ATOC: Automated Test Oracle Construction Based on Large Language Models

Xinyang Yin
Department of Electronics and Information Science,
Xi’an Jiaotong University, China
Email: yinxinyang@stu.xjtu.edu.cn

***Abstract*—Deep learning (DL) frameworks are now becoming more and more popular due to their wide applications in society, while testing DL frameworks presents immense obstacles despite their required high reliability. Current DL framework testing still focuses on assessing the models or APIs by running themselves using various fuzzing tools, neglecting the application of large language models (LLMs) which may assist in constructing test oracles automatically.**

To the best my knowledge, it’s the first time to try to generate test oracles for DL Libraries automatically based on LLMs. A pivotal challenge in DL framework testing is accurately giving out the expected test oracles for a given input, which demands profound understanding of both the DL frameworks and its internal operating limitations. Traditional testing approaches, relying on manually coding test cases and test oracles, fall short when meeting with the complexity and extensive using of current DL frameworks. However, LLMs can simplify the process.

Thus, I propose ATOC, an innovative strategy that utilizes LLMs to automatically construct test oracles for DL frameworks. By utilizing LLMs’ advanced natural language processing abilities, ATOC analyzes and comprehends DL frameworks, facilitating automated test oracle generation and expected output determination. To evaluate the effectiveness of my approach, I apply ATOC on testing Pytorch framework with 1500 APIs, and the results demonstrate that ATOC is effective in detecting bugs and inconsistencies, especially on crashes (100.0%), flaky (75.0%) and hangs (66.7%).

***Index Terms*—Test Oracle, Large Language Models, Automation.**

I. Introduction

In recent years, the application of artificial intelligence (AI) technology in the field of network security has become increasingly widespread, especially in automated testing, threat detection and response. By automating the construction of corresponding test oracles through AI technology, not only can testing costs be reduced and testing coverage improved, but potential security vulnerabilities can also be discovered in a timely manner, ensuring the security and stability of the system.

Currently, AI technology is gradually integrating into various fields of cybersecurity, especially in automated testing. Through advanced technologies such as DL and natural language processing, AI can automatically generate test cases, simulate attack behavior, and analyze system responses in real-time, thereby improving the efficiency and accuracy of testing. In the future, with the continuous development of AI technology, its application in cybersecurity testing will become more extensive and in-depth.

To evaluate the effectiveness of ATOC, I answer the following research questions:

RQ1: Can ATOC detect bugs and inconsistencies in deep learning frameworks?

RQ2: Can ATOC qualitatively give out test oracles on code performance when certain parameters change?

RQ3: Can ATOC explain the test oracles and give out a confidence score?

In this paper, I make the following contributions:

- A new approach to testing DL frameworks by clearing and connecting API documentation with test cases;
- The first approach to qualitatively determining changes in various code performance on DL Libraries by LLMs; and
- A further evaluation of testing modules on DL frameworks with the newest stable version.

II. Background

In this section, I demonstrate the background of this paper from three aspects: DL Frameworks, DL APIs, and DL Framework Testing.

A. DL Frameworks

DL have emerged as essential tools in the field of Machine learning, while DL frameworks are the foundation for implementing DL algorithms, providing various APIs that make it easier for developers to code and train DL models. These frameworks abstract the complex mathematical details of DL, such as gradient computations and so on, which allows DL frameworks to efficiently promote the training process and significantly enhance the model performance.

Besides, more and more popular DL frameworks, such as PyTorch [1], TensorFlow [2], CNTK [3], and Theano [4], offer abstraction of implementation logic, enabling DL developers to train and run DL models more freely, without needing to dig deep into the logic of code implementation or understand the steps to implement the algorithms.

B. DL APIs

As is demonstrated in the former part, DL frameworks offer abstraction of implementation logic, which is integrated into several APIs. DL APIs offer access to calling operation functions defined inside the frameworks. Taking Pytorch [1] as a typical example, such operations include Tensor Operations (e.g. torch.add for add operation), Neural Network Layers (e.g. torch.nn for the base of complex neural network architecture), Optimizers (e.g. torch.optim for multiple optimization algorithms), and so on. With the correct parameters given to the APIs, they can run independently without the interference of developers, which provides a lot of convenience for developers to focus more on the approaches and algorithms instead of how the APIs will run inside.

