Optimizing Memory Efficiency of Code Generated by Large Language Models

Arnav Joshi & Aditya Vinodh

Abstract

In an era where large language models (LLMs) are transforming software development, addressing non-functional requirements, such as memory efficiency, remains a critical challenge. This project investigates methods to optimize the memory efficiency of code generated by LLMs, building upon benchmarks like NoFunEval and EffiBench. Our initial experiment involved fine-tuning a pre-trained LLM using real-world examples of memory-optimized code from the NoFunEval dataset. Despite efforts, the small dataset and model limitations led to incoherent outputs and minimal functional correctness. To overcome these issues, we explored generating synthetic data to expand the training set with explicitly defined memory inefficiencies. This synthetic data highlighted inefficient patterns, enabling targeted fine-tuning. Our evaluation, using EffiBench tasks, revealed insights into the effectiveness of this approach for teaching LLMs to autonomously generate memory-efficient solutions. We further proposed alternative techniques, such as direct preference optimization and instruction tuning, to improve the LLM's response quality without exacerbating data limitations. This study underscores the importance of scalable and domain-specific datasets in advancing the capabilities of LLMs for optimized code generation, paving the way for more efficient and practical software development solutions. The paper also released the code and datasets generated for further usage and future work here: https://github.com/joshiarnav/Memory-Efficient-Code-LLM.

---

1. Introduction/Motivation

Large language models (LLMs) have demonstrated remarkable capabilities in code generation, assisting software developers in automating repetitive tasks, offering intelligent solutions, and accelerating the development cycle. However, while LLM-generated code often meets functional requirements, it frequently overlooks non-functional requirements, such as memory efficiency, leading to suboptimal solutions that may hinder software performance, especially at scale. Addressing these challenges is essential for optimizing the deployment of LLMs in real-world applications where memory efficiency is critical.

The prior work in this domain includes benchmarks like NoFunEval, which evaluates LLM-generated code not only for functional correctness but also for its efficiency, latency, and maintainability (Singhal et al., 2024). EffiBench, another benchmark, focuses specifically on code efficiency, offering a platform for evaluating LLM outputs against human-written canonical solutions Huang et al., 2024). Despite the insights gained from these benchmarks, a significant gap remains in optimizing LLM-generated code for memory efficiency. This project aims to bridge that gap by exploring fine-tuning methods to help LLMs generate memory-efficient code, starting with an experiment involving the NoFunEval dataset and extending the training with synthetic data generation.

2. Problem Statement

The primary goal of this project is to explore and develop methods for improving the memory efficiency of code generated by LLMs. Specifically, we aim to evaluate the effectiveness of fine-tuning an LLM using memory-optimized code examples and synthetic data to enable the model to autonomously generate more memory-efficient code in response to software development tasks.

3. Methodology and Results

Our approach to optimizing memory efficiency in LLM-generated code involved two primary experiments.

3.1 Experiment 1: Fine-tuning using NoFunEval Dataset

We started by attempting to fine-tune a pre-trained LLM using the NoFunEval dataset, which contains code examples and their memory-optimized counterparts. The dataset is structured around non-functional requirements such as memory optimization, with source code and optimized target code serving as training pairs. The fine-tuning process was performed on a Llama-3.2-1B-HF model using LoRA (Low-Rank Adaptation) (Hu et al. 2021), aimed at adapting the model's behavior to improve memory efficiency. The evaluation was carried out using EffiBench, a benchmark focused on code efficiency. The goal was to have the model learn and/or align itself with this specific domain while ignoring everything with a curated dataset which already focused on the goal of the research.

An example from the Resource Utilization branch of the NoFunEval evaluation dataset is shown below. The entire JSON file consists of only 40 rows, with a 30/10 train-validation split, which was determined after data preprocessing and splitting datasets, after investing a lot of time and resources into the dataset, so the experiment was seen through regardless.

Non-Functional Requirement: e.g. "Optimize memory management in Java code."

