An Empirical Study on Performance Bugs in Deep Learning Frameworks

Tarek Makkouk*, Dong Jae Kim*, Tse-Hsun (Peter) Chen*
*Software PEformance, Analysis and Reliability (SPEAR) Lab,
Concordia University, Montreal, Canada
{t_makkou, k_dongja, peterc}@encs.concordia.ca

***Abstract*—Machine Learning (ML) and Deep Learning (DL) applications are becoming more popular due to the availability of DL frameworks such as TensorFlow and PyTorch. Therefore, the quality of DL frameworks is essential to ensure DL/ML application quality. Given the computationally expensive nature of DL tasks (e.g., training), performance is a critical aspect of DL frameworks. However, optimizing DL frameworks may have its own unique challenges due to the peculiarities of DL (e.g., hardware integration and the nature of the computation). In this paper, we conduct an empirical study on the performance bugs in DL frameworks. We conduct our study on TensorFlow and PyTorch by identifying the performance and non-performance bugs by mining the GitHub repositories. We find that 1) the proportion of newly reported performance bugs increases faster than fixed performance bugs, and the ratio of performance bugs among all bugs increases over time; 2) performance bugs take more time to fix, have larger fix sizes, and more community engagement (e.g., discussion) compared to non-performance bugs; and 3) we manually derived a taxonomy of 12 categories and 19 sub-categories of the root causes of performance bugs by studying all performance bug fixes. Finally, we present some actionable implications for researchers and developers.**

***Index Terms*—empirical, software engineering, machine learning, deep learning, performance bugs**

I. Introduction

Deep Learning (DL) has gained tremendous popularity in recent years in academia and in numerous industrial and commercial settings due to the availability of big data, improved hardware, and its capability to produce high predictive accuracy. One crucial stepping stone in DL adoption in practice is the availability of open-sourced DL frameworks, which abstract complex mathematical formulas and empower developers to implement DL models efficiently in practice. Unfortunately, as DL applications become an integral part of our everyday lives, the quality of both the DL applications and DL frameworks, especially their performance, becomes of utmost importance. For example, a small delay in the time it takes to make an inference can lead to life-threatening situations (e.g., self-driving cars).

Many previous works have focused on characterizing and resolving performance issues in traditional software settings. However, there are new challenges that arise when developing DL software due to its difference from traditional software applications [1]–[3]. DL requires communication amongst numerous hardware components (e.g., GPU, CPU), finer-grained memory management, is data-driven, and is primarily based on gradient computations, requiring countless iterations through large datasets to improve model accuracy. Such characteristics of DL frameworks and the repetitiveness of DL tasks risk the quality assurance of DL applications and frameworks. While prior research aims to understand and study the characteristics of bugs in DL frameworks [4]–[6], many studies focus on bugs in a single DL framework, and there is a lack of detailed empirical studies on characterizing performance issues in DL systems. We believe that studying performance issues in the DL framework is of paramount importance for both researchers and framework developers: (1) it can direct future research efforts toward developing better test oracles to detect better performance issues; (2) it can provide new evidence on how performance issues arise and are fixed in DL systems for future tool development; and (3) it can help prevent future encounters of performance bugs.

While there are prior studies that study bugs or performance bugs in deep learning frameworks, there are no studies in the literature comparing performance bugs in two of the most popular frameworks simultaneously. Furthermore, no previous studies conducted a detailed empirical study on performance bugs in DL frameworks and provided finer-level insights on the cause of performance bugs. To fill this gap, we conducted an empirical study of performance bugs in two of the most popular DL frameworks: TensorFlow [7], and PyTorch [8]. We collect performance and non-performance bug to study the trend of open and fixed performance bugs over time and the differences between performance and non-performance bugs across different dimensions (i.e., complexity and community engagement). Finally, we build a detailed taxonomy of performance bugs in DL frameworks by manually studying 141 fixed true performance bugs. In particular, we seek to answer the following research questions (RQs):

RQ1: What is the trend of fixed and reported performance bugs over time? We investigate how the prevalence of performance bugs in DL frameworks evolves through time. We find that despite increasing efforts to fix performance bugs in DL frameworks over time, their prevalence is still increasing.

RQ2: What are the differences between performance and non-performance bugs in terms of fixes and community engagement? We compare performance and non-performance bugs across two dimensions (i.e., the complexity of the bugs and the amount of engagement the bugs attract). We find that performance bugs take more time to fix (median is 20 vs 10

days), and the fixes are larger (median is 38 vs 28 LOC) than non-performance bugs. We also find that performance bugs attract more discussions and involve more distinct developers. **RQ3: What are the causes of performance bugs in DL frameworks?** We manually study all fixed performance bugs (141 bugs) in the two DL frameworks and derive a comprehensive and detailed taxonomy of the root causes of the bugs. Our taxonomy contains 12 categories and 19 sub-categories.

In summary, we highlight the potential research directions in developing better test oracles and differential testing to improve performance. Our findings also reveal both differences and similarities between the performance bugs in traditional and DL systems. Finally, we highlight best practices to fix trivial bugs to assist developers in practice.

Paper organization. Section II presents the background of DL frameworks and related work. Section III describes our data collection process. Section IV presents the results to the research questions. Section V discusses the implications of our findings. Finally, Section VII concludes the paper.

II. Background and Related Work

An Overview of DL Frameworks. Machine Learning (ML) is a branch of Artificial Intelligence (AI) that focuses on learning patterns from data to solve complex problems. Deep Learning (DL) is a subcategory of ML that uses a neural network and gradient-based computations to extract patterns by continuously iterating through a large dataset while adjusting parameters. Unlike traditional ML, DL allows for automated feature selection and increases model precision and accuracy [9]–[12]. However, the DL model is notoriously complex and colossal, which introduces a new set of challenges, such as optimizing a wide range of configurations [1].

DL frameworks aim to abstract the complex mathematical details of DL algorithms (e.g., gradient computations) to allow ease of DL implementation in practice. Moreover, DL frameworks implement performance optimizations by representing gradient computations as computational graphs. These are directed graphs where each node represents a mathematical computation, and each edge represents a tensor (i.e., a multi-dimensional array) [8]. Such representation allows DL frameworks to distribute the model training and inference processes across many CPU or GPU cores for better performance. Finally, DL frameworks abstract the underlying hardware so that ML developers can train and run DL models on various devices (e.g., CPU or GPU) and environment settings (e.g., different operating systems).

While many DL frameworks have been released (e.g., TensorFlow, PyTorch, Caffe2, MxNet), TensorFlow and PyTorch have become the most popular DL frameworks [13]. Google's Brain team developed TensorFlow [7], and PyTorch was developed by Facebook's AI research group [8]. Therefore, in this paper, we focus our study on TensorFlow and PyTorch.

Since DL frameworks have become prevalent in various domains of industry and commercial settings, it is of utmost importance to validate their quality to avoid severe consequences. Hence, prior research has studied tremendously characterizing bugs in DL frameworks (e.g., [5], [6], [14]). Among those bugs are performance bugs, which may significantly impact the downstream tasks. Despite this, prior research has identified performance bugs as the most difficult type of bug to address in DL systems [15], and there have been few detailed studies on the characteristics of performance bugs in DL frameworks. Hence, in this paper, we conduct a detailed empirical study on the performance bugs in TensorFlow and PyTorch, the two most widely used DL frameworks today. As an essential first step, we quantitatively analyze how the performance bugs in DL frameworks compare to non-performance bugs. We then do a detailed qualitative study to derive a taxonomy of performance bugs and their causes. Our study highlights the key characteristics of performance bugs in DL frameworks and may inspire future research to address performance problems.

