An Empirical Study of Fault Triggers in Deep Learning Frameworks

Xiaoting Du, Yulei Sui, Zhihao Liu, and Jun Ai

Abstract—Deep learning frameworks play a key rule to bridge the gap between deep learning theory and practice. With the growing of safety- and security-critical applications built upon deep learning frameworks, their reliability is becoming increasingly important. To ensure the reliability of these frameworks, several efforts have been taken to study the causes and symptoms of bugs in deep learning frameworks, however, relatively little progress has been made in investigating the fault triggering conditions of those bugs. This paper presents the first comprehensive empirical study on fault triggering conditions in three widely-used deep learning frameworks (i.e., TensorFlow, MXNET and PaddlePaddle). We have collected 3,555 bug reports from GitHub repositories of these frameworks. A bug classification is performed based on fault triggering conditions, followed by the analysis of frequency distribution of different bug types and the evolution features. The correlations between bug types and fixing time are investigated. Moreover, we have also studied the root causes of Bohrbugs and Mandelbugs and investigated the important consequences of each bug type. Finally, the analysis of regression bugs in deep learning frameworks is conducted. We have revealed 12 important findings based on our empirical results and have provided 10 implications for developers and users.

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

1 Introduction

In recent years, deep learning has rapidly developed and achieved tremendous success in many domains, such as face recognition [1], image analysis [2], natural language processing [3] and many other fields [4], [5], [6]. Due to the rapid proliferation of deep learning systems, even a single bug with few lines of error code may cause disastrous consequences [7]. In fact, there are reports of real-world accidents caused by deep learning systems. For example, a bug in the Uber autonomous driving system cause the death of a pedestrian [8]. Deep learning frameworks, as the basis of constructing deep learning systems, bugs in them can adversely affect a larger number of users and cause more serious results than a specific deep learning model. Especially when they are applied to safety- and security-critical applications, such as autonomous driving [9] and healthcare [10]. It is critical to ensure the reliability of deep learning frameworks.

Same as traditional software systems, there are bugs in deep learning frameworks. To understand the features and characteristics of bugs in deep learning frameworks and help to fix these bugs, several studies have been conducted. For example, Zhang et al. [11] present an empirical study on detecting and locating programming mistakes in applications built on top of TensorFlow. The authors collected 175 TensorFlow coding bugs from GitHub issues and StackOverflow questions and reported their symptoms and causes, as well as the challenges on bug detection and localization. In [12], Li et al. analyzed bugs inside deep learning framework TensorFlow. They also studied the causes and symptoms of bugs, and the distribution of bugs among different components. In [13], Islam et al. performed a study of five deep learning libraries, including Caffe [14], Keras [15], TensorFlow [16], Theano [17] and Torch [18]. 2716 posts from Stack Overflow and 500 bug fix commits from GitHub were analyzed to explore the causes and effects of bugs. Although the above efforts have studied the causes and symptoms of bugs inside existing deep learning frameworks, none of them analyzed the factors that trigger a fault and/or propagate a fault into a failure in deep learning frameworks. These fault triggering conditions are important for both development and maintenance of deep learning frameworks.

Fault triggering conditions are usually complex, involving not only the timing of inputs and operations but also the interaction with other systems. According to the complexity of fault activation and/or error propagation conditions, Grottke and Trivedi [19] divided bugs into Bohrbugs (BOHs) and Mandelbugs (MANs). Among them, BOH is a kind of bug whose activation and error propagation conditions are simple. In addition, BOHs are easy to reproduce and isolate. In contrast, the activation and error propagation of MANs are complex. They make the system exhibit chaotic and even non-deterministic behavior during operation. In addition, a MAN can be further classified as a non-aging related Mandelbug (NAM) or an aging-related bug (ARB).

