# Investigating and Testing Performance Issues in Deep Learning Frameworks

**Tarek Makkouk**

**A Thesis**

**in**

**The Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Master of Computer Science (Computer Science) at**

**Concordia University**

**Montréal, Québec, Canada**

**June 2023**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Tarek Makkouk**

Entitled: **Investigating and Testing Performance Issues in Deep Learning Frameworks**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Computer Science (Computer Science)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

____________________________________ Chair
*Dr. Yang Wang*

____________________________________ Examiner
*Dr. Shin Hwei Tan*

____________________________________ Examiner
*Dr. Yang Wang*

____________________________________ Supervisor
*Dr. Tse-Hsun (Peter) Chen*

Approved by ____________________________________
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

____________ 2023 ____________________________
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

# Abstract

Investigating and Testing Performance Issues in Deep Learning Frameworks

Tarek Makkouk

Machine Learning (ML) and Deep Learning (DL) applications are becoming more popular due to the availability of DL frameworks such as PyTorch, Keras, and TensorFlow. 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 thesis, we first aim to better understand performance bugs in DL frameworks by conducting an empirical study. We conduct our study on PyTorch and TensorFlow by mining and studying their performance and non-performance bug reports from their respective 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 in DL frameworks by studying all performance bug fixes.

We then aim to investigate the potential of differential testing as a viable technique to detect and prevent performance bugs in DL frameworks. To do so, we train and evaluate two state-of-the-art CNN and RNN architectures (i.e., the Lenet-5 architecture on the MNIST dataset and the LSTM architecture on the IMDB movie review dataset), using different DL frameworks (i.e., PyTorch, Keras, and TensorFlow), and different configurations (i.e., the training dataset sample size, the batch size, the number of epochs, the weight initialization technique, the data type, the hardware used, the learning rate, and the dropout rate). To assess the performance of the DL models, we use a variety of performance metrics (i.e., training/inference time, hardware (CPU or GPU) usage during training/inference, and memory (RAM or GPU VRAM) usage during training/inference). Then, we compare the performance of the DL models across the DL frameworks. We train and evaluate 21,870 Lenet5 models and 21,870 LSTM models across the DL frameworks, for a grand total of 43,740 models. Our experiments took over 42 days. We find that 1) differences in performance between different DL frameworks, for the same task, may be indicative of a performance optimization opportunity/performance bug; 2) our approach is viable when training and evaluating a smaller number of DL models, which makes it more accessible for developers.

Finally, we present some potential avenues for future work that aim to further study performance bugs in DL frameworks.

# Acknowledgments

Firstly, I would like to thank my parents for their unconditional love and their endless support and patience throughout my entire life, and especially during my educational journey. Without their belief in me, and the sacrifices they have made, I would not be here today, and for that, I am eternally grateful. I hope to one day make them as proud as I am for being their son.

In addition, I would like to extend my deepest gratitude to my supervisor, Dr. Tse-Hsun (Peter) Chen. His insightful feedback, and constructive criticism were pivotal in driving the quality of my work, and his leadership, and mentorship were truly inspiring during this journey. I am appreciative to have had the chance to work under his guidance.

I also would like to thank fellow researcher at the Software PErformance, Analysis, and Reliability (SPEAR) lab, and PhD student, Dong Jae Kim, for being an excellent collaborator, and a constant source of inspiration throughout this journey. Our countless discussions, and brainstorming sessions were driving forces in getting meaningful results.

Finally, I would like to thank the committee members, Dr. Shin Hwei Tan and Dr. Yang Wang for the precious time they took to read and review my work, and for their invaluable feedback.

# Related Publications

Part of the work presented in this thesis has been previously published, as follows:

- **Tarek Makkouk**, Dong Jae Kim and Tse-Hsun (Peter) Chen, ”An Empirical Study on Performance Bugs in Deep Learning Frameworks,” *In IEEE International Conference on Software Maintenance and Evolution, ICSME.* 2022.

# Contributions of Authors

**An Empirical Study on Performance Bugs in Deep Learning Frameworks**

Tarek Makkouk: quantitative analysis, manual analysis, program implementation, writing, editing, proofing

Dong Jae Kim: manual analysis, experimental guidance, writing, editing, proofing

Tse-Hsun (Peter) Chen: research supervisor, funding, experimental guidance, writing, editing, proofing

# Contents

# List of Figures

# List of Tables

# Chapter 1

# 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 their underlying 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).

Performance bugs in DL refer specifically to software defects that impede the optimal utilization of system resources, such as memory, computation power, and data access. These bugs may have different causes, including inefficient algorithms, sub-optimal memory management, improper usage of hardware resources, or poorly optimized libraries and APIs (Makkouk, Kim, & Chen, 2022). While these bugs may not directly affect the correctness of the DL models, they can severely hamper their efficiency, resulting in increased training and inference times, higher computational costs, and reduced overall performance.

Many previous works have focused on characterizing and resolving performance bugs in traditional software settings (Han & Yu, 2016; Jin, Song, Shi, Scherpelz, & Lu, 2012; Y. Liu, Xu, & Cheung, 2014; Nistor, Jiang, & Tan, 2013; Zaman, Adams, & Hassan, 2011, 2012). However, there are new challenges that arise when developing DL software due to the fundamental differences that exist between traditional and DL software applications (Amershi et al., 2019; Ma, Zhang, et al., 2018; Sculley et al., 2015). 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 (J. Chen, Liang, Shen, & Jiang, 2022; Jia, Zhong, Wang, Huang, & Lu, 2020, 2021), many studies focus on bugs in a single DL framework, and there is a lack of detailed empirical studies on characterizing and detecting specifically performance bugs. We believe that studying performance bugs in DL frameworks is of paramount importance for both researchers and framework developers, as: (1) it can direct future research efforts toward developing better test oracles to better detect performance bugs; (2) it can provide new evidence on how performance bugs arise and are fixed in DL systems for future tool development; and (3) it can help prevent future encounters of performance bugs.

As a stepping-stone to help bridge research gaps, we collect performance and non-performance bugs from the GitHub repositories of two of the most popular DL frameworks (Hale, 2018), PyTorch and TensorFlow, and quantitatively study the characteristics and the prevalence of performance bugs in DL frameworks. Then, we manually study and categorize 141 fixed performance bugs from the DL frameworks into a taxonomy of 12 categories and 19 sub-categories of their root causes.

After which, we aim to build on this foundation by exploring the viability of differential testing for detecting performance bugs in DL frameworks. To do so, we train and evaluate 43,740 DL models across two state-of-the-arts architectures (i.e., Lenet5 and LSTM), using different DL frameworks (i.e., PyTorch, Keras, and TensorFlow v1.X), and configurations, and compare their performance across different performance metrics in search of optimization opportunities.

## 1.1 Thesis Overview

### 1.1.1 An Empirical Study on Performance Bugs in Deep Learning Frameworks

Firstly, we collect performance and non-performance bugs from the PyTorch and TensorFlow GitHub repositories and investigate how the prevalence of performance bugs in DL frameworks evolves over time. We find that despite increasing efforts to fix performance bugs in DL frameworks over time, their prevalence is still increasing.

Then, we compare performance and non-performance bugs collected 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.

Finally, 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.

### 1.1.2 An Exploratory Study on the Viability of Performance Differential Testing for Deep Learning Frameworks

Our primary objective is to explore the potential for identifying optimization opportunities through the utilization of differential testing. To accomplish this, we investigate the existence of significant performance variations across the multiple DL frameworks (i.e., PyTorch, Keras, and TensorFlow V1.X), along with various configuration combinations.

In particular, we implemented and trained a total of 43,740 DL models (21,870 Lenet5 and 21,870 LSTM models) using multiple of DL frameworks, as well as various configurations, including training sample size, batch size, number of epochs, weight initialization technique, hardware device, precision mode (when applicable), and learning rate.

We collect numerous performance metrics as indicators of the performance of the DL models. Namely, we collect training time, inference time, hardware (GPU/CPU) utilization, and memory consumption. We find many performance variations across different DL frameworks implemented for the same DL model. Such variations help us uncover performance trade-offs and optimization opportunities.

Finally, to allow the adoption of our approach for industry, we investigate the reproducibility of our results when training and evaluating smaller samples of DL models. We do so by selecting random samples of varied sizes (i.e., 5%, 25%, 50%, and 75% of the Lenet5 and LSTM model sets, each of which consists of 2,430 CPU models and 4,860 GPU models for each of the three DL frameworks, for 21,870 trained and evaluated models each). The samples are then used to train linear regression models to infer the performance metrics of the DL models from their configurations (i.e., given the configurations used to train and evaluate a given DL model, the regression models are trained to infer the performance metrics of the DL model). Lastly, we examine the Root Mean-Square Errors (RMSEs) we obtain using the various sample sizes using the trained regression models on the set of unselected models.

We repeat the experiment a hundred times for each sample size, using the Monte Carlo method, and provide the average RMSEs. We find that we lose minimal predictive power when training regression models with a 5% sample compared to a 75% sample, which implies that our approach is valid even when training and evaluating a smaller number of DL models.

In summary, we find that differential testing may have the potential to be an effective tool for finding performance issues in DL frameworks, but future research is needed to further evaluate this approach.

## 1.2 Thesis Contributions

In this section, we present the main contributions of this thesis.

### 1.2.1 An Empirical Study on Performance Bugs in Deep Learning Frameworks

The main contributions of our study of performance bugs in DL frameworks are the following.

- We web scrape and extract information (e.g., number of comments, date of first comment, whether fixed or not, fix size if fixed) from 17,893 performance and non-performance bug reports from TensorFlow and 16,284 from PyTorch, for a total of 34,177 bug reports. This data can be further studied by researchers to better understand bugs in DL frameworks. We make this data available online (Chap3data, 2022).

- We study quantitative statistics on the prevalence of performance bugs from different aspects. Namely, we study how the prevalence of open performance bug reports (i.e., unresolved performance bug reports) among all open bug reports and fixed performance bug reports (i.e., performance bugs for which a fix was committed) among all fixed bug reports evolves over time using more than 4 years worth of bug reports collected from the PyTorch and TensorFlow GitHub repositories. We also make this data available online (Chap3data, 2022). We find that performance bug reports are opening at a faster rate than performance bug reports being fixed, which indicates that performance is increasingly becoming a concern in DL frameworks.

- We study and compare the fix time, community engagement, and the fix sizes of performance and non-performance bugs. We find that performance bugs take more time to fix, despite benefiting from higher community engagement, and that their fixes tend to be larger. This is indicative of the complexity and importance of performance bugs in DL frameworks.

- We performed a qualitative analysis on fixed performance bugs collected from the PyTorch and TensorFlow GitHub repositories (i.e., 141 fixed performance bugs), and categorized them into a taxonomy compromised of 12 categories and 19 subcategories of their root causes.

### 1.2.2 An Exploratory Study on the Viability of Performance Differential Testing for Deep Learning Frameworks

The main contributions of our exploratory study of the viability of using performance differential testing for DL frameworks are the following.

- We train and evaluate 21,870 Lenet5 models and 21,870 LSTM models implemented in three different DL frameworks (i.e., PyTorch, Keras, and TensorFlow V1.X) and across different combinations of configurations (i.e., 486 unique combinations for CPU models and 972 unique combinations for GPU models, for each architecture and DL framework), for a total of 43,740 models. We evaluate these models across different performance metrics (i.e., training/inference time, hardware (GPU or CPU), and memory (GPU VRAM or RAM) usage during training/inference), and compare them across multiple DL frameworks. We do so in search of variations in performance between the implementations that may map to performance optimization opportunities for DL frameworks. The experiments took over 42 days. We make this data available online (Chap4data, 2023).

- We find that differences in performance metrics between the same models implemented in different DL frameworks may lead to uncovering performance bugs and optimization opportunities, which is indicative of the potential of differential testing as a tool to test and prevent performance bugs in DL frameworks.

- We find that, when training and evaluating a smaller sample of DL models, we lose little information, which indicates the potential scalability of differential testing for performance bugs in DL frameworks.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows; Chapter 2 provides a general background on DL frameworks and the literature review. Chapter 3 presents our empirical study of performance bugs in DL frameworks. Chapter 4 presents our study on the viability of using differential testing for performance bugs in DL frameworks. Finally, Chapter 5 provides a summary of the thesis and potential research directions for future work.

# Chapter 2

# Background and Related Works

In this chapter, we discuss the important background to our work, such as the Deep Learning (DL) frameworks that we studied throughout this thesis. Furthermore, we discuss relevant works to our research, such as works on performance bugs, DL system quality, and benchmarking DL frameworks.