Related Work. *Bugs in DL Systems and Frameworks.* Many works investigated bugs in DL systems, with a focus on the use of DL framework APIs. For example, Zhang et al. [14] and Humbatova et al. [16] analyze common mistakes that happen when using DL frameworks from Stack Overflow (SO) and GitHub projects. Islam et al. [17] studied the characteristics of common bug patterns in DL systems and their impact. Chen et al. [18] built a taxonomy of bug symptoms related to the deployment of DL systems on mobile applications. Cao et al. [19] study performance issues in using APIs from DL frameworks from SO. Zhang et al. [15] conducted an empirical study on 715 DL-related questions from SO. Their research shows that there are five main reasons why users are having problems: improper use of APIs; wrong choice of hyper-parameters; computation on GPUs; static graph computation; and lack of support for debugging and profiling. The main difference between those studies and ours is that they investigate DL framework API misuses in DL systems, while our paper focuses on performance bugs in the DL frameworks themselves.

There are also some studies that look into bugs in DL frameworks. Jia et al. [4] studied the symptoms and causes of bugs in TensorFlow by analyzing the bug reports.The authors later extend their work [5] by also studying the fixes and multi-language bugs. Chen et al. [6] conducted a larger-scale study by collecting and analyzing the root causes and symptoms of 800 bugs from different popular DL frameworks (i.e., TensorFlow, PyTorch, MXNet, and DL4J). While these studies focus on general bugs that occur in DL frameworks, we take a finer-grained approach by studying a specific type of bug (i.e., performance bugs). Performance bugs in DL frameworks may have different characteristics and require special attention from the research community.

Empirical Studies on Performance Bugs. There are many prior studies that try to understand the characteristics of performance bugs in traditional software systems. For example, Jin et al. [20] conducted a comprehensive study on 109 performance bugs from five systems (i.e., Apache, Chrome, GCC, Mozilla, and MySQL). Similar studies on bug reports from Android and

TABLE I: Breakdown of the bugs we collected for each framework.

	Performance bugs			Non performance bugs		
	Open	**Closed**	**Fixed**	**Open**	**Closed**	**Fixed**
PyTorch	405	603	113	3,919	11,357	2,675
TensorFlow	301	952	47	1,874	14,766	1,348
Total	706	1,555	160	5,793	26,123	4,023

iOS applications have been conducted by Liu et al. [21] and Afjehei et al. [22]. They developed a static analysis tool based on the patterns they observed, and were able to detect previously unknown performance bugs. Zaman et al. [23] compared 400 performance and non-performance bugs collected from Firefox and Google Chrome. Nistor et al. [24] studied over 420 performance and non-performance bugs and studied how the bugs are discovered, reported, and fixed by developers. Many prior studies focus on the characteristics of performance bugs in traditional systems. They found that performance bugs have different characteristics (e.g., different fix sizes), and the empirical findings allow researchers to develop bug detection tools. In this paper, we focus on performance bugs that occur in DL frameworks, which feature new challenges that are unknown to traditional software systems. Our findings may inspire future research to further help developers improve the performance of DL frameworks.

III. Data Collection

In this section, we discuss our data collection process.

Collecting the Bug Reports. Our goal is to study the characteristics of performance issues that exist in DL frameworks. For our study, we consider TensorFlow and PyTorch as the most widely used DL frameworks [13], as other DL frameworks are no longer maintained. To achieve this goal, we implemented a crawler to collect all the bug reports in PyTorch (i.e., 19,098 bug reports) and TensorFlow (i.e., 29,941 bug reports) up to February 2021. We then filter out the bug reports that are labeled as non-bugs (e.g., feature requests). After this step, 17,893 bug reports remain from TensorFlow and 16,284 remain from PyTorch, for a total of 34,177 bug reports.

Collecting Bug Fixing Commits. The next step is to identify the fixed bugs and collect their fixing commits. We collect the fixes to compare the characteristics between performance and non-performance bugs (e.g., fix size), and to manually derive a taxonomy of the causes of performance bugs in deep learning frameworks. To collect the bug fixing commits for a given bug, we implemented a web crawler that checks whether there exists a commit whose log contains a URL link to the bug report. If such a commit exists, then the commit is considered to be the bug-fixing commit. There may be some cases where developers are not following the standard practice of providing the link to the bug report in the commit log. To mitigate this concern, we follow a similar process to that done by prior studies [25]–[29] and use a regular expression to capture the fixing commits based on bug report IDs (e.g., a commit that mentions the bug with ID 1234 in TensorFlow is considered to be a fixing commit for *TensorFlow#1234*). We found a fixing commit for a total of 5,578 bugs. Note that some bugs may have more than one fix commit. For such bugs, we consider all fixing commits to be a single fixing patch.

Identifying Performance Bugs. Next, we aim to identify performance bugs from their non-performance counterparts among the bug reports collected. As aforementioned, some of the bug reports may be categorized through labels. Thus, we consider bug reports whose labels suggest an association with the performance of the framework (e.g., the *performance* label) as performance bug reports. However, those labels are optional and may not be used for all bug reports. To increase the recall of identifying the performance bug reports, we follow prior studies and use a keyword-search (e.g., *slow*, *laggy*) to further identify performance-related bug reports [20], [21], [23], [30], [31]. We make the list of the full list of keywords used publicly available online [32]. In total, we collect a total of 2,261 performance bugs and 31,916 non-performance bugs. We manually investigate the fixed performance bugs and discover that 141 out of 160 bugs identified as performance-related are true positives, implying that our heuristic had a precision of around 88.13%, which is consistent with previous work that used a similar heuristic (e.g., [23]).

IV. Case Study Results

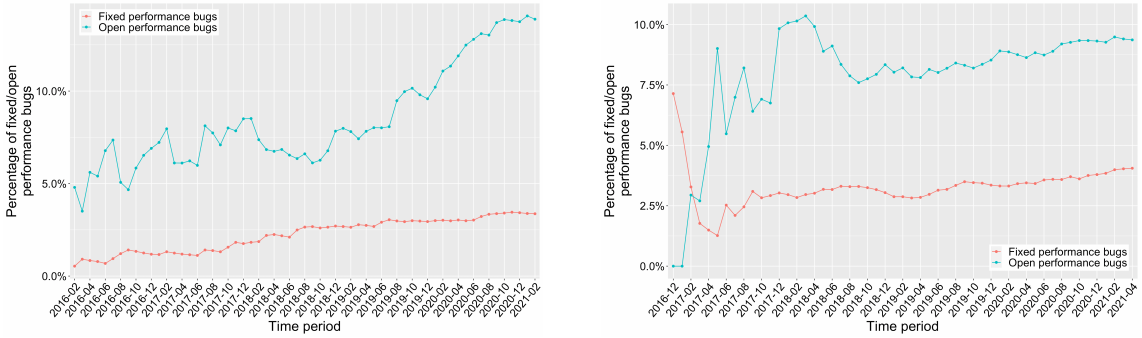
In this section, we first conduct a quantitative study to compare performance and non-performance bugs, and to better understand their prevalence and differences in characteristics. Then, we conduct a qualitative study and derive a taxonomy for the causes of the performance bugs.

RQ1: What is the trend of reported and fixed performance bugs over time?

