Using Large Language Models to Test Deep Learning Libraries

Jack Rowlands
University of Adelaide
Student
jack.rowlands@student.adelaide.edu.au

Dr Sean (Xiaogang) Zhu
University of Adelaide
Supervisor
xiaogang.zhu@adelaide.edu.au

ABSTRACT

Fuzzing deep learning libraries is crucial for identifying errors and ensuring their reliability, yet automating this process for complex systems remains challenging due to intricate API dependencies and semantic constraints of deep learning libraries. Traditional fuzzing techniques often result in low-quality or invalid test cases for such libraries. This study explores the use of large language models (LLMs) to generate syntactically and semantically correct test programs and parameter sets, enabling automated testing of deep learning libraries like PyTorch. LLMs have demonstrated remarkable capabilities in understanding and generating code, suggesting their potential as powerful tools for generating programs and mutating values of the parameters intelligently to explore the vast state space of these libraries across both CPU and GPU environments.

Results indicate high potential for LLM-based fuzz testing, with substantial challenges regarding the complex API dependencies and mathematical constraints of deep learning libraries. With a successful execution rate of 32.54%, a significant portion of tests encountered execution errors, indicating areas that require improvement in error handling, domain-specific LLMs, and prompt engineering. Although substantial developments are required to achieve reliable, automated fuzz testing, the results highlight the opportunity of LLMs to significantly reduce the manual effort required to create fuzz drivers. In addition to offering suggestions for future research directions targeted at improving test coverage, accuracy, and reliability, this study demonstrates the benefits and drawbacks of employing LLMs for testing in deep learning contexts.

1 INTRODUCTION AND MOTIVATION

Ensuring the reliability and correctness of deep learning libraries such as PyTorch [6] and TensorFlow [5] is crucial as these libraries form the backbone of many machine learning applications such as facial recognition systems, recommendation engines, fraud detection systems, and medical systems, which have a high impact on various aspects of daily life. The reliability and correctness of these libraries directly affects the safety and efficacy of the technology we increasingly rely upon. Any undetected bugs or vulnerabilities in these libraries could potentially lead to catastrophic consequences, ranging from privacy breaches to significant errors in critical systems.

Fuzzing, is a dynamic testing technique which involves providing generated inputs to a system and then testing the correctness of the output. While fuzzing has proven effective for many software systems, traditional fuzzing approaches have limitations with the unique challenges posed by deep learning libraries, creating a significant gap in the ability to effectively test these complex systems. These libraries have intricate interconnected API dependencies, complex data types such as multi-dimensional tensors, and require semantically correct inputs to trigger various code paths.

The two main traditional fuzzing techniques struggle to generate valid inputs that can effectively explore the vast state space of these libraries. This leads to low code coverage and difficulty in uncovering subtle bugs. Grammar-based fuzzers, while highly effective at generating structured valid inputs, struggle with the dynamic aspect of deep learning and cannot efficiently explore the full code. Mutation-based fuzzers on the other hand often fail to preserve the validity of the inputs, resulting in a high number of invalid test cases which waste computational resources. These limitations result in low code coverage, missed edge cases, and undetected errors.

Furthermore, the creation of programs that invoke APIs, also called fuzz drivers, for deep learning libraries typically require substantial manual effort with domain expertise in deep learning. The developer must understand the architecture, how the APIs are used, and the potential edge cases. The manual process of these drivers is time consuming, error-prone, and often struggles to keep up with the fast-paced development cycles of modern deep learning libraries, which stay on the breaking edge of the field and have to be highly optimised.

All these challenges leave a significant gap in the field of software testing for deep learning libraries. There is a need for more efficient automated approaches that can generate high-quality fuzz drivers and parameters capable of fully testing deep learning libraries. These approaches must be able to fully understand the architecture, create diverse and valid parameters, and reduce the manual effort required to test. They must also be adaptable to diverse new innovations in deep learning and able to quickly pick up new architectures and optimisations.

In recent years, the advent of Large Language Models (LLMs) has opened up new possibilities in many different domains of computer science, including code generation and understanding. LLMs, such as GPT-4o, Claude 3.5 Sonnet, and Llama 3.1 [1, 2, 3], have demonstrated remarkable capabilities in understanding and generating complex code across many languages and domains. These models are trained on a large variety of code and natural language and have shown abilities to understand complex code and generate valid code snippets.