## 2.1 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 optimize a loss function and 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 (Arel, Rose, & Karnowski, 2010; Bengio, Courville, & Vincent, 2013; Bottou, Chapelle, DeCoste, & Weston, 2007; Sestili, 2018. [Online]). However, the DL model is notoriously complex and colossal, which introduces a new set of challenges, such as optimizing a wide range of configurations (Sculley et al., 2015).

DL frameworks aim to abstract the complex mathematical details of DL algorithms (e.g., gradient computations) to allow developers to efficiently implement DL systems 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) (Paszke et al., 2017). 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), PyTorch and Keras/TensorFlow have become the most popular DL frameworks (Hale, 2018). PyTorch was developed by Meta's AI research group (Paszke et al., 2017). Google's Brain team developed TensorFlow (Abadi et al., 2016), and made Keras the default execution mode with the release of TensorFlow V2.X. Therefore, in this thesis, we focus our study on those frameworks.

Given the widespread adoption of DL in various domains of industry and commercial settings, it becomes crucial to validate their quality to mitigate severe consequences. Hence, prior research efforts have studied and characterized bugs in DL frameworks (e.g., J. Chen et al. (2022); Jia et al. (2021); Y. Zhang, Chen, Cheung, Xiong, and Zhang (2018)). Among those bugs are performance bugs, which may significantly impact the downstream tasks due to the repetitive and computationally expensive nature of DL. However, prior research has identified performance bugs as the most difficult type of bug to address in DL systems (T. Zhang, Gao, Ma, Lyu, & Kim, 2019), and there have been limited comprehensive studies that focus on the characteristics of performance bugs in DL frameworks. Hence, in this thesis, we conduct a comprehensive empirical study on the performance bugs within two widely used DL frameworks today (Hale, 2018), PyTorch and TensorFlow. As part of this thesis, we compare and contrast performance bugs with non-performance to uncover distinct nature of performance bugs. we quantitatively analyze how the performance bugs in DL frameworks compare to non-performance bugs. Moreover, we perform a comprehensive and detailed qualitative study to derive a taxonomy of performance bugs and their underlying root causes. Our study sheds light on key characteristics that define performance bugs in DL frameworks and may inspire future research to address performance problems.

## 2.2 Empirical Studies on Performance Bugs

Many prior studies aim to characterize performance bugs in traditional software systems. For example, Jin et al. (2012) 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 iOS applications have been conducted by Y. Liu et al. (2014) and Afjehei, Chen, and Tsantalis (2019). They developed a static analysis tool based on the patterns they observed and were able to detect previously unknown performance bugs. Zaman et al. (2012) compared 400 performance and non-performance bugs collected from Firefox and Google Chrome. Nistor et al. (2013) 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 insights gained from those studies have facilitated the development of bug detection tools. In this thesis, Our focus centers on performance bugs that occur in DL frameworks, which present 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.

## 2.3 Quality Assurance of Deep Learning Systems

There is a large number of prior work on the software quality assurance of DL systems. One line of research is bugs in DL systems, with a focus on bugs associated with the use of DL framework APIs. For example, Y. Zhang et al. (2018) and Humbatova et al. (2020) analyze common mistakes that happen when using DL frameworks from Stack Overflow (SO) and GitHub projects. Islam, Nguyen, Pan, and Rajan (2019) studied the characteristics of common bug patterns in DL systems and their impact. Z. Chen et al. (2021) built a taxonomy of bug symptoms related to the deployment of DL systems on mobile applications. Cao, Chen, Sun, Hu, and Peng (2021) study performance issues in using APIs from DL frameworks from SO. T. Zhang et al. (2019) conducted an empirical study on 715 DL-related questions from StackOverflow (SO). Their research shows that there are five main reasons why users are having problems with DL framework APIs: improper use of APIs;

wrong choice of hyper-parameters; computation on GPUs; static graph computation; and lack of support for debugging and profiling.

There are also some studies that look into bugs in DL frameworks. Jia et al. (2020) studied the symptoms and causes of bugs in TensorFlow by analyzing the bug reports. The authors later extend their work (Jia et al., 2021) by also studying the fixes and multi-language bugs. J. Chen et al. (2022) 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.

Finally, another line of research focuses on testing the quality of deep learning models (Ma, Juefei-Xu, et al., 2018; Pei, Cao, Yang, & Jana, 2017; Sun et al., 2018; M. Zhang, Zhang, Zhang, Liu, & Khurshid, 2018). For example, DeepXplore (Pei et al., 2017) uses a differential testing approach to detect inconsistencies in the prediction output of DL models to detect issues. Some other work applies concolic testing (Sun et al., 2018), and metamorphic relations to test the models (Ma, Juefei-Xu, et al., 2018; M. Zhang et al., 2018). In addition to models, much research focuses on the quality of DL application code. TheDeepChecker (Braiek & Khomh, 2022) uses property-based testing to help debug issues and misconfigurations when using DL frameworks. Qian et al. (2021) found that there are fairness issues in DL frameworks where different runs of the same configuration can result in different results.

While understanding bugs associated with the usage DL framework APIs is beneficial for end-users, we focus on performance bugs inherent to DL frameworks. These bugs may impact any DL application that uses the corresponding DL framework API. Thus, by studying performance bugs in DL frameworks, we contribute to overall improvements of DL frameworks, which also enhance end-user usages of DL frameworks.

## 2.4 Benchmarking of DL frameworks

Several studies benchmarked the prediction accuracy or performance of the DL frameworks (El Shawi, Wahab, Barnawi, & Sakr, 2021; Guo et al., 2019; L. Liu et al., 2018; Pham et al., 2020; Shams, Platania, Lee, & Park, 2017). Shams et al. (2017) analyze the performance of DL models on different hardware environments (e.g., single v.s. multi-node setup). Similarly, Guo et al. (2019) studied the accuracy differences across DL frameworks, DL architectures, and deployments (e.g., desktop v.s. mobile), using the same configuration settings. Both studies found that there are performance or accuracy differences on different hardware environments. L. Liu et al. (2018) found that the default configuration for a dataset trained using a specific DL framework may not work well when switching to another framework. El Shawi et al. (2021) and Pham et al. (2020) benchmarked the accuracy and training time of the same model configuration across DL frameworks and DL architectures. They found that there can be a large variation in accuracy and training time across frameworks.

We are the first to conduct a comprehensive study on the performance differences across frameworks with different combinations of configuration values through differential testing. In comparison to prior works, which only focus on one set of default configurations, we also focus on a variety of different configuration settings. In total, we consider 1,452 unique sets of configurations (i.e., 486 unique combinations for CPU models and 972 unique combinations for GPU models). Our findings show that differential testing can locate performance differences, which can help us conduct root cause analysis further down the line.

# Chapter 3

# An Empirical Study on Performance Bugs in Deep Learning Frameworks

Performance bugs in DL frameworks may have catastrophic consequences as DL becomes widely adapted in multiple domains, including life-critical ones. Moreover, performance bugs in DL frameworks involve new challenges due to the peculiarities of DL, and its repetitive and resource-hungry nature. Despite this, in Chapter 2, we discover a lack of research focusing on performance bugs in DL frameworks by reviewing the available literature. Thus, in this chapter, we aim to gain a better understanding of the characteristics, prevalence, and root causes of performance bugs in DL frameworks by investigating performance bugs reported in two of the most popular DL frameworks (i.e., PyTorch and TensorFlow) (Hale, 2018).

The rest of this chapter is organized as follows; Section 3.1 describes our data collection process. Section 3.2 presents the results of our case study. Section 3.3 discusses the implications of our findings. Section 3.4 discusses the threats to validity to our work in this chapter. Finally, Section 3.5 concludes the paper.

## 3.1 Data Collection

Our goal is to investigate performance issues present in the studied systems. We study their characteristics of performance issues by leveraging bug reports. In this section, we discuss our data

Table 3.1: 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 |

collection process.

### 3.1.1 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 they are the most widely used DL frameworks (Hale, 2018). To collect data for our study, we developed a tool capable of gathering bug reports from both PyTorch and TensorFlow. We have successfully retrieved a substantial number of bug reports reported from the release date of the frameworks until February 2021, amounting to 19,098 reports for PyTorch and 29,941 reports for TensorFlow. 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.

### 3.1.2 Collecting Bug Fixing Commits.

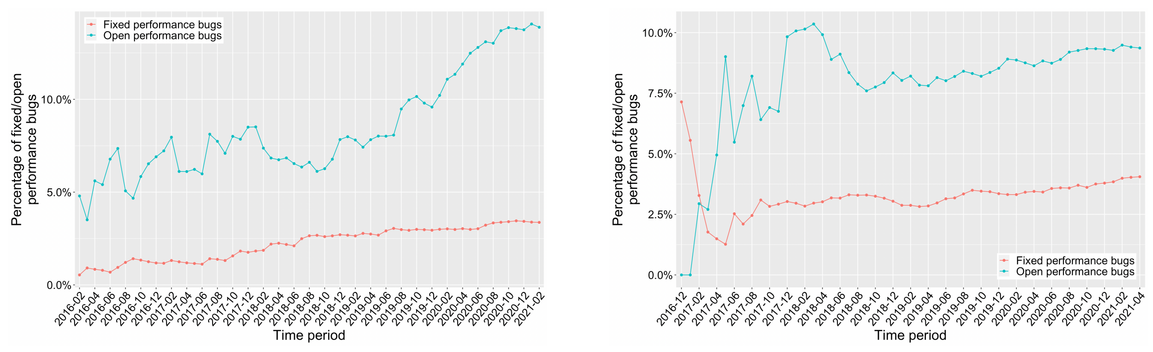
Given the collection of a substantial number of bug reports, the next step is to identify bugs that have been resolved with associated fixing commits. We collect the fixes to compare the characteristics between performance and non-performance bugs (e.g., fix size), and to manually derive a taxonomy of the causes of performance bugs in DL 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 a 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 (Bachmann & Bernstein, 2009; Schröter, Zimmermann, Premraj, & Zeller, 2006; Sliwerski, Zimmermann, & Zeller, 2005; Zhong & Su, 2015; Zimmermann, Premraj, & Zeller, 2007) 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.

### 3.1.3 Identifying Performance Bugs.

From the collected bug reports, 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 (T.-H. Chen et al., 2016; Jin et al., 2012; Y. Liu et al., 2014; Smith & Williams, 2001; Zaman et al., 2012). We make the list of the full list of keywords used publicly available online (Chap3data, 2022). 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., Zaman et al. (2012)).

## 3.2 Case Study Results

In this section, we conduct both quantitative and qualitative analyses to comprehensively examine performance bugs in the DL frameworks. In particular, we quantitatively study the prevalence of reported and fixed performance bugs in DL frameworks to better understand the concern that they represent. Then, we compare performance and non-performance bugs to identify key characteristics that differentiate them and gain deeper insights into performance bugs in DL systems. We do so by comparing performance and non-performance bugs across different dimensions (i.e., bug complexity and community engagement). Finally, we conduct a qualitative study and derive



(a) TensorFlow (b) PyTorch

Figure 3.1: Percentage of open and fixed performance bugs among **ALL** open and fixed bugs in DL frameworks, respectively

a taxonomy of the root causes for performance bugs in DL frameworks by manually studying 141 fixed performance bugs.

### 3.2.1 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 reported to be properly 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 (Cox & Stuart, 1955) to investigate the existence of an increasing trend. To quantify the significance of the observed data, we report the p-values obtained from the statistical tests we conduct (i.e., the lower the p-value, the higher the probability that a significant increasing trend exists).

## Results

***Despite increasing efforts to fix performance bugs in DL frameworks over time, their prevalence is still increasing.*** Figure 3.1 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 about 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.

### 3.2.2 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 (Han & Yu, 2016; Jin et al., 2012; Nistor et al., 2013; Zaman et al., 2011, 2012). Traditional software systems and DL frameworks have major differences. Namely, DL tasks are computation-heavy and require hardware integration (Amershi et al., 2019; Sculley et al., 2015).

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 (T.-H. Chen, Nagappan, Shihab, & Hassan, 2014):

***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 (T.-H. Chen et al., 2014; Sobreira, Durieux, Madeiral, Monperrus, & Maia, 2018). 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 (Lee, Carver, & Bosu, 2017). 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:

***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 (Arya, Wang, Guo, & Cheng, 2019; Zaman et al., 2012).

***Number of commentators.*** We consider the number of distinct commentators in each bug report. More distinct commentators may reflect more interest from the community (Arya et al., 2019).