Motivation. The performance of DL frameworks may influence all the applications that use them. Notably, performance bugs in DL frameworks may exacerbate the infamously lengthy training process of DL models and their inference time, which may be critical in certain applications (e.g., self-driving cars). In this RQ, we study the prevalence of performance bugs in DL frameworks and how it changes over time. An increasing trend may indicate the need for more attention from the research community to assist DL framework developers in addressing performance bugs.

Approach. We study the trend of reported and fixed performance bugs across the studied time periods. We considered all bugs (both fixed and open) from February 2016 to February 2021 for TensorFlow, and from December 2016 to April 2021 for PyTorch. We chose the start date as three months after the release of each framework to make sure the frameworks had enough bugs reportedto be studied.

We compute the percentage of fixed and open performance bugs among all fixed and open bugs (including non-performance bugs). For each month, we calculate the percentage based on **all the bugs** reported in the prior months (e.g., we compute the percentage of fixed performance bugs in September 2018 by considering all the bug reports that are fixed up to September 2018). We use the right-sided Cox-Stuart statistical test [33] to investigate the existence of an



(a) TensorFlow (b) PyTorch

Fig. 1: Percentage of open and fixed performance bugs among **ALL** open and fixed bugs in DL frameworks, respectively

increasing trend. To quantify the significance of the observed data, we report the p-values obtained from the statical tests we conduct (i.e., the lower the p-value, the higher the probability that a significant increasiing trend exits).

Results. ***Despite increasing efforts to fix performance bugs in DL frameworks over time, their prevalence is still increasing.*** Figure 1 shows the trend of the percentage of open and fixed performance bugs against all bugs in the two studied DL frameworks. We obtain low p-values for both the rate of open (i.e., p-value $<$ 0.0001 for TensorFlow and p-value $<$ 0.005 for PyTorch) and fixed (i.e., p-value $<$ 0.0001 for TensorFlow and p-value $<$ 0.0001 for PyTorch) performance bugs. In short, for both DL frameworks, the percentage of performance bug reports (either open or fixed) continues to rise, which shows the prevalence and increased concern on performance issues.

The performance bugs are fixed at a higher rate compared to non-performance bugs. Namely, the percentage of fixed bugs in TensorFlow was 0.54% in February 2016 and increased to 3.37% in February 2021 (i.e., an increase of around 2.83%). However, TensorFlow's percentage of open performance bugs shows a much higher increase. For example, there is an increase of 9.1% between February 2016 and February 2021 (i.e., from 4.79% to 13.89%). The finding may indicate that the rate of new performance bug reports increases much faster than the rate at which developers fix performance bugs.

We observe a similar trend in PyTorch. The percentage of open performance bugs increased by 9.37% between December 2016 and April 2021 (i.e., from 0.0% to 9.37%). The percentage of fixed performance bugs is higher initially, then stabilized and started to increase steadily to around 4% in April 2021. After some manual investigation, we found that the higher percentage of fixed performance bugs at the beginning of the studied period is due to PyTorch having fewer bug reports. Similarly, there were fewer open bug reports in PyTorch around the end of 2017 and the beginning of 2018, causing the percentage of open performance bugs to be higher. Nevertheless, we observe a trend where the percentage of performance bugs (either open or fixed) continues to increase.

The rate of newly reported performance bugs increases faster than fixed performance bugs. Moreover, the ratio of performance bugs among all bugs also increases, which shows that performance problems may be an increasing concern in DL frameworks.

RQ2: What are the differences between performance and non-performance bugs in terms of fixes and community engagement?

Motivation. Prior studies have shown that performance bugs in traditional systems have different characteristics (e.g., bug fixing time) than non-performance bugs [20], [23], [24], [34], [35]. Traditional software systems and DL frameworks have major differences. Namely, DL tasks are computation-heavy and require hardware integration [1], [3]. In this RQ, we quantitatively study the characteristics of performance bugs in DL systems by studying the differences between performance and non-performance bugs along different dimensions (i.e., their complexity and the community engagement they attract). The findings may provide preliminary insights on the characteristics of performance bugs in DL frameworks.

Approach. We compare performance and non-performance bugs by analyzing two dimensions: the complexity of the bugs and the engagement they receive from the community.

Bug complexity. We calculate two metrics as proxies to measure bug complexity [36]:

Bug fixing time. We compute the time difference between the initial report date and the close date. For the bugs that have been re-opened, we consider the latest date on which the bugs were closed. A longer fixing time may be indicative of more complex bugs.

Bug fix size. We compute the total number of lines of code deleted, added, and modified in every bug fix as a proxy for the fix complexity [36], [37]. Note that if a bug fix is composed of multiple commits, we take the sum of all the commits.

Community engagement. Developers may focus on bugs that impede their progress in a specific use-case [38]. A higher community engagement in a particular category of bugs may indicate that those bugs have a wider impact. Thus, we calculate three metrics as proxies for community engagement:

TABLE II: The median values and effect size of the computed metrics for performance and non-performance bugs. The larger distributions are indicated by a (*).

	Complexity of bugs				Engagement					
	Fix time (in days)		Fix size		Time before first comment (hours)		Number of comments		Number of commentators	
	Median	Cliff's Delta	Median	Cliff's Delta	Median	Cliff's Delta	Median	Cliff's Delta	Median	Cliff's Delta
Performance bugs	20.34*	0.248 (small)	38.0*	0.111 (negligible)	14.7*	0.034 (negligible)	5.0*	0.191 (small)	3.0*	0.168 (small)
Non-performance bugs	10.46		28.0		12.1		3.0		2.0	

Time before first comment. We compute the time difference between the bug report's creation date and the date of the first comment (i.e., how fast the community reacts to the bug). We only consider bug reports that have at least one comment when computing this metric, and the first comment cannot be posted by the user who created the bug report.

Number of comments. We compute the total number of comments posted for a given bug as a proxy to measure the amount of associated discussion. A higher amount of discussion may indicate a higher engagement with the given bug and may correlate with more complex bugs [23], [39].

Number of commentators. We consider the number of distinct commentators in each bug report. More distinct commentators may reflect more interest from the community [39].

We use the Wilcoxon rank-sum test to determine whether there is a statistically significant difference between the performance and non-performance bugs, as it is a non-parametric test that does not make any assumptions about the underlying data distribution. To quantify the significance of the observed data, we report the p-values from the statistical tests we obtain (i.e., the smaller the p-value, the higher the probability that there exits a difference between performance and non-performance bugs).We also apply Cliff's delta [40] to measure the effect size. We quantify the effect size by using the thresholds proposed by Romano et al. [41] (i.e., effect size is *negligible* if $|d| \leq 0.147$, *small* if $0.33 < |d| \leq 0.474$, *medium* if $0.33 < |d| \leq 0.474$ and *large* if $0.474 < |d| \leq 1$).

Results. ***Performance bugs require 94.46% more time to fix compared to non-performance bugs, and their fix sizes are also 35.71% larger.*** Table II shows the median of the computed metrics for performance and non-performance bugs and their statistical test to compare distributions. Namely, the performance bugs take more time to fix (median of 20.34 v.s. 10.46 days, with a p-value < 0.0001 and a small effect size), and the fix sizes are larger (median is 38 v.s. 28 LOC, with a p-value = 0.024 and a negligible effect size). In short, the median fix time and fix size for performance bugs are 94% larger and 35% more, respectively, and all the metrics are statistically significant (i.e., p-value ≤ 0.05). Although we do not compare the results with bugs in traditional software systems, a prior study by Nistor et al. [24] found that performance bugs in traditional systems (e.g., Mozilla) take only, on average, 37% more time to fix compared to non-performance bugs. The finding may provide an initial hint that performance bugs in DL frameworks may have unique characteristics and require more attention from the research community.