The use of LLMs could significantly reduce the manual effort required in creating fuzz drivers, being used to create code snippets that are both syntactically correct and semantically meaningful. It can also allow for developers with less domain expertise to contribute to testing more effectively. Moreover, LLMs' ability to understand and generate code based on natural language can be leveraged to create more intelligent mutation strategies. Instead of generating random mutations, a LLM could make more informed decisions on how to modify the parameters, considering the semantic requirements and common usage patterns of the APIs being tested. This would lead to more meaningful and effective test cases which would lead to an increase in exploration of the libraries' state space. [12]

However, there are some potential challenges with the application of LLMs to fuzzing. One significant concern is the potential for LLMs to introduce biases into the fuzzing process. [9] Pretrained LLMs rely on vast amounts of historical training data to function effectively. [10] However, this data may not include sufficient examples of certain specific APIs, especially those that are newer or less commonly used. For entirely new code, the training data may lack any relevant instances. These limitations could affect the model's ability to generate effective test cases for scenarios involving such APIs or new code, potentially leading to incomplete or inaccurate test coverage. Additionally, LLMs require a large amount of computational resources, which could increase cost and time.

Another important consideration is the need for prompt engineering for the specific task of fuzzing deep learning libraries. [11] The creation of fuzz drivers and parameters will require sophisticated prompting techniques, with a careful balance about the amount of context provided, as cost and latency increase as the length of the context given to the LLM increases.

Despite these potential challenges, the potential benefits of using LLMs for fuzzing deep learning libraries is significant enough to warrant serious investigation. This approach could potentially lead to more comprehensive testing of these critical systems. It could also potentially reduce the time and domain expertise required to effectively fuzz complex libraries. These insights could also have broader implications for software testing. The techniques discovered could be adapted to other domains of software development and other libraries.

In conclusion, the application of LLMs to the task of fuzzing deep learning libraries represents a promising new potential technique to address critical gaps in the current traditional testing methodologies. By using the code understanding and generation capabilities of LLMs, there is potential to develop more effective and efficient automated fuzzing techniques.

2 LITERATURE REVIEW

The application of LLMs to fuzz testing is rapidly evolving, with many recent significant advancements. Two pivotal studies within this area are: "White-box Compiler Fuzzing Empowered by Large Language Models" by Yang et al., and "How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation" by Zhang et al.

2.1 White-box Compiler Fuzzing Empowered by Large Language Models [8]

Yang et al. introduce WhiteFox, a white-box compiler fuzzer using LLMs. WhiteFox uses a dual model framework which consists of an analysis LLM, which looks at the compiler source code for optimisation triggers and produces requirements for a high-level program which triggers this optimisation, and a generation LLM which generates the test project based on these requirements.

This approach addresses limitations of existing fuzzing approaches that often struggle to generate test programs that can trigger complicated interconnected optimisations. WhiteFox was able to generate test programs with up to eight times more optimisations compared to state-of-the-art fuzzers and discover numerous previously unknown bugs.

However, the authors note some challenges and limitations with their approach. One limitation they had with this approach was the need for careful prompt engineering to guide the LLMs effectively. When creating their prompts, they decided to use few shot in-context learning. [13] This technique uses examples given within the context to extrapolate to perform the given task. Additionally, the cost of the LLM was a limitation, with it being unreasonable to use GPT-4 for code generation as with the feedback loop, the context grew large with many calls. However, they mostly overcame this by using a cheaper LLM specialised on code generation to continuously generate tests efficiently.

2.2 How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation [7]

Zhang et al. present the first in-depth study targeting the issues of using LLMs to generate effective fuzz drivers. This work provides valuable insights into the practical applications of LLMs for generating fuzz drivers, evaluating different prompting strategies, models, and temperature settings. The authors used the generated LLM drivers and compared them to drivers used in industry.

The problem that the authors sought to overcome with this paper was that existing traditional program analysis-based generators

are not generalised and struggle to adapt to different APIs, while also being difficult to understand to human reviewers. To address this problem, the authors used an LLM-based approach for generating fuzz programs. To discover the optimal method to generate the best drivers, the authors used six different prompting strategies designed to explore various aspects of LLM-based generation across five state-of-the-art LLMs, including both closed-source and open-source.
To evaluate the strategies, the authors used a dataset of 86 fuzz driver generation questions from 30 widely used C projects, then compared the LLM-generated drivers against industry drivers.
The study found that LLM-based fuzz driver generation demonstrates strong potential, with the optimal configuration resolving 91% of the test questions and the top twenty configurations resolving at least half of the questions. The effectiveness of the LLM-based generation is heavily influenced by the prompting strategy, model choice, and temperature setting. This is especially evident in the question resolve rate increasing from 10% to 91% when using the optimal configuration compared to the naive strategy.