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 exists a difference between performance and non-performance bugs).

We also apply Cliff's delta (Long, Feng, & Cliff, 2003) to measure the effect size. We quantify the effect size by using the thresholds proposed by Romano and Kromrey (2006) (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 $0.474 < |d| \leq 1$).

Table 3.2: 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 | |

## 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 3.2 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% and 35% larger, respectively, and the differences for all the metrics are statistically significant (i.e., p-value $\leq$ 0.05). Although we do not compare the results with bugs in traditional software systems, a prior study by Nistor et al. (2013) 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 3.2, 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), which also tends to be the case in traditional software settings (e.g., (Zaman et al., 2012)). Our findings show more community engagement in performance-related bug reports, which may reflect the community's interest in the problem.

Similarly to performance bugs in traditional software systems, performance bugs in DL frameworks 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.

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

#### Motivation

Traditional software suffers from frequent and costly performance issues (Han & Yu, 2016; Jin et al., 2012; Y. Liu et al., 2014; Nistor et al., 2013; Zaman et al., 2012). 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 3.1. The author of this thesis and another contributor (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: 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 (Viera & Garrett, 2005). We make the taxonomy and the corresponding bug report IDs publicly available online (Chap3data, 2022).

### Results

Table 3.3 shows the taxonomy of performance bugs in DL frameworks. Below, we discuss each category in detail.

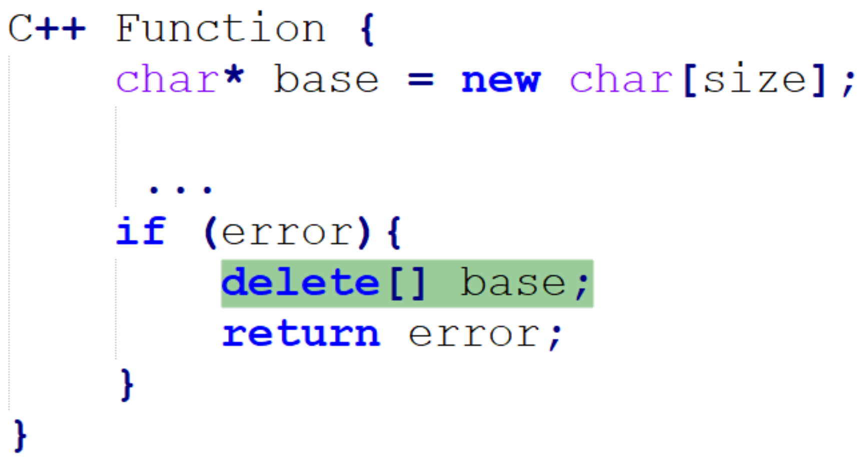
– **Memory inefficiencies (37/141, 26.24%).** Memory inefficiencies are the most common performance bugs in DL frameworks. One potential reason may be that a large part of DL frameworks are implemented in the C/C++ and CUDA programming languages for performance gains and finer-grained memory management. 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., *TensorFlow#14800* (TensorFlow#14800, 2017)). This case can be observed in *TensorFlow#14800* (TensorFlow#14800, 2017), which reports that a certain function does not release the memory it allocated if its execution ends in an error. The developers fixed this issue by simply making sure the funtion releases the memory resources it allocated when execution ends, as shown in Figure 3.2.

In some cases (e.g., *PyTorch#50522* (PyTorch#50522, 2021)), 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* (PyTorch#50522, 2021)). This highlights the need for better test oracles that consider different workloads to improve the detection of memory bugs in DL frameworks.

Table 3.3: Qualitative Analysis: Taxonomy of Performance Bugs in Deep Learning Frameworks (Chap3data, 2022).

| 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 specific 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), it may lead to significant performance degradation. | (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 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%)** |



```
C++ Function {
    char* base = new char[size];

        ...
    if (error){
        delete[] base;
        return error;
    }
}
```

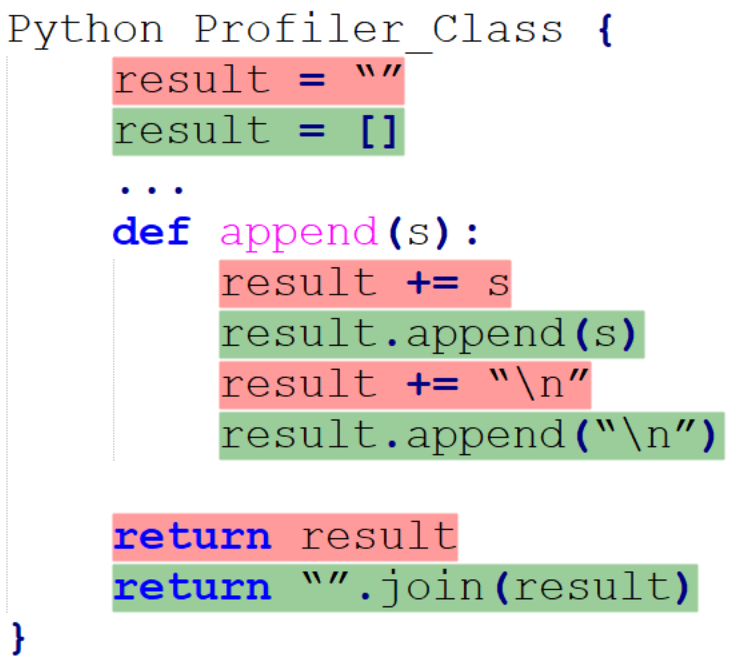
Figure 3.2: The fix for a performance bug due to unreleased memory allocations (*TensorFlow#14800* (TensorFlow#14800, 2017)).

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* (TensorFlow#14572, 2017), 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* (PyTorch#5611, 2018) and *PyTorch#6222* (PyTorch#6222, 2018)).

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 (Jin et al., 2012; Sujon, Shafiuzzaman, Rahman, & Rahman, 2016; Sánchez, Delgado-Pérez, Medina-Bulo, & Segura, 2020), 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* (PyTorch#10851, 2018), 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, as shown in Figure 3.3.



```
Python Profiler_Class {
    result = ""
    result = []
    ...
    def append(s):
        result += s
        result.append(s)
        result += "\n"
        result.append("\n")

    return result
    return "".join(result)
}
```

Figure 3.3: The fix for a performance bug due to traditional SE inefficiencies (*PyTorch #10851* (PyTorch#10851, 2018)).

Many previous works have investigated these performance bugs (Han & Yu, 2016; Jin et al., 2012; Nistor et al., 2013; Zaman et al., 2011, 2012). 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* (PyTorch#9646, 2018), the excessive use of atomic operations when managing CUDA streams causes the runtime to be synchronized and, hence, causes 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* (TensorFlow#3470, 2016), 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* (PyTorch#24080, 2019), developers found that the calculation for the number of threads varied across different multi-threading 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.

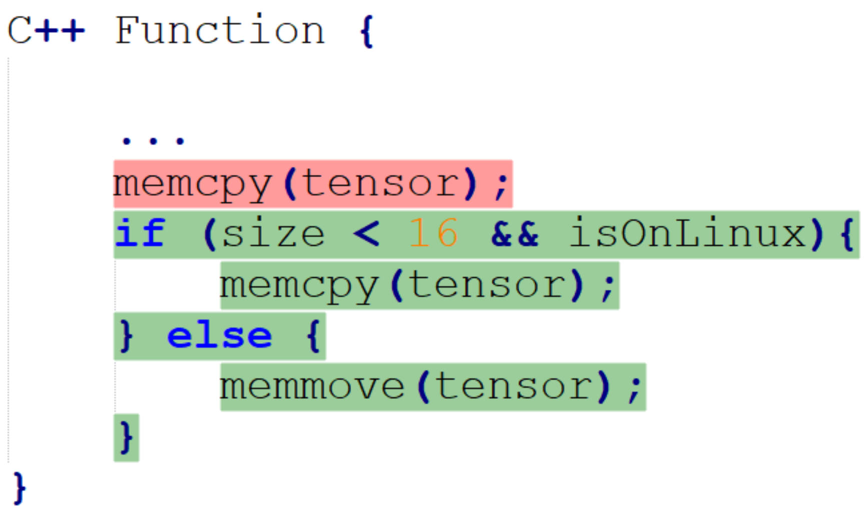
While parallelization can help circumvent the computationally-demanding nature of certain DL tasks, there are challenges associated with determining the most efficient threading configuration (e.g., number of threads) based on different factors (e.g., across a varying number of CPU/GPU cores). Future 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* (PyTorch#42265, 2020), 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, 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 (Tomov, Dongarra, & Baboulin, 2010) 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* (TensorFlow#17246, 2018), 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, as shown in Figure 3.4.

We also find some matrix computation *inefficiencies for specifc sizes and/or shapes* (2/141, 1.42%). For example, in *PyTorch#12006* (PyTorch#12006, 2018), a large batch size results in low GPU utilization. To fix this, the developers opted to transpose and reshape the inputs when the batch



```
C++ Function {

    ...
    memcpy(tensor);
    if (size < 16 && isOnLinux){
        memcpy(tensor);
    } else {
        memmove(tensor);
    }
}
```

Figure 3.4: The fix for a performance bug due to trade-offs of using different linear algebra operators (*TensorFlow#17246* (TensorFlow#17246, 2018)).

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* (PyTorch#17206, 2019), 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., (Elshawi, Wahab, Barnawi, & Sakr, 2021; Ganeshan, Elumalai, & Achar, 2020; L. Liu et al., 2018)), 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* (PyTorch#7883, 2018), the short-time Fourier transform is slower on PyTorch than on Librosa (McFee et al., 2015). This is due to the fact that, unlike PyTorch, Librosa's implementation is based on the fast Fourier transform (Cochran et al., 1967). Another example is *PyTorch#11931* (PyTorch#11931, 2018), 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 implementation and the weighted random sampling algorithm (Efraimidis & Spirakis, 2016).

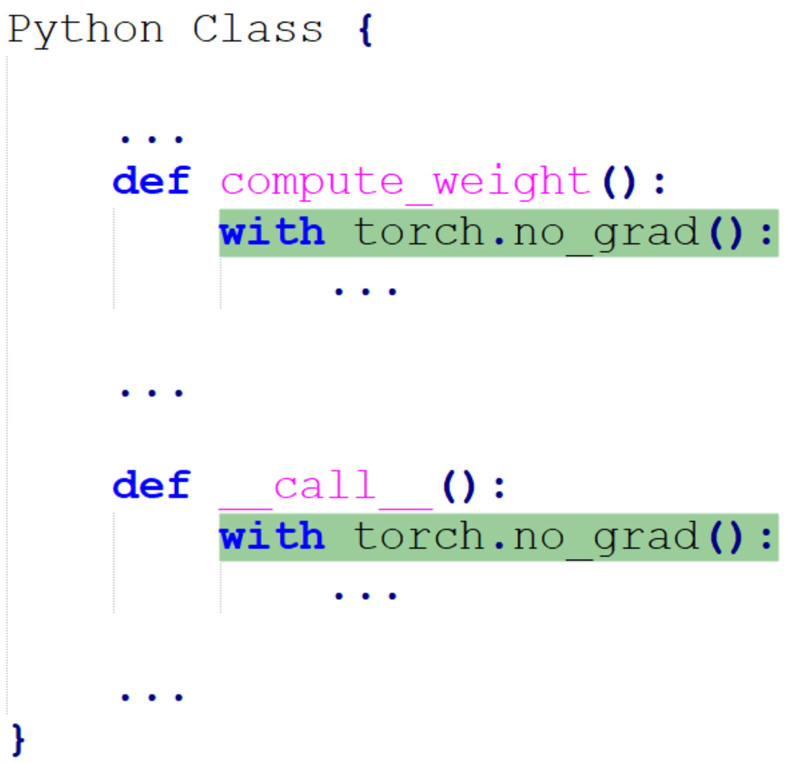
We observe that these bugs were discovered by comparing the performance of different implementations of the same operation. This suggests approaches such as performance differential testing may help uncover possible performance bugs or help developers choose more efficient libraries.