While the first comments on performance bug reports take around 15.75% longer to be posted compared to non-performance bugs, the discussions involving performance bugs have around 66.67% more comments and 50% more commentators than non-performance bugs. As shown in Table II, the first comments take approximately 14.7 and 12.1 hours (median) for performance and non-performance bugs, respectively. While the differences are statistically significant (i.e., p-value < 0.001), the effect size is negligible. However, we find performance bugs have significantly more comments than non-performance bugs (median is 5 v.s. 3 comments, with a p-value < 0.0001 and a small effect size). Moreover, the discussions for performance bugs have significantly more distinct commentators than for non-performance bugs (median is 3 v.s. 2 commentators, with a p-value < 0.0001 and a small effect size). Our findings show more community engagement on performance-related bugs, which may reflect the community's interest in the problem.

Performance bugs take more time to fix, and the fix sizes are often larger. Moreover, there is a higher level of community engagement in discussing performance bugs. Our findings show initial evidence of performance bugs' complexity and importance in DL frameworks.

RQ3: What are the causes of performance bugs in DL frameworks?

Motivation. Traditional software suffers from frequent and costly performance issues [20], [21], [23], [24], [35]. In DL frameworks, such performance bugs may have different characteristics and may scale many-fold due to the repetitiveness of the training loop (e.g., involving many epochs through the same input data). Our goal is to derive a **comprehensive and detailed** taxonomy of the causes of performance bugs in DL frameworks. We believe our taxonomy will inspire future research and provide insights into the maintenance of performance problems in DL frameworks.

Approach. We conduct a comprehensive qualitative study on the causes of performance bugs in deep learning frameworks. Our manual study is composed of the following phases:

Phase I: To create the taxonomy for the performance bugs, we manually study all the fixed performance bugs that we identified in Section III. The first two authors of the paper (A1 and A2) independently derived an initial list of the causes by manually inspecting the relevant commit messages, source code, and bug reports. We found 12 main categories of performance bugs in DL frameworks.

Phase II: Authors A1 and A2 unified the derived reasons and compared the assigned reason for each performance bug. Any disagreement was discussed until a consensus was reached. The list of categories remains unchanged in this step. The inter-rater agreement of the coding process has a Cohen's kappa of 0.92, indicating almost a perfect agreement level [42].

TABLE III: Qualitative Analysis: Taxonomy of Performance Bugs in Deep Learning Frameworks [32].

Cause	Description	Impact (speed, memory, both)	Frequency
Memory inefficiency		**(4, 31, 2)**	**37/141 (26.24%)**
Unreleased memory resources	Memory resources not released after serving their purpose (e.g., stream not cleared, object reference count mishandling).	(1, 23, 0)	24/141 (17.02%)
Unnecessary memory allocation	Allocating unnecessary memory resources/references to implement the same functionality in the implementation.	(3, 8, 2)	13/141 (9.22%)
Traditional SE inefficiency	Traditional coding errors found in traditional software systems can result in downstream performance-related faults (e.g., inefficient loops, unnecessary copy when converting a read-only array to a tensor without flagging as read-only array, unnecessary caching caused by incorrect cache index, inefficient API usage).	**(22, 6, 1)**	**29/141 (20.57%)**
Threading inefficiency		**(11, 2, 1)**	**14/141 (9.93%)**
Lack of parallelization	Failure to reap the full benefits of multithreading (e.g., excessive usage of atomic operators resulting in unnecessary device synchronization.	(7, 0, 0)	7/141 (4.96%)
Excessive usage of multithreading	Making use of too many threads may lead to adverse consequences (e.g., thread starvation).	(1, 1, 1)	3/141 (2.13%)
Challenges in fine-tuning the optimum number of threads	Some tasks (e.g., iterating over a dataset) require the work to be divisible among the cores to fully utilize them. In some cases, if not done correctly, task division can lead to performance degradation.	(2, 0, 0)	2/141 (1.42%)
Excessive usage of thread-local variables	Thread-local variables are variables that are created and can only be accessed by the same thread. The excessive usage of such variables may lead to excessive memory consumption during forward propagation.	(0, 1, 0)	1/141 (0.71%)
Thread not returning on time	Certain multithreaded tasks need to wait for a thread to return before continuing the computation (e.g., loading data during training process). In those cases, if a thread does not return as soon as it can, this will cause a performance degradation.	(1, 0, 0)	1/141 (0.71%)
Matrix computation inefficiency		**(9, 2, 1)**	**12/141 (8.51%)**
Trade-off of using different linear libraries/operators	Failure to use the appropriate linear algebra library/operator (e.g., the trade-off of using Magma and cuBlas for certain sizes of input)	(3, 0, 0)	3/141 (2.13%)
Use of intermediate matrices	Making use of temporary intermediate matrices in the implementation of certain linear algebra operations, leading to extra operations or extra memory consumption.	(2, 1, 0)	3/141 (2.13%)
Inefficiency for specifc sizes and/or shapes	Failure to optimize execution for inputs of specific sizes and/or shapes.	(1, 1, 0)	2/141 (1.42%)
Inefficient broadcasting	Broadcasting is usually done so that two matrices, involved in an operation, have compatible shapes. However, when done inefficiently (e.g., through the usage of an expensive *expand* operation, which is used to expand a tensor to a larger size).	(1, 0, 1)	2/141 (1.42%)
Lack of vectorization	Vectorization is the process of transforming a scalar program into a vectorized program. Given the prevalence of linear algebra operations in DL applications, not vectorizing inputs in those operations can cause a performance degradation.	(2, 0, 0)	2/141 (1.42%)
Inefficient AI algorithm implementation	Inefficient implementation of certain core algorithms (e.g., wrong choice of sampling algorithms).	**(7, 1, 0)**	**8/141 (5.67%)**
Inefficiency for certain input data types	Slowdown occurring for certain input data types (e.g., using half-precision data types (e.g., *float16*) on GPU may result in slowdowns).	**(6, 0, 0)**	**6/141 (4.26%)**
Unnecessary computation		**(0, 3, 2)**	**5/141 (3.55%)**
Unnecessary gradient computation	Computing the gradient of inputs that are not needed to implement the wanted functionality.	(0, 2, 1)	3/141 (2.13%)
Unnecessary node creation	Unnecessarily root node creation in a computational graph that has only one node.	(0, 1, 0)	1/141 (0.71%)
Unnecessary operation	Unnecessary matrix multiplication by 1.	(0, 0, 1)	1/141 (0.71%)
Inefficient hardware optimization		**(4, 1, 0)**	**5/141 (3.55%)**
Wrong device placement	Incorrectly or inadvertently executing a task on the wrong device (e.g., executing the training process on CPU when GPU is available).	(2, 1, 0)	3/141 (2.13%)
Lack of support for hardware optimization libraries	Not making use of hardware optimization libraries due to a lack of support for the libraries.	(2, 0, 0)	2/141 (1.42%)
Hardware components communication overhead		**(6, 0, 0)**	**6/141 (4.26%)**
Local communication overhead	Overhead associated with communication between hardware components on the same local machine (e.g., transferring a sequence from CPU to GPU locally).	(4, 0, 0)	4/141 (2.84%)
Remote communication overhead	Overhead associated with the communication between different components in a distributed runtime (e.g., overhead associated with workers communicating with a remote server).	(2, 0, 0)	2/141 (1.42%)
Inefficient caching	Inefficiently making use of cache (e.g., lack of caching or excessive usage of caching). This includes the unnecessary caching of certain gradient values.	**(2, 2, 0)**	**4/141 (2.84%)**
C/C++ abstraction	Bugs caused by the abstraction of the C/C++ backend using a higher level language (e.g., Python).	**(1, 2, 0)**	**3/141 (2.13%)**
Other	Performance bugs that do not fit into any of the categories identified (e.g., user-end bugs, not enough discussion).	**(8, 4, 0)**	**12/141 (8.51%)**

We make the taxonomy and the corresponding bug report IDs publicly available online [32].