A major challenge identified is the LLMs struggle to generate fuzz drivers which require complex API usage specifics. To minimise this limitation, they identified three useful techniques, repeatedly calling the LLM, calling the LLM with extended information such as code examples, and iteratively calling the LLM with error feedback. However, these techniques significantly increased token costs. Another limitation they identified was 34% of the APIs required manual validation to ensure semantic correctness of the generated drivers, due to false positives.

Comparing LLM-generated drivers with those used within industry, it was found that LLM-generated drivers can produce comparable fuzzing outcomes to manually written drivers. However, LLMs tend to generate drivers with minimal API usage, which leaves room for improvement.
This study provides a comprehensive understanding of the current state of LLM-based fuzz driver generation.

2.3 Overall Review

Both studies demonstrate the significant potential of using LLMs for fuzz testing, showing that LLMs can generate high-quality test programs and fuzz drivers which are competitive with manually written programs and drivers. However, there were several unresolved issues within these studies, with areas for future research. Both studies highlighted the challenge of ensuring semantic correctness within LLM-generated code. While LLMs excel in generating syntactically correct code, it struggles with generating code which requires complex API dependencies or specific execution contexts.

The use of LLMs can incur significant costs, especially if using feedback loops or few-shot techniques. Finding how to optimise this cost while maintaining the same quality is an important challenge to solve. Both of the studies emphasised the importance of effective prompt engineering. Developing standardised, effective prompting strategies for different fuzzing and testing scenarios is an open research question.

LLMs may struggle with APIs or code patterns that are underrepresented in their training data. Developing techniques to improve LLM performance in such cases is crucial for deep learning libraries, as innovations within this domain are rapid. Developing automated techniques to validate and refine LLM-generated fuzz drivers could significantly enhance the practicality of these approaches.

3 Methodology

The project leverages LLMs to automate the fuzz testing process for PyTorch APIs. The methodology consists of four main components: test program generation, parameter generation, execution of test programs, and result analysis.

3.1 Test Program Generation

An LLM was used to generate test programs for specified PyTorch API functions. To ensure the generated code was both syntactically correct and semantically meaningful, detailed prompt engineering techniques were employed.

3.1.1 Few-Shot Learning [13,14]

Few-shot learning is a prompt engineering technique that consists of including a set of high-quality examples, with each consisting of the input and desired output. This technique helps the LLM understand the expected format and style of the output, leading to more consistent and correct code generation.

3.1.2 Chain of Thought [15]

Chain-of-Thought (CoT) prompting is a technique for eliciting complex multi-step reasoning, which was implemented through the use of structured step by step instructions, and high-quality examples. The CoT process was limited to the beginning of the prompt, using both XML tags and Markdown code blocks to be kept separated, to ensure the correctness of the extraction of the code. There were four sections of the reasoning; firstly, the LLM had to identify the function name and the return type of the function, helping the LLM understand which function is to be tested, and to ensure that the outputs could be extracted out of the environment. Secondly, it had to consider the specific requirements of this function and the potential edge cases. Then, the LLM had to plan out the structure of the test program. Finally, the LLM could write the program code, using all of the above information to improve its quality.

3.1.2 Prompt Caching [18]

To reduce costs, improve efficiency, and decrease latency, prompt caching was also implemented. By structuring the prompts in a specific format where the static components, such

as the introductory system message, CoT analysis structure and example outputs, were at the beginning of the prompt, the same initial content could be reused throughout the different LLM API calls. Only the dynamic part - the specific API signature - was appended at the end of each prompt, significantly cutting the price per API call.

Overall, these approaches provided the LLM with a structured way to break down the problem and high quality reasoning chains to emulate, while not increasing prices significantly.

3.2 Parameter Generation

For each test program, the LLM generated sets of parameters, which were used for the fuzz testing.

