Improving Compilation and Accuracy for C Leetcode Problems

Asim Baral, Azeez Ishaqui, Rachit Chadha, Rishi Pathak
Georgia Institute of Technology
{abaral7, aishaqui3, rchadha33, rpathak38}@gatech.edu

Abstract

Our project explores the adaptation of transformer models to better understand and solve C programming challenges. Initially, we assess the capabilities of baseline transformer models using a dataset of C programming problems sourced from Leetcode. We then enhance these models through two phases of fine-tuning: the first targets general model improvements, and the second integrates C-specific grammatical structures to guide comprehension and generation processes. Our comprehensive analysis benchmarks each stage of development against the initial baseline, highlighting the effectiveness of fine-tuning and grammar-enhanced fine-tuning. The results demonstrate significant advancements in model compilation performance, providing a promising approach to applying NLP in programming language applications. This work not only furthers the integration of NLP technologies in software development and education but also opens avenues for future research in specialized language model training.

1. Introduction

In the rapidly evolving field of software development, automation and efficiency in coding practices are highly sought after. Natural Language Processing (NLP) holds the potential to significantly impact this area by enabling models to parse, interpret, and solve programming challenges automatically. Such capabilities are invaluable in educational contexts, where they can provide immediate feedback and support to learners, and in professional settings, where they enhance code review processes and facilitate bug detection. However, the unique complexities of programming languages, especially those with intricate syntax like C, pose substantial challenges. These challenges necessitate specialized adaptations in NLP models to effectively understand and engage with code.

Transformers have led to breakthroughs across various NLP tasks, achieving state-of-the-art results in language translation, summarization, and more. Yet, their application to programming languages, especially for statically typed languages and low-level memory accessing languages like C have been limited. Standard transformer models struggle with the syntactic and semantic specifics of programming languages without considerable customization and fine-tuning. Our project is dedicated to enhancing transformer models to adeptly handle C programming tasks. By systematically evaluating baseline models, fine-tuning with targeted data, and incorporating C-specific grammar, we aim to develop proficient models that excel in both code comprehension and problem-solving efficacy.

The primary problem this project addresses is the limited effectiveness of general-purpose transformer models in understanding and solving C programming problems. This issue stems from the distinct structural and syntactical nuances of C, which are not typically well-captured by models trained on general language data. The novel challenge lies in adapting these transformers not only to comprehend C code but also to generate syntactically correct and logically coherent solutions to programming problems.

What makes this project innovative is its dual-phase fine-tuning approach which first optimizes general performance and then specifically targets the grammatical structures of C programming. This method departs from traditional applications of NLP in programming, which have largely focused on more syntactically forgiving languages like Python. By pioneering this tailored approach, the project sets a foundation for future explorations into other complex programming languages and demonstrates a potential pathway for significantly improving the utility of NLP in software development contexts.

2. Related Work

CFG's are an under-utilized tool within coding contexts, due to their stringent nature. SynCode is a framework which utilizes a context-free grammar (CFG) defined for a programming language to guide decoding for LLMs [5]. By constructing an efficient lookup table called the DFA mask store, SynCode ensures that the LLM are able to generate syntactically valid tokens while rejecting invalid ones. Similarly, we utilize C-specific grammatical structures during fine-tuning to improve model's comprehension and generation of C code.

While Syncode reduces syntax errors for Python and Go generation, we focus on C from programming challenges sourced from LeetCode. Through analyzing baseline transformer models and benchmarking performance improvements during grammar-enhanced fine-tuning, we showcase the potential of NLP techniques in solving complex programming tasks.

TinyPy Generator also uses a carefully crafted CFG to generate Python snippets of varying complexity [6]. There are several rules for assignments, arithmetic expressions, conditionals, and loops, ensuring that generated programs are executable, comprehensive, and free of compilation errors. Our project incorporates C-specific grammatical structures to guide the comprehension and generation processes of transformer models.