Results. Table III shows the taxonomy of performance bugs in DL frameworks. Below, we discuss each category in detail.

– Memory inefficiencies (37/141, 26.24%). Memory inefficiencies are the most common performance bugs in DL frameworks. One potential reason may be that a large part of DL frameworks are implemented in the C/C++ and CUDA programming languages for performance gains and finer-grained memory management. However, such memory resource management is often manual and may more likely result in performance bugs. Notably, DL requires a constant exchange of in-memory data between the different components (i.e., RAM, CPU, GPU, etc.), which makes those memory inefficiencies ever more frequent.

Unreleased memory allocations (24/141, 17.02%) is the most frequent cause of memory-related performance bugs. For example, when an error happens, developers may forget to free up memory resources, which can lead to a memory leak (e.g., (e.g.,*TensorFlow#14800* [43]). In some cases (e.g., *PyTorch#50522* [44]), in a distributed runtime, Remote Procedure Calls (RPCs), which allow users to communicate and train models across multiple machines, are only cleared from memory once when timed out. This causes PyTorch to use memory resources for needlessly long periods (i.e., memory bloat). The severity of this bug depends on the number of RPCs used. If the number of RPCs used is small, the memory bloat may be too insignificant to detect. Developers also often discuss the importance of using various grain sizes to expose inefficient memory handling that only manifests under specific workloads (e.g., *PyTorch#50522* [44]). This highlights the need for better test oracles that consider different workloads to improve the detection of memory bugs in DL frameworks.

Another type of memory-related performance bug is *unnecessary memory allocations* (13/141, 9.22%). Such bugs are caused by unnecessarily copying or creating objects. For example, in *TensorFlow#14572*, loading datasets and checkpoints from Amazon Web Services (AWS) cloud storage service caused unnecessary copies of the retrieved data, causing both a slowdown and memory bloat. Other examples include unnecessary copies of a tensor because developers wrongly assumed that a function would return a new tensor (e.g., *PyTorch#5611* [45] and *PyTorch#6222* [46]).

Our findings show that DL frameworks are prone to memory-related bugs due to the usage of lower-level programming languages and the constant exchange of in-memory data between the different components at play. Hence, developers may benefit from the creation of automatic tools for better memory management mechanisms, especially for heterogeneous computation environments.

Many memory-related performance bugs are related to the usage of low-level languages and the complex exchange of in-memory data between the different components or external resources.

– Traditional SE inefficiency (29/141, 20.57%). Despite

the fundamental differences between DL-based and traditional software systems, the performance bugs that affect traditional software systems, such as function misuses, inefficient usage of APIs or unnecessary conditions [20], [47], are also common in DL frameworks. Specifically, many trivial bugs may cause a significant slowdown due to the repetitive nature of the training loops and the prevalence of large datasets in DL frameworks. For example, in *PyTorch #10851* [48], using a profiler is particularly slow due to the usage of the += operator for string concatenation, which causes repeatedly copying a new string object, instead of a more efficient *append* call. Many previous works have investigated these performance bugs [20], [23], [24], [34], [35]. More work is needed to adopt prior tools to help improve the performance of DL frameworks.

Traditional performance bugs are common in DL frameworks. Such bugs may be exacerbated by the repetitive and resource-hungry nature of the model training process. Future research may adapt existing tools to further analyze and improve the performance of DL frameworks.

– Threading inefficiency (14/141, 9.93%). While multithreading is critical for improving DL framework performance for resource-intensive tasks, inefficient thread usage can cause performance issues. Some issues include not being able to compute the optimal number of threads for a given task and/or adopting threading configurations based on the runtime environment (e.g., the libraries used and CPU vs GPU).

Lack of parallelization (7/141, 4.96%) is the most prominent cause of threading inefficiencies, which happens when a low number of threads is being used or when the runtime becomes sequential. We find that this issue is commonly found in GPU-related code due to the prevalence of parallelization in GPU computations. One example is the excessive use of atomic operations. These operations help developers manage concurrent variable accesses and are thus asynchronous. For example, in *PyTorch#9646* [49], the excessive use of atomic operations when managing CUDA streams causes the runtime to be synchronized and, hence, causes a performance degradation. Developers fixed the bug by overhauling the stream creation process to implement a priority system.

Another common cause is the *excessive usage of multithreading* (3/141, 2.13%). Using too many threads to execute a task may lead to performance degradation due to overheads associated with multithreading (e.g., context switches). For example, in *TensorFlow#3470* [50], an excessive number of threads are used for training in a distributed environment due to the creation of an unbounded number of threads that do not depend on the completion of the previous ones.

The final causes are the *challenges in fine-tuning the optimum number of threads* (2/141, 1.42%). We find that fine-tuning the optimum number of threads for a given task may be challenging in some cases, and a suboptimal number of threads could lead to performance degradation. Deciding the optimal number of threads depends on numerous factors, such as the number of available CPU/GPU cores or the workload associated with the task. For example, in *PyTorch#24080* [51], developers found that the calculation for the number of threads varied across different multithreading frameworks. Such imprecision led to performance degradation during DL training tasks. There are also other threading issues (i.e., *excessive usage of thread-local variables* and *threads not returning on time*) that we found in the DL frameworks. Although the number of instances is small, these issues still caused significant performance impacts.

While parallelization can help circumvent the computationally-demanding nature of certain DL tasks, there are challenges associated with determining the most efficient threading configuration (e.g., number of threads) based on different factors (e.g., across a varying number of CPU/GPU cores). Future studies may assist developers in automatically tuning those configurations based on the environment to improve performance.

The optimum threading configurations depend on numerous factors, including the environment and the task at hand. Developers may benefit from future works that focus on automating such configurations.

– Matrix computation inefficiency (12/141, 8.51%). Matrices represent the multidimensional containers of algebraic elements and the data structure used in model computation (e.g., training and inference). Therefore, computations associated with matrices and their efficiency play a crucial role in DL frameworks. Some bugs in this category are the result of a *trade-off of using different linear algebra libraries/operators* (3/141, 2.13%). Such cases occur when certain linear algebra operators from diverse libraries have different performances under specific conditions (e.g., slower on GPU/CPU or a particular matrix size). For example, in *PyTorch#42265* [52], a specific matrix operation during inference suffers from a performance slowdown when using GPU when the input is a small matrix. This is because of the developers' usage of cuBlas [53], which executes the entire operation on GPU. For such small workloads, the overhead of transferring the data to GPU may outweigh the benefits of using GPU acceleration.

On the other hand, Magma [54] executes such operations on the CPU before moving the output to GPU. The developers were not aware of the differences between the libraries and used the libraries inefficiently (i.e., not using each library in their best-suited scenario). To fix the issue, developers added a manual condition to choose the most optimal library to use at runtime. Another example is *TensorFlow#17246* [55], where the *memcpy* operator, which is used to copy tensors around different memory addresses in linear algebra operations, is slower for large tensors on the Linux OS. Developers did not find an explanation for this behavior, and instead used a different operation (i.e., *memmove*) to prevent the slowdown.

