A Study of PyTorch Bug Patterns and Memory-Related Challenges

Brian Yu
Skyview High School
Vancouver, Washington
brianhyu08@gmail.com

Rubayet Rahman Rongon
Washington State University
Vancouver, Washington
r.rongon@wsu.edu

Chen Cao
Pennsylvania State University Behrend
Erie, Pennsylvania
ccao@psu.edu

Xuechen Zhang
Washington State University
Vancouver, Washington
xuechen.zhang@wsu.edu

***Abstract*—This study presents an in-depth manual analysis of memory-related bugs within the PyTorch deep learning framework, leveraging a filtered dataset of 1,678 closed issues from the official PyTorch GitHub repository. The selected issues span a three-year period from January 1, 2020, to March 23, 2023, allowing for a comprehensive examination of trends, patterns, and solutions. This study aims to understand the correlations between the characteristics of PyTorch bugs and also the composition of the root causes behind memory bugs. The findings reveal that Correctness and Runtime Error bugs occur most frequently, with a lack of a correlation between Affected Components and Bug Symptoms. Our results highlight the need for more integrated inter-component debugging tools. Furthermore, the findings show that indexing errors occur most frequently among memory bugs. We determine that, to address the severe impact of such memory bugs, there exists a need for more comprehensive and redundant test cases. Through this analysis, this work aims to provide actionable insights for developers to improve the robustness of PyTorch, improving its reliability in machine learning applications.**

***Index Terms*—Deep Learning, Bug Analysis, PyTorch**

I. Introduction

Deep neural networks (DNNs) have become foundational to many modern software systems, enabling the modern demand for data extraction, transformation, and processing. Many developers and researchers have turned to DNN frameworks to abstract the complexities of designing, training, and deploying models. One of these DNN frameworks is PyTorch [1]. PyTorch is currently the second most popular DNN framework on GitHub and is actively maintained and improved upon.

This work comprehensively studies and analyzes bugs and their solutions within PyTorch. By studying and analyzing these bugs in PyTorch, this work aims to identify trends and patterns to assist in understanding the composition of bugs and create references for debugging and building more robust systems.

This work further analyzes memory-related bugs. Memory bugs within PyTorch have especially fatal consequences. Crashes and segmentation faults can block other system functionality, requiring immediate fixes. Hence with a more specialized analysis of memory bugs, this work further aims to understand the root cause and prevention of these bugs. Similarly to the prior analysis, the insights create a better understanding of the composition of the issues, which will help in building more reliable systems.

We first examined 1,678 closed issues from the PyTorch GitHub repository, collected over three years, from January 1, 2020, to March 23, 2023. After carefully examining issue attributes, this subset was then manually labeled for its characteristics. Our analysis discovered that correctness and runtime errors were most prominent in the data set. We further discover that core PyTorch components make up a lower percentage of the total issues than the issues with a critical severity (classification discussed in Section IV). In a similar manner, runtime errors are discovered to comprise 23.6% of total issues but 41.2% of critical severity issues.

Next, we analyze memory bugs that reside within the prior subset that also have at least one accessible patch. We focus on the root cause and prevention steps that can be taken for these bugs. This is determined primarily by further discussion and the code changes in the pull request. The root causes are then generalized between the bugs to create three categories, being indexing, empty tensor, and device mismatch errors, with indexing bugs comprising 42.8% of the total. Between all three categories, the need for more comprehensive and redundant testing is exhibited.

The rest of the paper is organized as follows. In Section II, we discuss the related works. In Section III, we describe the methodology of this work. In Section IV we analyze overall correlations and patterns within the bugs. In Section V we determine the root causes and mistakes of memory bugs. In Section VI we discuss the findings and implications. Finally, in Section VII, we discuss our conclusions.

