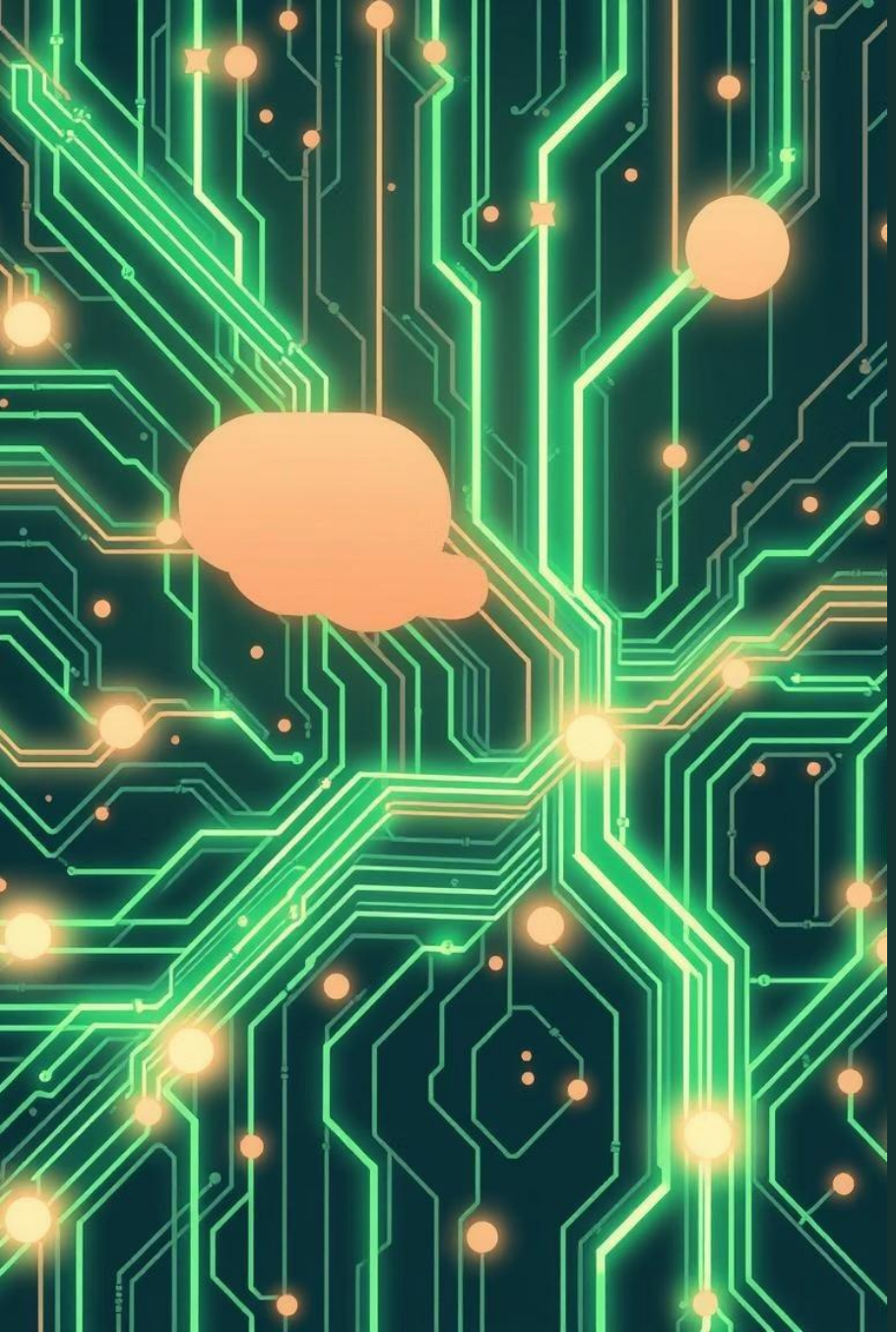


AI-Powered Code Optimizer: Enhancing Python with LLMs

This presentation details an AI system designed to automatically improve Python code for enhanced performance, clarity, and readability. Developed during an LLM training internship at Tips Hindawi, this project showcases practical applications of fine-tuned language models in software development.

Project Overview & Core Technologies

AI-Powered Code Optimisation

An intelligent system that refines Python code.