TinyPy focuses on generating Python code for machine learning applications, like training language models, while we use C programming challenges from LeetCode. We fine-tune models for performance improvements through targeting grammatical structures of C programming, and specialize our model training.

Tarassow employs an LLM similar to GPT-3.5 and evaluates its performance on various programming tasks specific to gretl [3] Similarly, our project utilizes the llama-2-7b-chat-hf model and assesses its capabilities using a dataset of C programming problems. Our studies coincide in benchmark the effectiveness of LLMs in coding contexts that differ from mainstream programming languages. Their findings show LLMs can better understand, write, and improve gretl code, despite the limited availability of resources. It can generate descriptive docstrings, translate between them code, and explain poorly documented code. By establishing the potential of LLMs in low-resource and domain-specific programming contexts, Tarassow's work sets a foundation for our project's exploration of C programming.

3. Dataset

Initially, we could not find any large dataset with skeletons as well as solutions alike for inferencing which included comprehensive and robust C code. Our approach involved the development of a specialized dataset for C programming challenges through curating a diverse selection of LeetCode questions, spanning various difficulty levels, and meticulously extracting both code function skeletons as well as their solutions. This detailed process was crucial for capturing a wide array of programming concepts, data structures, and algorithms, essential for models that understand and generate code.

Our approach ensured high-quality control, with each question being thoroughly reviewed to ensure clarity and relevance to C programming. We had two principal datasets, the inference dataset and our training dataset,

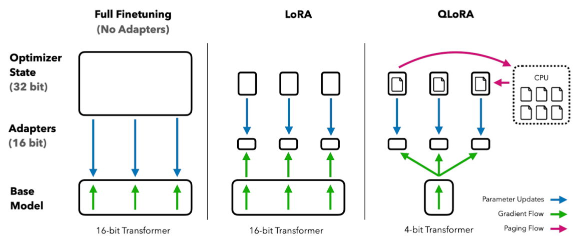

Figure 1. Finetuning methods and their memory requirements adapted from QLoRA [1]

which were 100 problems and 150 problems respectively. The inferencing (test) dataset was collected without solutions, and we scraped the descriptions using Selenium, while obtaining the starter code manually. Problems were randomly sampled for both datasets with the following difficulty distribution: 47% easy, 48% medium, 5% hard. This was vital for preserving the dataset's integrity and relevance to real-world programming scenarios. Our curated datasets are available here.

The skeleton structure of functions combined with their corresponding solutions and descriptions, gave us a solid foundation for training transformer models to generate syntactically accurate and functionally precise C code.

After the initial gathering, we formatted the data in the following format for fine-tuning:

```
<s>
<<SYS>> System Prompt <</SYS>>
[Inst]Prompt + Starter Code[/Inst]
Solution
</s>
```

4. Technical Approach

After the prompt dataset was established, we worked on testing out the baseline model and fine-tuning it. We decided to choose the model NousResearch/llama-2-7b-chat-hf [2], due to its acceptable performance, ease of use, and space-efficient qualities. Due to resource and time costs, we weren't able to fine-tune the raw model. But rather, we quantized the model using guidelines from LoRA [4] and QLoRA [1] to reach similar performance as the original, as claimed by these papers, while using significantly less computation and memory.

To quantize the models we found that a LoRA attention dimension of 16, a LoRA alpha of 16, and LoRA dropout probability of 0.1 was the most efficient in terms of performance and due to the constrained memory of a single GPU. We also used 8-bit quantization and a compute type of torch.bfloat16, instead of 4-bit because 8-bit was generally more accurate, while not exacerbating runtime. We rigorously tested this initial baseline model and trained our dataset on the quantized model for 5 epochs with a batch size of 38 prompts for fine-tuning.