Commit: e.g. "https://github.com/aaronjwood/PortAuthority/commit/….."

Source Code: e.g. Java code for port scanning and MAC vendor lookup using AsyncTask and database queries.

Target Code: e.g. Optimized code to handle database queries and resource closure (Cursor, Database).

Programming Language: e.g. "Java"

Chain of Thought: e.g. "Focus on optimizing memory and preventing resource leaks (proper cursor closure). "

One-Shot Instruction: e.g. "Optimize memory usage in Java by improving resource management."

Base Prompt: e.g. "Improve Java code by addressing potential memory leaks in port scanning and database operations."

Figure 1: Example of resource_util code pair used for fine-tuning Experiment 1

Results from Experiment 1

The fine-tuning experiment with the NoFunEval dataset did not yield positive results. Despite a slight increase in DiffBLEU scores, the model generated incoherent outputs after a few tokens. Furthermore, none of the outputs were executable, failing the test cases for memory-efficient solutions. This failure can be attributed to several factors: the small size of the dataset, the limitations of the model, and the complexity of transferring memory optimization patterns.

```
null) backOff()\n        else if (taskTail.compareAndSet(tn,
n)) execute(n)\n      } catch {\n        case ex:
InterruptedException => return\n        case ex: Throwable =>
onError(ex)\n      }\n    }\n  }\n\n  private def execute(n:
TaskNode) {\n    n.task.run()\n    n.task = null // to avoid
holding of task reference when queue is empty\n    backOffs =
0\n  }\n\n  private def backOff() {\n    backOffs += 1\n    if
(backOffs < 2) Thread.`yield`()\n    else if
(backrearearearearearearearearearearearearearearearearearearear
earearearearearearearearearearearearearearearearearearearearear
earearearearearearearearearearearearearearearearearearearearear
earearearearearearearearearearearearearearearearearearearearear
earearearearearearearearearearearearearearearearearearearearear
earearearearearearearearearearearearearearearearearearearearear
earearearearearearearearea
```

Figure 2: Example of fine-tuned model's code generation from Experiment 1

Based on this, we proposed 3 methods to improve our model's performance, while keeping in mind the limitation

S. No	**Proposed Follow Up**	**Rationale**	**Limitation**
1	DPO (Direct Preference Optimization)	Optimizes based on learned feedback algorithm, improving the model's output quality in specific contexts.	Neither address data limitations or model overfitting, crucial in our failed fine-tuning attempt.
2	Instruction Tuning	Improves how the model interprets and responds to prompts, refining task-specific behavior.	
3	Few shot Prompting	Adjusts behavior with examples, improving task execution on the fly without changing model weights.	

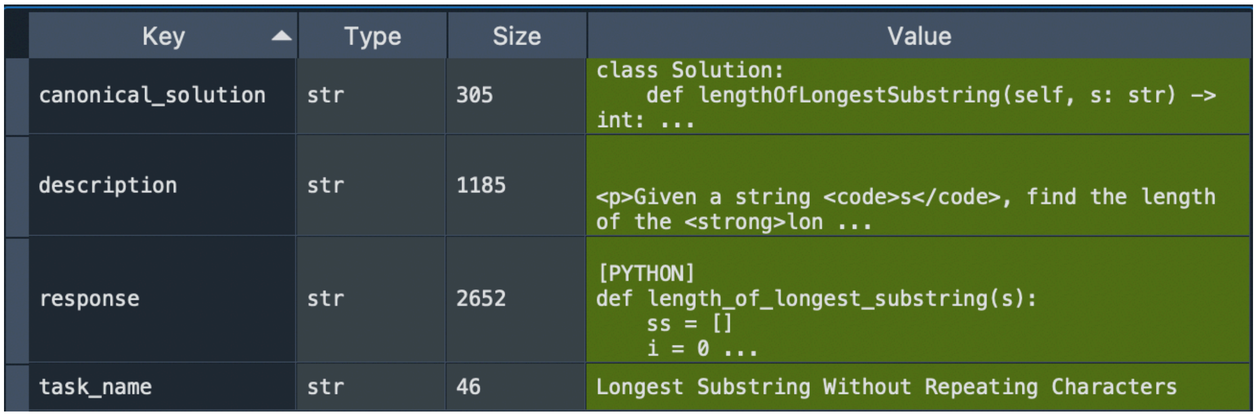
Table 1: Proposed directions for experiment 2