3.2.1 Structured Output [17]

To generate the parameter sets, a dynamic JSON schema was used in conjunction with the Structured Output mode in the OpenAI API. This mode constrains the OpenAI models to match the supplied schema. This ensures that the LLM generates the correct number of parameters per set, and the correct number of sets, as well as ensuring correct parsing of the outputs. A limitation of this mode is that it enforces a limit of the amount of JSON entries to a total of 100, limiting the amount of sets of parameters per LLM message. To overcome this limitation, the JSON schema was dynamically created to include the maximum number of sets allowed, as each program had a different number of parameters per set. The amount of parameter sets per PyTorch API is calculated using the following equation:

$$\text{Number of Parameter Sets} = \left\lfloor \frac{100}{\text{Number of Parameters} + 1} \right\rfloor$$

The benefits of generating all of the sets in a single LLM call were the prevention of repeat sets, a lower cost, and lower latency. Additionally, these JSON schemas were included in the prompt caching, reducing cost and latency.

3.3 Execution of Test Programs

The generated test programs were executed with the generated parameters under controlled conditions.

To ensure safe execution, both memory limits and execution timeouts were used. Using Python's `resource` module, the memory usage was capped per test environment. This was key due to the LLM generated parameters potentially creating extremely large tensors. Execution timeouts were also implemented to terminate tests exceeding a set time limit, preventing potential hangs.

Tests were executed in parallel using a thread pool to optimise system resource utilisation and reduce runtime. This multithreaded execution enabled efficient handling of a large number of test cases.

3.4 Result Analysis

Results were systematically collected, analysed, and logged, including comparisons of CPU and GPU outputs using `torch.allclose`, which is a method to compare PyTorch tensors, with customisable tolerances.

4 Experimental Setup

The experimental setup facilitated the automated generation and execution of fuzz tests for PyTorch APIs.

4.1 Software Environment

The testing framework was implemented using Python, with PyTorch facilitating tensor operations on both CPU and GPU. The code was organised into three main modules, `programs.py`, `parameters.py`, and `run.py`, managing the generation or loading of test programs, parameter sets, and executing the test programs respectively. The OpenAI API was used to interact with the LLM for code and parameter generation. `GPT-4o-mini` was used as the LLM generating both the programs and parameters. It was chosen as it was the quickest and most cost efficient LLM with the Structured Output mode and the capabilities to create syntactically correct Python programs consistently. The default parameters were used for the LLM, as they ensure a balance between creativity and correctness.

4.2 Hardware Configuration

The experiments were conducted on a system equipped with both CPU and GPU capabilities to facilitate output comparisons. Additionally, at least 350GB of storage is required.

4.3 PyTorch APIs

A list of 1,583 different PyTorch API definitions was sourced from the TitanFuzz repository. [4] All of these APIs were implemented in the CPU and CUDA systems, and the definition contained their full path and name, all parameters, both optional and required, as well as the default values for the parameters. The additional information assisted with the LLM program generation.

4.4 Execution Strategy

Test programs were generated for each API, and multiple parameter sets were created to explore different input conditions. Multithreading was used to parallelize LLM generation and the execution of tests, improving efficiency and reducing overall runtime. Each test was executed within their own test environment, with a timeout of 90 seconds and a maximum of 4GB of memory to ensure the tests did not exceed resource constraints.

To extract the results from their environments, the CPU and GPU outputs were printed to the environments standard output. These outputs were then compared using `torch.allclose` with relative tolerance set to 1e-2, and the absolute tolerance set to 1e-3, with the comparison and outputs logged. Additionally, the errors for each test program were logged.

4.5 Evaluation Metrics

The evaluation metrics used in this study provide a comprehensive picture of the fuzz testing framework's

effectiveness, robustness, and coverage in testing PyTorch APIs, using a systematic approach to understand how successfully each API performs under various test conditions. First, the metric of *Total Runs* represents the overall testing scale, quantifying the total number of test cases executed across all combinations of APIs and parameter sets. This metric is constrained by the limitation of the Structured Outputs method. This baseline allows us to measure the overall reliability by looking at two primary outcomes: the *Success Rate* and *Mismatch Rate.* The *Success Rate* highlights the proportion of tests where the CPU and GPU outputs match, with no errors, confirming the consistency of computation across hardware. Meanwhile, the *Mismatch Rate* identifies where CPU and GPU results diverge, with no errors, highlighting potential computational inconsistencies and implementation errors which require deeper investigation.

Alongside these, the *Error Rate* metric measures the occurrences of execution failures, pinpointing the programs or parameters most prone to errors. This error analysis goes further through *Error Categorization,* which classifies the errors into types such as `RuntimeError`, `TypeError`, `ValueError`, and `Timeout`. By identifying these error types, it is possible to observe patterns and identify where specific issues tend to occur, which is crucial for refining the testing framework.