- *This work was supported by the National Natural Science Foundation of China under Grants 61772055 and 61872169. (Corresponding author: Jun Ai.)*

- *Xiaoting Du and Zhihao Liu are with the State Key Laboratory of Software Development Environment, and the School of Automation Science and Electrical Engineering, Beihang University, Beijing, China. E-mail: xiaoting_2015@buaa.edu.cn; lzh123698745@buaa.edu.cn*
- *Yulei Sui is with the Australian Artificial Intelligence Institute, University of Technology Sydney, Sydney, Australia. E-mail: yulei.sui@uts.edu.au*
- *Jun Ai is with the School of Reliability and Systems Engineering, and Science & Technology on Reliability and Environmental Engineering Laboratory, BeiHang University, Beijing China. E-mail: aijun@buaa.edu.cn*

Among them, ARB is a kind of bug that can cause software aging, that is, the cause of an increase in failure rate and/or performance degradation [20]. According to the above classification, Cotroneo et al. [21] proposed a more detailed classification of ARBs and NAMs.

In this study, we make the first attempt to explore the bug characteristics in deep learning frameworks based on fault triggering conditions. We have conducted an extensive study on three widely-used deep learning frameworks, i.e., TensorFlow [22], MXNET [23] and PaddlePaddle [24], where 3,555 bug reports are analyzed. We have investigated bug characteristics from several aspects, including (1) the frequency distribution of different types of bugs, and the evolution of different bug types over time; (2) the fixing time of bugs and the correlation between bug types and fixing time; (3) the root causes of Bohrbugs and Mandelbugs; (4) the impacts of Bohrbugs and Mandelbugs; and (5) different features of regression bugs in deep learning frameworks. For each report, we have manually examined the bug descriptions, comments, linked pull requests and the corresponding commits. The contributions of our work provide answers to the following five research questions.

RQ1: What is the distribution of different bug types in deep learning frameworks?

To answer this question, bugs in TensorFlow, MXNET and PaddlePaddle are classified based on their fault triggers. Bug type distribution in these deep learning frameworks are analyzed. In addition, we have studied the proportions of the bug types in these frameworks and how these bug types evolve over time. This RQ is answered in Section 3.

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

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. This RQ is answered in Section 4.

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

We have classified bugs into Bohrbugs and Mandelbugs in RQ1. The root causes of Bohrbugs and Mandelbugs are further investigated in this RQ to understand the nature of bugs. Five root causes are identified, including environment/configuration, memory, compatibility, concurrency and semantic. This RQ is answered in Section 5.

RQ4: What are the impacts of Bohrbugs and Mandelbugs?

We have studied the distribution of impacts and the correlation between impacts and bug types. Identifying the impacts of bugs helps us to understand how severe a Bohrbug or a Mandelbug is. Through manual examination, failures caused by Bohrbugs and Mandelbugs include crash/exception, hang/no response, wrong output, operation failure and warning style error. This RQ is answered in Section 6.

RQ5: What is the feature of regression bugs in deep learning frameworks?

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 deep learning frameworks could help developers with their debugging and program repair, thus preventing future regression bugs. This RQ is answered in Section 7.

The contributions of this paper are summarized into 12 findings, as shown in TABLE 1. The detailed implications of these findings are illustrated in relevant sections of this paper. These results provide valuable insight for deep learning developers and users.

This paper expands and improves our previous work [25]. Several new analyses are conducted. For example, (1) two other deep learning frameworks (i.e., MXNET and PaddlePaddle) are analyzed in this paper; (2) we study the impact of each bug to understand the severity of Bohrbugs and Mandelbugs in deep learning frameworks; (3) we perform the analysis of MXNET and PaddlePaddle from five dimensions, including distribution and evolution of different bug types, bug fixing time, root causes, impacts and regression bugs, and compare the results between different frameworks.

The rest of the paper is structured as follows. Section 2 presents the study methodology utilized in this paper. We give the answers for the five research questions in Sections 3-7. Section 8 reports the threats to validity of our study. Section 9 discusses some implications based on the results obtained. Section 10 describes related work. Finally, conclusions and future work are given in Section 11.

2 Study Methodology

This section describes our methodology of studying bugs in deep learning frameworks, including data source, bug classification based on fault triggering conditions, bug report classification procedure, definitions of root causes and impacts of Bohrbugs and Mandelbugs, and the metric we used to analyze the correlation between different bug types.

