Fault Triggers in the TensorFlow Framework: An Experience Report

Xiaoting Du*, Guanping Xiao†, Yulei Sui‡
*School of Automation Science and Electrical Engineering, Beihang University, China
†College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
‡Australian Artificial Intelligence Institute, University of Technology Sydney, Australia
xiaoting_2015@buaa.edu.cn, gpxiao@nuaa.edu.cn, yulei.sui@uts.edu.au

***Abstract*—TensorFlow is one of the most popular machine learning frameworks for developing machine learning algorithms. Because of the popularity and large-scale use of TensorFlow, even a single bug may lead to severe consequences and impact a large number of users. With a growing number of safety-critical systems built upon TensorFlow, its reliability is becoming increasingly important. An essential step to ensure TensorFlow's reliability is to understand the characteristics of bugs that occurred in TensorFlow.**

This paper presents the first comprehensive empirical study on fault triggering conditions in TensorFlow. 2,285 bug reports from TensorFlow's GitHub repository are collected. A bug classification is performed based on fault triggering conditions, followed by the frequency distribution of different types of bugs and the evolution features of varying bug types over time. Then the relationships between bug types and fixing time are also investigated. In addition, the root causes of Bohrbugs and Mandelbugs are studied. Five root causes are discovered. Furthermore, the analysis of regression bugs in TensorFlow is conducted. We have revealed 10 important findings based on our empirical results. There are 8 implications based on these findings are provided for developers and users.

***Index Terms*—TensorFlow, machine learning framework, fault triggers, Mandelbug, empirical study**

I. Introduction

Machine learning projects are popular in various domains such as data mining [1], image analysis [2], [3], pattern recognition [4], and many other fields [5]–[7]. Due to the large-scale use of machine learning projects, even a single bug with few lines of error code may affect a large number of users and cause disastrous consequences, especially for safety-critical applications. For example, a subtle bug in a self-driving system may cause car accidents [8], [9]. A vulnerability exploited by an attacker in a credit card system may cause cracking of bank accounts [10]. A bug in a web application may lead to web site defacement [11]. In particular, the quality of many machine learning projects highly depends on the quality of the frameworks they build on [12]–[15]. Among all those frameworks, TensorFlow is arguably the most popular one, which is flexible and can be used to support a variety of algorithms and has been adopted to build more than 36,000 applications hosted on GitHub [16].

TensorFlow's reliability is becoming critically important due to the widely use of TensorFlow in dozens of safety-critical applications. To understand the features and characteristics of bugs in TensorFlow, researchers have conducted several empirical studies on TensorFlow. In particular, Zhang et al. [16] conducted an empirical study on deep learning applications built on top of TensorFlow and examined the root causes and symptoms of bugs. Islam et al. [17] studied 2,716 posts from Stack Overflow, and 500 bug-fixing commits from Caffe [13], Keras [18], TensorFlow [12], Theano [15], and Torch [19]. Root causes and impacts of bugs were analyzed. The study in [20] analyzed 715 questions in Stack Overflow related to three popular deep learning frameworks, and summarized them into seven frequently asked questions. Although these studies have investigated the root causes and symptoms of bugs in TensorFlow clients and the TensorFlow framework, none of them has analyzed the factors that trigger a fault and/or propagate a fault inside TensorFlow. These fault triggers are expected to provide valuable insights into TensorFlow's development and maintenance phases.

Generally, fault triggers are complex, not only including the timing of inputs and operations, but also involving the interactions with other systems. In [21], Grottke and Trivedi divided bugs into two categories: Bohrbug (BOH) and Mandelbug (MAN), according to the complexity of fault activation and/or error propagation conditions. Among them, Bohrbug is easy to be isolated, and their manifestation is consistent under a well-defined set of conditions. In contrast, Mandelbug is a special kind of faults, whose activation and/or error propagation is complex and difficult to be discovered by traditional techniques. In addition, a Mandelbug can be further categorized as a non-aging related Mandelbug (NAM) or an aging-related bug (ARB). Aging-related bug is a kind of bug that can lead to the software aging phenomenon, i.e., to an increase in the failure rate and/or performance degradation [22]. According to the above classification, the work presented in [23] extended a more fine-grained classification for aging-related bugs and non-aging related Mandelbugs.

In this paper, we make the first attempt to explore the bug characteristics in the TensorFlow framework based on fault triggering conditions. We have conducted an extensive study of 2,285 bug reports from TensorFlow's GitHub repository[1]. Bug characteristics of TensorFlow are investigated from several

*Corresponding author: Xiaoting Du.

[1] TensorFlow's GitHub repository: https://github.com/tensorflow/tensorflow

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00010

aspects, including (1) the frequency distribution of different types of bugs, and the proportions of different bugs during their evolution over time; (2) the relationship between a bug type and its bug fixing time; (3) the five root causes of Bohrbugs and Mandelbugs, including environment and configuration, memory, compatibility, concurrency, and semantic; and (4) different features of regression bugs in TensorFlow. For each report, we have examined the bug descriptions, comments, linked pull requests and the corresponding commits. The contributions of our work provide answers to the following four research questions.

RQ1: What is the distribution of different bug types in TensorFlow?

To answer this question, bugs in TensorFlow are classified based on their fault triggers. In addition, we have investigated the evolution of different types of bugs over time. Many findings are obtained, for example, 78.17% of bugs in TensorFlow are Bohrbugs. The detailed results and analysis are reported in Section III.

RQ2: How much time is spent to fix different types of bugs?

To answer this question, we calculate the fixing time of different types of bugs, including the average fixing time and the median fixing time. We have also analyzed the correlations between bug types and their fixing time. We obtain several findings, for example, fixing a Mandelbug costs much more time than that when fixing a Bohrbug. The detailed results are shown in Section IV.

RQ3: What are the root causes of Bohrbugs and Mandelbugs?

In RQ1, bugs are classified into Bohrbugs and Mandelbugs. Root causes of Bohrbugs and Mandelbugs are further investigated in this section. The root cause of a bug helps us understand the nature of the bug. Through cross-analysis of fault triggering conditions and root causes, we observed several important findings. For example, about half of Mandelbugs in TensorFlow are caused by memory bugs. We introduce the results in detail in Section V.

RQ4: What is the feature of regression bugs in TensorFlow?

Regression bug is a type of bug that causes a feature, which worked normally in previous versions but stopped working after a certain code commit. Investigating the distribution of regression bugs and the features of these bugs in TensorFlow could help TensorFlow developers with their debugging and program repair, thus preventing future regression bugs. Detailed analysis is conducted in Section VI. To the best of our knowledge, this paper is also the first time to explore the proportion and characteristics of regression bugs in the TensorFlow framework.