A metric which investigates each individual PyTorch API complexity is the *Number of Parameters per API*. This records how many input parameters each PyTorch API takes, which is an approximate proxy for the API's complexity. For additional depth, *Number of Parameters per Set* measures the number of input parameters each test program takes, approximating a proxy for the test program complexity. Moreover, to consider the diversity of the parameter data generated, the *Parameter Types* were recorded.

5 Results

A total of 29,140 tests were executed across 1,583 generated test programs targeting 1,583 different PyTorch APIs over the course of 13 hours.

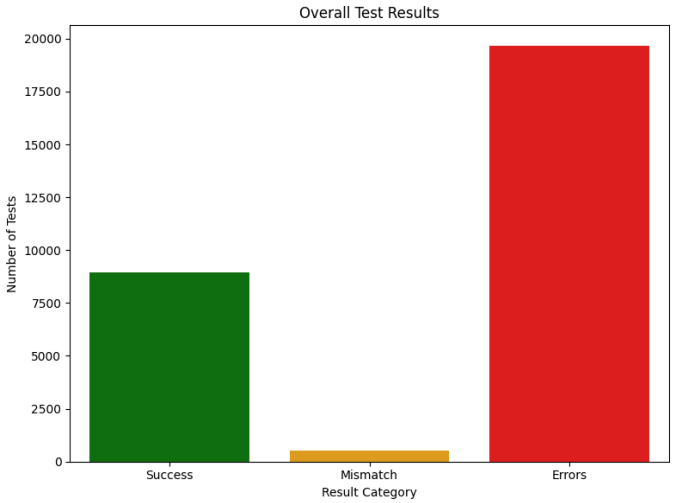

Figure 1: Overall Test Results

As shown in Figure 1, the majority of the tests executed resulted in errors (67.46%), with only 32.54% of tests executing without failure. This highlights a significant issue with the framework, considering over two thirds of the completed tests failed to execute, and therefore were unable to detect errors between the CPU and GPU implementations.

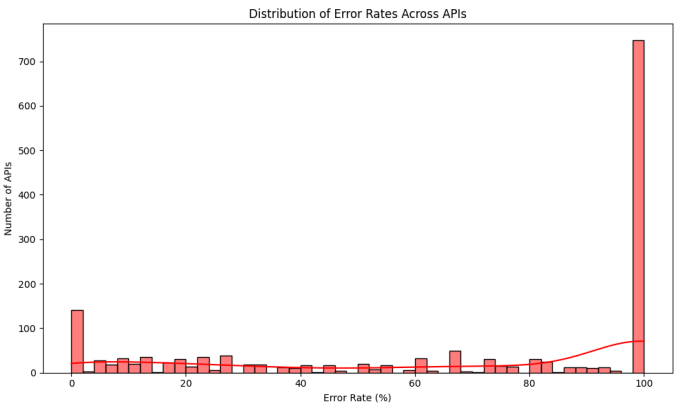

Figure 2: Distribution of Error Rates Across APIs

The vast majority of these errors were caused by the program generation, as evidenced by Figure 2. There was a significant outlier at the 100% error rate, indicating that most errors came from programs which did not successfully execute for any parameter provided.