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 (Sapunov, 2020). 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 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* (TensorFlow#41715, 2020)). Such bugs that only occur under certain configurations are also common in traditional software systems and are notoriously hard to detect (He et al., 2020). 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 (Alzamil & Korel, 2005). 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* (PyTorch#7261, 2018), the unnecessary gradient calculation of an argument in the spectral normalization implementation increased the memory consumption and caused an Out-of-Memory exception.



```
Python Class {

        ...
        def compute_weight():
            with torch.no_grad():
                ...

        ...

        def __call__():
            with torch.no_grad():
                ...

        ...
}
```

Figure 3.5: The fix for a performance bug due to unnecessary gradient computations (*PyTorch#7261* (PyTorch#7261, 2018)).

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* (TensorFlow#32138, 2019), 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, in *PyTorch#19797* (PyTorch#19797, 2019), PyTorch does not support Intel's MKL-DNN for the average pooling operation. 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* (PyTorch#158, 2016), 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* (TensorFlow#11411, 2017), 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* (PyTorch#33334, 2020), 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* (TensorFlow#40758, 2020), 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., (Harmanen & Mikkonen, 2016)). 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* (PyTorch#18853, 2019)), bugs that were not discussed enough (e.g., *Pytorch#18405* (PyTorch#18405, 2019)) and bugs that are due to bugs in third-party libraries (e.g., *PyTorch#82* (PyTorch#82, 2016)).

## 3.3 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.

### 3.3.1 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 (L. Liu et al., 2018), 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 re-occur 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.

### 3.3.2 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.

## 3.4 Threats to Validity

### 3.4.1 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 (Hale, 2018), well-maintained, consistently updated, and used in various commercial settings.

### 3.4.2 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 (T.-H. Chen et al., 2016; Jin et al., 2012; Y. Liu et al., 2014; Smith & Williams, 2001; Zaman et al., 2012) 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 (Zaman et al., 2012).

### 3.4.3 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.

## 3.5 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 focuses on improving the quality of DL frameworks and DL systems.

Chapter 4

# An Exploratory Study on the Viability of Performance Differential Testing for Deep Learning Frameworks

In Chapter 3, we analyze performance bugs in two of the most popular DL frameworks (i.e., PyTorch and TensorFlow) in order to better understand them, asses their prevalence, and understand their causes. Building on this foundation, we aim to assess the viability of using differential testing for detecting these performance bugs, as well as optimization opportunities in DL frameworks. To do so, we compare two state-of-the-arts architectures (i.e., Lenet5 and LSTM), while considering different configurations, across different DL frameworks (i.e., PyTorch, Keras, and TensorFlow v1.X), using a variety of performance metrics.

Differential testing is a technique that compromises of comparing two or more implementations of the same algorithm, in search of discrepancies or differences in output or behavior, that could be indicative of optimization opportunities or bugs McKeeman (1998). Differential testing is already widely used in the industry (Gulzar, Zhu, & Han, 2019), but its effectiveness in a DL environment is yet to be explored.

In particular, all DL frameworks aim to abstract the complex mathematical details of DL algorithms to allow users to efficiently implement and deploy DL models. Thus, comparing the implementations of the same DL model across different DL frameworks may represent an opportunity to develop better test oracles for testing the performance of DL frameworks.

Moreover, DL frameworks allow for the use of numerous configurations (e.g., data precision) and support multiple platforms (e.g., CPU/GPU, different OS). In such cases, differential testing may be useful to detect inefficiencies that only occur under certain configurations or platforms, such as certain categories of performance bugs discussed in Chapter 3 (e.g., inefficiencies for certain data types, inefficient hardware optimization).

The rest of this chapter is organized as follows; Section 4.1 describes the setup for our case study. Section 4.2 presents the results of our case study. Section 4.3 discusses the implications of our findings. Section 4.4 discusses the threats to validity to our work in this chapter. Finally, Section 4.5 concludes the chapter.

## 4.1 Case Study Setup

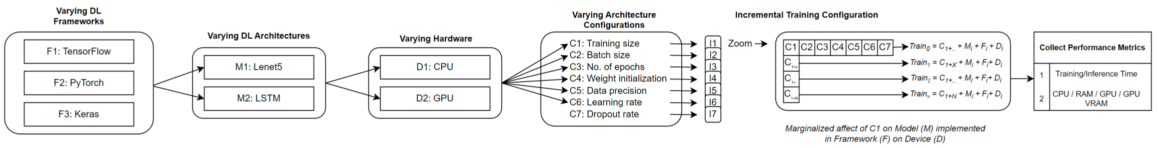

Varying DL Frameworks
F1: TensorFlow
F2: PyTorch
F3: Keras
Varying DL Architectures
M1: Lenet5
M2: LSTM
Varying Hardware
D1: CPU
D2: GPU
Varying Architecture Configurations
C1: Training size
C2: Batch size
C3: No. of epochs
C4: Weight initialization
C5: Data precision
C6: Learning rate
C7: Dropout rate
I1
I2
I3
I4
I5
I6
I7
Zoom
Incremental Training Configuration
C1 C2 C3 C4 C5 C6 C7
$C_{1+X}$
$C_{1+..}$
$C_{1+N}$
$Train_0 = C_{1+..} + M_i + F_i + D_i$
$Train_1 = C_{1+X} + M_i + F_i + D_i$
$Train_2 = C_{1+..} + M_i + F_i + D_i$
$Train_N = C_{1+N} + M_i + F_i + D_i$
Marginalized affect of C1 on Model (M) implemented in Framework (F) on Device (D)
Collect Performance Metrics
1 Training/Inference Time
2 CPU / RAM / GPU / GPU VRAM

Figure 4.1: The overview of the end-to-end process of collecting performance metrics for differential analysis.

Figure 4.1 shows the outline of our differential testing step. We apply variations in three main sources: (1) DL framework, (2) architecture, and (3) configurations. We first discuss the types of variations we introduced for differential testing, and we outline the methodology we used to train and test our DL models under different variation configurations. Finally, we present the performance metrics we collected for each variation configuration. These metrics are used to detect any differences in behavior resulting from incremental changes in the variations, helping us locate potential performance issues. In total, we trained 43,740 models with 1,452 unique combinations of configurations for each architecture and framework.

### 4.1.1 Variations Types in Different Testing

***Framework Variation.*** We perform differential testing on three popular DL frameworks; Pytorch (v1.10.0), Tensorflow v1.X (v2.5.0), and Keras (v2.5.0).

*F1: TensorFlow.* TensorFlow was created by Google's Brain team and made available as an open-source project in 2015. The framework enables users to build static computation graphs, where each edge is a tensor (i.e., a multi-dimensional array) and each node denotes an action.

*F2: Pytorch.* PyTorch was released by Facebook in 2016.PyTorch employs dynamic computational graphs, in contrast to the majority of DL frameworks, allowing for greater flexibility when developing models (Paszke et al., 2017).

*F3: Keras.* Keras was released by Google as an open-source project in 2015. In contrast to the DL frameworks listed above, Keras is an API designed to be used on top of other DL frameworks in order to provide additional abstraction. Since 2019, TensorFlow runtime uses Keras as the default.

***Architecture Variation.*** For our experiments, we selected state-of-the-art CNN (i.e., Lenet5) and RNN (i.e., LSTM) architectures. We employ Lenet5 models on the MNIST dataset and LSTM models on the IMDB dataset. Below, we discuss the implementation of these models.

*M1: Lenet5.* Lenet5 is one of the earliest Convolutional Neural Network (CNN) architectures widely used for image classification tasks (Lecun, Bottou, Bengio, & Haffner, 1998). It consists of several layers of convolution, pooling, and fully connected layers that extract features from input images and make predictions based on those features. To train the Lenet5 model, we utilize the popular MNIST dataset which consists of a large number of grayscale images of handwritten digits. The dataset is randomly split into two parts: a training dataset containing 60K images and a testing dataset containing 10K images.

*M2: LSTM.* LSTM is a type of Recurrent Neural Network (RNN) architecture that is commonly used for natural language processing tasks (Hochreiter & Schmidhuber, 1997). For our study, we use an LSTM architecture for binary sentiment analysis on the IMDB movie review dataset. The dataset contains 50K reviews in the form of text files, which are randomly split into a training set and a testing set with 25K samples each. The LSTM architecture we use consists of an embedding layer with a size of 32, followed by a dropout layer, an LSTM layer with a hidden size of 100, and a second dropout layer. Finally, a dense layer with a sigmoid activation function is used to produce a single output representing the sentiment of the review.

***Varying Hardware.*** We train the Lenet5 and LSTM models on both CPU and GPU. This allows us to compare the performance of DL frameworks on different processing units.

*D1: CPU.* We use the CPU to train and evaluate the DL models, alongside the optimization available to Intel CPUs (e.g., MKL).

*D2: GPU.* We use the CPU to train and evaluate the DL models, alongside the optimization available to Nvidia GPUs (e.g., cuDNN).

***DL Configuration Variation.*** We use seven configurations to comprehensively study the performance differences of the deep learning (DL) frameworks. We vary each configuration across a range of pre-defined values to compare how the models behave to the configuration changes across the different frameworks. We consider the following configurations:

*C1: Training Size.* Given the resource-intensive and time-consuming nature of training deep learning (DL) models, we aim to find differences between DL frameworks by training models on a portion of the original datasets. Specifically, we use 20%, 35%, and 50% of the dataset to train both models.

*C2: Batch Size.* We vary the batch size during both the training and inference phases. The batch size defines the number of data samples that go through the model before updating the model parameters. Hence, batch sizes may affect the accuracy and performance (e.g., a larger batch size uses more memory). In particular, we consider batch sizes of 64, 256 and 512 for both the Lenet5 and LSTM models.

*C3: Number of Epochs.* The number of epochs defines the number of times that training data is fed to the model for training. We use 5, 15, and 30 epochs for the training phase of the Lenet5 models, while we use 3, 5, and 10 epochs for the training phase of the LSTM models.

*C4: Weight Initialization.* We employ different weight initialization techniques to initialize the initial weights of our models. We consider two popular techniques, Xavier and He initialization, for both Lenet5 and LSTM models. This enables us to evaluate the impact of weight initialization techniques on the performance of the models across DL frameworks.

*C5: Precision Mode.* When using GPU-acceleration (i.e., GPU runtime), we use both *float32* data type and mixed-precision for the training and inference phases of both models. Larger datatypes may result in higher precision in computation but may take more time and memory.

*C6: Learning Rate.* A higher learning rate means that the weights in the model are adjusted faster, which may result in faster training time but reduce the model's accuracy. We use learning rates of 0.01, 0.05, and 0.01 during the training phase of the Lenet5 models. On the other hand, we use, learning rates of 0.01, 0.005, and 0.001 for the LSTM models.

*C7: Dropout Rate.* Dropout rate defines the percentage of nodes to be dropped randomly during training to reduce overfitting. We use dropping rates of 0, 0.15, and 0.25 for the training phases of both the Lenet5 and the LSTM models.

### 4.1.2 Experimental Runtime Phase

To thoroughly evaluate the performance of DL architecture across different frameworks, we assess their effectiveness during both the training and inference phases.

***Collecting Inference-Related Performance Metrics.*** Since model inference is often significantly faster than training, we artificially increased the test sets, such that model inference on test sets lasts long enough for us to collect reliable performance metrics. For the Lenet5 models, we replicate the original IMDB testing dataset 100 times, resulting in 1 million samples. For the LSTM models, we use eight times the original IMDB testing dataset, resulting in 200K samples.

***Training DL Models Using Incremental Configuration Changes.*** To ensure comprehensive evaluation, we train and evaluate DL models resulting from every possible combination of adjustable parameters discussed in Section 4.1.1. We evaluated a total of 43,740 DL models, with 21,870 models for each LSTM and Lenet5 architecture. These configurations resulted from three sources of differential testing variation, including 3 frameworks, 2 DL architectures, 2 hardware devices used (i.e., GPU and CPU) and 6 different configurations. Specifically, we varied 6 configuration sources, resulting in 1,452 possible configurations (i.e, 486 on CPU and 972 on GPU). We evaluated these configurations times across 3 frameworks, which amounts to 4,374 DL models for each DL architecture. To ensure accuracy and reliability, we replicate these experiments five times, resulting in a total of 21,870 DL models for each DL architecture. We use a standalone machine equipped with an Intel(R) Core(TM) i9-9900K CPU operating at 3.60GHz and an NVIDIA Quadro RTX 4000 GPU with 8GB of VRAM for our experiments. The entire process is time-consuming and computationally expensive. The entire training and inference took over 42 days. We believe the collected data is important not only for replicating this work but also for future research. Thus, we made the data publicly available online (Chap4data, 2023).

***Collected Performance Metrics*** To evaluate and compare the performance of the models across different DL frameworks, architectures, and configurations, we collect six performance metrics from each possible combination of configurations.