C. *DL Framework Testing*

DL framework testing constructs test oracles of several input test cases, checking the framework itself by confirming whether those test oracles meet the developer expectations. Sometimes developers use assertions to check the correctness of the exact codes, or maybe they will calculate the result in other methods to check whether these two results are close or not (e.g. torch.allclose in Pytorch [1]. Through these steps, developers can check for bugs or inconsistencies in DL frameworks and then report them to the framework developers, making contributions to the community.

III. Approach

In this section, I describe how I validate the effectiveness of the approach and how ATOC works to detect and qualitatively give the expected results.

A. *Document Crawler*

The Document Crawler is essential in my approach to systematically collecting API documents from specified DL frameworks [5]. This step ensures that I have comprehensive documentation to work with and that references can be provided for later test cases.

The document crawler operates by browsing through the structured documentation on PyTorch, collecting all of the necessary details about the APIs. This process involves parsing HTML content, extracting relevant data such as function name, parameter descriptions, return types, and usage examples. The selected version should meet with the environment I have built, which is crucial as it ensures that my analysis aligns with the current version of the frameworks.

After collecting the API documents, a significant preprocessing phase is undertaken to provide convenience for LLMs. The raw documents often contain HTML tags and other formatting contents that can interfere with the model's understanding and processing capabilities. Therefore, I take a series of cleaning steps to remove messy codes, fix garbled text areas, and standardize the formats across all documents. This involves removing unnecessary HTML tags and ensuring a consistent layout for better understanding by the LLMs.

Furthermore, the preprocessed documents are stored in a structured format, i.e. plain text files, providing convenience for accessing and generating accurate outputs by LLMs. Through these methods, I can enhance their ability to understand and answer complex questions related to PyTorch functions.

B. *Effectiveness Validation*

Figure 1 illustrates the validation process applied to GitHub issues [1], [6] in order to assess the capability of LLMs in identifying and understanding bugs and inconsistencies within popular DL frameworks. The validation method is designed to test the LLM's abilities in both finding bugs and inconsistencies within minified repros and telling why bug happened based on bug reports.

In this effectiveness validation, I first gathered several bug issues reported for the framework on GitHub, for their wide acknowledge in the community. These issues are meticulously selected to ensure they represented a diverse range of bug and inconsistency types, providing better testing datasets for the LLMs.

The collected bug issues are then divided into two parts: minified repros and detailed bug reports. Minified repros are simplified versions of the original bug-inducing codes, which are designed to isolate the issue and minimize extraneous details, making them easier for both humans and models to analyze. Bug reports typically include a more detailed description of the bugs and inconsistencies, detailing the steps leading to the bug, expected status and actual behavior, and any other relevant logs or error messages.

Before I feed these inputs into the LLMs, a cleaning process is essential to ensure the clarity of the inputs. This involves removing irrelevant comments and formatting inconsistencies. This step is essential as it helps LLMs to focus more attention on the relevant parts of the codes, enhancing its ability to accurately generate the results.

Next, the cleaned repros, along with the corresponding API documents of the DL frameworks, will be fed into the LLMs. The objective here is to assess the LLM's ability to analyze the code, understand the context provided by the API documentation, and accurately identify the presence of the bugs or inconsistencies within the repro. This capability is essential for automated test oracle construction, as it allows for quick and precise prediction of codes' behavior without running.

C. *Dataset Training*

Figure 2 shows how I train the LLM with fuzzing. Training codes is developed to mix code generation together with confidence validation and result checking.

1) Code Generation: To generate test cases of frameworks, code generator is coded based on TitanFuzz [7]. TitanFuzz uses Codex to automatically create high-quality seed programs and runs InCoder to obtain mutations by filling the code. To perfectly match with the whole program, it's necessary to extensively rewrite and customize TitanFuzz.