3.2 Experiment 2: Fine-tuning using Synthetic Data

After the results from the initial fine-tuning experiment, the experiment was shifted to generating synthetic data due to the lack of optimal data in the domain. This approach involved creating 1000 examples of memory-inefficient code derived from EffiBench's canonical solutions. These synthetic examples were designed to demonstrate specific memory inefficiencies, providing the model with clear, contrastive examples of inefficient code alongside their efficient counterparts. We hypothesized that training the model on these examples would teach it to generate more memory-efficient code autonomously.

The model used for every component of this experiment was Llama-3.1-8B, a larger version than the one used in Experiment 1. Firstly, it was utilized to gain generate the inefficient solutions which was determined to be relatively optimal through preliminary analyses. Next, this model was utilized for the main generation and fine-tuning experiments, which allowed for better learning of memory efficiency patterns. This was due to a more optimal LoRA setup than the initial model as the newer Llama models do not have fully constructed LoRA setups available as easily due to their novelty. This LoRA setup enabled faster training and inference than before.

The structure of the synthetic data outputs (consisting of canonical_solution, description, response, and task_name fields) is shown below. We also illustrate an example code output that was used for fine-tuning in Experiment 2.



Key	Type	Size	Value

canonical_solution	str	305	class Solution:
			    def lengthOfLongestSubstring(self, s: str) ->
			int: ...

description	str	1185	<p>Given a string <code>s</code>, find the length of the <strong>lon ...

response	str	2652	[PYTHON]
			def length_of_longest_substring(s):
			    ss = []
			    i = 0 ...

task_name	str	46	Longest Substring Without Repeating Characters

Figure 3: Structure of synthetic data outputs

```
 prompt = f"""You are an expert at writing inefficient Python
code. Given a programming problem and its efficient solution,
your task is to write a memory inefficient version of the
solution. The inefficient solution should:

1. Use excessive memory allocation
```

```
2. Still produce the correct output

3. Be significantly less memory efficient than the original


Problem Description:

{description}


Efficient Solution:

{efficient_solution}


Write ONLY the inefficient solution code, without any
explanations:"""
```

Figure 4: Example code output from Llama-2-7B-HF, synthetic data generation (Experiment 2)

We also conducted a preliminary evaluation of how far the synthetic code deviates from canonical solutions in terms of their average memory usage, and if this difference was promising enough to indicate meaningful optimization potential.

Model	**Memory Usage (MB)**
Canonical Solutions	**0.0007658**
Synthetic Inefficient Solutions	**0.0640196**

Table 2: Synthetic pairs of inefficient solutions generated based on Effibench canonical solutions, with difference in memory usage, showing potential for finr-tuning.

We provide a side-by-side comparison of the data pairs fed as input for training, highlighting the explicit differences between canonical and optimized code. Our rationale is that consistently providing the LLM inefficient code and having it model efficient code will teach it to generate more efficient code.

```
"""Below is a programming problem description and an
inefficient solution. Your task is to provide a more efficient
```

```
solution.

### Problem Description:
{}

### Inefficient Solution:
{}

### Efficient Solution:
{}"""

EOS_TOKEN = tokenizer.eos_token
```

Figure 5: Example of input pairs and descriptions fed into our LLM for fine-tuning. The area in brackets is to be replaced with the aforementioned code examples (Effibench and synthetically generated examples).