II. Related Work

Recent related works have analyzed root causes, symptoms, pipeline steps, and bug types for DNN frameworks and their applications. They have further discovered relationships between such characteristics. Jia et. al manually analyzed 202 Tensorflow bugs, looking at symptoms, root causes, and repair methods [2]. Similarly, Ho et. al examined PyTorch, acting as a replication of Jia et. al's study to compare the two frameworks [3]. Chen et. al further expanded the scope, analyzing 1000 bugs across TensorFlow, PyTorch, MXNet, and DL4J [4]. Zhang et. al focused on symptoms and root causes of bugs for applications built upon TensorFlow, looking through 175 stack overflow posts and GitHub commits [5]. Islam et. al analyzed 970 bugs also on applications built on DNN frameworks,

looking through Stackoverflow posts and GitHub commits across the 5 most popular DNN frameworks: Caffe, Keras, Tensorflow, Theano, and Torch [6]. Thung et. al focused on three ML systems, searching for bug frequencies, bug types, the severity of the bug, bug-fixing duration, bug-fixing effort, and bug impact [7]. Rongon et. al studied the bugs related to the data path of machine-learning applications [8]. Building on these works, our research focuses on manual analysis of correlations between components, symptoms, and severity with a larger dataset of specifically GitHub issues, focusing on the PyTorch framework itself rather than applications built on top of it. Furthermore, we focus on analyzing the root cause and prevention of memory bugs within PyTorch.

III. Methodology

PyTorch was selected due to its significant role in both academic research and industry applications. PyTorch provides a comprehensive set of tools for building, training, and deploying DNNs. Its ease of use whilst also including the complexities has made it a top choice for many users.

PyTorch's prominence is seen on GitHub, where it is the second most popular DNN framework. With an extensive repository of contributions, bug reports, and patches it provides a substantial dataset for analyzing real-world issues encountered by users. Furthermore, PyTorch's large, active community constantly works together to improve the framework, making it an ideal candidate for studying how bug patterns and resolutions have evolved and improved.

Data was primarily collected in the form of GitHub issues. We collected closed issues from the PyTorch GitHub repository, focusing on those reported between January 2020 to March 2023. The dataset was filtered based on the presence of the keywords "bug" and "fix," ensuring that the selected issues had a substantial problem and was not just an improvement. Each issue was evaluated to ensure it was marked closed, with an emphasis on those linked to a pull request. Issues without pull requests were included only when significant discussion and resolution steps were available. However, there were exceptions such as issue #29779 [9] where the issue was indicated to be discussed and resolved elsewhere. Through this process, 1678 issues were collected and analyzed.

Next, these 1678 issues were examined based on the patch, description, further discussion, and tags. The bugs were assessed on an issue-by-issue basis and then were classified into four characteristics—component, symptom, severity, and date solved—according to predefined criteria (See Section IV for the criteria). For instance, severity was assessed based on the impact of the bug on system stability and user experience (e.g., crashes = major severity).

Finally, due to the prominence of such bugs within high-severity issues and their especially fatal consequences, memory-related runtime errors are examined for their root causes and prevention. First, the issues were narrowed down to 28 issues based on the symptom of the bug (eg. Segmentation Fault) and the information provided in the further discussion. Next, each bug was manually examined and categorized into one of 3 common root causes within the memory bugs. This was done by deeply analyzing the further discussion of the GitHub issue and the associated pull request/patch. The classification of the characteristics is discussed in Section V

IV. Results

This section of this study provides a detailed analysis of the symptoms, components, and severities of the bugs identified. By categorizing bugs into distinct groups based on their symptoms, components, and severity, this study reveals patterns and trends that can inform future bug identification and mitigation efforts. A majority of the bugs fell into six key symptom categories, with correctness and runtime errors being the most prevalent. This section further examines the distribution of bugs across PyTorch's core components and how severity levels correlate with these components.

A. Symptoms

The bug symptoms have been categorized into 6 different labels. 91.6% of the bugs fell under these 6 common categories. For the remaining 8.4%, they were grouped under "Other", due to the distinctiveness between each of them.

1. Correctness (28.8%): This category encompasses bugs where the program produces inaccurate or unexpected results. For example, issue #51036 shows incorrect index checking resulting in incorrect gradient calculations [10].

2. Runtime Error (25.8%): Bugs in this category cause the program to crash or terminate unexpectedly during execution. For example, in issue #41768, the program crashes when one of the function arguments has a dimension of size 0 [11].