The primary modifications involve refining the strategy of code generation. Specifically, the generator is enhanced to include essential import statements necessary for the context of the testing frameworks. Additionally, the testing phase after generation is omitted, which is originally designed to

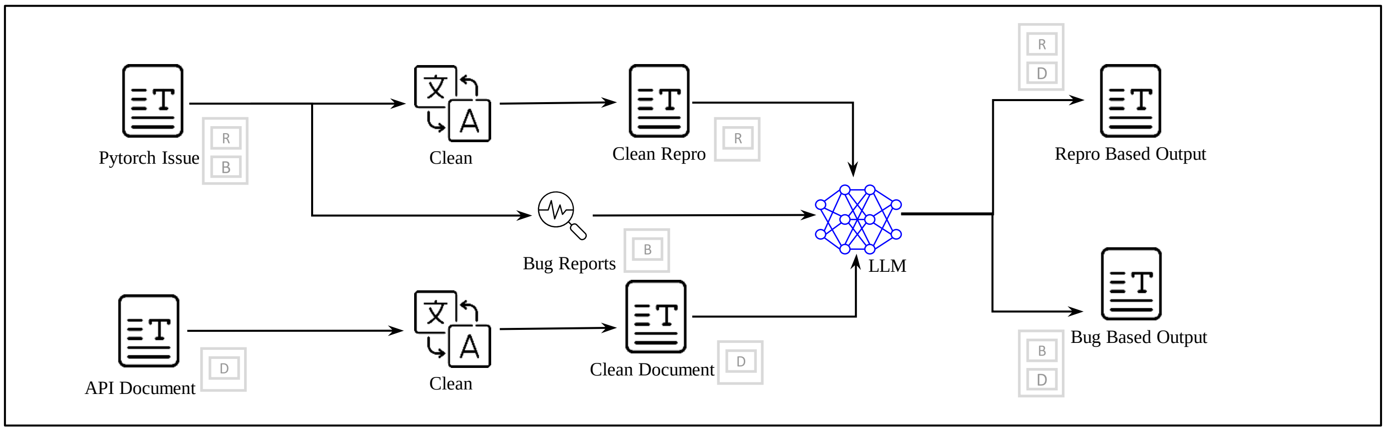


Fig. 1. Validate Process on Issue

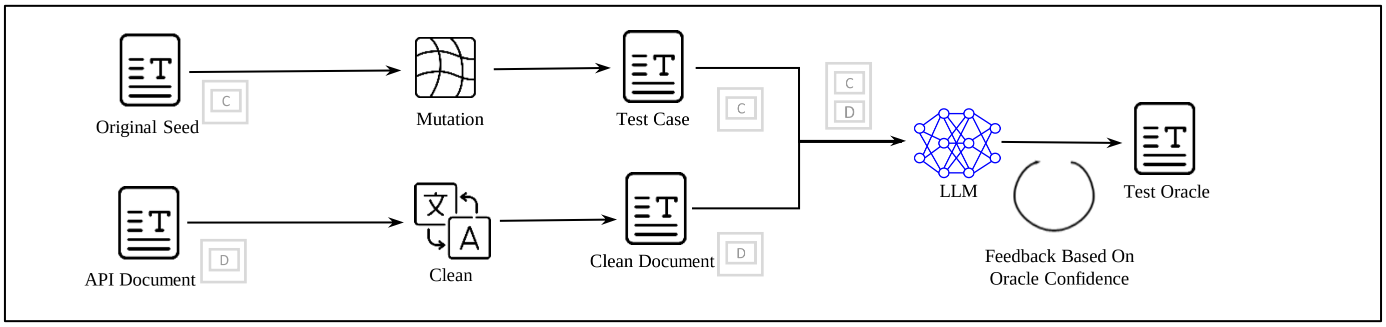


Fig. 2. Train Process on Fuzzing

immediately validate the mutated code within TitanFuzz. This decision is made to simplify the process, as my focus is on generating a diverse set of test cases rather than immediately testing their validity. The generated test cases will be fed into the LLMs for outputs of test oracles.