*Training time (in seconds).* We collect the time it takes to train the models (i.e., the time it takes for the model to get through the training phase).

*Inference time (in seconds).* We collect the time it takes the models to get through the inference phase.

*GPU usage (in %).* For the models that are on GPU, we collect the GPU consumption for both the training and the inference phases of the models.

*GPU'S VRAM usage (in %).* For the models that are on GPU, we collect the GPU VRAM consumption for both the training and inference phases of the models.

*CPU usage (in %).* For the models that are on CPU, we collect the CPU consumption for both the training and inference phases of the models.

*CPU'S RAM usage (in %).* For the models that are on CPU, we collect the RAM consumption for both the training and inference phases of the models.

## 4.2 Case Study Results

In this section, we first conduct a quantitative study to compare the performance of the DL frameworks across the different configurations and architectures and investigate whether those differences may lead to optimization opportunities. Then, we use linear models to analyze the influence of the incremental configuration changes on the performance of the DL models across the different DL frameworks and to better comprehend the performance differences found. Finally, we compare the predictive abilities of linear models trained on smaller samples of DL models to investigate whether our approach is viable when considering a smaller sample of DL models.

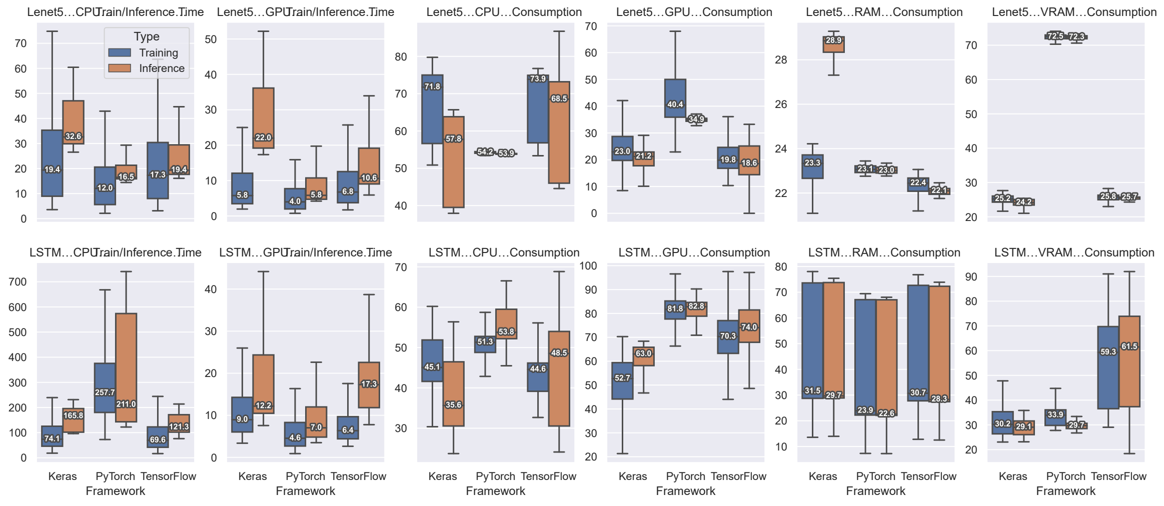
### 4.2.1 RQ1: How does the performance of the DL models compare across different DL frameworks and different combinations of configurations?

#### Motivation

PyTorch, TensorFlow, and Keras are among the many DL frameworks that support the development of large-scale DL applications. While these frameworks can be used interchangeably to solve DL tasks, it is crucial for developers to understand their trade-offs, particularly in terms of performance. Hence, several studies benchmarked the performance of the DL frameworks (El Shawi et al., 2021; Guo et al., 2019; L. Liu et al., 2018; Pham et al., 2020; Shams et al., 2017). However, these studies focused more on benchmarking and did not consider the differential differences across frameworks with various combinations of configurations. In this RQ, we compare all the configurations studied in previous research to gain a comprehensive understanding of the performance of these DL frameworks.

#### Approach

We conducted a statistical analysis to identify performance differences across various frameworks and configurations. We first utilize one-way ANOVA (Heiberger, Neuwirth, Heiberger, & Neuwirth, 2009) to compare the distributions of performance metrics across Keras, PyTorch, and TensorFlow, checking for statistically significant differences. We use ANOVA parametric test as the assumption is that the performance metric being compared across the framework is normally distributed and their variances should be equal. We then apply Cliff's Delta to measure the effect size for performance metrics that are statistically significantly different. We quantify the effect size by using the thresholds proposed by Romano et al. (Romano & Kromrey, 2006) (i.e., the 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 $0.474 < |d| \leq 1$). From Cliff's Delta, we focus our differential testing analysis on the performance metrics with large effect sizes that may be more concerning for both framework and application developers.



| Model | Framework | CPU Train Time | CPU Inference Time | GPU Train Time | GPU Inference Time | CPU Consumption Train | CPU Consumption Inference | GPU Consumption Train | GPU Consumption Inference | RAM Consumption Train | RAM Consumption Inference | VRAM Consumption Train | VRAM Consumption Inference |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Keras | - | +0.92* (large) | - | +0.87* (large) | - | - | - | - | - | +1.0* (large) | - | - |
| Lenet5 | PyTorch | - | -0.75* (large) | - | -0.79* (large) | -0.94* (large) | - | +0.94* (large) | +0.99* (large) | - | - | +1.0* (large) | +1.0* (large) |
| | TensorFlow | - | - | - | - | - | - | - | - | -0.66* (large) | -0.85* (large) | - | - |
| | Keras | - | - | - | - | - | - | - | - | - | - | - | - |
| LSTM | PyTorch | +0.84* (large) | +0.61* (large) | - | -0.67* (large) | - | - | +0.87* (large) | +0.77* (large) | - | - | - | - |
| | TensorFlow | - | - | - | - | - | - | - | - | - | - | +0.93* (large) | +0.73* (large) |

Figure 4.2: Comparing the performance metrics of Lenet and LSTM Models: Analyzing all differential testing combinations with boxplots (Top) and effect sizes (Bottom).

## Results

We present a comprehensive analysis of our findings from multiple perspectives. Firstly, we use a box plot (Figure 4.2) to display the distribution of performance metrics collected from various differential testing configurations (as shown in Figure 4.1). To supplement this, we also include a table in Figure 4.2 highlighting the statistical analysis results, identifying the performance metrics exhibiting statistically significantly different distributions across the studied DL models and frameworks. Interestingly, our ANOVA analysis reveals that the differences across all performance metrics are statistically significant. *This indicates that DL frameworks achieve different performance outcomes, with some frameworks performing more efficiently than others.*

The table at the bottom of Figure 4.2 also report the Cliff's Delta effect sizes. We only show statistically significant samples with large effect sizes that should be first focused on by the framework developers. We report either positive or negative effect sizes. For example, a positive effect size for PyTorch CPU inference time indicates that PyTorch takes the longest inference time while using CPU compared to Keras and TensorFlow. Based on these large effect sizes automatically acquired through our statistical analysis, we list several observations of performance differences in this RQ and report them to developers of frameworks.

**Result A: Training and Inference Time Comparison.**

RQ1-A Observation 1: ***While PyTorch has the fastest median training and inference time in both CPU and GPU for Lenet5 models, interestingly, it has the slowest training and inference times for LSTM when using CPU. However, its performance improves significantly when using GPU, with the highest speed observed.*** As shown in Figure 4.2, Pytorch achieves the fastest median training time for Lenet5 models in both CPU (12 seconds) and GPU (4 seconds), and inference time in both CPU (16.5 seconds) and GPU (5.8 seconds). Note that, all the differences across the frameworks are statistically significant (p-value $< 0.001$), and the differences in Lenet5 inference time have a large effect size for both CPU and GPU. Similarly, PyTorch's GPU execution achieves the fastest training time for the LSTM architecture, with a median of 4.55 seconds, compared to Keras (8.959 seconds) and TensorFlow (6.352 seconds). Despite its fastest performance time in most cases, based on a combination of statistical tests and data visualization, we find that PyTorch's LSTM architecture has the slowest median training time (with a large effect size) when executed on CPU (257.7 seconds), compared to Keras (74.06 seconds) and TensorFlow (69.59 seconds). The significant slowdown observed in the CPU implementation of PyTorch's LSTM architecture suggests a potential performance bug. We reported this issue to PyTorch developers (*PyTorch#94457* (PyTorch#94457, 2023)), and the developers confirmed the bug was partly due to a hardware optimization library (i.e., oneDNN) not being utilized and provided a bug fix. Note that this bug fits under the *"inefficient hardware optimization"* category and the *"lack of support for hardware optimization libraries"* subcategory of our taxonomy of the root causes for performance bugs in DL frameworks discussed in Chapter 3.

Overall, the issue highlights the importance and benefits of differential testing in DL frameworks to detect potential performance bugs.

RQ1-A Observation 2: ***As anticipated, all frameworks show improved training and inference times when utilizing GPU. However, Keras exhibits significantly slower inference times (28% slower) when using CPU and even slower inference times (107.8% slower) when using GPU.*** Figure 4.2 illustrates that when using Keras, the inference process takes almost twice as long on both CPU (median time 32.64s) and GPU (21.99s) compared to the other DL frameworks (with a large effect size), where PyTorch's median inference time is 16.49s and 5.84s seconds, and TensorFlow is 19.36s and 10.62s on CPU and GPU, respectively. Since we do not observe such performance disparities during training, it is possible that Keras' implementation of the Lenet5 architecture's forward pass is inefficient. It is worth noting that we used cuDNN for all frameworks during our experiments, which means that they use the same low-level code for DL-related computations. Therefore, the difference in training time may be implementation-specific. We reported the potential area of improvement to developers and are waiting for their response (*TensorFlow#60462* (TensorFlow#60492, 2023)). Overall, we find that PyTorch may be the most beneficial framework for improving DL development due to its faster inference times.

We observed several discrepancies between different frameworks. We reported the discrepancies to developers. One of the reported issues is confirmed and fixed by developers (we are still waiting for responses on other issues).

**Result B: CPU and GPU Consumption Comparison.**

RQ1-B Observation 1: ***Despite PyTorch consuming the lowest CPU usage in Lenet5 training and inference, it achieves higher performance than Keras and TensorFlow. In contrast, PyTorch consumes the highest CPU usage for LSTM training and inference.*** Figure 4.2 demonstrates that PyTorch has lower median CPU usage for Lenet5's training (54.25%) and inference (53.88%) processes than Keras (71.76% and 57.76%) and TensorFlow (73.86% and 68.52%). Despite lower CPU usage, PyTorch outperforms the other frameworks during CPU execution, confirming that higher hardware usage does not always result in better performance (El Shawi et al., 2021). This also suggests potential inefficiencies in Lenet5's training and inference processes on Keras and TensorFlow during CPU execution. In contrast, Figure 4.2 shows PyTorch has higher average CPU usage for the LSTM architecture's training (51.33%) and inference (53.79%) processes compared to Keras (45.05% and 35.65%) and TensorFlow (44.58% and 48.5%), yet underperforms in performance. This may indicate inefficiencies in PyTorch's LSTM training and inference processes during CPU execution.

RQ1-B Observation 2: ***PyTorch has high GPU consumption during the training and inference***

***processes of both Lenet5 and LSTM models. In the case of Lenet5, higher GPU consumption leads to faster training and inference times. In contrast, for LSTM models, higher GPU consumption results in slower training time but faster inference time.*** In Figure 4.2, PyTorch shows the highest GPU consumption in both training (40.45%) and inference (34.91%) processes of Lenet5, compared to 22.953% and 21.23% for Keras and 19.83% and 18.58% for TensorFlow. Based on Figure 4.2, a similar trend exists for training (81.76%) and inference (82.83%) of LSTM, compared to 70.31% and 74.03% for TensorFlow and 52.73% and 63.00% for Keras. Moreover, we observe the high variations in GPU consumption when training Lenet5, indicating the influence of configurations on GPU consumption. In contrast, during inference, PyTorch demonstrates consistent GPU consumption, with an IQR of only 0.1% compared to IQRs of 5.% for Keras and 10.71% for TensorFlow. Hence, PyTorch provides consistent performance during the inference process of the Lenet5 architecture, regardless of configuration changes. We reported the potential area of improvement to developers and are waiting for their response (*PyTorch#100425* (PyTorch#100425, 2023)).