The rest of the paper is structured as follows. Section II presents the study methodology utilized in this paper, including our research data, bug classification approach, bug classification procedure and the metric used to analyze the correlation among different types of bugs. We present the answers for the four research questions in Sections III-VI. Section VII reports the threats to validity of our study. In Section VIII, we review the related work. Finally, the conclusion is given in Section IX.

TABLE I
DETAILS OF DATA SET

Project	Time frame	Status	# of reports
TensorFlow	Nov. 26, 2015- Nov. 26, 2019	Closed	2,285

II. Study Methodology

In this section, we first introduce the research data and the bug terminologies used in this paper. Then, we discuss the root causes of Bohrbugs and Mandelbugs in TensorFlow. Next, we present the procedure that is conducted to classify bugs. Lastly, we describe the method used for correlation analysis of different bug types.

A. Research Data

The bug reports are collected from TensorFlow's GitHub repository. There are two steps for collecting data set, i.e., report filtering and retracting bug reports and related information. Each step is described in detail as follows.

Report filtering. In the TensorFlow's GitHub repository, there are two types of bug reports: opened bug report and closed bug report. To ensure the correctness of the results, only closed bug reports are studied. Bug reports still under the opened status are excluded because they are under discussion between reporters and developers. Thus their types cannot be determined. To narrow down the scope of bug reports, we filtered closed bug reports with the label *"type: bug"*. During the process of our analysis, bug reports are continuously closed, so we chose the data ranging over a period from Nov. 26 of 2015 to Nov. 26 of 2019, i.e., four years from the time when the first bug report was submitted. Finally, 2,285 bug reports are obtained. The details of the data set are shown in TABLE I.

Retracting bug reports and related information. We obtained relevant bug information through the GitHub API, including the reporters' descriptions, opened time, closed time, and comments of a bug. In addition, a bug can be fixed by one or more pull requests with multiple code commits. We identified the corresponding pull requests or commits in the form of links in the bug reports. For further analysis, we also collected all these relevant pull requests and commits.

B. Bug Classification based on Fault Triggering Conditions

Our bug classification method is adopted from [23]. According to the conditions related to the fault activation and error propagation, a bug can be classified as a Bohrbug (BOH) or a Mandelbug (MAN). The definitions of Bohrbug and Mandelbug are given as follows.

- **Bohrbug:** a bug whose activation and/or error propagation are simple and can be reproduced consistently under a well-defined set of conditions.
- **Mandelbug:** a bug whose activation and/or error propagation are complex. The following reasons may cause the

complexity of the triggering conditions: there is a time lag between the fault activation and failure occurrence; the possible influence of indirect factors, such as the interactions between the software application with its system-internal environment; the timing of inputs and operations; and the relative sequence of inputs and operations.

According to whether a Mandelbug would cause a software aging problem, Mandelbug is further divided into two subtypes, i.e., non-aging related Mandelbugs (NAMs) and aging-related bugs (ARBs). The definitions of NAM's subtypes are listed below.

- **LAG:** There exists a time lag between fault activation and the occurrence of its failure;
- **ENV:** The interaction of the software application with its system-internal environment would affect the activation of the bug and/or error propagation;
- **TIM:** The timing of inputs and operations has impacts on the fault activation and/or error propagation;
- **SEQ:** The sequence (i.e., the relative order) of inputs and operations would influence the fault activation and/or the propagation of error.

The definitions of ARB's subtypes are listed below:

- **MEM:** ARBs caused by the accumulation of errors because of improper memory management (e.g., memory leaks, buffers not being flushed);
- **STO:** ARBs caused by the accumulation of errors as a result of improper storage space management, for example, disk space is consumed by the bug;
- **LOG:** ARBs caused by the leaks of other logical resources, e.g., inodes or sockets that are not freed after usage;
- **NUM:** ARBs caused by the accumulation of numerical errors, for instance, integer overflows and round-off errors;
- **TOT:** ARBs caused by the increase of the fault activation or error propagation rate with total system runtime, but it is not induced by the accumulation of internal error states.

C. Root Causes of Bohrbugs and Mandelbugs in TensorFlow

In this part, we aim to identify the root causes of Bohrbugs and Mandelbugs that appeared in TensorFlow. Through analyzing the bugs in TensorFlow and referring to the definition of root causes in [24] and [25], we identified five root causes of bugs in TensorFlow, as listed below:

- **Environment and configuration:** The error reside the dependent libraries, underlying operating systems, or the non-code that affects the function;
- **Concurrency:** The synchronization problems exist among the concurrent threads or processes in concurrent programs;
- **Memory:** The bugs are caused by improper handling of memory objects;
- **Compatibility:** A program cannot normally run on a specified CPU architecture, operating system, or web browser, etc.;

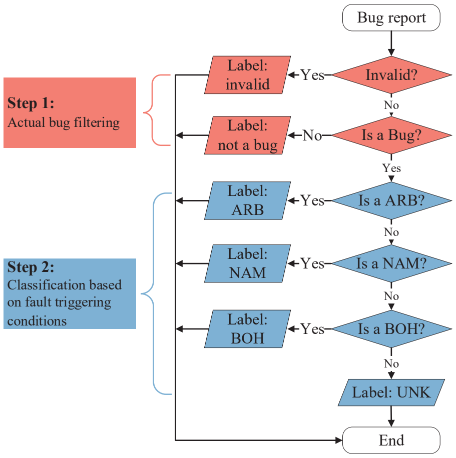

Fig. 1. Process of bug report filtering.

- **Semantic:** The bugs that are inconsistent with requirements or the programmers' intention and do not fall into the above categories.

D. Bug Report Classification Procedure

We analyzed bug reports of TensorFlow's GitHub repository, ranging from Nov. 26, 2015, to Nov. 26, 2019. We focused on closed bug reports labeled with *"type: bug"* and manually looked into the description and comments of each bug report. The descriptions in bug reports were sometimes ambiguous, so we further exploited related pull requests and commits. For a given bug report, the classification process is divided into two steps, as shown in Fig. 1. Each step is described as follows.

Step 1: Actual bug filtering. A bug report should first be examined to make sure that it contains an actual bug. In our study, invalid bug reports were filtered out first, i.e., bug reports containing too little information to determine whether it is a bug or not. For example, many bug reports were closed due to lack of activity, stale, deprecated and out of the range of TensorFlow's GitHub repository. Then, a bug report which did not contain an actual bug was filtered out, i.e., requests for new features or enhancements, documentation issues (e.g., missing information, outdated documentation, or harmless warning outputs), compile-time issues (e.g., cmake errors or linking errors), operator errors and duplicate bug reports.

Step 2: Classification based on fault triggering conditions. We carefully checked the descriptions and comments of bugs, together with linked pull requests and commits, to find out what is the activation condition of each bug, such as the operations or inputs before a bug was triggered; how the bug was propagated, for example, what parameters or states of the program were changed by the bug, and how the changed parameters or states were propagated; what the user observed when a failure occurred.