Mistral-7B LLM

Built upon a powerful open-source large language model.

LoRA Fine-Tuning

Leveraged LoRA for efficient model adaptation.

Custom Code Dataset

Trained on a bespoke dataset of unoptimized vs. optimized code examples.

Streamlit & ngrok

Frontend for interactive usage and seamless deployment.

Key Project Goals

Build a Lightweight, Fine-Tuned Model

Specialised in improving Python code by reducing time complexity, removing redundancy, and producing clean, readable output.

Provide an Easy-to-Use Interface

Developed with Streamlit, offering intuitive browser-based access for developers.

Allow Public Access via ngrok

Facilitates easy sharing of demos with instructors and teammates, ensuring broad accessibility.

System Architecture

The system architecture outlines the flow from user input to the refined, optimized Python code, leveraging a seamless integration of frontend and AI backend.

This streamlined process ensures efficient interaction and delivery of high-quality code optimization.

Dataset Preparation for Fine-Tuning

Dataset Columns

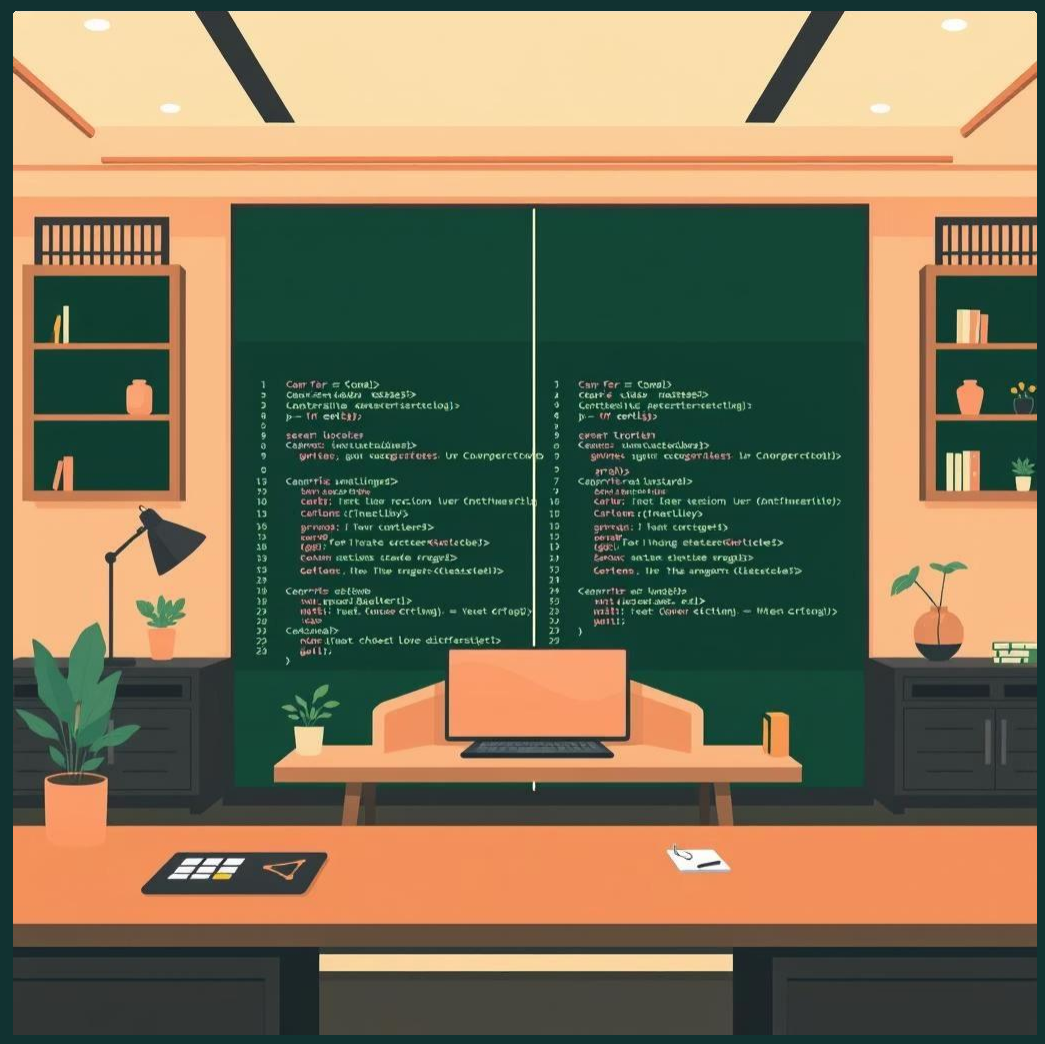
- `unoptimized_code`: The original Python code snippet.
- `optimized_code`: The improved version of the code.
- Optional metadata: `optimization_type`, `complexity_change`.