2.1 Data Source

To determine the target deep learning frameworks for our study, we have investigated 14 popular deep learning frameworks, which have been widely used in recent years [13], [26], [27], [28], [29], including TensorFlow, MXNET, PaddlePaddle, Keras, Pytorch, Caffe, CNTK, Torch, Deeplearning4j, Caffe2, Sonnet and Chainer. First, we ranked all 14 frameworks based on the number of stars marked in their GitHub repositories, and 7 of them were starred by more than 15k GitHub users. They are TensorFlow, MXNET, PaddlePaddle, Keras, PyTorch, Caffe and CNTK. Second, we searched for bug reports under the "closed" status and with the label "type:bug". Bug reports in GitHub have two statuses, namely, "closed" and "open". Considering that the reports under the "open" state are still under discussion and their types cannot be determined, only "closed" bug reports will be studied in this work. Furthermore, to narrow down the scope of bug reports to actual bugs, we used the "type:bug" label to filter out the closed bug reports, which are added by developers during the process of fixing these bugs. However, in the GitHub repositories of Keras and PyTorch, there is no "type:bug" label. Finally, we considered whether the number of bug reports obtained in each framework was sufficient. In Caffe and CNTK's repositories, only a few dozen closed bug reports were labeled as "type:bug",

TABLE 1
Summary of Findings Related to Bug Characteristics in Deep Learning Frameworks

Findings on bug types	
#1	The proportions of actual bugs in TensorFlow, MXNET and PaddlePaddle are 41.7%, 45.5% and 50.6%, respectively.
#2	More than two-thirds of actual bugs are BOHs in all the three deep learning frameworks (i.e., TensorFlow, MXNET and PaddlePaddle).
#3	MEM is the major subtype of ARB. The proportions of MEMs in TensorFlow, MXNET and PaddlePaddle are 84.1%, 50% and 52.6%, respectively.
#4	In MXNET and PaddlePaddle, the major subtype of NAM is ENV, the proportions are 65% and 60.7%, respectively. In TensorFlow, TIM is the major subtype of NAM, accounting for 46.4%.
#5	The proportions of BOHs in MXNET and PaddlePaddle tend to grow over time, while the proportion of MANs tends to decrease. In TensorFlow, the proportion of BOHs tends to shrink in the first year around, and then increases slowly afterwards. On the contrary, the ratio of MANs in TensorFlow starts to increase in a period of time and then goes down slowly.
Findings on fixing time	
#6	It takes more time to close an invalid bug report than fixing an actual bug.
#7	It takes more time to fix a Mandelbug (MAN) than fix a Bohrbug (BOH).
Findings on root causes	
#8	The major root cause of BOHs is semantic bugs and the primary root cause of ARBs is memory bugs.
#9	A BOH is prone to be caused by a semantic bug or a compatibility bug; an ARB is likely caused by a memory bug; and a NAM is more likely caused by an environment/configuration bug or a concurrency bug.
Findings on impacts	
#10	More than half of BOHs and MANs would result in crashes/exceptions.
Findings on Regression Bugs	
#11	The proportions of regression bugs in TensorFlow, MXNET and PaddlePaddle are 9.02%, 13.55% and 4.81%, respectively.
#12	Among all regression bugs in TensorFlow, MXNET and PaddlePaddle, BOHs account for 78%, 77% and 80%, respectively, and MANs account for 15%, 23% and 10%, respectively.

TABLE 2
Details of Data Set

Project	Time frame	# of reports
TensorFlow	Nov. 26, 2015-Nov. 26, 2019	2,285
MXNET	Sep. 12, 2015-Sep. 12, 2020	859
PaddlePaddle	Aug. 31, 2016-Aug. 31, 2020	411
Total	3,555	

and the conclusions reached based on these data may not be statistically significant. Finally, TensorFlow, MXNET and PaddlePaddle were obtained.

Among them, TensorFlow is the most popular deep learning framework. It is developed by Google and can be used to support a variety of algorithms and has been adopted to build more than 36,000 applications hosted on Github [22]. MXNET is a well-known deep learning framework and the choice of Amazon Web Services. It blends declarative symbolic expression with imperative tensor computation. MXNET is computation and memory efficient and runs on various heterogeneous systems [23]. PaddlePaddle is developed by Baidu and has unique features in supporting distributed training with ultra-large data and fast inference on the server, mobile as well as edges [24].