According to the definition and characteristics of each bug type, we checked in turn whether the bug belongs to ARB, NAM, or BOH, respectively. For the classification of

TABLE II
EXAMPLES OF BOHRBUGS AND MANDELBUGS

Bug ID	Bug Type	Description
9012	BOH	"While running the above script, despite the device has been specified to be GPU, tensorflow still try to do the BiasGradOp on CPU and will cause an error because of the data format..."
10954	LAG/NAM	"Supervisor: SummaryWriter and Saver stop after some time: ... It works all well for up to 30mins - 1h30mis. But after that time the summary_writer stops to write events, and..."
4135	ENV/NAM	"...I was able to reproduce this on Mac OS X with the nightly. So this appears to be a Mac only bug..."
9125	TIM/NAM	"...The ordering of ops that are copied is not deterministic so this error pops up somewhat randomly...Example code snippet (note: you may need to run this multiple times to get a failure)..."
18266	SEQ/NAM	"Cache lockfile already exists: ...I get an error because the evaluation starts before that all cache is written on the filesystem...If the cache is written before the first evaluation, I don't get the error..."
6599	MEM/ARB	"memory leak in tensorflow_gpu 0.12.1: ...after some upgrade in cuda and tensorflow, I see large memory leak in our server with 32GB RAM..."
28798	STO/ARB	"tf.data.Dataset::cache doesn't cleanup unused lock and tmp files: ...tf.data.Dataset::cache doesn't cleanup unused lock and tmp files..."
5401	LOG/ARB	"Creating and fitting a Trainable leaks file descriptors: ...During the run "lsof" is indicating that the number of file descriptors used is increased on every loop iteration..."
34390	NUM/ARB	"[TF2.0] tf.reduce_mean crashes Python (Floating point exception) if the count becomes zero due to overflow: ...Crashes the Python interpreter (e.g. Floating point exception (core dumped)). (Likely as 256 overflows in uint8 to 0, leading to an uncaught division by zero..."
4820	TOT/ARB	"Thread created by SummaryWriter not killed: ...the _EventLoggerThread created by summary writer does not get killed by the close method, which will make the number of threads keep increasing until it exceeds the system capacity..."

TABLE III
EXAMPLES OF ROOT CAUSES AND THEIR CORRESPONDING BUG TYPES

Bug ID	Bug Type	Description
12037	Environment/configuration	"Missing tf_python_protos_cc library dependency in tf_tutorials.cmake: ...can't build tf_tutorials_example_trainer due missing library dependency to tf_python_protos_cc.lib..."
6378	Concurrency	"WhereOp: Race condition between counting the number of true elements and writing them"
33960	Memory	"Docker/Kubernetes memory limits not respected? OOMKilled when deployed to GCP"
6171	Compatibility	"tfprof: Python3 incompatibility: ...This line in tfprof_logger.py uses dict.iteritems(), which breaks my Python 3 code..."
9312	Semantic	"Typo in seq2seq.attention_wrapper.py: ...I think there is a small typo in contrib.seq2seq.attention_wrapper.py, would someone like to check it?..."

ARB subtypes, if a bug was classified as an ARB, but there was not enough information to determine which subtype it belongs to, then it was labeled as ARU. Similarly, NAU is a NAM, but there was insufficient information to decide its specific subtype. Finally, if a bug report did not have enough information to be classified as ARB, NAM, or BOH, it was labeled as UNK.

The classification was implemented manually by two authors who are familiar with machine learning projects and have experience in developing machine learning applications based on TensorFlow. We have released our dataset online[2]. When encounter suspicious classified cases, a cross-check will be taken. To clarify the classification, TABLE II shows some examples of BOHs and MANs and their partial descriptions.

Moreover, after dividing bugs into BOH, ARB, and NAM, we aim to find out the root cause of each bug. We identified the root cause of a bug by manually examining the information contained in bug reports (i.e., descriptions and comments, linked pull requests and commits). For the classification, we first read the description and comments of a bug report, and then read corresponding code changes in the linked pull requests and their commits. Finally, we summarized five root causes of bugs in TensorFlow, including environment and configuration, memory, compatibility, concurrency, and semantic, and classify each bug report accordingly. In TABLE III, examples of different root causes and their corresponding bug types are listed.

E. Correlation Metric

To study the correlation between two types of bugs, we used a statistical metric called $lift$ [26], [27]. The $lift$ value of category A_i and category B_j is represented by $lift(A_i, B_j)$,

[2] Classified data: https://2.gp/a74xm

and calculated as $P(A_iB_j)/(P(A_i) * P(B_j))$, where $P(A_i)$ and $P(B_j)$ are the probability of A_i and B_j, respectively. $P(A_iB_j)$ is the probability that a bug belongs to both category A_i and B_j. For instance, if there are 100 actual bugs in total, 80 of which are BOHs, 70 of which are semantic bugs, and 60 of which are BOHs caused by semantic bugs, we can calculate the $lift$ correlations between BOHs and semantic bugs as $lift(A_i, B_j) = P(A_iB_j)/(P(A_i) * P(B_j)) = (60/100)/((70/100)*(80/100)) = 1.07$, where A_i is semantic bug, B_j is BOH.

If $lift(A_i, B_j)$ is equal to 1, it means that $P(A_iB_j) = P(A_i) * P(B_j)$, which indicates that category A_i and B_j are not correlated. If $lift(A_i, B_j)$ is greater than 1, categories A_i and B_j are positively correlated, that is, a bug belongs to A_i is more likely to belong to B_j too. Similarly, if $lift(A_i, B_j)$ is less than 1, it means that category A_i and B_j are negatively correlated. If a bug belongs to A_i, it is unlikely to belong to B_j. In the above example, $lift(A_i, B_j)$ is 1.07, which means that BOHs are more likely to be caused by semantic bugs.

III. Bug Classification

This section answers the RQ1. As depicted in Section II-A, after filtering out bug reports with label *"type:bug"*, 2,285 closed bug reports were obtained in the period of 4 years, from Nov. 26, 2015, to Nov. 26, 2019. In this section, the distribution of bugs classified based on the fault triggering conditions is investigated.

A. Distribution of Actual Bugs, Non-bugs and Invalid Bug Reports among All the Bug Reports

In this section, the distribution of actual bugs, non-bugs, and invalid bug reports are analyzed, results are shown in Fig. 2.