3. Build Failure (13.8%): Build failures occur when the codebase cannot be compiled. For example, in issue #79449, the program fails to compile due to a variable being referenced before assignment [12].

4. Functional Failure (10.4%): This category includes cases where the program does not perform its intended function. Although correctness failures could be considered a subset of functional failures, they were separated here due to their prevalence. For example in issue #90500, changes to certain parameters did not affect gradient calculations, leading to incorrect training behavior [13].

5. Test Failure (6.8%): This symptom accounts for when tests break or do not comprehensively check all test cases. For example in issue #28958, the CI test gives different results with the same test case [14].

6. Performance Degradation (6.2%): Performance issues arise when the software runs significantly slower than expected or consumes more resources than necessary. For example in issue #48049, a particular function consumed approximately ten times the GPU memory compared to similar operations [15].

7. Other (8.4%): This category includes less common issues such as data corruption, typos, or misconfigurations. Due to their varied nature, these issues are harder to group under a single symptom type but are still important to address.

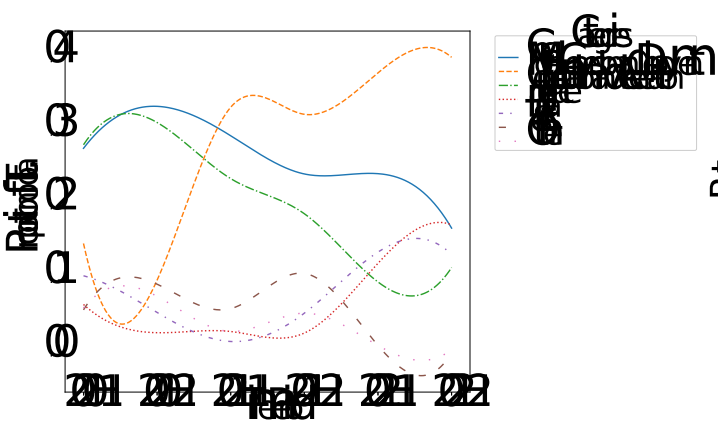

Fig. 1. Distribution of Components from 2020-2022

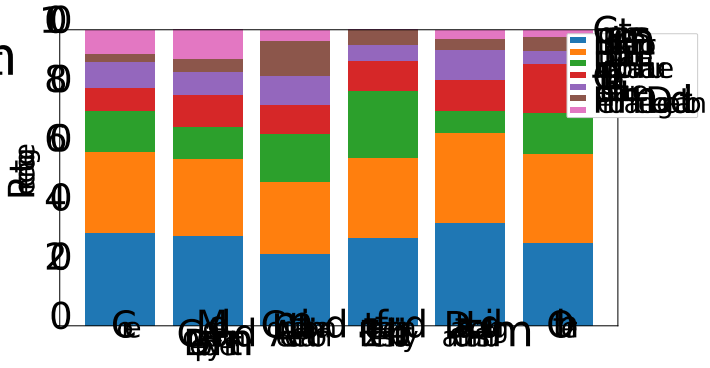

Fig. 2. Distribution of Consequences across Components

It is shown that correctness and runtime errors are the most frequent symptoms of the bugs, accounting for 28.8% and 25.4% respectively. This is particularly worrying due to the difficulty in resolving both of these kinds of bugs. Correctness bugs are often silent, making them difficult to discover and resolve. On the other hand, runtime errors terminate the entire program and can be considered "loud", though they require running of the program to discover them which can be especially time-consuming.

B. Components

A "component" refers to a functional module or subsystem that provides a specific function. These components can vary widely in their purpose, though they generally fall into 6 of the groups described below. The distribution of the affected groups of components of PyTorch issues are also shown below.

1. Core (27.1%): The Core component contains the fundamental functionalities of PyTorch, such as tensor operations and autograd. Given its central role in the framework, issues in this area can have far-reaching consequences, affecting a wide range of models and workflows.

2. Model Conversion and Deployment (27.1%): Bugs within this category typically involve the conversion of models into formats suitable for deployment, such as ONNX. Conversion processes such as quantization are also included under this category.