The code generator operates on an original set of seed codes provided, which serve as the foundation upon generating various test cases. For the original seed provided, the generator mutate them and generate more different test cases, which can help the LLM to enlarge the train datasets, exploring various potentially problematic paths that maybe appear when using the API.

The diversity of the generated test cases is crucial because it allows the LLMs to learn from a wider range of datasets, which can assist the models to deal with various cases that might happen during actual deployment. Furthermore, extending datasets enhances the generalization abilities of LLMs, making it more reliable in constructing test oracles of the test cases.

2) Prompt: To facilitate the model training process more effectively, I set up a prompt template. This template serves as a structured guide for the model to understand every part in the prompt and what needs to be predicted. Given the documentation of the critical API in the needed version, the model will need to construct test oracles in the desired format under several limitations.

First of all, there is a test case that the model needs to analyze and predict the output for. Next, for the clear understanding of context, there is some explanation about the API that is crawled before. Based on the test case and the API explanation, the model needs to answer in the structured format. By following the structured prompt template, the model can provide clear, concise, and accurate responses that meet with the expected format and requirements.

This prompt template includes the following parts:

Result: The model needs to predict the output of test cases and provide concise and accurate results under the requirements based on the explanation of the API.

Score: The model needs to provide a confidence score about the result it gives out of the test cases, which should be a number between 0 and 100.

Final answer: The final answer of the model should be a summary of its prediction and confidence score, which is a final result based on the value of the confidence score and threshold provided.

Explanation: In this section, the model needs to explain the reasons for its predictions and evaluations. It should explain what it find between the test case and API documentation and how it gains the final answer, citing specific content from the test case and API documentation as its support.

3) Evaluator: To evaluate the result of LLMs and reduce the probability of hallucinations, which refer to the generation of factually incorrect or unreliable information, I utilize a confidence score within the evaluator. The confidence score ranges from 0 to 100, providing a quantitative measure of the reliability of the LLM's outputs [8]. To significantly minimize the false alarm rate, I choose a sufficiently large threshold value of 90 for the confidence score [9]. This strict criterion helps in filtering out less reliable responses.

However, only relying on a single confidence score generated by the LLMs might not always construct the most precise results. Therefore, to achieve a more accurate confidence score evaluation, I adopt a multi-run approach. Specifically, I execute the LLMs multiple times with the same input and calculate the occurrences of each unique output. The confidence of a particular result is defined as the proportion of times that result appears among all runs, rather than relying on the single output of the LLMs. This approach takes advantage of statistics to provide a more reliable confidence score.

Mathematically, the confidence score can be defined as follows [8]:

$$confidence = \frac{\max_{i=0}^{n} times_i}{\sum_{j=0}^{n} times_j} \tag{1}$$

$times_i$ represents the number of times the i-th unique result is generated in several runs of the LLMs. The numerator of the equation identifies the times most frequent result appears, while the denominator sums the total occurrences of all unique results. This ratio provides a clear indication of the probability that the most common output is accurate and reliable, therefore reducing the probability of hallucinations in the final evaluation.

D. Experimental Settings

1) Framework: The whole experiment is running on Pytorch framework, although when I generate the test cases, both Pytorch and TensorFlow are frameworks I considered.

2) Trained Model: The training is based on Llama-3.1, which provides test oracles of various test cases [10].

3) Hardware and Infrastructure: I utilize multiple Anaconda environments to switch among generator, evaluator and different backends. I run all experiments on Ubuntu 20.04.6 LTS with 251Gi of Mem and eight NVIDIA GPUs. For the convenience of performance analysis, I run the whole program utilizing a single GPU.

IV. Results

A. RQ1: Can ATOC detect bugs and inconsistencies in deep learning frameworks?

The primary objective of this research question is to assess the capability of ATOC in detecting bugs and inconsistencies within DL frameworks. The results presented in Table I and Table II offer a comprehensive overview of ATOC's performance in this regard.