Finding #1: *Among all the classified 2,285 bug reports, each of the 41.71% reports contains an actual bug, while 44.86% of them do not contain bugs, and the number of invalid bug reports accounts for 13.44%.*

Fig. 2 illustrates the distribution of the extracted bug reports. After manual examination, it can be observed that actual bugs account for 41.71% of all extracted bug reports. The percentage of non-bugs is 44.86%, and the proportion of invalid bug reports is 13.44%. It should be noted that in this study, as described in Section II-D, invalid bug reports were those that lack of information. Therefore they cannot be determined whether they contain actual bugs or not. After classification, 307 bug reports were labeled as invalid. There are several reasons for these invalid bug reports being closed. The most common reason is inactivity, i.e., the bug reports have not been discussed for months or even years after the last comment. In addition to inactivity, there were other situations, such as bugs corresponding to deprecated features or not within the scope of the TensorFlow's GitHub repository. Non-bugs are whose reports are related to (1) the requests of features or enhancements, (2) the descriptions of compile-times issue or documentation issues, or (3) duplicated reports. According to our examination, most of them are simply requesting new features or enhancements. The high proportion of non-bugs is because developers mistakenly labeled non-bugs, such as feature requests, performance or builds, as bugs.

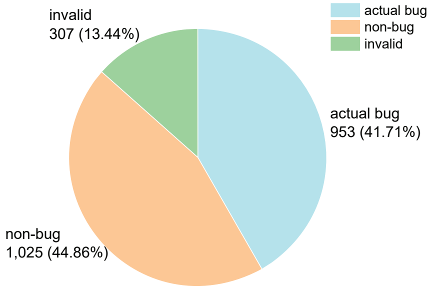

Fig. 2. Numbers and percentages of actual bug, non-bug and invalid bug report.

Implications: This finding indicates that invalid reports and non-bugs account for a large proportion of all classified bug reports. For invalid reports, the inactivity of reporters and assigned developers is the primary reason that bug reports falling in inactivity or stale and closed without being handled. On the one hand, for reporters, once a report is submitted, reporters should pay attention to the developers' comments and promptly assist the developers with reproducing the bug. On the other hand, for maintainers, automatic mechanisms could be introduced to identify bug reports' status to be able to close inactivity bug reports timely. Moreover, tools that automatically assign developers can be used as an auxiliary to reduce the time and resource overhead caused by the repeated assignment of new developers. For non-bugs, reporters should be encouraged to label their submitted reports, for example, labeling [feature request], [build], or [performance] at the beginning of the bug report. A tool could be integrated to identify duplicate bug reports. By recommending correlated bug reports to reporters, it can reduce the repetition rate of bug reports and the burden of TensorFlow's maintainers.

B. Distribution of BOH, ARB and NAM among All the Actual bugs

In this section, actual bugs are divided into Bohrbugs (BOHs) and Mandelbugs (MANs), and Mandelbugs are further classified into aging-related bugs (ARBs) and non-aging related Mandelbugs (NAMs). If there is insufficient information to divide the actual bug into BOH or MAN, it is labeled as UNK.

Finding #2: *Among 953 actual bugs, the proportions of BOHs and MANs are 78.17% and 16.47%, respectively. The proportion of UNK is 5.35%.*

Fig. 3 shows the number and percentage of each bug type (i.e., BOH, ARB, NAM, and UNK). As depicted in Fig. 3, more than two-thirds (i.e., 78.17%) of the actual bugs in TensorFlow are BOHs. Compared to other software systems, the proportion of BOHs in TensorFlow is higher than that of Linux kernel (i.e., 55.82% [28]), Android (i.e., 65.2% [29]), and MySQL (i.e., 56.6% [23]). Moreover, MANs, including ARBs and NAMs, account for 16.47% of actual bugs. Compared with traditional software systems, the proportion

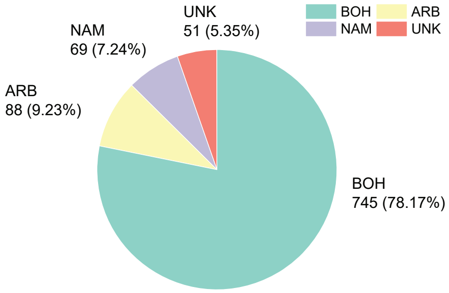

Fig. 3. Numbers and proportions of BOH, ARB and NAM.

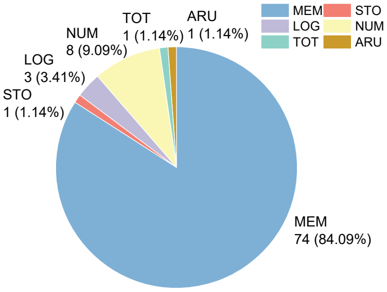

Fig. 4. Subtype distribution of aging-related bug.

of MANs in TensorFlow is lower than that of Linux kernel system (i.e., 36.34% [28]), Android OS (i.e., 31.4% [29]), MySQL (i.e., 38% [23]), and space mission on-board software (i.e., 36.5% [30]). The results show that in TensorFlow, BOH accounts for a larger proportion even though they can be easily reproduced and debugged under a well-defined set of conditions. This phenomenon may be caused by the following three reasons. First, it is more challenging to test TensorFlow than traditional software projects [31], [32]. Compared with traditional software projects, which are relatively more deterministic, machine learning testing is more challenging due to the fundamentally different nature and construction methods of machine learning systems. For example, it usually requires a complicated testing environment and a large testing space. For these reasons, machine learning systems are sometimes regarded as "non-testable" software. The second reason is that in order to adapt to the rapidly increasing functional requirements, new functions have been constantly added in TensorFlow, which introduce BOHs at the same time. Another reason is that the activation and/or propagation conditions of MANs are more complex than BOHs, making MANs more difficult for users to discover. As a result, the number of MANs is much fewer than that of BOHs.

Implications: Since BOH accounts for more than two-thirds of bugs in TensorFlow, the testing of BOH should be the focus of TensorFlow testing. We suggest conducting sufficient testing before releasing a version. For example, static program analysis [33]–[35] could be used to detect bugs in code, and automatic program repair tools could be used to solve typical bugs, such as null pointer dereferences [36]. Considering that TensorFlow has been deployed to a wide variety of platforms ranging from the mobile devices to large-scale systems with thousands of GPUs, developers should test a feature in different environments.

C. Subtype Distribution of Aging-related Bugs

Finding #3: *The major subtype of ARBs is MEM (i.e., 84.09%).*

In this section, we further explore the proportion of ARB's subtypes. Fig. 4 depicts the numbers and percentages of ARB's subtypes, among which MEM accounts for more than four-fifths (i.e., 84.09%) of the total ARBs. The result is higher than that of the Linux kernel (i.e., 68.78% [28]) and Android (i.e., 76.2% [29]). That is because TensorFlow is often used to process large-scale images or to train complex neural networks, which consume too much memory and rely heavily on memory devices. Unreleased resource in the iterative loop is an important reason for out-of-memory. For example, Bug ID-14181: *"...with the increasing time the whole process starts consuming more and more RAM although it should clean it up..."*. In this report, TensorFlow has the memory cleanup issue, and as the loop iteration increases, it will cause excessive memory consumption.