Results from Experiment 2

Fine-tuning the model on the synthetic data led to a more promising outcome. The model demonstrated a better understanding of memory inefficiencies and generated more coherent code outputs. While some inefficiencies were still present, the improvements were significant compared to the base model (with few-shot and zero-shot prompting). The evaluation on the EffiBench test split showed better functional correctness and fewer errors in memory optimization. Additionally, it proved the experiment successful.

Model	Avg Diff Bleu	Edit Similarity	AST Similarity	Bleu Score	Memory Usage (MB)
Fine-tuned Model II	0.6640	0.6598	0.7833	0.5085	0.000802
Base	0.5596	0.5329	0.7539	0.3904	0.013090

Table 3: Evaluation of LoRA fine-tuning performed on 1000 synthetic data pairs of Effibench canonical solutions and memory inefficient versions of them against a base and fine-tuned Llama-3.1-8B model. All solutions were compared to the ground truth canonical solution from Effibench.

4. Discussion

The results highlight several key insights:

1. **Dataset Limitations**: A small training dataset, as seen in Experiment 1, hinders the model's ability to generalize and learn from code patterns. The limited variety of examples resulted in overfitting and incoherent outputs.
2. **Synthetic Data**: The use of synthetic data significantly improved the model's ability to understand and generate memory-efficient code. The contrast between canonical and inefficient code allowed the model to identify key inefficiency patterns and apply them effectively.
3. **Model Limitations**: Despite the improvements observed with synthetic data, the model still struggled with more complex tasks, suggesting that larger models or further fine-tuning are necessary to fully capture the nuances of memory optimization.
4. **Data Augmentation**: The synthetic data generation approach proved valuable in overcoming the dataset limitations of Experiment 1, highlighting the importance of scalable and domain-specific datasets in training LLMs for non-functional requirements.

The experiment did, however, eventually proved the hypothesis that an LLM can learn, even through synthetically generated code examples, how to improve the memory efficiency of an inefficient code example. The results show that, through fine-tuning on efficient code examples, even those not necessarily focused on the specific domain of memory utilization (but happening to affect memory utilization as a byproduct of efficiency) demonstrate these performance gains in memory utilization.

5. Related Work

Several approaches have been proposed to address memory efficiency in LLM-generated code. EffiLearner (Huang et al.) attempted to optimize LLM-generated code by iteratively refining it based on runtime and memory usage profiles. However, this approach has limitations, as optimizing memory consumption through iterative refinement can lead to lower quality code outputs on functional correctness. Our work builds on this by exploring the use of synthetic data generation and fine-tuning techniques to explicitly teach the model memory optimization strategies.

6. Future Work/Conclusion

In future work, we plan to extend our approach by exploring larger and more diverse datasets, potentially incorporating user feedback and performance metrics to refine the training process. Additionally, we will investigate the potential of combining direct preference optimization (DPO) and instruction tuning to further enhance the LLM's memory efficiency without relying on large datasets.

This study contributes to the growing body of research on optimizing non-functional requirements in code generated by LLMs. It highlights the importance of both dataset quality and model capacity in addressing memory efficiency challenges. With further improvements in dataset scaling, model architectures, and fine-tuning techniques, LLMs could become more adept at generating memory-efficient code autonomously, facilitating more efficient and sustainable software development practices.

7. References

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., & Chen, W. (2021). LORA: Low-Rank adaptation of Large Language Models. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2106.09685

Huang, D., Dai, J., Weng, H., Wu, P., Qing, Y., Zhang, J. M., Cui, H., & Guo, Z. (2024). SOAP: Enhancing efficiency of generated code via Self-Optimization. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2405.15189

Huang, D., Zhang, J. M., Qing, Y., & Cui, H. (2024). EFFIBench: Benchmarking the efficiency of automatically generated code. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2402.02037

Singhal, M., Aggarwal, T., Awasthi, A., Natarajan, N., & Kanade, A. (2024). NoFunEVaL: Funny How code LMs falter on requirements beyond functional correctness. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2401.15963