Using the fine-tuned model, we created an additional model which added a CFG using the Outlines library. The purpose of the CFG is to define the language syntax. When this grammar is taken in by the Outline's generate function, it changes how the logits of the model output are processed to ensure that it satisfies the C grammar. As the Outlines library didn't include a C grammar, we decided to make it ourselves. This specific grammar included rules about how the complex C syntax should be written.

Finally, using the initial test dataset, we ran the prompts on all three models, the regular baseline model before fine-tuning, the fine-tuned model and the fine-tuned model with CFG. We first ran the 100 prompts on the baseline model and after further analysis we realized that compilation was increasingly difficult with medium and hard questions, with steeper logic and concepts than easy questions. In these generations for easy questions, most errors were due to syntax and at compile-time, less due to logic. For medium and hard questions, a majority of solutions failed due to incorrect logic as well as incorrect syntax. We followed this approach closely with just easy questions as fine-tuning and CFGs don't improve logic, but focus on language syntax.

After we evaluated the prompts on each of the models, we went through and ran the solutions manually on Leetcode. We recorded the number of correct cases versus total cases, and also noted if the solutions didn't compile with specific technical excerpts.

5. Experiments and Results

Our key evaluation metric was compilation rate. The most prevalent issue throughout our three principal models was predicated on general syntax issues or tiny function calling errors which completely shut down runtime. From a small malloc line, to a misplaced pointer, or even to invalid method calling these issues completely trumped the training process as they were completely unable to evaluate.

5.1. Key Observations

In general, C language specifics and incorrect syntax seemed to be the root cause of the majority of these compilation errors. Code snippets would use something like strcmp() for integer comparisons rather than the standard comparison operators ($e.g., ==, <, >$).

Instantiation and pointers were another constant cause of errors, either from classic syntax issues or dereferencing. For example, a snippet would generate perm = arr; rather than copying arr contents to perm using a loop or memcpy(). Memory allocation is inherently linked to instantiation within C contexts, often times generated code didn't properly calculate the size of the allocated memory, like using sizeof(char*) instead of sizeof(char) when allocating for a character array. This also came with errors from failing to properly deallocate memory with malloc() using free(), which can cause memory leaks.

Table 1. Condensed Analysis of Code Errors in Models

Category	Fine-tuned Model	Baseline Model
Syntax	"Word Pattern": Incorrect if-else usage (`else if (pattern[i] == 'A')`).	"Excel Column Title": Incorrect string literals (`return ""A""`).
Variables	"Kth Largest Element": Undefined `add()` in `KthLargest`.	"Get Max in Generated Array": Uses undefined `nums` array.
Type Definitions	"Design an Ordered Stream": Structs `ListNode` not *completely* defined.	"Height Checker": Uses undeclared `expected` array.
Syntax Correctness	Correct syntax with func calls and `malloc` in "String from Binary Tree".	Many errors, e.g., undefined `MIN` macro in "Min Depth of Binary Tree".
Logic	Correct looping and comparison logic in "Pattern of Length M", barely incomplete.	Incoherent structures, just returns complete subtree max/min.
Memory Management	Memory leaks in "Construct String from Binary Tree" (no `free()`).	Similar issue in "Positions of Large Groups" (memory not freed).
Error Handling	Missing NULL checks in "Construct String from Binary Tree".	Also missing in "Min Depth of Binary Tree", causing runtime errors.

5.2. Baseline Model

For the baseline model, we found a lot of errors due to incorrect syntax as seen in Figure 2. It seem that the original method wasn't trained extensively on C syntax. We weren't able to test about half of the solutions from the baseline model due to compilation errors. After analyzing these errors it was clear that it wasn't just because of incorrect logic, but also due to overall incorrect syntax.