Table I details the findings from the validation process, indicating that ATOC successfully detected 1041 inconsistenciess out of the 3897 test cases that are pre-identified in the DL frameworks under investigation of GitHub. These bugs are selected because they cause unique errors that could potentially impact the performance and reliability of ATOC. Notably, in the samples built from data above, 421 inconsistencies out of 1800 test cases are highlighted for study in Table II. This suggests that ATOC demonstrates a strong ability to identify bugs, although there is still room for improvement in its coverage and accuracy.

The types of bugs and inconsistencies that are used in the validation process are diverse, including a range of issues such as Type Issues, Incorrect Algorithm Implementations, API Incompatibilities, and others. This variety ensures that the validation is thorough and can represent the types of issues that may arise in practice. It is worth noting that as the training phase progressed, the types of bugs and inconsistencies detected by ATOC will expand [11].

To further analyze the results, as is shown in Table II, I studied the FP Error in the above test cases. I classified them into 3 classes with each class having its own classifications. It's notable that most FP Error is happened due to the abilities of LLMs, so it's essential to select the generating LLMs with better confidence.

TABLE I
RESULTS OF DETECTION (FIRST APPROACH)

Unknown Tag	**True**	**False**	
Label	**None**	**True**	**False**
Sample	388	991	421
Total	518	2338	1041

Overall, the results of this validation demonstrate that ATOC has the potential to be a valuable tool for detecting bugs and inconsistencies in DL frameworks. However, the presence of undetected bugs and inconsistencies highlights the need for improvements in detection process. Future research could explore ways to better enhance ATOC's accuracy and coverage, such as utilizing more advanced DL techniques or expanding the types of bugs and inconsistencies that it is trained to detect.

B. RQ2: Can ATOC qualitatively give out test oracles on code performance when certain parameters change?

I explore the capability of ATOC to qualitatively assess code performance and provide test oracles when specific parameters changes. This assessment is essential for understanding how well ATOC can identify bugs and inconsistencies in codes' behavior, particularly under different parameter conditions.

The code generation process within my testing includes six distinct status: crash, exception, flaky, hangs, notarget, and valid. Among these status, 'notarget' is not viewed as a bug or inconsistency. The status 'exception' is not detected in the testing. 'Valid' status, however, indicates a correctly functioned test case, which has no need to use for detecting bugs and inconsistencies.

My primary focus, therefore, falls on the remaining three status: crash, flaky, and hangs. Each of these status represents

TABLE II
FP ERROR STUDY

Error Type	Details	Explanation / Examples
Prediction Error	Type Misunderstand	Regular/Cloned Tensor is different from Tensor
	Relative Parameters	Parameter atol and rtol must be either specified or omitted
	Added Limitations	Think dtype: torch.int32 is not supported but actually supported
	Reason Dismatched	Move tensor to cpu will decrease the time use
	Value Misunderstand	Think input_data.clone().clone() is different from input_data
	Think Only About Change	No concern about the accuracy of the program's status
	Not Care About Context	TypeError: can't convert cuda:0 device type tensor to numpy
Generations Dismatch	New Program Change But Dismatch	Generation is different from the new codes
	New Program No Change	Parameter Changed but Program Not Changed
	Changed More/Less Than Needed	Other parameter changed also
	Existed Changed Parameter Added	For high, torch.randint(2, (5, 3), dtype=torch.int64, high=10)
	Import Deleted/Added	torch.nn is not defined
	Dismatch Implicit Calls	Parameter value judge error
	Messy Codes	New program full of messy codes
	Misunderstand Of Documentation	Think parameter A impacts the use of funcs
Original Code Error	Original Code With No This API	Original code only includes import words
	Original Code With Multi-API	Can't judge the change of the parameter

a significant deviation from expected behavior and could have significant influence on the reliability of the LLMs.

After conducting a series of experiments using ATOC, I observe that its accuracy in detecting test cases in the 'crash' status is quite high, as is shown in Table III, approximating 100.0%. This indicates that ATOC is highly effective in identifying situations where the code terminates abnormally due to an unhandled bugs or inconsistencies.