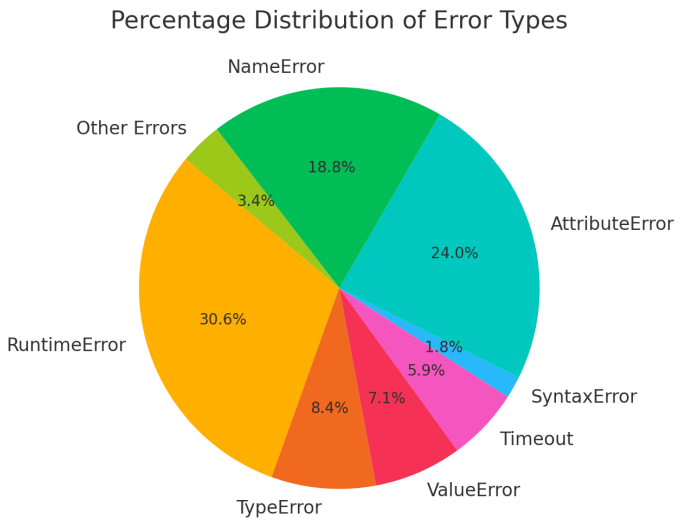

Figure 3: Percentage Distribution of Error Types

Figure 3 demonstrates the breakdown of error type encountered during testing, with the most frequent issues being `RuntimeError` at 30.6% and `AttributeError` at 24%. `RuntimeError` occurs when the code is syntactically correct but encounters an error during execution, predominantly caused by incorrect parameters. `AttributeError` was mainly caused by a variable or function not having a `.cpu()` or `.cuda()` attribute, which occurred when the program generation unnecessarily included it on a variable or function where it was not required. `NameError`, which occurred in 18.8% of the errors, was commonly caused by the program generation not referring to the output variables by the correct name. `SyntaxError` only occurred in 1.8% of the errors, which is a significant improvement over previous prototypes in this project. This improvement is attributed to the Structured Output mode, as manual parameter extraction previously led to structural errors in approximately 20% of generations.

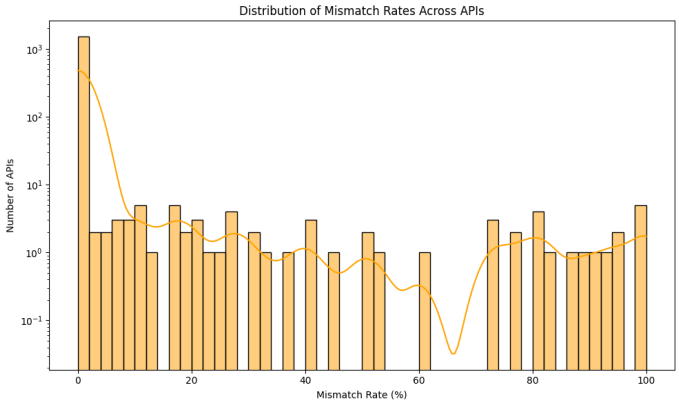

Figure 4: Distribution of Mismatch Rates Across APIs

Only 1.8% of the tests executed resulted in a mismatch between the CPU and GPU results. Figure 4 shows that a high majority of the PyTorch APIs were consistent between GPU and CPU implementations, with a significant spike at the 0% mismatch rate. Between a 5% and 40% mismatch rate, there is a moderate frequency of APIs. This range could represent APIs that are mostly reliable but exhibit discrepancies under specific conditions. These APIs might be particularly promising to investigate further, as they show inconsistencies under certain conditions. Within the higher mismatch rate range, there exists a smaller subset of APIs. This could suggest either severely erroneous PyTorch APIs or errors within the program generation, with the program not completing the same operations between the CPU and GPU sections.

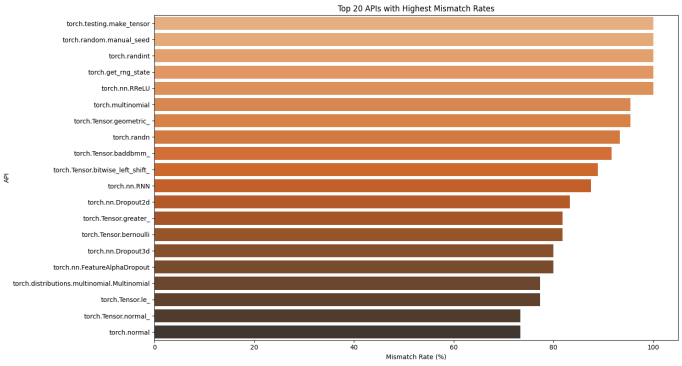

Figure 5: Top 20 APIs with Highest Mismatch Rates

This issue is explicitly seen in Figure 5, which shows the APIs with the highest mismatch rates. The top 5 APIs are explicitly related to randomness. Therefore, it is expected that the CPU and GPU answers should not return the same values.

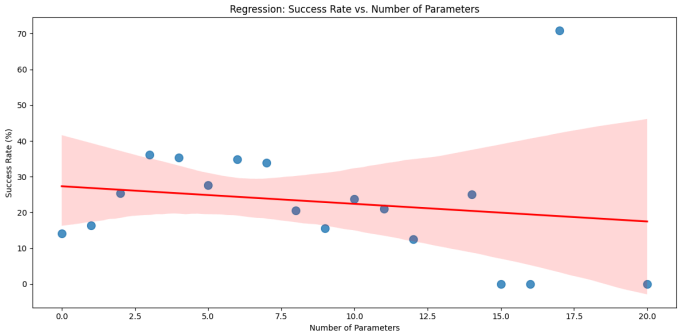

Figure 6: Regression: Success Rate vs. Number of Parameters

Figure 6 clearly shows a slight negative correlation between the success rate and the number of parameters of the API. As stated before, the number of parameters of the API is an approximate proxy of the complexity of the API, so this result is expected.