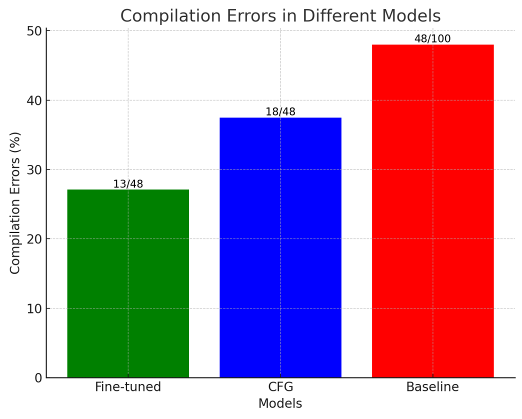

Figure 2. Model Compilation Rates

5.3. Fine-Tuned Model

With the fine-tuned model, we found that it significantly improved the output compared to the baseline model. Most of these solutions contained correct syntax. As seen in Figure 2 , only 27% of the fine-tuned models didn't compile, which is significantly better than the baseline results of 48%. We were able to achieve these metrics with only 150 different Leetcode questions. The large increase from 27% to 48% indicates that our method was effective in changing how the syntax of C is perceived by the LLM.

5.4. Fine-Tuned Model W/ CFG

The fine-tuned model with CFG showed more compilation errors than the our fine-tuned model 2. We were initially surprised by this result as we believed that by using a C grammar, we could force the LLM to have correct syntactical code. After further analysis, we learned that this was still the case. The code was syntactically correct but the logic that the original fine-tuned model had got lost to the point where there were more compilation errors. It would call functions that weren't previously written, return the wrong type of values (int instead of int*) and etc. Also, there were errors with quotation marks. Instead of putting quotes around the number 1, as "1" it would add additional quotes leading to ""1"". We tried to fix this by adjusting our grammar but we were unable to fix this part as Outlines wouldn't be able to process our solution.

6. Limitations and Constraints

One of the limitations of our study is the lack of high-performance GPUs due to resource constraints. Also, due to time constraints, we were unable to fine-tune the model on a larger dataset. But nevertheless, we do believe that our work is significant because we were able to show significant improvement in the model with fine-tuning with a very low number of data points.

Through the reliance on a simplified CFG for the C during fine-tuning. The generated CFG captures the essential syntactic structures of C, and it may not encompass all the nuances and intricacies. This simplification could potentially limit the models' ability to handle more advanced or estranged C programming constructs.

7. Conclusion and Future Work

In conclusion, by curating a diverse and comprehensive dataset from LeetCode and employing a dual-phase fine-tuning process, we've demonstrated that transformers can be effectively adapted to understand and generate C code. We have found that fine-tuning a model using even a few data points can significantly improve the model's performance. When adding a CFG grammar to the model, it can lead to an improvement of syntax, but not necessarily an improvement of code logic as seen in Figure 2.

While our manually curated dataset provides a rich set of C programming challenges, it may not cover all possible variations and edge cases that exist in real-world codebases. One future work could involve expanding the dataset by incorporating C programming challenges from other sources beyond LeetCode, such as open-source codebases, programming competitions, and academic coursework. This expansion would expose the models to a wider variety of coding styles, design patterns, and problem domains, enhancing their generalization and inferencing capabilities. Also, Refining the C grammar is also something that needs to be done. We made the C grammar ourselves, but it wasn't comprehensive due to restrictions from the Outlines library.

References

[1] A. Author and B. Collaborator. Title of the paper. *arXiv e-prints*, page arXiv:2305.14314, May 2023. 2

[2] NousResearch. Llama-2-7b chat model. Hugging Face Model Repository, 2023. 2

[3] Artur Tarassow. The potential of LLMs for coding with low-resource and domain-specific programming languages. *arXiv e-prints*, page arXiv:2307.13018, July 2023. 2

[4] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Are Pre-trained Convolutions Better than Pre-trained Transformers? *arXiv e-prints*, page arXiv:2106.09685, June 2021. 2

[5] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Improving llm code generation with grammar augmentation, 2024. 1

[6] Kamel Yamani, Marwa Naïr, and Riyadh Baghdadi. Automatic generation of python programs using context-free grammars, 2024. 2