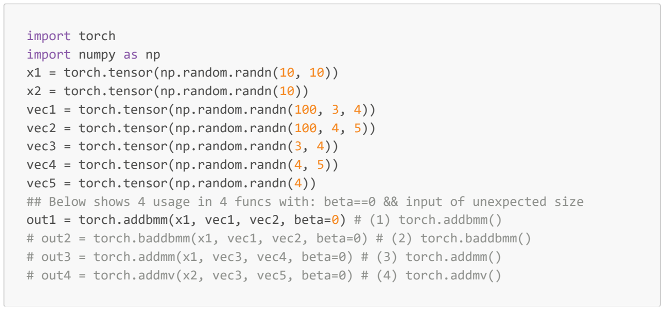
However, when I focus on the 'flaky' and 'hangs' status, ATOC's accuracy decreases. Specifically, for 'flaky' test cases, ATOC's accuracy drops to 75.0% in Table III. This decrease suggests that while ATOC can identify some of flakiness, it sometimes couldn't judge well when codes meet with 'flaky' phenomenon.

Similarly, for 'hangs' status, where the code fails to complete execution within a reasonable timescale, Table III shows that ATOC's accuracy further decreases to 66.7% when I focus on 'hangs' status. This lower accuracy may be attributed to the complexity involved in detecting hangs, which includes the threshold of 'hangs', background processes and other system-level factors.

TABLE III
RESULTS ON PARAMETER CHANGE (FIRST APPROACH)

	Crash	Flaky	Hangs
Accuracy	100.0%	75.0%	66.7%

Contributions are made to report these bugs or inconsistencies in the community on GitHub. Take the series of torch.addbmm() as an example [12]. It is described in the official documentation that if beta is 0, then input will be ignored, and nan and inf in it will not be propagated, which means that whether input is an expected matrix or not, there shouldn't be an error happened for the misuse of input. Instead, it should ignore this just like input doesn't exist. But now when beta is set to 0, with input of unexpected size, which is shown in Figure 3, it will raise error. This bug is now fixed by changing the description into ignoring the content of input, which also proved the validation of ATOC.



```
import torch
import numpy as np
x1 = torch.tensor(np.random.randn(10, 10))
x2 = torch.tensor(np.random.randn(10))
vec1 = torch.tensor(np.random.randn(100, 3, 4))
vec2 = torch.tensor(np.random.randn(100, 4, 5))
vec3 = torch.tensor(np.random.randn(3, 4))
vec4 = torch.tensor(np.random.randn(4, 5))
vec5 = torch.tensor(np.random.randn(4))
## Below shows 4 usage in 4 funcs with: beta==0 && input of unexpected size
out1 = torch.addbmm(x1, vec1, vec2, beta=0) # (1) torch.addbmm()
# out2 = torch.baddbmm(x1, vec1, vec2, beta=0) # (2) torch.baddbmm()
# out3 = torch.addmm(x1, vec3, vec4, beta=0) # (3) torch.addmm()
# out4 = torch.addmv(x2, vec3, vec5, beta=0) # (4) torch.addmv()
```

Fig. 3. Issue Example

Despite these limitations, the results demonstrate that ATOC has pretty potential in qualitative assessment of code performance, particularly in identifying critical failures such as crashes. The challenges observed in detecting flaky and hangs status highlight areas for future improvement. Overall, these findings provide valuable insights into ATOC's strengths and weaknesses, guiding future research and development efforts in automated test oracle construction.

C. RQ3: Can ATOC explain the test oracles and give out a confidence score?

As demonstrated in RQ1, during the training phase of ATOC, I broaden the types of bugs and inconsistencies it could detect. This includes a diverse range of issues such as Type

Issue, Tensor Shape Misalignment, Incorrect Algorithm Implementation, Environment Incompatibility, API Incompatibility, API Misuse, Incorrect Assignment, Incorrect Exception Handling, Misconfiguration, Numerical Issue, Concurrency Issue, Dependent Module Issue and so on [11]. This comprehensive coverage is designed to ensure that ATOC could effectively identify and address widely potential problems in the DL frameworks. Figure 4 shows the loss change in the training phase.