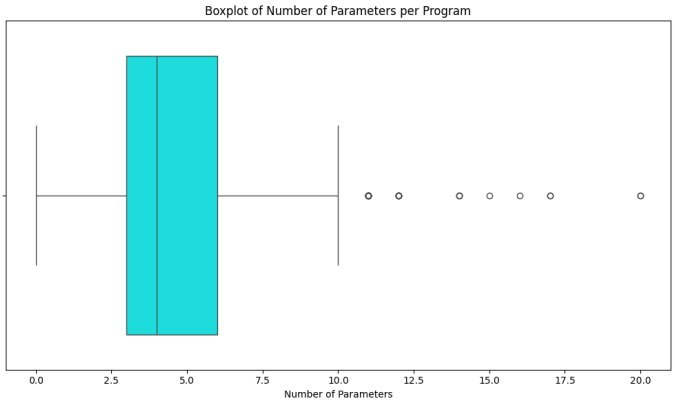

Figure 7: Boxplot of Number of Parameters per Program

The majority of programs have between 2.5 and 6 parameters, as evidenced by Figure 7, which corresponds to the section with the highest success rate in Figure 6. Therefore, there may be some benefit of setting a limit to the number of parameters per program, as most programs will not be constrained, and the overall success rate would increase.

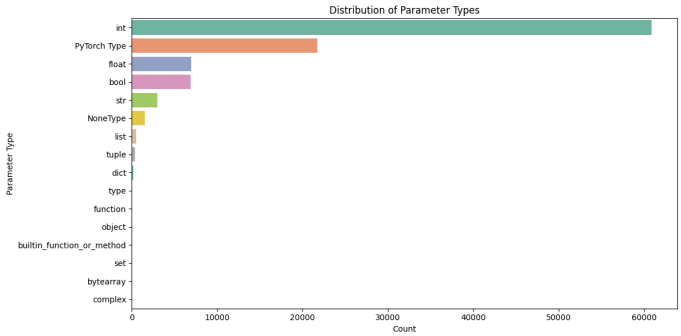

Figure 8: Distribution of Parameter Types

As illustrated in Figure 8, a majority of parameters are of the `int` type, followed by PyTorch specific types, then other fundamental types. Complex types such as `dict`, `tuple`, and `list` are less frequent, revealing a limitation in the parameter generation. More complex and less common types are more likely to uncover errors, as they utilise less frequently tested code paths.

6. Discussion

This study examined the use of Large Language Models (LLMs) to automate fuzz testing for PyTorch APIs, aiming to address the challenges of deep learning libraries' intricate dependencies and complex parameter requirements. A total of 29,140 tests were conducted across 1,583 PyTorch APIs, by generating test programs and parameter sets with the LLM GPT-4o-mini, allowing for insights into the strengths and limitations of this framework.

6.1 Key Findings and Significance

The results of these tests revealed both promising outcomes and limitations of LLM-generated tests. Most notably, a significantly high error rate was observed, with only 32.54% of the tests completing successfully. This suggests that generating syntactically and semantically code and parameters for complex deep learning API calls remains challenging. Analysis of these errors showed that `RuntimeError` and `AttributeError` were the most common errors, which often were caused by incorrect parameters and incorrect code respectively. Among successfully executed tests, CPU and GPU outputs were largely consistent, suggesting a small level of errors within the CUDA and CPU implementations of PyTorch. A slight negative correlation between the number of API parameters and test success rates further emphasised the difficulty in handling complex parameterised API calls, where each additional parameter introduced further potential for misconfiguration and errors.

6.2 Comparison with Previous Studies

These findings align with earlier studies by Yang et al. and Zhang et al., who also investigated LLM for software fuzz testing in different contexts. Yang et al. reported up to eight times more optimisations in compiler fuzzing by using an LLM-based framework, while Zhang et al found that LLM-generated fuzz drivers could resolve up to 91% of test questions with optimal configurations. Despite these achievements, this study faced a higher error rate in test execution, which may have been caused for a multitude of reasons. A major difference between this study and other studies performed on LLM assisted fuzz testing is that in other studies, the parameter generation is assisted with traditional techniques as well. In TitanFuzz by Y. Deng et al. [4], the parameter generation was a combination of traditional parameter mutation and LLMs, whereas in this study, the parameters are only generated by LLMs. Additionally, testing deep learning libraries involves unique challenges, due to the intricate and layered dependencies of each function, requiring a large level of domain-specific knowledge.