Transformation to Instruction-Response Pairs

Each row was converted into a structured instruction-response format for model training:

```
Instruction Template:
Optimize the following Python function/code for performance and clarity.
# Type: | Complexity:
### Unoptimized code:

Response:
[Optimised code]
```



Final SFT Format

`[INST] instruction [/INST] response`

Dataset Size & Split

A total of 3,000 samples were selected and rigorously split to ensure robust training and validation:

- 90% for training the model.
- 10% for validation to assess performance.

Model Fine-Tuning Details

Base Model Selection: Mistral-7B-v0.1

Mistral-7B was chosen for its exceptional capabilities in coding tasks and its efficiency.

- **Strong Performance:** Renowned for high accuracy in code generation and analysis.
- **Lightweight & Efficient:** Optimised for performance with reduced computational overhead.
- **Instruction Formatting:** Adept at processing and generating structured responses.

Quantization for GPU Efficiency

BitsAndBytes 4-bit quantization was applied to minimise GPU memory usage without significant performance degradation.

- `nf4` quantization
- `double quant` for further precision reduction.
- `bfloat16` compute for faster operations.

LoRA Configuration

- `r = 8`
- `alpha = 32`
- `dropout = 0.1`
- `task_type = causal LM`

Training Arguments

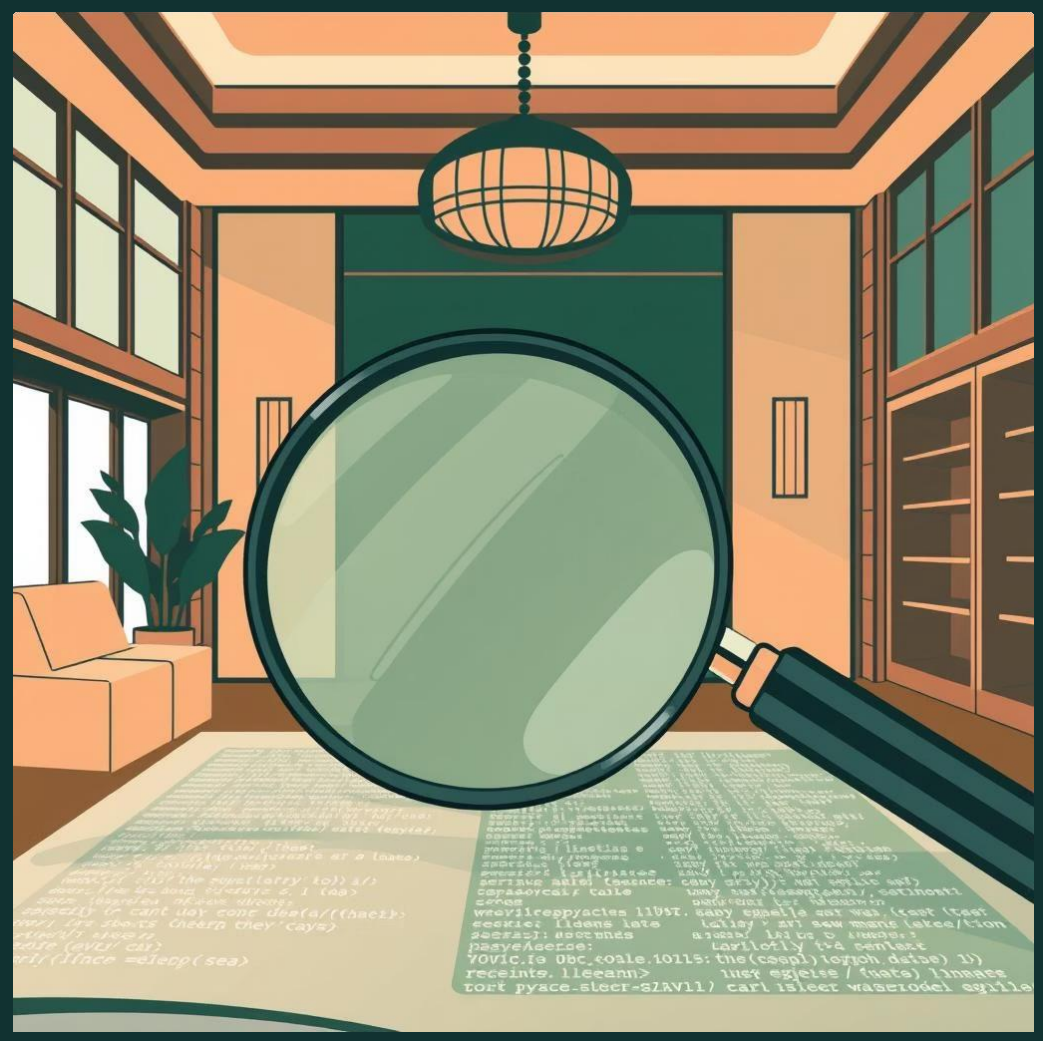
- Batch size: 2
- Gradient accumulation: 4
- Learning rate: 2e-4
- Warmup steps: 5
- Epochs: 1
- Evaluation: Every 50 steps
- Saving: Every 200 steps
- `bf16` enabled