We also find some matrix computation *inefficiencies for specifc sizes and/or shapes* (2/141, 1.42%). For example, in *PyTorch#12006* [56], a large batch size results in low GPU utilization. To fix this, the developers opted to transpose and reshape the inputs when the batch size for this operation exceeds a certain number. These bugs highlight the need for

more thorough testing that involves different libraries and operators alongside different input sizes and shapes.

Moreover, matrices of different dimensions and sizes cannot be added, subtracted or used in most arithmetic operations. In such cases, developers often use broadcasting, which duplicates a smaller matrix across the larger matrix's dimension so that their shapes are compatible. We find that one cause for matrix computation inefficiencies is the *inefficient implementation of the broadcasting process* (2/141, 1.42%). For example, in *PyTorch#17206* [57], the use of extra *expand* and *reshape* operations, instead of making use of broadcasting, results in a computation time of almost nine minutes, when it was supposed to take seconds.

We find that many performance bugs related to matrix computations are caused by incorrect library usage and the peculiarities of linear algebra operations. While many previous works have benchmarked DL frameworks (e.g., [58]–[60]), there is little empirical evidence that discusses the trade-offs between using different linear algebra libraries and operators. Given the number of available libraries and the wide variety of hardware, future studies may further assist developers in understanding the trade-offs of using the different tools available across different situations. Future studies may also investigate potential code smells or optimization opportunities when doing matrix computation (e.g., choosing the optimal input shape) to further improve the performance of DL frameworks.

Performance bugs associated with linear algebra operations may lead to exacerbated effects. Future studies may focus on more exhaustively testing these operations and on helping developers abstract complex linear algebra optimization and parallelization.

– Inefficient AI algorithm implementation (8/141, 5.67%). DL is an active research area, with new and more efficient AI algorithms constantly being proposed. However, developers may not keep up with the latest research and may use less efficient AI algorithms. For example, in *PyTorch#7883* [61], the the short-time Fourier transform is slower on PyTorch than on Librosa [62]. This is due to the fact that, unlike PyTorch, Librosa's implementation is based on the fast Fourier transform [63]. Another example is *PyTorch#11931* [64], where sampling numerous elements with no replacement from a multinomial distribution is slow. To fix this, the developers implemented a fast-path inspired by the NumPy [65] implementation and the weighted random sampling algorithm [66].

We observe that these bugs were discovered by comparing the performance of different implementations of the same operation. This suggests approaches such as performance differential testing may help uncover possible performance bugs or help developers choose more efficient libraries.

Developers may use less efficient algorithms in their implementations. In addition to adopting the newest algorithms, developers compare similar implementations across libraries to uncover performance bugs and optimization opportunities.

– Inefficiency for certain input data types (6/141, 4.26%). DL frameworks support various data types [67]. We find that there may be performance bugs specific to certain data types. The majority of such bugs are concerned with using half-precision data types (e.g., *float16*) on GPU. Using these data types for the training and inference processes should result in faster execution time with lower memory consumption, at the expense of some precision in the outputs. However, in some cases, using these data types may lead to a performance degradation. For all the manually studied bugs, developers do not know why such performance bugs occur. As a workaround, developers cast half-precision inputs to a single-precision data type (e.g., *float32*) and then cast the output back to half-point (e.g., *TensorFlow#41715* [68]). Such flakiness reveals the need for further work to better understand the advantages and disadvantages of using different data types and their expected and actual performance on different configurations (e.g., CPU/GPU and Windows/Linux).

There is unexpected performance degradation when running certain data types on different configurations. There may be opportunities for performance optimization when training/applying the models using different data types.

– Unnecessary computation (5/141, 3.55%). Unnecessary computations are a common cause of performance degradation in traditional software systems [69]. We find that such issues also exist in DL frameworks but for DL-specific contexts. The most prevalent cause of these bugs is *unnecessarily gradient computations* (3/141, 2.13%), where gradients are calculated through backward propagation when not required. For example, in *PyTorch#7261* [70], the unnecessary gradient calculation of an argument in the spectral normalization implementation increased the memory consumption and caused an Out-of-Memory exception. Another cause of such bugs is an *unnecessary creation of nodes* (1/141, 0.71%) in computational graphs. In PyTorch and TensorFlow, computational graphs are the foundations of backward propagation in calculating the gradients of neural networks. Our study finds cases where nodes in the graph are unnecessarily created, causing additional computation and memory consumption.

Similar to traditional software systems, unnecessary computations may also affect the performance of DL frameworks, especially during backward propagation.

– Inefficient hardware optimization (5/141, 3.55%). DL frameworks enable users to use numerous types of devices (e.g., CPU and GPU). Typically, the CPU is used for the data preprocessing tasks, while the GPU is preferred for the computationally demanding training tasks. However, we find cases where a *wrong device placement* (3/141, 2.13%) causes performance degradation. For example, in *TensorFlow#32138* [71], iterating through a dataset automatically places the computation on CPU due to the assumption that iterating through the data must be for preprocessing. Such assumption is false for customized training loops, where iterating through a dataset is done to train a model. In such cases, the computation (i.e., the training) was inadvertently moved to CPU, resulting in

performance degradation. Another cause of such bugs is a *lack of support for hardware optimization libraries* (2/141, 1.42%). Many hardware manufacturers (e.g., Nvidia) develop libraries that optimize DL computations on their devices. However, DL frameworks may not support some of those libraries, which results in suboptimal performance. For example, PyTorch does not support Intel's MKL-DNN [72] for the average pooling operation (*PyTorch#19797* [73]). Thus, despite having the compatible hardware (i.e., an Intel CPU) to benefit from the library's improved performance, the user was still unable to do so for such operation. Future studies should help developers automatically integrate various hardware optimization libraries for optimal performance across multiple platforms.

Failing to properly integrate the usage of hardware optimization libraries specific to certain hardware devices may cause performance degradations. Future research may focus on aiding developers in properly supporting those libraries for optimal performance across various platforms.

– Hardware components communication overhead (6/141, 4.26%). Some bugs are due to the overhead associated with the communication between the numerous hardware components in DL (i.e., RAM, CPU, GPU, etc.). These bugs may be associated with a *local communication overhead* (4/141, 2.84%). For example, in *PyTorch#158* [74], the process of creating a tensor on GPU from a sequence on CPU, suffers from a slowdown, due to the sequence not being buffered. These bugs may have exacerbated adverse consequences on the performance of DL frameworks, notably due to some DL tasks involving large amounts of data or a constant exchange of data between different hardware devices (e.g., CPU and GPU). These bugs may also be associated with a *remote communication overhead* (2/141, 1.42%). For example, in *TensorFlow#11411* [75], there is an overhead associated with different workers fetching data from remote servers. The performance degradation that ensues is exacerbated by using GPU-acceleration due to the overhead associated with the data being serialized on CPU before being passed on GPU. Future studies should focus on further enabling efficient communication between these components and on better understanding the performance bugs that are caused by these interactions.

Efficient communication between different hardware components is important for ensuring the performance of DL frameworks. Future studies may focus on these interactions and on the performance bugs they may cause.