Implications: We suggest that developers (1) pay special attention to the release and consumption of resources in TensorFlow, and (2) provide guidelines for users to configure the parameters and structure of their models to avoid insufficient memory problems. In addition, tools [37], [38] for code analysis could be used to debug memory leak bugs in TensorFlow.

D. Subtype Distribution of Non-aging Related Mandelbugs

Finding #4: *The major subtypes of NAMs are TIM (i.e., 46.38%) and ENV (i.e., 31.88%).*

Fig. 5 shows the numbers and proportions of NAM's subtypes, among which the percentage of TIM is 46.38% and the percentage of ENV is 31.88%. These two categories are the two major subtypes of NAM. This result is close to that of the Linux kernel (i.e., the proportion of TIM is 37.23%, the percentage of ENV is 36.51% [28]). Due to the characteristics of TensorFlow, it is reasonable that TIM and ENV have a higher proportion. TensorFlow inherently must handle concurrent activities and access shared resources, which would inevitably cause timing-related problems. Typical TIM bugs in TensorFlow are deadlock and data race. Deadlock occurs when two or more threads attempt to access shared resources held by other threads, and neither is willing to give them up [39]. Data race occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [40]. For example, a deadlock occurs in Bug ID-932: *"ThreadPool dtor does not pop waiters from waiters_ list... thread pool deadlocks because some notifications are consumed by the leftover dead waiters instead of alive threads..."*. An environment-related bug occurs during interaction with platforms. The complexity and diversity of the operating environment of TensorFlow resulting in the high ratio of ENV bugs. For example, Bug ID-4135 is

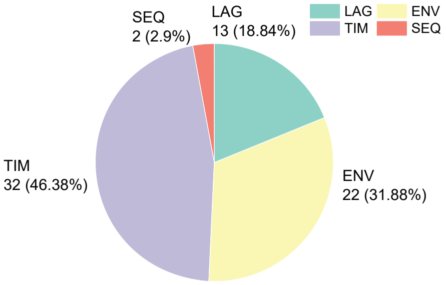

Fig. 5. Subtype distribution of non-aging related Mandelbugs.

a Mac only bug and does not crash on Linux. Bug ID-4521, which is a GPU specific bug, works fine on the CPU.

Implications: Testing NAMs in TensorFlow should focus on TIM and ENV bugs. More specifically, in order to deal with TIMs, it is recommended to use pairwise testing to expose concurrency bugs in TensorFlow. Concurrency bug detectors, such as race free-type technique, static analysis technique, dynamic analysis technique and hybrid technique, could be used to detect deadlock and data race. For ENV bug testing, it should focus on fault-tolerance techniques since the environment is unpredictable and uncontrollable.

E. Evolution of the Proportion of Bohrbugs and Mandelbugs over Time

In the following, we analyze the evolution trend of proportion of Bohrbugs and Mandelbugs over time, as shown in Fig. 6.

Finding #5: *With the development of time, in the first two years, the proportion of BOH tends to decrease, while in the following two years, the proportion of BOH tends to increase. For MAN, this ratio tends to grow in the first two years and then declined in the following two years.*

Fig. 6 shows the evolution of BOH and MAN's ratio in the four years from Nov. 2015 to Nov. 2019. It can be clearly seen from Fig. 6 that in the first two years around, the proportion of BOH dropped obviously, and then slowly increased in the next two years. Compared with BOH, the proportion of MAN tended to increase in the first two years and then decreased gradually. The evolution trend of the proportion was tested through the Mann-Kendall trend test [41], [42], as shown in TABLE IV. The Mann-Kendall test results indicate that for a significance level of $\alpha = 0.05$, the above conclusions are all statistically significant. Through further research, we found that these phenomena may be correlated with the evolution of TensorFlow. With the development, TensorFlow began to support distributed operations and multi-platforms in Apr. 2016 and Jun. 2016, respectively, which led to an increase in the number of MAN, especially TIM and MEM bugs. After version 1.0, the first official version of TensorFlow was released in Feb. 2017, TensorFlow's development became matured. With the increase and expansion of functions, the proportion of BOH gradually increased, which in turn caused the percentage of MAN to decreased progressively.

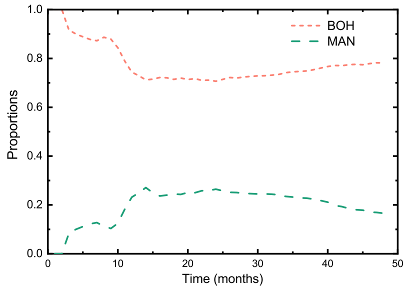

Fig. 6. The evolution of proportion of Bohrbugs and Mandelbugs among all the valid bug reports.

TABLE IV
RESULTS OF MANN-KENDALL TREND DETECTION FOR FIG. 6

Type	Time frame	p value	Trend
BOH	Nov. 26 2015-Nov. 26 2017	<0.001	decreasing
BOH	Nov. 27 2017-Nov. 26 2019	<0.001	increasing
MAN	Nov. 26 2015-Nov. 26 2017	<0.001	increasing
MAN	Nov. 27 2017-Nov. 26 2019	<0.001	decreasing

IV. FIXING TIME OF BUGS

This section answers the RQ2. In this part, the relationship between bug type and the corresponding fixing time is analyzed. Since the time to fix a bug is not recorded in the bug report, we estimate the fixing time based on the difference between the time when the report was submitted and the time when it was last closed.

Finding #6: *It takes more time to close an invalid bug report than close an actual bug.*

From TABLE V, we can observe that it takes the longest time to close an invalid bug report, whether it is the average time (i.e., 120.2 days) or the median time (i.e., 63.2 days). It is followed by the actual bug. Closing an actual bug takes an average of 96.4 days, with a median time of 37.3 days. The results are further verified by the Wilcoxon-Mann-Whitney test [43] with a significance level of $\alpha = 0.05$. After performing the test, we obtained a p value of 0.000262. Therefore, it tends to take more time to close invalid reports than close actual bugs. The reason why it takes more time to close invalid bug reports is because a large number of them have not received any response for months or even years, and finally they have to be closed because of inactivity or stale. There are two typical reasons for inactivity. One of them is that after a report is submitted by the reporter and assigned to a developer, the developer does not respond to the bug report, causing the report to fall into the crack and be closed. Another reason is the inactivity of reporters who cannot provide sufficient information to help the developers reproduce or locate the bug. Besides, the average time of closing a non-bug is 76.4 days, and the median time is 27.3 days. It is a great waste of developers' time to dealing with invalid bug reports or answering questions that should be submitted in the Q&A platform (e.g., Stack Overflow).