PyTorch requires more GPU memory compared to Keras and TensorFlow for both training and inference processes of the Lenet5 and LSTM architecture. The increase in GPU consumption improves training and inference time while providing consistent performance in the inference, which may be suitable for situations where reliability is critical.

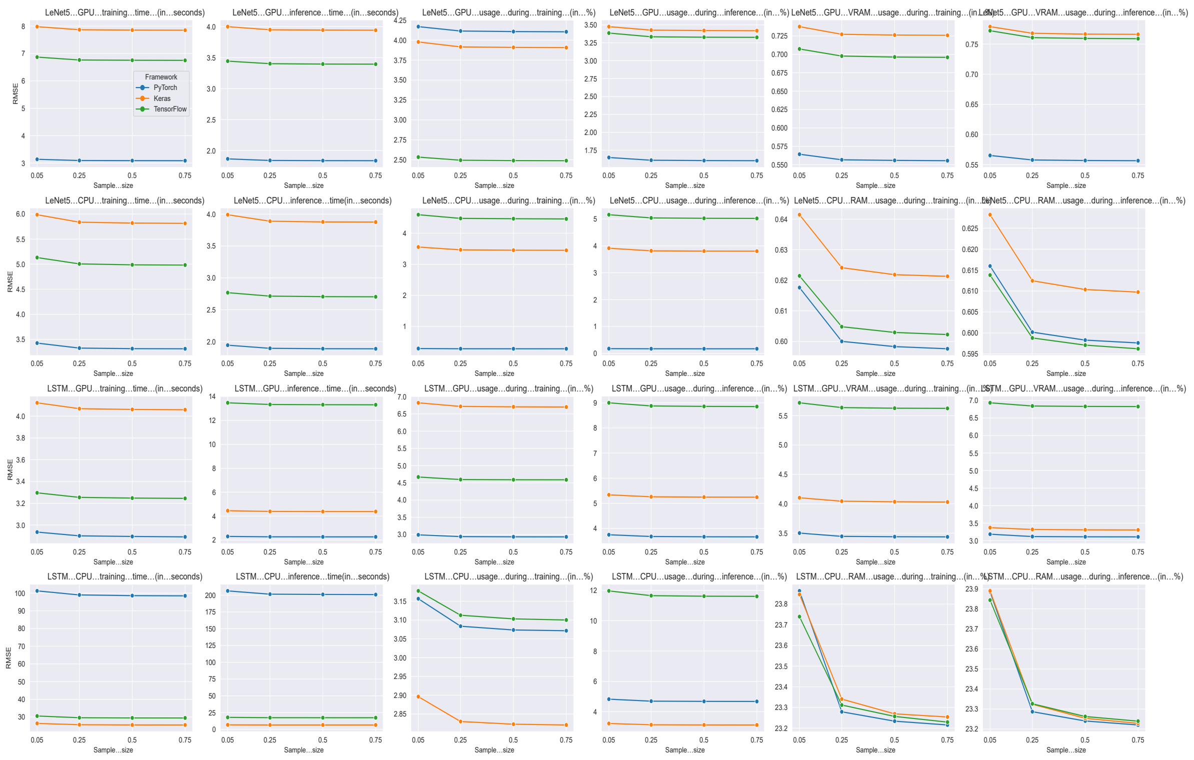
**Result C: CPU's RAM and GPU's VRAM Comparison.**

RQ1-C Observation 1: ***Keras exhibits the highest CPU RAM consumption specifically during inference for Lenet5. Given that Keras also experienced the slowest inference time, there may be potential performance issues associated with Keras.*** Figure 4.2 shows that RAM consumption stay generally consistent around 23% to 24% across all DL frameworks for Lenet5. However, we observe that using Keras yields a higher average RAM usage during the training (23.34 %) and inference (28.85%) processes of the Lenet5 architecture, followed by PyTorch (i.e., medians of 23.06% and 23.01%), and with TensorFlow yielding slightly lower RAM usage (i.e., medians of 22.45% and 22.10%). The differences are all statistically significant and the difference between Keras and other frameworks have a large effect size. Given the higher CPU RAM consumption during inference for Lenet5 in Keras, combined with its slower inference time, could indicate potential performance inefficiencies within the Keras framework. We reported the potential area of improvement to developers and are waiting for their response (*TensorFlow#60494* (TensorFlow#60494, 2023)).

RQ1-C Observation 2: ***PyTorch exhibits the highest GPU VRAM consumption during both training and inference for Lenet5, whereas TensorFlow's LSTM architecture presents high GPU VRAM consumption.*** As shown in Figure 4.2, PyTorch's Lenet5 implementation had a higher median GPU VRAM usage of 72.50% and 72.34% during training and inference, compared to 25.21% and 24.16% for Keras and 25.80% and 25.70% for TensorFlow (the differences are statistically significant with a large effect size). We reported the potential area of improvement to developers and are waiting for their response (*PyTorch#100617* (PyTorch#100617, 2023)).

Figure 4.2 also shows that TensorFlow's LSTM implementation has higher and more variable GPU VRAM consumption during training and inference, with median usages of 59.30% and 61.50%, and IQRs of 33.19% and 45.04%, respectively (the differences are statistically significant with a large effect size), whereas PyTorch yields a lower median GPU VRAM's of 33.93% and 29.67%, and a 30.23% and 29.08% for Tensorflow. We also reported this performance issue to developers and are waiting for their response (*TensorFlow#60495* (TensorFlow#60495, 2023)). Interestingly, LSTM's VRAM fluctuates significantly and consumes much more VRAM than PyTorch. Despite this, PyTorch achieves faster training and inference times, suggesting potential memory issues associated with the TensorFlow LSTM implementation.

The Lenet5 architecture's Keras implementation uses a little bit more RAM than the other frameworks, which might point to a place for optimization. Moreover, during both phases of the LSTM architecture, Keras and TensorFlow show higher RAM usage, which might indicate another area for optimization.



| Model | Framework | CPU Train Time | CPU Inference Time | GPU Train Time | GPU Inference Time | CPU Consumption Train | CPU Consumption Inference | GPU Consumption Train | GPU Consumption Inference | RAM Consumption Train | RAM Consumption Inference | VRAM Consumption Train | VRAM Consumption Inference |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lenet5 | Keras | 0.173 | 0.115 | 0.132 | 0.058 | 0.106 | 0.111 | 0.071 | 0.057 | 0.020 | 0.018 | 0.012 | 0.013 |
| | PyTorch | 0.113 | 0.057 | 0.051 | 0.29 | 0.008 | 0.005 | 0.065 | 0.045 | 0.020 | 0.018 | 0.009 | 0.009 |
| | TensorFlow | 0.150 | 0.065 | 0.119 | 0.049 | 0.136 | 0.138 | 0.045 | 0.060 | 0.019 | 0.018 | 0.011 | 0.013 |
| LSTM | Keras | 0.881 | 0.190 | 0.064 | 0.072 | 0.075 | 0.098 | 0.122 | 0.093 | 0.592 | 0.661 | 0.072 | 0.066 |
| | PyTorch | 2.790 | 5.520 | 0.044 | 0.037 | 0.085 | 0.151 | 0.059 | 0.084 | 0.648 | 0.664 | 0.065 | 0.076 |
| | TensorFlow | 1.089 | 0.488 | 0.513 | 0.166 | 0.077 | 0.360 | 0.081 | 0.150 | 0.510 | 0.604 | 0.095 | 0.107 |

Figure 4.3: Comparing the performance metrics of Lenet and LSTM Models: Investigating the feasibility of using subsets of DL configurations to evaluate unseen configurations with lineplots (Top) and RMSE increase when training linear models with 5% subsets instead of 75% subsets (Bottom).

### 4.2.2 RQ2: How do the differences between different implementations generalize when a smaller sample of models is used?

#### Motivation

In our previous research question (RQ), we utilized differential testing techniques to identify areas for performance optimization in DL frameworks. Subsequently, we shared our findings with framework developers to help them improve framework performance. However, this process was highly resource-intensive and time-consuming, requiring the training and evaluation of over 40K DL models for different combinations of configurations. Therefore, in this RQ, we aim to investigate the feasibility of utilizing a smaller subset of configurations of DL models to identify the same performance trend on new unseen configurations. Namely, getting comparable results with a smaller subset of models would make the testing process less resource-intensive and beneficial for framework developers.

## Approach

To test the feasibility of using a subset of configurations to achieve comparable results, we randomly sampled 5%, 25%, 50%, and 75% of the configurations for each framework and DL architecture. We used these subsets to build linear regression models, which were trained to predict the expected performance metrics such as training and inference time. Finally, we evaluated the accuracy of these regression models using the rest of the unseen (unselected) configurations and compared the predicted and actual DL model performance results using Root Mean Squared Error (RMSE), which is calculated as shown below;

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2}$$

| | |
|---|---|
| $N$ | number of unseen configurations |
| $x_i$ | inferred value for a given performance metric |
| $y_i$ | actual value for a given performance metric |

Our goal is to determine the extent to which the predictive power of the linear regression models is preserved across different configuration sample sizes. We repeat the experiment a hundred times for each sample size with different samples, using the Monte Carlo method, and provide the average RMSEs.

## Results

***There is a minimal change in the root mean squared error (RMSE) when training regression models with a 5% sample compared to a 75% sample. This suggests that the regression models maintain their predictive power even when trained on smaller sample sizes.*** Figure 4.3 illustrate the change in RMSEs of the linear regression models across different sample sizes for both Lenet5 and LSTM.

The table at the bottom of Figure 4.3 showcases the average RMSE increase when going from a linear model trained with 75% of the training samples to linear models trained with 5% of the training samples. In Figure 4.3, most changes in RMSE show a plateau between using 5% and 75% training sample; hence RMSE remains relatively stable across different sample sizes and maintains its predictive power even when trained with fewer trained samples.

For example, for Lenet5, the average increase in RMSE when reducing sample size from a 75% to 5% training sample is only around 3.03% for the Keras CPU models, around 1.66% for the Keras GPU, around 3.10% for the PyTorch CPU models, 1.79% for the PyTorch GPU models, 2.89% for the TensorFlow CPU models, and 1.70% for the TensorFlow GPU models. We observe similar trends for LSTM; the average increase in RMSE from 75% to 5% training sample is 2.93% for the Keras CPU models, 1.77% for the Keras GPU models, 2.87% for the PyTorch CPU models, 1.96% for the PyTorch GPU models, 2.82% for the TensorFlow CPU models, and 1.59% for the TensorFlow GPU models.

Therefore, despite a greater risk of missing out on optimization opportunities as some performance differences may only be visible under particular configurations, we find that working with a smaller set of DL models leads to similar performance trends.

We find that utilizing smaller training samples does not result in a substantial loss of predictive ability of the regression models. This demonstrates that our technique is viable even when training a substantially smaller number of DL models, making it more accessible to developers.

## 4.3 Implications and future work

Based on our findings, we believe differential testing can locate performance differences across DL frameworks, which enables us to conduct root cause analysis further down the line, and in turn, find optimization opportunities. This opens up several avenues for future research in this area.

Firstly, this study focused on a limited set of configurations and architectures. Future works can expand to a broader set of configurations and models to find a wider spectrum of performance bugs (e.g., using different library versions for the same task may reveal an inefficient use of APIs).

Moreover, future work may also incorporate machine learning techniques and search-based algorithms into the differential testing process, to better pinpoint the given optimization opportunity or to better the choice of configurations and their values.

Future work may also focus on the potential of differential testing for performance bugs in DL frameworks as a complementary approach to be used with other techniques that are used to test these frameworks (e.g., unit testing).

Finally, future work should further study the influence of the configurations considered on the performance of the different DL frameworks, as this could be key to finding performance bugs that are specific to a certain set of configurations.

Overall, the findings of this study indicate that differential testing has the potential to be an effective approach for detecting performance problems in DL frameworks, highlighting the need for additional research in this area.

## 4.4 Threats to Validity

### 4.4.1 External Validity

Our research was carried out using PyTorch, Keras, and TensorFlow v1.X. As a result, our findings may not be applicable to all DL frameworks. These three frameworks, on the other hand, are the most popular DL frameworks, well-maintained, consistently updated, and used in a variety of commercial situations. Our research was also limited to two architectures (i.e., Lenet5 and LSTM), which may also restrict the generalizability of our findings. We aim to reduce this effect in part by using state-of-the-art and popular CNN (i.e., Lenet5) and RNN (i.e., LSTM) architectures.