– Inefficient caching (4/141, 2.84%). Some performance bugs are due to the inefficient usage of caching techniques, which are used to speed up data access operations. For example, in *PyTorch#33334* [76], concatenating a sequence of tensors is slow due to the input being consistently loaded instead of being loaded once and then cached. This bug only occurs when using a single core on CPU. This may highlight the need for comprehensive DL testing, which considers different sets of environment configurations. Moreover, such bug may be exacerbated by the use of large datasets (i.e., tensors).

Failing to efficiently store data in DL frameworks may cause significant performance degradation. This is partly due to the data-intensive nature of certain DL tasks. We also find that these bugs may only occur in specific conditions (e.g., certain environment configurations).

– C/C++ abstraction (3/141, 2.13%). Interactions and clashes in behavior between C/C++ and higher-level languages (e.g., Python) are also the cause of some performance bugs in DL frameworks. For example, in *TensorFlow#40758* [77], a clash of behaviour between TensorFlow's C/C++ backend and its GO implementation led to a cache blow-up. Previous research has focused on software systems that use the polyglot architecture (i.e., systems written in multiple programming languages) (e.g., [78]). However, no such studies have been conducted on the adoption of the polyglot architecture for DL frameworks. These bugs highlight the need for such works. This may also highlight the need for tools that may aid developers in addressing such bugs.

The abstraction of the C/C++ backend using a higher-level language may cause erroneous behaviour, leading to huge performance degradation. This highlights the need for future research into the effects of using the polyglot architecture for DL frameworks.

– Other (12/141, 8.51%). The bugs under this category do not correspond to the aforementioned categories. The bugs include user-end bugs (e.g., *PyTorch#18853* [79]), bugs that were not discussed enough (e.g., *Pytorch#18405* [80]) and bugs that are due to bugs in third-party libraries (e.g., *PyTorch#82* [81]).

V. Implications and Future Work

Based on our empirical findings, we present actionable implications and future work for two groups of audiences: 1) researchers and 2) framework developers.

A. Researchers

R1: There may be unexpected performance differences when DL frameworks are used with different combinations of environment configurations (e.g., the data type used at runtime). Future studies should further explore such unexpected performance differences under different combinations of configurations. As seen in RQ3, there were cases where performance bugs only occur under certain conditions. For example, we find cases where using the *float16* data type causes a slowdown instead of a performance improvement. In this case, flakiness in the slowdown was difficult to rationalize, and for the inefficacy of data types, developers applied a workaround to bypass them (e.g., temporary conversion). While there is tremendous research studying flakiness in traditional systems, there is a lack of research studies in DL systems in benchmarking performance trade-offs for different configurations [58], especially in detecting flakiness in performance. Our study raises new research opportunities for better configuration recommendations in future DL systems by studying their trade-offs.

R2: Similar libraries may have different performances, especially when operating in different environments. Future research should help DL framework developers choose the most performant library given various scenarios. We found that there were several scenarios where a lack of understanding of the performance differences between external libraries led to performance degradation. For example, the trade-offs between linear algebra libraries (e.g., cuBlas v.s. Magma) cause matrix inefficiencies. We also find cases where particular hardware optimization libraries are not supported by the DL framework, leading to suboptimal performance. Managing external libraries is a known challenge in many traditional systems. Similarly, in DL frameworks, we find that while there may be multiple external libraries that offer many similar functionalities (e.g., matrix solver), each library may have different performance optimization based on factors such as the OS and the underlying hardware. Hence, creating a performance benchmark on how different libraries perform under different environments, hardware devices, or even versions may help DL framework developers make better decisions in choosing the libraries. Future research may also help DL frameworks automatically choose or recommend libraries that have better performance given an environment setting.

R3: There are future research opportunities for finding the optimal number of threads for different workloads. As we found in RQ3, parallelization can help circumvent the resource-intensive nature of DL tasks. However, there are challenges associated with determining the most efficient threading configuration (e.g., number of threads) based on different factors (e.g., across many CPU/GPU cores and workloads). Future studies may assist developers in automatically tuning the threading configuration based on the environment.

R4: Future studies should help DL framework developers improve and optimize hardware utilization (i.e., CPU and GPU usage) from the perspective of linear algebra operations. We find that specific matrix sizes and shapes may cause hardware underutilization, resulting in performance degradation. For example, we observe cases where certain batch shapes lead to a low GPU usage and hence, sub-optimal performance. In this case, while DL developers manipulate the matrix into different shapes (e.g., reshape) to allow higher GPU utilization, the issue is partially fixed, and problems reoccur in the future. In another case, we find slowdowns caused by matrix reshaping when dealing with matrix operations for different shapes and sizes. Our study opens an interesting direction for future research to optimize matrix operations involving calculating and reshaping matrices to improve and optimize hardware utilization. Notably, future research is required to identify and remove code smells related to inefficient matrix operations to improve the performance of DL systems.

B. Framework Developers

F1: DL framework developers may benefit from a better understanding of the state-of-the-art in SE research. We find that traditional SE programming errors (e.g., String append v.s. += operator) also appear in DL systems. Such problems are more susceptible to performance slowdown due to large data usage and the repetitiveness of computations (e.g., gradients) in certain DL tasks. Many cutting-edge studies in SE research have already investigated such performance bugs and how to better address them. Hence, we find that developers can benefit tremendously from having a better understanding of the state-of-the-art in SE research in order to improve the quality of DL frameworks.

VI. Threats to Validity

External Validity. We conducted our study on TensorFlow and PyTorch. Thus, our findings may not generalize to all DL frameworks. However, these two are the most popular DL frameworks [13], well-maintained, consistently updated, and used in various commercial settings.

Internal Validity. We conducted our study based on the performance bug reports on GitHub repositories. To increase the precision and recall of finding the bug reports, we follow prior studies [20], [21], [23], [30], [31] and use both developer-provided labels and performance-related keywords (e.g., *slow*, *laggy*) to identify performance bugs. Our manual verification found that the precision of our approach is above 88%, which is similar to what was reported in a prior study [23].

Construct Validity. DL frameworks are undergoing continuous development, so some studied issues may be related to certain releases/versions of hardware or libraries. However, with constant hardware innovations (e.g., new Nvidia GPUs) and software changes, such new releases would always be part of DL framework evolution. Hence, we believe our trend analysis (RQ1), which includes around five years of data, should reflect such patterns to a certain degree. Our manual study may have biases on the causes of the performance bugs. Thus, two authors independently examined all available software artifacts. The inter-rater agreement was high between the two authors. We do not claim to find all performance bugs. However, we show the existence of such issues and identify further research opportunities.

VII. Conclusion

Deep Learning (DL) has gained tremendous popularity in recent years due to the availability of open-sourced DL frameworks, which empower developers to implement DL models efficiently. However, optimizing DL frameworks may have its unique challenges due to the peculiarities of DL. In particular, fixing performance bugs is vital to avoid severe life-threatening consequences. In this work, we collected the performance bug reports from the TensorFlow and PyTorch GitHub repositories and studied their characteristics and prevalence compared to non-performance bugs. We find that: 1) performance bugs may increasingly become a concern in DL frameworks; 2) performance bugs take significantly more time and necessitate significantly larger fixes; and 3) we build a comprehensive and detailed taxonomy of the root causes of performance bugs in DL frameworks. We hope our findings can inspire future research that focus on improving the qualiy of DL frameworks and DL systems.

REFERENCES

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, 2015, pp. 2503–2511.

[2] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 100–111.

[3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 291–300.

[4] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *Database Systems for Advanced Applications*, 2020, pp. 604–620.