3. Computation and Acceleration (22.5%): The Computation and Acceleration category covers bugs related to performance optimizations, hardware accelerators, and distributed computing. CUDA, torch.multiprocessing, and torch.distributed related issues are hence labeled under this category

4. Interface and Extensibility (7.4%): This component includes issues related to PyTorch's APIs, user interfaces, and platform-specific support. For example, Mac support belongs under this category.

5. Data Transformation and Loading (7.2%): This category contains essential data preparation, extraction, transformation, and loading. Modules such as torchvision and dataloader are included in this category for their data transformation and loading capabilities respectively.

6. Tests (5.1%): This category accounts for bugs related to tests and testing modules. CI tests and torch.testing are both included in this category. Furthermore, debugging tools such as tensorboard are included in this category.

7. Other (3.6%): This category includes bugs that don't fit into the main component categories, such as documentation errors, small inconsistencies, or other less common issues.

As expected, core components take up the largest percentage of the issues. This percentage is inflated by the sheer amount of components related to core framework components. However, Model Conversion and Deployment also take up 27.1%. After further analysis, we discover a significant growth in such bugs. This is shown in Figure 1, where we see a drastic jump in Model Conversion and Deployment in the first half of 2021.

C. Distribution of consequence across components

As shown in Figure 2, the distribution of bug consequences is relatively uniform across the different PyTorch components. This highlights the lack of correlation between component and consequence, further demonstrating that no component is especially vulnerable or immune to a certain kind of bug.

D. Severity

Severity does not have a formal definition and hence is dependent on the context of the situation. In this work, bug severity was classified into five levels: trivial, minor, major, critical, and blocker. Analyzing the distribution of these severities is crucial for prioritizing and allocating resources to address the most impactful bugs.

1. Trivial (0.11%) Trivial issues include issues where it was not necessary to fix the bug. This is seen in issue #74978, where a variable is turned from a signed to unsigned integers [16]. As many "trivial" changes are improvements rather than bug fixes, there are very little within the entire data set. Hence, trivial issues are not further analyzed in this study.

2. Minor (25.3%) This category includes issues where the issue could be mitigated and would not affect core functionality. For example, in issue #49932, the lack of return type annotation can cause separate code to misinterpret the object

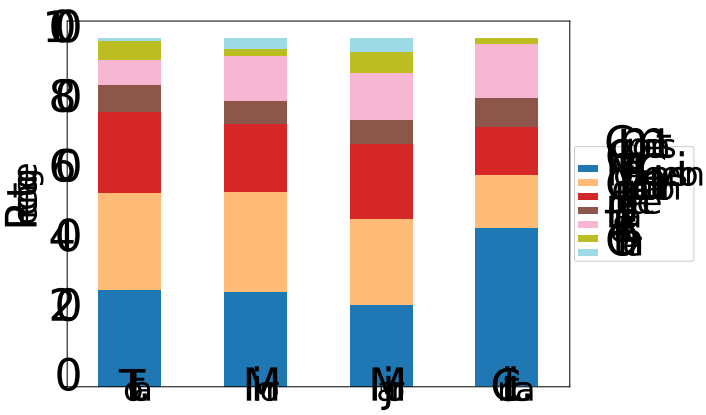

Fig. 3. Distribution of Components across Severities

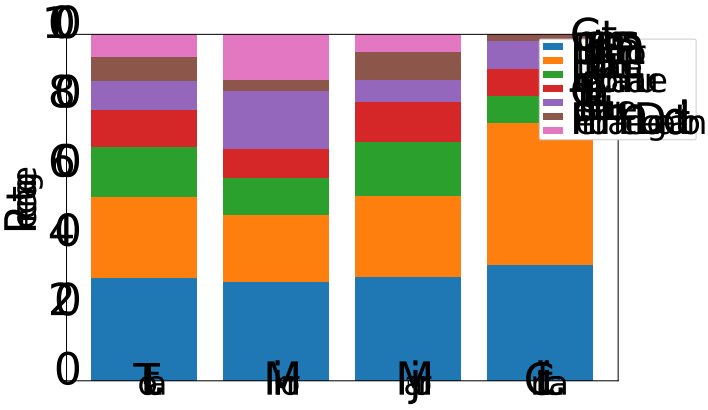

Fig. 4. Distribution of Consequences across Severities

type [17]. This bug doesn't affect the functionality of any other components.

