`https://colab.research.google.com/github/tejaswi693/llama2-finetune/blob/main/final-model/llama2_7b_hf_fine_tuned_metrics_eval.ipynb?authuser=2#scrollTo=teBNsk6EHgc1`

6. Waiting Room

While we identified promising extensions to enhance model performance and semantic understanding, we were unable to implement them within the project timeline due to time constraints. Below are two key directions for future exploration:

1. **Neuro-Symbolic Feedback Loop**
   We intended to integrate a symbolic executor or static analyzer to evaluate patches generated by the QLoRA-tuned model. The idea was to use pass/fail feedback to iteratively refine the LoRA adapters and enforce formal correctness. Due to time limitations, this neuro-symbolic feedback mechanism remains a future enhancement.

2. **Graph-Neural Adapters over ASTs**
   Another proposed extension was to embed lightweight Graph Neural Network (GNN) modules that process Abstract Syntax Trees (ASTs) during fine-tuning. This would help the model capture control-flow and data-flow structures for deeper semantic understanding and more accurate bug fixes. Although conceptually explored, it was not implemented due to project constraints.