[5] ——, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[6] J. Chen, Y. Liang, Q. Shen, and J. Jiang, "Toward understanding deep learning framework bugs," *Computing Research Repository (CoRR)*, 2022.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[8] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS 2017 Workshop on Autodiff*, 2017.

[9] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, *Scaling Learning Algorithms toward AI*, 2007, pp. 321–359.

[10] I. Arel, D. C. Rose, and T. P. Karnowski, "Deep machine learning - a new frontier in artificial intelligence research [research frontier]," *IEEE Computational Intelligence Magazine*, vol. 5, no. 4, pp. 13–18, 2010.

[11] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[12] C. Sestili, "Deep learning: Going deeper toward meaningful patterns in complex data," Carnegie Mellon University's Software Engineering Institute Blog, Feb. 12, 2018. [Online]. [Online]. Available: http://insights.sei.cmu.edu/blog/deep-learning-going-deeper-toward-meaningful-patterns-in-complex-data/

[13] J. Hale, "Deep learning framework power scores 2018," Sep 2018. [Online]. Available: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

[14] 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 (ISSTA)*, 2018, pp. 129–140.

[15] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 104–115.

[16] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[17] 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.

[18] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 674–685, 2021.

[19] J. Cao, B. Chen, C. Sun, L. Hu, and X. Peng, "Characterizing performance bugs in deep learning systems," *Computing Research Repository (CoRR)*, 2021.

[20] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *Sigplan Notices - SIGPLAN*, vol. 47, 2012.

[21] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, 2014.

[22] S. Afjehei, T.-H. P. Chen, and N. Tsantalis, "iperfdetector: Characterizing and detecting performance anti-patterns in ios applications," *Empirical Software Engineering*, vol. 24, 2019.

[23] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 199–208.

[24] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 237–246.

[25] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, 2005, pp. 1–5.

[26] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, 2007, pp. 9–9.

[27] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk..." in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*, 2006.

[28] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: A historical view on open and closed source projects," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 2009, pp. 119–128.

[29] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 913–923.

[30] C. Smith and L. Williams, "Software performance antipatterns; common performance problems and their solutions." in *Int. CMG Conference*, 2001, pp. 797–806.

[31] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Trans. Softw. Eng.*, vol. 42, no. 12, pp. 1148–1161, 2016.

[32] hhttps://github.com/dlframeworkperfbugs/performance-bugs-in-dl-frameworks.

[33] D. Cox and A. Stuart, "Some quick sign tests for trend in location and dispersion," *Biometrika*, vol. 42, pp. 80–95, 1955.

[34] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: A case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 93–102.

[35] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016.

[36] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 82–91.

[37] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[38] A. Lee, J. C. Carver, and A. Bosu, "Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: A survey," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 187–197.

[39] D. Arya, W. Wang, J. L. C. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 454–464.

[40] J. D. Long, D. Feng, and N. Cliff, *Ordinal Analysis of Behavioral Data*. American Cancer Society, 2003, ch. 25, pp. 635–661.

[41] J. Romano and J. Kromrey, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys?" in *Annual Meeting of the Florida Association of Institutional Research*, 2006.

[42] A. Viera and J. Garrett, "Understanding interobserver agreement: The kappa statistic," *Family medicine*, vol. 37, pp. 360–363, 2005.

[43] https://github.com/tensorflow/tensorflow/issues/14800, accessed: 2021-08-26.
[44] https://github.com/pytorch/pytorch/issues/50522, accessed: 2021-08-26.
[45] https://github.com/pytorch/pytorch/issues/5611, accessed: 2021-08-26.
[46] https://github.com/pytorch/pytorch/issues/6222, accessed: 2021-08-26.
[47] M. Sujon, M. Shafiuzzaman, M. M. Rahman, and R. Rahman, "Characterization and localization of performance-bugs using naive bayes approach," in *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, 2016, pp. 791–796.
[48] https://github.com/pytorch/pytorch/issues/10851, accessed: 2021-08-26.
[49] https://github.com/pytorch/pytorch/issues/9646, accessed: 2021-08-26.
[50] https://github.com/tensorflow/tensorflow/issues/3470, accessed: 2021-08-27.
[51] https://github.com/pytorch/pytorch/issues/24080, accessed: 2021-08-26.
[52] https://github.com/pytorch/pytorch/issues/42265, accessed: 2021-08-24.
[53] "cublas," https://docs.nvidia.com/cuda/cublas/index.html, accessed: 2010-09-30.
[54] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.
[55] https://github.com/tensorflow/tensorflow/issues/17246, accessed: 2021-08-24.
[56] https://github.com/pytorch/pytorch/issues/pytorch12006, accessed: 2021-08-26.
[57] https://github.com/pytorch/pytorch/issues/17206, accessed: 2021-08-26.
[58] L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin, and Q. Zhang, "Benchmarking deep learning frameworks: Design considerations, metrics and beyond," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1258–1269.
[59] S. Ganeshan, N. K. Elumalai, and R. Achar, "A comparative study of magma and cublas libraries for gpu based vector fitting," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.
[60] R. Elshawi, A. Wahab, A. Barnawi, and S. Sakr, "Dlbench: a comprehensive experimental evaluation of deep learning frameworks," *Cluster Computing*, vol. 24, no. 3, pp. 2017–2038, 2021.
[61] https://github.com/pytorch/pytorch/issues/7883, accessed: 2021-08-26.
[62] B. McFee, C. Raffel, D. Liang, D. Ellis, M. Mcvicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th python in science conference*, 2015, pp. 18–24.
[63] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch, "What is the fast fourier transform?" *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, 1967.
[64] https://github.com/pytorch/pytorch/issues/11931, accessed: 2021-08-27.
[65] https://www.openmp.org/, accessed: 2021-08-26.
[66] P. Efraimidis and P. P. Spirakis, *Weighted Random Sampling*. New York, NY: Springer New York, 2016, pp. 2365–2367.
[67] G. Sapunov, "Fp64, fp32, fp16, bfloat16, tf32, and other members of the zoo," May 2020. [Online]. Available: https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407
[68] https://github.com/tensorflow/tensorflow/issues/41715, accessed: 2021-08-26.
[69] Z. Alzamil and B. Korel, "Application of redundant computation in software performance analysis," in *Proceedings of the 5th International Workshop on Software and Performance*, 2005, pp. 111–121.
[70] https://github.com/pytorch/pytorch/issues/7261, accessed: 2021-08-26.
[71] https://github.com/tensorflow/tensorflow/issues/32138, accessed: 2021-08-26.
[72] https://oneapi-src.github.io/oneDNN/v0/index.html, accessed: 2021-08-26.
[73] https://github.com/pytorch/pytorch/issues/19797, accessed: 2021-08-26.
[74] https://github.com/pytorch/pytorch/issues/158, accessed: 2021-08-27.
[75] https://github.com/tensorflow/tensorflow/issues/11411, accessed: 2021-08-27.
[76] https://github.com/pytorch/pytorch/issues/33334, accessed: 2022-16-01.
[77] https://github.com/tensorflow/tensorflow/issues/40758, accessed: 2021-08-27.
[78] J. Harmanen and T. Mikkonen, "On polyglot programming in the web," *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, pp. 102–119, 2016.
[79] https://github.com/pytorch/pytorch/issues/18853, accessed: 2021-08-27.
[80] https://github.com/pytorch/pytorch/issues/18405, accessed: 2021-08-27.
[81] https://github.com/pytorch/pytorch/issues/82, accessed: 2021-11-08.