3. Major (71.0%) Issues are defined to have a major severity when the bug in subject has a significant impact on functionality, but will not completely stop the overall program from running. For example, in issue #89560, the model fails when the input shape is 1D [18]. Although this does impact the functionality of the specific component heavily, other components are not severely impacted.

4. Critical (3.6%) Critical issues occur when a bug heavily impacts the program and causes it to fail. While it does not prevent the entire framework from running, there will still be significant impacts on other components. For example, in issue #71094, the function crashes, disrupting the entire program [19].

5. Blocker (0%) Blocker issues occur when the entire software is prevented from working. No bug in the dataset prevented the entire PyTorch framework from functioning.

E. Correlation of Components across Severities

From Figure 3, we see that core PyTorch components make up a higher percentage of critical severity issues. For Core components, they make up 27.1% of total issues but 45.8% of critical components. It is also observed that model and data-related issues make up a lower percentage of critical issues. Model Conversion and Deployment issues make up 27.1% of total issues and 15.3% of critical issues. Similarly, Computation and Acceleration issues make up 22.5% of total issues and 13.6% of critical issues. This significant disparity between the distribution for total and critical severity issues is contrasted with minor and major severity issues, where we see that the composition of affected components do not differ significantly.

F. Correlation of Consequences across Severities

We see from Figure 4 that performance degradation makes up a much larger portion of minor issues, from 6.2% of total issues to 13.1% of minor issues. This is explainable, as while performance flaws are serious bugs, it is not something that needs to be immediately fixed as it does not directly affect functionality. Figure 4 further shows that for critical severity issues, runtime errors make up a much larger portion, making up 41.2% of critical issues but only 27.1% of the total.

V. Memory Bugs in PyTorch

When further analyzing the composition of critical severity issues, we discovered that memory-related issues, particularly memory access violations, which can cause segmentation faults and crashes, comprised a small percentage of the total issues but a high percentage of the critical issues. Of the total issues memory issues make up 2.2%, but of the critical issues they make up 29.7%. In general, 25.0% of memory bugs are labeled with a critical severity as shown in Figure 5, which is held in stark contrast to the 3.6% of bugs labeled with critical severity overall. In this section, we address the root causes of these bugs and aim to generalize the steps to fix or prevent them.

1. Indexing (42.8%): An indexing bug refers to the oversight in the accessible indices, resulting in an inaccessible index being attempted to be accessed. In the case where there is a failure of bounds checking, the solution is to simply add an exception in the case of an inappropriate index. As seen in issue #77896, by adding bounds checks, it prevents undefined behavior, creating a more stable and debuggable function [20].

In issue #52715, an extremely large tensor fails to be processed due to the integer type of the indexing being too small [21]. The solution is to add 64-bit indexing as shown in the issue. Another solution implemented is to "chunk" the tensor into more manageable sectors, which can also help deal with larger tensors. Creating more comprehensive checks with all the edge cases (bounds, small, large) would help discover and prevent more of these issues.

2. Empty Tensors (28.6%) Within these issues, the bug involved a reference to an empty tensors. For example, in issue #46700, a check was added to disallow tensors of element size 0 [22]. Similarly, in [16], unary operators are set to output an empty tensor in the case of an input of an empty tensor. In many of these cases, the issue is an oversight of such an edge case, similar to indexing. Checking the error raised in the case

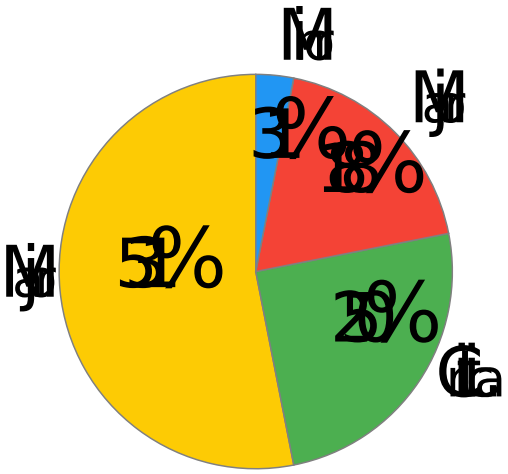

Fig. 5. Severity Composition of Memory Bugs

of an empty tensor would drastically reduce the occurrence of these issues.