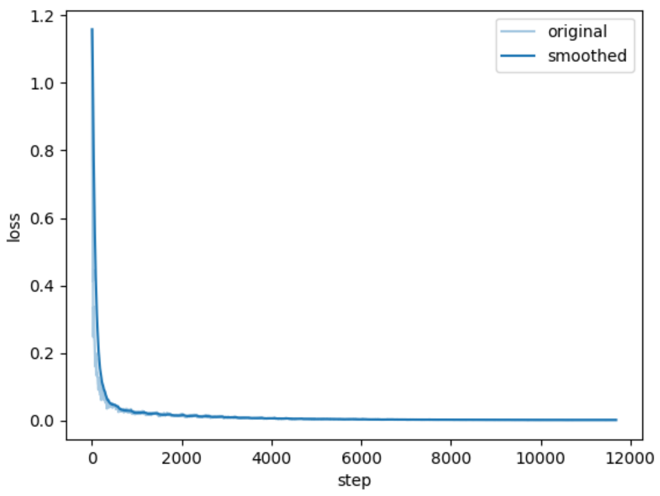


Fig. 4. Training Loss

When integrating LLMs into ATOC to generate confidence scores and reasoning, I observe an impressive accuracy rate of up to 76.4%. This high accuracy highlights the efficiency of ATOC in not only generating test oracles but also explaining them and providing a qualitative measure of the code performance.

The confidence score, generated from the LLM's analysis of the test cases and its corresponding context, serves as an indication of the probability of correctness. This confidence score can be particularly useful that by generating this additional information, ATOC can make decisions more wisely, constructing more reliable test oracles for users.

Moreover, the ability to explain the test oracles through reasoning is pretty useful when constructing test oracles automatiacally, which can help ATOC to better understand test cases and enhance the overall reliability of the test oracles.

So the findings from RQ3 demonstrate that ATOC is capable of not only generating test oracles but also explaining them and providing a confidence score to assess their reliability. This comprehensive approach to construct test oracles automatically may continue to evolve and improve in the future.

V. Limitations and Threats to Validation

Since I focus more on the bugs and inconsistencies in DL frameworks, the model may lack precision in locating and fixing bugs and inconsistencies. Based on fine-tuning pre-trained models, the effectiveness of the model may be affected. Therefore, I have strengthened training and prompt engineering to limit the output as much as possible, which can make the results more credible.

My approach is based on LLM, which results in poor interpretability of the output. And also, some complex DL frameworks may contain non deterministic layers, so the output may differ slightly when given the same input. Alternatively, I can use various LLMs for cross training and validation, which remains as future work.

VI. Related Work

To the best my knowledge, I'm the first to generate test oracles for DL Libraries automatically based on LLMs.

Automatically testing machine learning libraries: Automatic testing of machine learning libraries has become active in the past several years. Dutta et al. [13] use ProbFuzz to test probabilistic programming systems. Srisakaokul et al. [14] detect inconsistencies between several implementations of common machine learning algorithms. Dwarakanath et al. [15] detect inconsistencies in machine learning libraries by transforming between training and testing datasets. All of them are focusing on the machine learning libraries.

Classification of bugs on DL frameworks: Makkouk et al. [16] study the performance and non performance deficiencies of the DL frameworks, comparing them quantitatively and qualitatively and classifying the reasons for the performance deficiencies. Guo et al. [17] propose Audee to automatically test logical errors, crashes, and NaN errors in DL frameworks, which can identify inconsistencies between DL frameworks and locate layers that cause inconsistencies or bugs. Chen et al. [11] propose TenFuzz, which is a prototype DL framework testing tool that provides a classification of the primary causes and symptoms of DL framework errors. TenFuzz proposes a promising direction that we may shorten the training process by simplifying models' structures and reducing the amount of training datasets. These classification methods can help me better understand DL frameworks and validate program effectiveness more efficiently.

Generating test cases on DL frameworks: Deng et al. [7] generate TitanFuzz to directly utilize LLMs to generate input programs for fuzzing DL libraries, using Codex to automatically create high-quality seed programs for evolutionarily fuzzing algorithms, and running InCoder to fill code to obtain mutations. TitanFuzz executes the generated differential testing programs on different backends finally to detect bugs in DL libraries. Georgescu et al. [18] study the effectiveness of differential testing on identifying bugs in Kotlin compiler, generating random test cases based on language features. However, these prior works focus on generating test cases only and don't construct test oracles later.