Implications: To improve the efficiency of fixing bugs, users should provide high-quality bug reports. Provide sufficient debugging information so that developers can reproduce

TABLE V
FIXING TIME OF ACTUAL BUGS, NON-BUGS, AND INVALID REPORTS

Fixing time (day)	Actual bug	Non-bug	Invalid
Average	96.4	76.4	120.2
Median	37.3	27.3	63.2

the bug as soon as possible. For example, a reporter submitted two reports (Bug ID-4651 and Bug ID-5394). Neither of them provided reproduce instructions and was eventually closed. TensorFlow's maintainers should assign bug reports to developers more accurately and update the status of bug reports in a timely manner. Developers should always respond to bug reports assigned to them, or reassign them to other relevant developers.

Finding #7: *In TensorFlow, it tends to take more time to fix a Mandelbug than fix a Bohrbug.*

TABLE VI
FIXING TIME OF BOHRBUGS AND MANDELBUGS

Fixing time (day)	BOH	MAN
Average	91.0	129.1
Median	33.6	69.2

TABLE VI shows the average fixing time and median fixing time of Bohrbugs and Mandelbugs. As can be seen from the results, the average time to fix a Mandelbug is 129.1 days. In comparison, the average time to fix a Bohrbug is 91.0 days. As for median fixing time, Mandelbug's median fixing time is 69.2 days, over twice that of Bohrbug's median fixing time (i.e., 33.6 days). We further verify the results by performing the Wilcoxon-Mann-Whitney test. For a given criteria ($\alpha = 0.05$), the p value is 0.000687, which means it is more likely to take a longer time to fix a Mandelbug than to fix a Bohrbug. The results is consistent with traditional software systems, such as MySQL [23], Linux kernel [28], and Android [29]. The definition of Bohrbug and Mandelbug can explain this phenomenon. Due to the complexity of error activation and/or error propagation conditions, Mandelbug is challenging to reproduce, while Bohrbug is easy to reproduce. To reproduce a Mandelbug, developers usually need more information to understand the underlying root cause in the code. Besides, the non-deterministic feature of Mandelbug requires a strict replication environment, and the construction of the reproduction environment is also an important reason for the long fixing time of Mandelbug. Moreover, it may need to run the code multiple times, or it takes a long period of time to trigger a Mandelbug.

Implications: Since Mandelbugs seem to be more difficult to reproduce and fix, specific strategies should be developed to deal with Mandelbugs. Mitigation approaches could be used in TensorFlow to prevent the appearance of Mandelbugs, such as fault tolerance [44] and software rejuvenation [45].

V. ROOT CAUSES

The root cause of a bug is important for understanding and fixing the bug. In this section, we aim to identify the root cause of Bohrbugs and Mandelbugs inside TensorFlow. Through manual examination, five root causes are discovered, including environment and configuration, memory, compatibility, concurrency, and semantic. The definitions of these root causes have described in Section II-C. Here, we investigate the root causes of BOHs, ARBs, and NAMs, respectively. The results are shown in the TABLE VII.

Finding #8: *98.93% of BOHs are caused by semantic bugs, 84.09% of ARBs are caused by memory bugs, and 39.13% of NAMs are caused by concurrency bugs.*

TABLE VII shows the root cause distribution among BOHs, ARBs, and NAMs. For BOHs, the major root cause is semantic bug, accounting for 98.93%. Compared with BOHs, only 14.77% of ARBs and 21.74% of NAMs are caused by semantic bugs. The reason for this phenomenon is that semantic bug is a kind of bug that corresponds to the inconsistencies with requirements or the programmers' attention. Quite a few semantic bugs are caused by incorrect functionality implementation or typos, which are prone to introduce BOHs. The major root cause of ARBs is memory bug, accounting for 84.09%, followed by the semantic bug, accounting for 14.77%. However, only 0.4% of BOHs and 7.25% of NAMs are caused by memory bug. It is reasonable that most ARBs are memory bugs since memory bug indicates the improper handling of memory objects. For example, a memory bug in Bug ID-16163, after closing a session and creating a new one, the memory of the previous session is not freed. As a result, memory consumption increases after each call of *session.run()*. Improper processing of memory objects leads to the accumulation of memory consumption, consistent with the definition of ARB. It should be noted that not all memory bugs would cause ARBs, only memory bugs that result in accumulation of memory consumption are ARBs. The primary root cause of NAM is concurrency bug, accounting for 39.13%, followed by environment/configuration bug and semantic bug, accounting for 27.54% and 21.74%, respectively. In contrast, concurrency bugs do not result in BOHs and ARBs. In order to further understand the correlation between root causes and bug types, we show the *lift* correlation (defined in Section II-E) in TABLE VIII. Numbers greater than 1 indicate positive correlation and are shown in bold.

Finding #9: *A NAM is prone to be an environment/configuration bug or a concurrency bug; an ARB is more likely to be a memory bug; a BOH is more likely to be a compatibility bug or a semantic bug.*

As presented in TABLE VIII, a NAM is more likely to be an environment/configuration bug or concurrency bug. NAM bug is a type of bug embodied in the four subtypes: LAG, ENV, TIM, and SEQ. Among them, TIM bug is mainly caused by concurrency bug, especially deadlock and data race. ENV bug is mainly caused by environment bug, i.e., errors in dependent libraries, underlying operating systems, or non-code that affects functionality. An ARB is preferably a memory bug. The reason why ARB is positively related to memory bug is that the major subtype of ARB is MEM, which is related to the accumulation of errors as a result of improper memory management. A BOH is more likely to be a compatibility

TABLE VII
Distribution of Root Causes among Different Bug Types

Root cause	BOH	ARB	NAM
Environment/configuration	0	0	19
Memory	3	74	5
Compatibility	5	0	0
Concurrency	0	0	27
Semantic	737	13	15
UNK	0	1	3
Total	745	88	69

TABLE VIII
Correlation between Bug Types and Root Causes

Root cause	BOH	ARB	NAM
Environment/configuration	0	0	**13.07**
Memory	0.04	**9.25**	0.80
Compatibility	**1.21**	0	0
Concurrency	0	0	**13.07**
Semantic	**1.17**	0.17	0.26

bug or a semantic bug. The reason is that a large number of semantic bugs are wrong functionality implementation or typos, and compatibility bug is a kind of bug that causes software cannot normally run on a particular CPU architecture, operating system, or Web browser, etc. Both of them are easy to trigger and can always be reproduced under certain conditions. Therefore, semantic bugs and compatibility bugs are more likely to be BOHs.

Implications: For NAMs, attention should be paid to environment/configuration bugs and concurrency bugs. The main point to prevent the appearance of ARBs is to solve memory bugs. For BOHs, developers could reference the solutions for semantic bugs and compatibility bugs.

VI. Regression Bugs in TensorFlow

In this section, we present the results of the RQ4. A regression bug in TensorFlow means that a bug causes a feature, which worked normally in previous versions, but stopped working after a certain event, such as fixing a bug or adding a new feature [46], [47]. A regression bug can be caused by a commit fixing an existing bug or implementation for a new system feature [48]. In this part, we make a statistic on the number of regression bugs and the distribution of bug types among regression bugs.