3. Device Disparity (10.7%): Memory access violation bugs under this category occurred due to a mismatch between the input and output devices. The disparity between two different devices can cause a memory location in one to be attempted to be accessed in the other, causing the memory access violation. As shown in issue #82531, the issue only occurs in the CUDA kernel and not the CPU may explain the oversight, once again demonstrating the need for more comprehensive tests [23].

4. Other (17.6%): The rest of the issues were highly contextual and hence could not be generalized to another group.

Our analysis of the root causes and prevention of memory access violations demonstrates the increased need for comprehensive bug checks. Indexing and empty tensor bugs, accounting for 71.4% of the total memory access violation issues, are a direct cause of oversights within edge cases. Although many of these edge cases simply cannot be foreseen, adding such comprehensive checks despite their redundancy can create more robust systems. Furthermore, as highlighted by the device disparity issues, testing on different environments as PyTorch's extensibility grows is also likely to help prevent further similar issues.

VI. Discussion

A. Correctness

We have seen in the analysis that correctness bugs make up a large proportion of the bugs. As previously discussed, these bugs are difficult to detect. A lack of test cases and case-checking led to this scenario. This implies that an increased emphasis on correctness testing, particularly fuzzing, can help minimize a very large proportion of the bugs. Extensive research has already been performed on fuzzing. However, with the development of automated fuzzing frameworks like the automated fuzzing framework presented in [25], the ease of use and time it takes to test can be significantly improved.

B. Relation between Component and Consequence

In the analysis, we also see the lack of a correlation between specific components and their consequences. While a more specialized module, such as torchvision, might naturally exhibit issues tied to the unique context in which it operates, the uniformity of bug impact across general component groups reveals this important insight. Additionally, this highlights the difficulty in predicting the consequences of bugs based solely on their component location. As many issues arise from interactions between various modules, bugs may not always show predictable symptoms until they surface during complex model executions. Similar to how traditional debugging tools are ineffective for DNN frameworks due to their interoperability, this interaction demonstrates the need for the development of more integrated debugging tools, ensuring that small bugs from interactions between seemingly unrelated components can still be caught.

C. Runtime Errors

Among the critical severity issues, analysis shows that there is a significant increase in runtime errors compared to the entire dataset. This is explained by the significant impact of runtime termination not only shutting down the respective component but also the framework as a whole. This indicates the need to resolve runtime errors earlier and more efficiently, which can only be done with a more thorough understanding of what causes runtime errors and the common developer mistakes that lead to them.

D. Bounds Checking and Empty Tensors

Memory access violations can occur when the program tries to access bounds exceeding the appropriate range or inputs empty tensors. These edge cases can often be difficult to predict, and may sometimes be completely irrelevant to the context of the operation. Despite the simplicity of these kinds of bugs, they occur extremely often. To prevent such bugs, more comprehensive testing procedures are required. Furthermore, due to the difficulty in predicting them, redundant checks need to be added.

E. Large Tensors

When the input tensor is very large, not using the appropriate data type (eg. floating-point tensor) can cause a memory access violation. The analysis suggests to ensure there is proper indexing support for the tensors and add appropriate checks for the respective data type.

F. Device Disparity

Device disparities cause memory problems in PyTorch when a program incorrectly tries to access a memory location on a different device than intended, such as attempting to access CPU memory from a tensor stored on a GPU (CUDA) or vice versa. This occurs because PyTorch tensors are device-specific; tensors allocated on a GPU must be accessed with

operations targeting that same GPU, and the same applies to CPU tensors. The frequency of this error shown from the analysis suggests that checks for device consistency across all different environments should be added.