Model Evaluation & Performance

Evaluation Logic

After the fine-tuning process, the model underwent rigorous evaluation using unseen validation samples.

- Output generation via a robust pipeline.
- Instruction prefix stripping for clean comparison.
- Direct comparison against ground-truth optimized code.
- Extensive manual verification for clarity and correctness.



Key Evaluation Results

- **Better Variable Selection:** The model consistently chose more appropriate and descriptive variable names.
- **More Concise Loops:** Redundant loop structures were simplified, leading to more efficient code.
- **Redundancy Removal:** Unnecessary conditions and duplications were effectively eliminated.
- **Structural Improvements:** Overall code architecture was enhanced, improving maintainability and readability.

These results underscore the effectiveness of the fine-tuning process in achieving the desired optimization goals.

Deployment with Streamlit & ngrok

Streamlit UI Features

- **Simple Textbox:** For direct input of Python code.
- **<Optimize= Button:** Triggers real-time model inference.
- **Output Card:** Displays the optimized code clearly.
- **Clean & Intuitive:** Designed for a seamless user experience.

ngrok Integration

ngrok was integrated to expose the local Streamlit application to the public internet.

```
ngrok authtoken
streamlit run app.py --server.port 8501
ngrok http 8501
```

Purpose of ngrok

- **Shareable URL:** Enables easy sharing with instructors and teammates.
- **Public Demo:** Facilitates demonstrations without complex server setups.

Tools & Technologies Utilised

Model: Mistral-7B

The foundational LLM for code optimization.

Training: Transformers, TRL, PEFT

Key frameworks for model fine-tuning and development.

Quantization: BitsAndBytes

Efficient memory management for GPU resources.

Interface: Streamlit

Interactive and user-friendly web application frontend.

Deployment: ngrok

Secure tunnelling for public access and sharing.

Dataset: Pandas, Huggingface Datasets

Tools for data manipulation and preparation.

Environment: Colab GPU

Cloud-based platform for accelerated training.

Challenges & Solutions

1

GPU Memory Limits

Overcome with 4-bit quantization techniques.

2

Dataset Inconsistencies

Addressed through meticulous metadata cleaning.

3

Instruction Formatting

Standardised using a consistent SFT format.

4

Easy Deployment

Achieved with the simple and effective ngrok tunnel.

Project Results

Successful LLM Fine-Tuning

Achieved a robust code optimization LLM.

Stable Performance

Maintained with low VRAM consumption.

Fully Functional Web Demo

Showcasing the model's capabilities in real-time.

Clean User Interface

Ensuring a positive user experience.

Easy Sharing

Facilitated by seamless ngrok integration.