Regression bugs are determined according to examining the textual information contained in bug reports (e.g., the description of bug and discussion comments) based on the definition of the regression bug. One of the most common situations of regression in TensorFlow is that a regression bug is introduced by a commit which is used to fix an existing bug. For example, TensorFlow regression bug ID-3035 (*"The following fails with UnboundLocalError after b80a4a8"*) was introduced by commit ID-*b80a4a8*, which is used to resolve improper exception handling problems of TensorFlow. Different from a bug fix, another situation is that after implementing some new features, a feature stops working, e.g., TensorFlow Bug ID-9708: *"tf.random_crop exception after upgrading to tf1.1 from tf1.0"*.

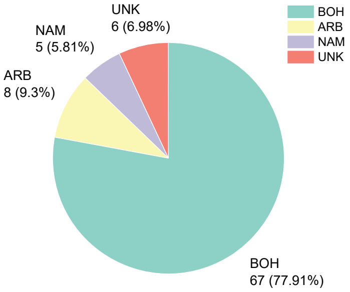

Fig. 7. Bug type distribution among regression bugs.

TABLE IX
Correlation between Bug Types and Regression Bugs

Correlation	BOH	MAN
Regression	1	0.92
Non-regression	1	**1.01**

Finding #10: *Among the 953 TensorFlow actual bugs we classified in this paper, 86 of them are regression bugs, accounting for 9.02%. In all regression bugs, BOHs, MANs and UNKs account for 77.91%, 15.12%, and 6.98%, respectively.*

We first calculate statistics for the number of regression bugs and non-regression bugs, which are 86 and 867, respectively. In TensorFlow, 9.02% of actual bugs are regression bugs, i.e., a feature of software stopped working after some changes. Among all the regression bugs, the number of BOHs account for 77.91%, ARBs and NAMs account for 9.30% and 5.81%, respectively, as shown in Fig. 7. Compared to traditional software projects, the proportion of regression bugs in TensorFlow is lower than that in Linux (50.1% [48]), and Google Chromium (51.09% [47]). To further study the correlation between bug types and regression bugs, we calculate the *lift* correlation to determine the bug types that regression bugs are more likely to occur. As shown in TABLE IX, the results indicate that non-regression bugs tend to be MANs.

Implications: For users, it is annoying to encounter regression bugs when upgrading to a new version of TensorFlow. It would make users lose confidence in the new version and refuse to upgrade. It is also a massive task for developers to locate commits that introduce regression bugs from dozens of changes. We recommend developers implement more regression testing before releasing a new TensorFlow version to reduce the occurrence of regression bugs. For regression bugs that have occurred, tools can be used to help programmers locate regression bugs. For example, CodePsychologist, a tool developed in [46] to assist the programmers in locating the lines of code that caused a regression bug.

VII. Threats to Validity

As other empirical studies, our study is naturally subject to limitations. We identify the following threats:

Threats to Construct Validity. This paper focuses on actual bugs, i.e., only bug reports with the label "type:bug" are analyzed. There may be some bug reports but are not labeled as "type:bug" by TensorFlow's maintainers. Besides, we only analyze fixed and closed bug reports since unfixed and unclosed reports may contain incomplete information. Bug types could be very different if unfixed and unclosed reports are considered.

Threats to Internal Validity. For each bug report, we have examined carefully all the related information, involving reporters' descriptions, comments, linked pull requests, and commits. We tried our best to avoid classification mistakes. To mitigate this threat, we use experienced developers to classify the bugs. Moreover, to ensure the consistency of the results obtained by different authors, cross-checks were performed, and conflicting cases were resolved through discussion to reach a consensus among the developers.

Threats to External Validity. Research objects in this paper are collected from TensorFlow's GitHub repository: *tensorflow/tensorflow*, some findings and implications may not hold in other machine learning frameworks or systems. To reduce this threat, we try not to expand the conclusions that apply only to TensorFlow to other machine learning frameworks.

VIII. Related Work

There are several existing studies focus on the empirical research of machine learning systems. These work can be divided into two categories. Some of them performed studies on machine learning applications, that is, clients programmed on top of machine learning frameworks. The others researched machine learning frameworks, i.e., the library or platform being used when building a machine learning model, such as TensorFlow, PyTorch, and Scikit-learn [49].

Zhang et al. [16] present an empirical study on bugs of deep learning applications programmed on top of TensorFlow. They collected 175 TensorFlow coding bugs from GitHub issues and Stack Overflow questions and examined their symptoms and root causes according to Q&A pages, commit messages, pull request messages, and issue discussions. They also studied the strategies deployed by TensorFlow users for bug detection and localization. Islam et al. [17] analyzed the clients of more deep learning frameworks such as Caffe, Keras, Theano, and Torch. They categorize bugs into 11 bug types, ten root causes, and seven impacts. According to their findings, data bug and logic bug are the most severe bug types in deep learning projects. While these works studied machine learning clients, our research object is machine learning framework.

Authors in [20] analyzed 715 questions in Stack Overflow related to three popular deep learning frameworks, including TensorFlow, PyTorch, and Deeplearning4j. The Datasets used in this paper were collected from Stack Overflow, a large-scale Q&A community dedicated to users' individual support. Compared to [20], our dataset comes from the TensorFlow's GitHub repository and have been labeled by developers as *"type:bug"*, which are actual bug information. Besides, their findings focus on the frequency distribution of asked questions in Stack Overflow, whereas our results are mainly relevant to actual bugs in TensorFlow.

In [24], Sun et al. performed a categorization based on the occurring reasons for bugs with 329 bugs classified into seven categories and twelve fix patterns. According to their findings, nearly 70% of machine learning bugs were resolved within a month, and around 40% of machine learning bugs used micro-repair model. Compare to [24], which performed an empirical study on Scikit-learn, Paddle, and Caffe, our research object is TensorFlow, one of the most popular frameworks [16]. Also, instead of analyzing the occurring reasons, we perform our research from the perspective of fault triggering conditions.

Our work differs from the above studies. It is the first empirical study on fault triggering conditions of the actual bugs in machine learning framework. By fault triggers, we mean the set of conditions activating a fault and propagating the resulting error into a failure [50]. To examine fault triggers comprehensively, Grottke and Trivedi [22] developed the definitions of Bohrbug and Mandelbug. There is a further subtype of Mandelbug, which is aging-related bug [51], [52]. An aging-related bug appears when software systems running continuously for a long time and tend to show a degraded performance and increased failure occurrence rate. In [23], Cotroneo et al. performed an extended analysis of triggers. According to fault triggering conditions, Qin et al. [29] and Xiao et al. [28] conducted empirical research on the Android and Linux kernel, respectively.