6.3 Implications for the Field

Despite the high error rate, this approach holds significant implications for software testing, especially in automating test generation for complex libraries. The LLM was able to generate correct and novel drivers for approximately half of all APIs supplied to it, with no human intervention needed. By automating the creation of the fuzz drivers, LLMs could significantly reduce the manual effort required in testing. The ability of LLMs to generate diverse test cases using logical reasoning has potential to test uncommon edge cases, which may not be covered by traditional testing methods.

6.4 Limitations and Further Potential Extensions of the Study

There were several limiting factors which impacted the effectiveness of LLM-generated fuzz tests in this study. One of the most significant limitations of this study was the use of

`GPT-4o-mini`. This model is the most cost effective model which supports all of the necessary features, however it is significantly weaker at coding and reasoning then other LLMs.[16] By utilising other LLM such as `GPT-4o`, `o1`, or `Sonnet-3.5`, it is highly likely the error rate of this framework would decrease significantly. Additionally, by including feedback loops similar to what is implemented by Yang et al., it would be expected that the majority of the errors would be resolved automatically, however this would increase the cost tremendously. The resource constraints, including the memory limit of 4 GB and execution timeouts of 90 seconds, also contributed to the amount of errors. Additionally further validation steps and error-handling measures are necessary to identify and mitigate issues introduced by test generation rather than genuine PyTorch errors.

7. Conclusion

This studies exploration into using LLMs for fuzz testing deep learning libraries like PyTorch demonstrates a set of promising directions and important challenges. The automation provided by LLM offers a pathway to more efficient and extensive testing. However, current limitations in both code and parameter generation accuracy and reliability must be addressed. Overall, continued research and development within this area is necessary to realise the full potential of LLMs in enhancing the robustness and reliability of critical software libraries in the field of artificial intelligence.

Link to Github Release:

https://github.com/Topics-Group-AI-For-Fuzzing-AI/topics-fuzzing-ai/releases/tag/Paper

REFERENCES

[1] OpenAI, "GPT-4 Technical Report," arXiv:2303.08774 [cs.CL], 2023.
[2] LlamaTeam, "The Llama 3 Herd of Models," Jul. 23, 2024. [Online]. Available: https://llama.meta.com/
[3] Anthropic, "Claude 3.5 Sonnet Model Card Addendum," 2024. Available: https://www.anthropic.com/news/claude-3-5-sonnet
[4] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep Learning Libraries via Large Language Models," arXiv:2212.14834 [cs.SE], 2023.
[5] TensorFlow, "TensorFlow," 2020. [Online]. Available: https://www.tensorflow.org
[6] PyTorch, "PyTorch," 2024.
[7] C. Zhang et al., "Understanding large language model based fuzz driver generation," arXiv:2307, 2023.
[8] C. Yang et al., "White-box compiler fuzzing empowered by large language models," arXiv:2310.15991, 2023.
[9] Huang, Dong, et al. "Bias assessment and mitigation in llm-based code generation." arXiv preprint arXiv:2309.14345 (2023).
[10] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. Language Models are Unsupervised Multitask Learners. OpenAI. (2019)
[11] Marvin, Ggaliwango, et al. "Prompt engineering in large language models." International conference on data intelligence and cognitive informatics. Singapore: Springer Nature Singapore, 2023.
[12] Huang, Linghan, et al. "Large language models based on fuzzing techniques: A survey." arXiv preprint arXiv:2402.00350 (2024).
[13] Brown, Tom B., et al. "Language models are few-shot learners. arXiv 2020." arXiv preprint arXiv:2005.14165 4 (2005).
[14] Weng, L., "Prompt Engineering," Lil'Log, Mar. 15, 2023. Available: https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/
[15] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y., "Large Language Models are Zero-Shot Reasoners," arXiv:2205.11916, 2022. Available: https://arxiv.org/pdf/2205.11916
[16] OpenAI, "OpenAI o1-mini: Advancing Cost-Efficient Reasoning," Sep. 12, 2024. Available: https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/
[17] OpenAI, "Introducing Structured Outputs in the API," Aug. 6, 2024. Available: https://openai.com/index/introducing-structured-outputs-in-the-api/
[18] OpenAI, "Prompt Caching in the API," Oct. 1, 2024. Available: https://openai.com/index/api-prompt-caching/