Constructing test oracles on DL frameworks: Zhang et al. [6] develop Citadel to collect existing error reports, identify problematic APIs, and generate test cases more effectively for testing, calling the stack to find similar APIs. Deng et al. [19] propose DeepREL to collect all API descriptions from documents and compare them using SBEncoder encoding, automatically inferring all possibly relational APIs based on API syntax and semanteme. None of these techniques construct test oracles by using LLMs.

VII. Conclusion

In this paper, I propose ATOC, a new approach to automatically construct test oracles of DL frameworks based on LLMs. ATOC operates by gathering API documents of specified DL frameworks using the document crawler, which serves as resources for LLMs to learn more about the exact API. Then, ATOC employs LLMs to detect and understand bugs and inconsistencies within DL frameworks. Its effectiveness in identifying the test cases is confirmed through meticulous experimental validation.

Furthermore, ATOC extends beyond mere bug detection. It can qualitatively assess code performance changes upon parameter adjustments and clarify test oracles, generating confidence scores to its predictions. This capability to generate confidence scores and reasons improves the efficiency and reliability of DL framework testing.

This paper calls for more attention for testing DL frameworks through LLMs not just by generating test oracles manually. It is noted that there are limitations to ATOC's accuracy in locating and fixing bugs with poor interpretability of models. Future work includes strengthening training and prompt engineering. Exploring the use of various LLMs for cross-training and validation may work as well. Overall, ATOC represents a significant step forward in automated test oracle construction and has the potential to improve the accuracy and reliability of testing in DL frameworks.

Acknowledgment

I would like to extend my heartfelt thanks to Minnan Luo and Heng Wang for their invaluable cultivation and support throughout this course. Additionally, I am immensely grateful to Xiaohong Guan, Chao Shen, Chenhao Lin and Xiaoyu Zhang for their meticulous guidance and assistance during the project. Their extensive professional knowledge, rigorous academic approach, and genuine concern for students' development have significantly enhanced my learning experience and deepened my understanding. I'm truly appreciative of their efforts and support.

References

[1] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[3] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 2135–2135.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. P. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python." in *SciPy*, 2010, pp. 18–24.

[5] "Pytorch documentation - pytorch 2.5 documentation." [Online]. Available: https://pytorch.org/docs/2.5/

[6] X. Zhang, J. Zhai, S. Ma, S. Wang, and C. Shen, "Citadel: Context similarity based deep learning framework bug finding," *arXiv preprint arXiv:2406.12196*, 2024.

[7] 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," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[8] Z. Lin, S. Trivedi, and J. Sun, "Generating with confidence: Uncertainty quantification for black-box large language models," *arXiv preprint arXiv:2305.19187*, 2023.

[9] J. Chen, J. Yoon, S. Ebrahimi, S. O. Arik, T. Pfister, and S. Jha, "Adaptation with self-evaluation to improve selective prediction in llms," *arXiv preprint arXiv:2310.11689*, 2023.

[10] "Hugging face – the ai community building the future." [Online]. Available: https://huggingface.co/

[11] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[12] "Fix ignore description in torch.addbmm(), torch.addmm(), torch.addmv() and torch.baddbmm()." [Online]. Available: https://github.com/pytorch/pytorch/issues/146611

[13] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 574–586.

[14] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software." in *AAAI Workshops*, 2018, pp. 384–391.

[15] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 118–128.

[16] T. Makkouk, D. J. Kim, and T.-H. P. Chen, "An empirical study on performance bugs in deep learning frameworks," in *2022 ieee international conference on software maintenance and evolution (icsme)*. IEEE, 2022, pp. 35–46.

[17] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 486–498.

[18] C. Georgescu, M. Olsthoorn, P. Derakhshanfar, M. Akhin, and A. Panichella, "Evolutionary generative fuzzing for differential testing of the kotlin compiler," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 197–207.

[19] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 44–56.