IX. Conclusion

We present the first empirical study on the fault triggering conditions in the TensorFlow framework. By examining 2,285 bug reports, our analysis aims to answer four research questions: bug type distribution, fixing time, root causes, and regression bugs. It is found that more than two-thirds of bugs in TensorFlow are Bohrbugs. In addition, we have analyzed the distribution of ARB's subtypes and NAM's subtypes, and the evolution of bug types. To better understand the features of Bohrbugs and Mandelbugs, we have also investigated the bug-fixing time and the five root causes in Bohrbugs and Mandelbugs. Through correlation analysis, we have found that a NAM is prone to be an environment/configuration bug or a concurrency bug; an ARB is likely to be a memory bug; and a BOH is likely to be a compatibility bug or a semantic bug. Finally, the characteristics of regression bugs in TensorFlow were extensitively discussed.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Grants 61772055 and 61872169, in part by the Technical Foundation Project of Ministry of Industry and Information Technology of China under Grant JSZL2016601B003, in party by Equipment Preliminary R&D Project of China under Grant 41402020102, in part by the Start-up Fund for New Faculty of NUAA under Grant YAH20026, and in part by the Australian Research Council under Grant DP200101328.

References

[1] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for data mining and machine learning," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[2] E. Menasalvas and C. Gonzalo-Martin, "Challenges of medical text and image processing: Machine learning approaches," in *Machine Learning for Health Informatics*. Springer, 2016, pp. 221–242.

[3] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[4] N. Subrahmanya, P. Xu, A. El-Bakry, C. Reynolds *et al.*, "Advanced machine learning methods for production data pattern recognition," in *SPE Intelligent Energy Conference & Exhibition*. Society of Petroleum Engineers, 2014.

[5] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu, "Exploring topic models in software engineering data analysis: A survey," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2016, pp. 357–362.

[6] L. Wang, X. Sun, J. Wang, Y. Duan, and B. Li, "Construct bug knowledge graph for bug resolution," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*. IEEE, 2017, pp. 189–191.

[7] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[8] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[10] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," 2014.

[11] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, pp. 2007–2, 2007.

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

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[15] R. Alrfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv: Symbolic Computation*, 2016.

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

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

[18] M. Lux and M. Bertini, "Open source column: deep learning with keras," 2019.

[19] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Tech. Rep., 2002.

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

[21] M. Grottke and K. S. Trivedi, "A classification of software faults," *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[22] ——, "Software faults, software aging and software rejuvenation (special survey: New development of software reliability engineering)," *The Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[23] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *2013 IEEE 24Th international symposium on software reliability engineering*. IEEE, 2013, pp. 178–187.

[24] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *2017 24th Asia-Pacific Software Engineering Conference*. IEEE, 2017, pp. 348–357.

[25] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 2017, pp. 413–424.

[26] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[27] G. Xiao, Z. Zheng, B. Yin, K. S. Trivedi, X. Du, and K. Cai, "Experience report: Fault triggers in linux operating system: From evolution perspective," in *2017 IEEE 28Th international symposium on software reliability engineering (ISSRE)*. IEEE, 2017, pp. 101–111.

[28] G. Xiao, Z. Zheng, B. Yin, K. S. Trivedi, X. Du, and K.-Y. Cai, "An empirical study of fault triggers in the linux operating system: An evolutionary perspective," *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1356–1383, 2019.

[29] F. Qin, Z. Zheng, X. Li, Y. Qiao, and K. S. Trivedi, "An empirical investigation of fault triggers in android operating system," in *2017 IEEE 22Nd pacific rim international symposium on dependable computing*. IEEE, 2017, pp. 135–144.

[30] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *2010 IEEE/IFIP international conference on dependable systems & networks*. IEEE, 2010, pp. 447–456.

[31] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[32] I. Goodfellow and N. Papernot, "The challenge of verification and testing of machine learning," *Cleverhans-blog*, 2017.

[33] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*. ACM, 2016, pp. 265–266.

[34] Y. Lei and Y. Sui, "Fast and precise handling of positive weight cycles for field-sensitive pointer analysis," in *International Static Analysis Symposium*. Springer, 2019, pp. 27–47.

[35] M. Barbar, Y. Sui, and S. Chen, "Flow-sensitive type-based heap cloning," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*.

[36] X. Xu, Y. Sui, H. Yan, and J. Xue, "Vfix: value-flow-guided precise program repair for null pointer dereferences," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 512–523.

[37] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 254–264.

[38] ——, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[39] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[40] S. A. Asadollah, H. Hansson, D. Sundmark, and S. Eldh, "Towards classification of concurrency bugs based on observable properties," in *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems*. IEEE, 2015, pp. 41–47.

[41] H. B. Mann, "Nonparametric tests against trend," *Econometrica: Journal of the Econometric Society*, pp. 245–259, 1945.

[42] M. G. Kendall, "Rank correlation methods." 1948.

[43] M. Cortina-Borja, “Handbook of parametric and nonparametric statistical procedures, 5th edn,” *Journal of the Royal Statistical Society*, vol. 175, no. 3, 2012.

[44] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, “Recovery from failures due to mandelbugs in it systems,” in *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2011, pp. 224–233.

[45] Y. Qiao, Z. Zheng, Y. Fang, F. Qin, K. S. Trivedi, and K.-Y. Cai, “Two-level rejuvenation for android smartphones and its optimization,” *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 633–652, 2018.

[46] D. Nir, S. S. Tyszberowicz, and A. Yehudai, “Locating regression bugs,” in *Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007, Proceedings*, 2007, pp. 218–234.

[47] M. Khattar, Y. Lamba, and A. Sureka, “Sarathi: Characterization study on regression bugs and identification of regression bug inducing changes: A case-study on google chromium project,” in *Proceedings of the 8th India Software Engineering Conference*, 2015, pp. 50–59.

[48] G. Xiao, Z. Zheng, B. Jiang, and Y. Sui, “An empirical study of regression bug chains in linux,” *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 558–570, 2020.

[49] A. Swami and R. Jain, “Scikit-learn: Machine learning in python,” *Journal of Machine Learning Research*, vol. 12, no. 10, pp. 2825–2830, 2012.

[50] Russo, Stefano, Cotroneo, Domenico, Pietrantuono, Roberto, Trivedi, and Kishor, “How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation,” *Journal of Systems & Software*, vol. 113, pp. 27–43, 2016.

[51] M. Grottke and K. S. Trivedi, “Fighting bugs: Remove, retry, replicate, and rejuvenate,” *Computer*, vol. 40, pp. p.107–109, 2007.

[52] M. Grottke, R. Matias, and K. S. Trivedi, “The fundamentals of software aging,” in *2008 IEEE International Conference on Software Reliability Engineering Workshops*, 2009, pp. 1–6.