### 4.4.2 Internal Validity

DL models are known to have a large number of configurations, and we have conducted our study based on a number of designated configurations. As a result, our study is inherently limited by the number of configurations included, and more findings may be discovered if more are included. Given that the purpose of this research is to investigate the viability of employing differential testing for DL frameworks, we attempted to select a broad and inclusive set of configurations.

We also believe that domain expertise allows DL framework developers to uncover more optimization opportunities by picking better-suited configurations.

### 4.4.3 Construct Validity

DL frameworks continually undergo new optimizations and new releases, so some of the disparities and optimization opportunities identified in our study may no longer exist. However, due to the nature of our research, we contend that our emphasis should be on the existence of optimization opportunities rather than the optimization opportunities themselves.

## 4.5 Conclusion

Deep Learning (DL) is growing in popularity and, as a result, is rapidly being employed in a wide range of applications, including life-critical ones. This is due in part to DL frameworks, which enable developers to efficiently develop DL applications. However, because of the uniqueness of DL, optimizing and testing these frameworks may provide distinct challenges. In this work, we aim to study the viability of using differential testing for DL frameworks to find optimization opportunities. We train, evaluate and compare 21,870 Lenet5 models and 21,870 LSTM models, using numerous configurations and performance metrics, across three different DL frameworks (i.e., PyTorch, Keras, and TensorFlow v1.X. We find that: (1) differences in performance between the DL frameworks may lead us to performance optimization opportunities; and (2) this approach is also valid when using a smaller sample of DL models. We hope our findings can inspire future research that focus on further studying the potential of differential testing to improve the quality of DL frameworks and DL systems.

# Chapter 5

# Summary and Future Work

Deep Learning (DL) has revolutionized the field of Artificial Intelligence (AI), enabling several breakthroughs in image recognition, Natural Language Processing (NLP), and other complex tasks. This was made possible partly due DL frameworks, which enabled developers to efficiently develop DL systems. However, as with any complex software system, DL frameworks are prone to performance bugs and issues that can undermine their effectiveness and reliability. Moreover, performance bugs in DL environments can be challenging to detect and diagnose, as they often involve subtle interactions between hardware, software, and data inputs. Performance bugs in DL frameworks, in particular, are of utmost importance, as a performance bug in a DL framework may negatively affect a number of DL systems that relies on that DL framework. Moreover, as DL systems are increasingly being used in a wide spectrum of applications, these bugs may be increasingly likely lead to life-critical circumstances. Despite this, performance bugs in DL frameworks have received an underwhelming amount of attention from the research community.

This thesis has delved into this issue by analyzing the characteristics and the root causes of performance bugs in DL frameworks. Then we built on this understanding to explore the potential of differential testing as a strategy to detect and prevent them.

In this chapter, we conclude this thesis by providing a summary of our work and our findings, and by discussing avenues for future work that focus on performance bugs in DL frameworks.

## 5.1 Summary and Findings

### 5.1.1 An Empirical Study on Performance Bugs in Deep Learning Frameworks

In Chapter 3, we set out to study the characteristics of performance bugs in DL frameworks. To do so, we web-scrape and extract numerous metrics of 34,177 performance and non-performance bug reports from the PyTorch and TensorFlow GitHub repositories. We then use the data collected to study the characteristics of the performance bugs collected from those DL frameworks. More precisely, we explore the following Research Questions (RQs):

- What is the trend of fixed and reported performance bugs over time?
- What are the differences between performance and non-performance bugs in terms of fixes and community engagement?
- What are the causes of performance bugs in DL frameworks?

To address these RQs, we do the following:

- We study the prevalence of open performance bugs and fixed performance bugs in DL frameworks (i.e., PyTorch and TensorFlow) over a span of more than 4 years. We find performance bug reports are being opened at a faster rate than DL framework developers fixing them, which indicates that performance bugs are increasingly becoming a concern on DL frameworks.
- We compare performance and non-performance bugs in DL frameworks across different dimensions (i.e., complexity, and community engagement) using a variety of metrics (i.e., fix time, fix size, time for first comment, number of comments, and number of commentators). We find that performance bugs in DL frameworks tend to take more time to fix and have larger fixes, when compared to non-performance bugs. We also find that performance bugs tend to benefit from higher community engagement, which further underlines their importance for the community.
- We manually study and characterized 141 fixed performance bugs from the DL frameworks. We find 12 main categories of performance bugs in DL frameworks, which are composed of 19 smaller subcategories.

### 5.1.2 An Exploratory Study on the Viability of Performance Differential Testing for Deep Learning Frameworks

In Chapter 4, we set out to investigate the viability of using performance bugs for DL frameworks. To do so, we implement two state-of-the-arts CNN (i.e., Lenet5 for image classification on the MNIST dataset) and RNN (i.e, LSTM for sentiment analysis on the IMDB review dataset) architectures in three different DL frameworks (i.e., PyTorch, Keras, and TensorFlow). Then, we train and evaluate 21,870 models of each architecture, using a variety of different configurations (i.e., training sample size, batch size, number of epochs, weight initialization technique, precision mode, learning rate, and dropout rate). We evaluate those models collecting numerous performance metrics (i.e., training/inference time, hardware (CPU or GPU) usage during training/inference, and memory usage (RAM or GPU VRAM) during training/inference).

More precisely, we explore the following Research Questions (RQs):

- How does the performance of the DL models compare across different DL frameworks and different combinations of configurations?
- How do the differences between different implementations generalize when a smaller sample of models is used?

To address these RQs, we do the following:

- We compare the performance of the DL models across the three different DL frameworks, using the aforementioned performance metrics. We find that when the performance of the models implemented in one of the frameworks stands out, it may be indicative of performance optimization opportunities or performance bugs. This, in turn, is indicative of the potential of differential testing as an efficient tool to detect and prevent performance bugs in DL frameworks.
- We train linear regression models to predict the performance metrics of the DL models for each architecture (i.e., Lenet5 and LSTM), framework (PyTorch, Keras, and TensorFlow) and device (i.e., CPU or GPU) used, using randomly sampled 5%, 25%, 50%, and 75% of the

DL models evaluated. Then, we compare their accuracies by using the regression models to predict the rest of the unseen (unselected) DL models. We find that the regression models lose little predictive power when being trained with smaller samples. This suggests that our method can be applied by training a smaller number of DL models which, in turn, makes it more viable for developers.

## 5.2 Future Work

We believe that this thesis makes significant contributions towards addressing performance bugs in DL frameworks and towards the assessment of the potential of differential testing as an efficient tool for detecting and preventing them. However, our work also highlights the need for future work to address the significant challenges associated with handling performance bugs in a DL environment. We highlight some avenues for future work.

### 5.2.1 Handling Performance Bugs That only occur under certain configurations

We find that some unexpected performance discrepancies occur for certain configurations. For example, in Chapter 3, we find cases where using mixed-precision during GPU runtime cause a performance degradation instead of a performance improvement. Despite the existence of those bugs, there is a lack of studies that consider the trade-offs between different combinations of configurations for DL systems (L. Liu et al., 2018). We take the first step to fill this void by considering multiple configurations when training and evaluating the DL models used in our experiments in Chapter 4. However, future studies should further focus on exploring performance discrepancies that only occur under certain configurations.

### 5.2.2 Choosing the Optimal Libraries Based on The Environment Settings

In other cases, we find that libraries may perform differently for different tasks, and when operating in different environments. For example, we observe in Chapter 3, that in certain cases choosing the wrong linear algebra library may cause matrix inefficiencies. There exists numerous libraries that offer similar functionalities, but they may have different performance optimization based on factors such as OS and underlying hardware. Similar to what we have done in Chapter 4 for DL frameworks, benchmarking those libraries under different environment variables may help DL framework developers in selecting the optimal library in a given scenario. Moreover, future work may focus on automatically choosing or recommending the optimal libraries based on an environment setting.

### 5.2.3 Choosing the Optimal Threading Configurations For a Given DL Task

Parallelization is a critical component in mitigating the resource-intensive nature of DL. However, we observed in Chapter 3 that DL framework developers may face difficulties in establishing the most efficient threading configurations (e.g., number of threads) based on a variety of factors (e.g., across many CPU/GPU cores and workloads). Future research could help developers in automatically tuning the threading configurations based on the environment.

### 5.2.4 Identifying and Detecting Linear Algebra Code Smells in DL Systems

Linear algebra is a crucial part of DL, as it provides the mathematical foundation for many of its key operations and concepts. Thus, linear algebra code smells in DL frameworks may exorbitantly undermine their performance. For instance, we find in Chapter 3 that the usage of inefficient linear algebra operations or shapes may lead to sub-optimal hardware and memory usage, and thus, sub-optimal performance. Future work may assist developers in optimizing linear algebra operations and reshaping matrices in order to optimize hardware and memory usage in DL systems. Moreover, future work could focus on identifying and removing code smells related to linear algebra operations in order to improve the performance of DL systems.

### 5.2.5 Exploring the Potential of Differential Testing as a Tool for DL Systems

We find in Chapter 4 that differential testing may be a viable tool to detect performance optimization opportunities and prevent performance bugs in DL frameworks. However, our study of the usage of differential testing in a DL environment is exploratory, and further work is required to fully assess the potential of the method as an efficient tool. For instance, future work could focus on a more granular approach to how different DL frameworks react to the same configurations for the same task, in order to better detect performance discrepancies that only occur under certain configurations. Future work may also incorporate ML techniques and search-based algorithms to better choose the optimal configurations to be used in a given differential testing procedure, alongside their optimal values for finding performance differences. Moreover, future work may also integrate differential testing as a complementary approach to other testing methods used to test DL systems, such as unit testing and integration testing.

# References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., … Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th usenix conference on operating systems design and implementation* (pp. 265–283).

Afjehei, S., Chen, T.-H. P., & Tsantalis, N. (2019). iperfdetector: Characterizing and detecting performance anti-patterns in ios applications. *Empirical Software Engineering*, *24*.

Alzamil, Z., & Korel, B. (2005). Application of redundant computation in software performance analysis. In *Proceedings of the 5th international workshop on software and performance* (pp. 111–121).

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., … Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *Proceedings of the 41st international conference on software engineering: Software engineering in practice* (pp. 291–300).

Arel, I., Rose, D. C., & Karnowski, T. P. (2010). Deep machine learning - a new frontier in artificial intelligence research [research frontier]. *IEEE Computational Intelligence Magazine*, *5*(4), 13-18.

Arya, D., Wang, W., Guo, J. L. C., & Cheng, J. (2019). Analysis and detection of information types of open source software issue discussions. In *Proceedings of the 41st international conference on software engineering* (pp. 454–464).

Bachmann, A., & Bernstein, A. (2009). 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* (pp. 119–128).

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798-1828.

Bottou, L., Chapelle, O., DeCoste, D., & Weston, J. (2007). Scaling learning algorithms toward ai. In *Large-scale kernel machines* (p. 321-359).

Braiek, H. B., & Khomh, F. (2022, may). Testing feedforward neural networks training programs. *ACM Trans. Softw. Eng. Methodol.*.

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

Chap3data. (2022). `https://github.com/dlframeworkperfbugs/performance-bugs-in-dl-frameworks`.

Chap4data. (2023). `https://github.com/doej47766/dl_diff_testing`.

Chen, J., Liang, Y., Shen, Q., & Jiang, J. (2022). Toward understanding deep learning framework bugs. *Computing Research Repository (CoRR)*.

Chen, T.-H., Nagappan, M., Shihab, E., & Hassan, A. E. (2014). An empirical study of dormant bugs. In *Proceedings of the 11th working conference on mining software repositories* (pp. 82–91).

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016). Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Trans. Softw. Eng.*, *42*(12), 1148–1161.

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

Cochran, W., Cooley, J., Favin, D., Helms, H., Kaenel, R., Lang, W., ... Welch, P. (1967). What is the fast fourier transform? *Proceedings of the IEEE*, *55*(10), 1664-1674.

Cox, D., & Stuart, A. (1955). Some quick sign tests for trend in location and dispersion. *Biometrika*, *42*, 80-95.

Efraimidis, P., & Spirakis, P. P. (2016). Weighted random sampling. In *Encyclopedia of algorithms* (pp. 2365–2367). New York, NY: Springer New York.

Elshawi, R., Wahab, A., Barnawi, A., & Sakr, S. (2021). Dlbench: a comprehensive experimental evaluation of deep learning frameworks. *Cluster Computing*, *24*(3), 2017–2038.