VII. Conclusion

As deep learning frameworks are increasingly depended upon, the need for creating more robust systems only ever grows. By analyzing the trends within such data, we can better understand the situation and further steps to improve on reliability. In this paper, we comprehensively study the distribution and relations of bugs within PyTorch by analyzing 1678 closed issues from the PyTorch GitHub repository, dating from 1/1/2020 to 3/23/2023. These bugs are examined for four characteristics, being the framework component affected, severity, symptom, and date solved. We further analyze the root causes and preventions of memory bugs specifically. Our study found that there is no correlation between an affected Pytorch component and a consequence. Furthermore, indexing bugs are found to be the most populous root cause of memory issues. A limitation of our study is the time frame over which the bugs are identified, as it is subject to timing factors like the focused development of a certain feature, and could be repeated over the lifespan of the framework to reduce outliers and skewing in the data. The study also relies on manual analysis of metadata and qualitative descriptions, which adds further inconsistency to the inherent errors of manual analysis and could be improved upon by implementing more quantifiable measures of categorization. Future research could focus on analyzing the inter-module interactions and development of context-aware debugging tools, helping capture bugs more effectively.

VIII. Acknowledgment

We sincerely thank our team members Michelle Zhao and Iris Ta for collecting and labeling the raw data. This work was supported in part by NSF CNS-2245753, MRI-2216108, and OAC-2243980.

References

[1] "PyTorch," https://github.com/PyTorch/PyTorch.

[2] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," Journal of Systems and Software, vol. 177, p. 110935, 2021.

[3] S. C. Yin Ho, V. Majdinasab, M. Islam, D. E. Costa, E. Shihab, F. Khomh, S. Nadi, and M. Raza, "An empirical study on bugs inside PyTorch: A replication study," in 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023, pp. 220–231.

[4] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," 2023.

[5] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, pp. 129–140.

[6] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 510–520.

[7] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang, "An empirical study of bugs in machine learning systems," in 2012 IEEE 23rd International Symposium on Software Reliability Engineering, 2012, pp. 271–280.

[8] Rubayet Rongon, Chen Cao, and Xuechen Zhang, "A Study of Data-Path Bugs in PyTorch with a Focus on Memory Management", in Proceedings of the 2024 IEEE International Conference on Big Data (BigData'24), Washington DC, December 2024.

[9] "ROCm CI may hang/timeout after testing nnpack," https://github.com/PyTorch/PyTorch/issues/29779

[10] "Incorrect gradients for F.interpolate on GPU when mode=nearest," https://github.com/PyTorch/PyTorch/issues/51036

[11] "Segmentation fault in torch.orgqr when input size has element 0," https://github.com/PyTorch/PyTorch/issues/41768

[12] "UnboundLocalError: local variable 'ws' referenced before assignment," https://github.com/PyTorch/PyTorch/issues/79449

[13] "nn.LSTM not working together with functional_call for calculating the gradient," https://github.com/PyTorch/PyTorch/issues/90500

[14] "ParallelWorkersTest.testParallelWorkersInitFun is flaky," https://github.com/PyTorch/PyTorch/issues/28958

[15] "Native amp consumes 10x gpu memory," https://github.com/PyTorch/PyTorch/issues/48049

[16] "Change numModules type to unsigned," https://github.com/PyTorch/PyTorch/pull/74978

[17] "torch.fx.Graph has no return type annotations," https://github.com/PyTorch/PyTorch/issues/49932

[18] "Meta PReLU failing when input shape is 1D," https://github.com/PyTorch/PyTorch/issues/89560

[19] "embedding_bag will trigger segmentation fault in Linux," https://github.com/PyTorch/PyTorch/issues/71094

[20] "Segmentation fault in fused_moving_avg_obs_fake_quant," https://github.com/PyTorch/PyTorch/issues/77896

[21] "CUDA Illegal memory access for softmax," https://github.com/PyTorch/PyTorch/issues/52715

[22] "Fix segfault with torch.orgqr," https://github.com/PyTorch/PyTorch/pull/46700

[23] "[MPS] sub backward seg fault," https://github.com/PyTorch/PyTorch/issues/82531

[24] "Illegal memory access when printing a CUDA tensor (problem with torch.stack(out=))," https://github.com/PyTorch/PyTorch/issues/52044

[25] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis, "IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks", in Proceedings of the 32nd USENIX Security Symposium, 2023, pp. 2383-2400