El Shawi, R., Wahab, A., Barnawi, A., & Sakr, S. (2021, 09). Dlbench: a comprehensive experimental evaluation of deep learning frameworks. *Cluster Computing*, *24*, 1-22. doi: 10.1007/s10586-021-03240-4

Ganeshan, S., Elumalai, N. K., & Achar, R. (2020). A comparative study of magma and cublas libraries for gpu based vector fitting. In *2020 ieee 11th latin american symposium on circuits systems (lascas)* (p. 1-4).

Gulzar, M. A., Zhu, Y., & Han, X. (2019). Perception and practices of differential testing. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (p. 71-80). doi: 10.1109/ICSE-SEIP.2019.00016

Guo, Q., Chen, S., Xie, X., Ma, L., Hu, Q., Liu, H., . . . Li, X. (2019). An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *Proceedings of the 34th ieee/acm international conference on automated software engineering* (pp. 810–822). IEEE Press. Retrieved from https://doi.org/10.1109/ASE.2019.00080 doi: 10.1109/ASE.2019.00080

Hale, J. (2018, Sep). *Deep learning framework power scores 2018.* Retrieved from https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

Han, X., & Yu, T. (2016). 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.*

Harmanen, J., & Mikkonen, T. (2016). On polyglot programming in the web. *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, 102-119.

He, H., Jia, Z., Li, S., Xu, E., Yu, T., Yu, Y., . . . Liao, X. (2020). Cp-detector: Using configuration-related performance properties to expose performance bugs. In *2020 35th ieee/acm international conference on automated software engineering (ase)* (p. 623-634).

Heiberger, R. M., Neuwirth, E., Heiberger, R. M., & Neuwirth, E. (2009). One-way anova. *R* *through Excel: A spreadsheet interface for statistics, data analysis, and graphics*, 165–191.

Hochreiter, S., & Schmidhuber, J. (1997, nov). Long short-term memory. *Neural Comput.*, *9*(8), 1735–1780.

Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., & Tonella, P. (2020). Taxonomy of real faults in deep learning systems. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 1110–1121).

Islam, M. J., Nguyen, G., Pan, R., & Rajan, H. (2019). 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* (pp. 510–520).

Jia, L., Zhong, H., Wang, X., Huang, L., & Lu, X. (2020). An empirical study on bugs inside tensorflow. In *Database systems for advanced applications* (p. 604-620).

Jia, L., Zhong, H., Wang, X., Huang, L., & Lu, X. (2021). The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software*, *177*, 110935.

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. *Sigplan Notices - SIGPLAN*, *47*.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

Lee, A., Carver, J. C., & Bosu, A. (2017). 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)* (p. 187-197).

Liu, L., Wu, Y., Wei, W., Cao, W., Sahin, S., & Zhang, Q. (2018). Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In *2018 ieee 38th international conference on distributed computing systems (icdcs)* (p. 1258-1269).

Liu, Y., Xu, C., & Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. In *In proceedings of the 36th international conference on software engineering (icse 2014).*

Long, J. D., Feng, D., & Cliff, N. (2003). Ordinal analysis of behavioral data. In *Handbook of psychology* (p. 635-661). American Cancer Society.

Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., ... Wang, Y. (2018). Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (pp. 120–131). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3238147.3238202 doi: 10.1145/3238147.3238202

Ma, L., Zhang, F., Sun, J., Xue, M., Li, B., Juefei-Xu, F., ... Wang, Y. (2018). Deepmutation: Mutation testing of deep learning systems. In *Ieee 29th international symposium on software reliability engineering (issre)* (p. 100-111).

Makkouk, T., Kim, D. J., & Chen, T.-H. P. (2022). An empirical study on performance bugs in deep learning frameworks. In *2022 ieee international conference on software maintenance and evolution (icsme)* (p. 35-46). doi: 10.1109/ICSME55016.2022.00012

McFee, B., Raffel, C., Liang, D., Ellis, D., Mcvicar, M., Battenberg, E., & Nieto, O. (2015). librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference* (p. 18-24).

McKeeman, W. M. (1998). Differential testing for software. *Digit. Tech. J.*, *10*, 100-107.

Nistor, A., Jiang, T., & Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (msr)* (p. 237-246).

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... Lerer, A. (2017). Automatic differentiation in pytorch. In *Nips 2017 workshop on autodiff.*

Pei, K., Cao, Y., Yang, J., & Jana, S. (2017, October). Deepxplore: Automated whitebox testing of deep learning system. *Proceedings of the 26th Symposium on Operating Systems Principles*. Retrieved from http://dx.doi.org/10.1145/3132747.3132785 doi: 10.1145/3132747.3132785

Pham, H. V., Qian, S., Wang, J., Lutellier, T., Rosenthal, J., Tan, L., ... Nagappan, N. (2020). Problems and opportunities in training deep learning software systems: An analysis of variance. In *2020 35th ieee/acm international conference on automated software engineering (ase)* (p. 771-783).

PyTorch#100425. (2023). https://github.com/pytorch/pytorch/issues/100425. PyTorch. (Accessed: 2023-04-05)

PyTorch#100617. (2023). https://github.com/pytorch/pytorch/issues/100617. PyTorch. (Accessed: 2023-04-05)

PyTorch#10851. (2018). https://github.com/pytorch/pytorch/issues/10851. Pytorch. (Accessed: 2021-08-26)

PyTorch#11931. (2018). https://github.com/pytorch/pytorch/issues/11931. Pytorch. (Accessed: 2021-08-27)

PyTorch#12006. (2018). https://github.com/pytorch/pytorch/issues/pytorch12006. Pytorch. (Accessed: 2021-08-26)

PyTorch#158. (2016). https://github.com/pytorch/pytorch/issues/158. Pytorch. (Accessed: 2021-08-27)

PyTorch#17206. (2019). https://github.com/pytorch/pytorch/issues/17206. Pytorch. (Accessed: 2021-08-26)

PyTorch#18405. (2019). https://github.com/pytorch/pytorch/issues/18405. Pytorch. (Accessed: 2021-08-27)

PyTorch#18853. (2019). https://github.com/pytorch/pytorch/issues/18853. Pytorch. (Accessed: 2021-08-27)

PyTorch#19797. (2019). https://github.com/pytorch/pytorch/issues/19797. Pytorch. (Accessed: 2021-08-26)

PyTorch#24080. (2019). https://github.com/pytorch/pytorch/issues/24080. Pytorch. (Accessed: 2021-08-26)

PyTorch#33334. (2020). https://github.com/pytorch/pytorch/issues/33334. PyTorch. (Accessed: 2022-16-01)

PyTorch#42265. (2020). https://github.com/pytorch/pytorch/issues/42265. Pytorch. (Accessed: 2021-08-24)

PyTorch#50522. (2021). https://github.com/pytorch/pytorch/issues/50522. Pytorch. (Accessed: 2021-08-26)

PyTorch#5611. (2018). https://github.com/pytorch/pytorch/issues/5611. Pytorch. (Accessed: 2021-08-26)

PyTorch#6222. (2018). https://github.com/pytorch/pytorch/issues/6222. Pytorch. (Accessed: 2021-08-26)

PyTorch#7261. (2018). https://github.com/pytorch/pytorch/issues/7261. Pytorch. (Accessed: 2021-08-26)

PyTorch#7883. (2018). https://github.com/pytorch/pytorch/issues/7883. Pytorch. (Accessed: 2021-08-26)

PyTorch#82. (2016). https://github.com/pytorch/pytorch/issues/82. PyTorch. (Accessed: 2021-11-08)

PyTorch#94457. (2023). https://github.com/pytorch/pytorch/issues/94457. PyTorch. (Accessed: 2023-02-16)

PyTorch#9646. (2018). https://github.com/pytorch/pytorch/issues/9646. Pytorch. (Accessed: 2021-08-26)

Qian, S., Pham, V. H., Lutellier, T., Hu, Z., Kim, J., Tan, L., . . . Shah, S. (2021). Are my deep learning systems fair? an empirical study of fixed-seed training. In *Advances in neural information processing systems (neurips)* (Vol. 34, pp. 30211–30227).

Romano, J., & Kromrey, J. (2006). 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.*

Sapunov, G. (2020, May). *Fp64, fp32, fp16, bfloat16, tf32, and other members of the zoo.* Retrieved from https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407

Schröter, A., Zimmermann, T., Premraj, R., & Zeller, A. (2006). If your bug database could talk... In *Proceedings of the international symposium on empirical software engineering (isese).*

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., . . . Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Proceedings of the 28th international conference on neural information processing systems - volume 2* (pp. 2503–2511).

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

-going-deeper-toward-meaningful-patterns-in-complex-data/ (Accessed:2022-03-28)

Shams, S., Platania, R., Lee, K., & Park, S.-J. (2017). Evaluation of deep learning frameworks over different hpc architectures. In *2017 ieee 37th international conference on distributed computing systems (icdcs)* (p. 1389-1396). doi: 10.1109/ICDCS.2017.259

Sliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 international workshop on mining software repositories (msr)* (pp. 1–5).

Smith, C., & Williams, L. (2001). Software performance antipatterns; common performance problems and their solutions. In *Int. cmg conference* (p. 797-806).

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., & Maia, M. A. (2018). Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner).*

Sujon, M., Shafiuzzaman, M., Rahman, M. M., & Rahman, R. (2016). Characterization and localization of performance-bugs using naive bayes approach. In *2016 5th international conference on informatics, electronics and vision (iciev)* (p. 791-796).

Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., & Kroening, D. (2018). Concolic testing for deep neural networks. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (p. 109–119).

Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., & Segura, S. (2020). Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access*, *8*, 107214-107228. doi: 10.1109/ACCESS.2020.3000928

TensorFlow#11411. (2017). https://github.com/tensorflow/tensorflow/issues/11411. TensorFlow. (Accessed: 2021-08-27)

TensorFlow#14572. (2017). https://github.com/tensorflow/tensorflow/issues/14572. TensorFlow. (Accessed: 2021-08-27)

TensorFlow#14800. (2017). https://github.com/tensorflow/tensorflow/issues/14800. TensorFlow. (Accessed: 2021-08-26)

TensorFlow#17246. (2018). https://github.com/tensorflow/tensorflow/issues/17246. TensorFlow. (Accessed: 2021-08-24)

TensorFlow#32138. (2019). https://github.com/tensorflow/tensorflow/issues/32138. TensorFlow. (Accessed: 2021-08-26)

TensorFlow#3470. (2016). https://github.com/tensorflow/tensorflow/issues/3470. TensorFlow. (Accessed: 2021-08-27)

TensorFlow#40758. (2020). https://github.com/tensorflow/tensorflow/issues/40758. TensorFlow. (Accessed: 2021-08-27)

TensorFlow#41715. (2020). https://github.com/tensorflow/tensorflow/issues/41715. TensorFlow. (Accessed: 2021-08-26)

TensorFlow#60492. (2023). https://github.com/tensorflow/tensorflow/issues/60462. TensorFlow. (Accessed: 2023-04-05)

TensorFlow#60494. (2023). https://github.com/tensorflow/tensorflow/issues/60494. TensorFlow. (Accessed: 2023-04-05)

TensorFlow#60495. (2023). https://github.com/tensorflow/tensorflow/issues/60495. TensorFlow. (Accessed: 2023-04-05)

Tomov, S., Dongarra, J., & Baboulin, M. (2010). Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, *36*(5-6), 232–240.

Viera, A., & Garrett, J. (2005). Understanding interobserver agreement: The kappa statistic. *Family medicine*, *37*, 360-363.

Zaman, S., Adams, B., & Hassan, A. E. (2011). Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th working conference on mining software repositories* (pp. 93–102).

Zaman, S., Adams, B., & Hassan, A. E. (2012). A qualitative study on performance bugs. In *2012 9th ieee working conference on mining software repositories (msr)* (p. 199-208).

Zhang, M., Zhang, Y., Zhang, L., Liu, C., & Khurshid, S. (2018). Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In M. Huchard, C. Kästner, & G. Fraser (Eds.), *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering* (pp. 132–142).

Zhang, T., Gao, C., Ma, L., Lyu, M., & Kim, M. (2019). An empirical study of common challenges in developing deep learning applications. In *2019 ieee 30th international symposium on* *software reliability engineering (issre)* (p. 104-115).

Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., & Zhang, L. (2018). An empirical study on tensorflow program bugs. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis (issta)* (pp. 129–140).

Zhong, H., & Su, Z. (2015). An empirical study on real bug fixes. In *2015 ieee/acm 37th ieee international conference on software engineering* (p. 913-923).

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting defects for eclipse. In *Third international workshop on predictor models in software engineering (promise'07: Icse workshops 2007)* (p. 9-9).