We collected bug reports from the GitHub repositories of Tensorflow[1], MXNET[2] and PaddlePaddle[3]. After careful examination, 3,555 bug reports are obtained, and the detailed information is shown in TABLE 2. We obtain relevant bug information through the designed Web-Crawler, including the reporters' description, opened time, closed time and comments. Note that a bug can be fixed by one or more pull requests with multiple code commits. For further analysis, we also consider all these related pull requests and commits.

2.2 Bug Classification based on Fault Triggering Conditions

Based on the fault activation and error propagation conditions, bugs are classified into Bohrbugs (BOHs) and Mandelbugs (MANs) in [21]. In order to study the fault triggering conditions in deep learning frameworks, we adopted the bug classification method in [21]. The definitions of BOH and MAN are as follows:

1. https://github.com/tensorflow/tensorflow/issues
2. https://github.com/apache/incubator-mxnet/issues
3. https://github.com/PaddlePaddle/Paddle/issues

TABLE 3
Examples of Bohrbugs (BOHs) and Mandelbugs (MANs)

Project	Bug ID	Bug Type	Description
TensorFlow	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...”
MXNET	10438	LAG/NAM	“Loading an older model with a custom operator from Java/Scala more than twice causes MXNet to crash”
PaddlePaddle	3073	ENV/NAM	“Incorrect Inference.infer results when running with multiple GPUs.”
TensorFlow	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)...”
PaddlePaddle	2141	SEQ/NAM	“Swapping the order of the inputs of layer.fc will cause error.”
MXNET	10436	MEM/ARB	“FeedForward.scala and NDArrayIter.scala leak memory by not disposing of NDArrays”
TensorFlow	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...”
MXNET	11403	NUM/ARB	“When .norm() method is called on a float16 array it tries to compute the norm in fp16, which causes the squared sum to go out of range and causes nan as output.”
PaddlePaddle	12343	LOG/ARB	“Fix generator not closed when iteration end bug”
TensorFlow	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...”

- **Bohrbug:** a kind of bug which can always be reproduced given a well-defined set of conditions because its activation and/or error propagation are simple.
- **Mandelbug:** a kind of bug whose activation and/or error propagation are complex and can not always be reproduced even under the same condition. The complexity may caused by the following reasons: direct factors related to a time lag between the bug activation and the manifestation of the failure; indirect factors, for example, the interactions between a software application and its internal environment; the timing of inputs and operations; or the relative order of inputs and operations.

There is a special subtype of MAN, i.e., aging-related bug (ARB). It is a kind of bug that can cause an increasing failure rate and/or degraded performance. As a result, a MAN is either an aging-related bug (ARB) or a non-aging related Mandelbug (NAM). Based on the different kinds of complexity in fault triggering conditions, NAMs can be categorized into 4 subtypes. The definitions of NAM’s subtypes are listed as follows.

- **LAG:** a time lag exists between the bug activation and the occurrence of failure;
- **ENV:** The interaction of the software application with its system-internal environment may influence the activation of the bug and/or error propagation;
- **TIM:** The timing of inputs and operations is the factor that impact the fault activation and/or error propagation;
- **SEQ:** The sequence of inputs and operations has impacts on the fault activation and/or the propagation of error.

According to the underlying reasons for the software aging phenomenon, aging-related bugs are futher classified into 5 subtypes. The definitions of ARB’s subtypes are listed below:

- **MEM:** a kind of ARB caused by the accumulation of errors because of improper memory management, such as memory leaks, buffers not being flushed;
- **STO:** a kind of ARB caused by the accumulation of errors caused by improper storage space management, for example, disk space consumed by bugs;
- **LOG:** a kind of ARB caused by the leak of other logical resources, such as inodes or sockets that are not released after use;
- **NUM:** a kind of ARB caused by the accumulation of numerical errors, such as integer overflows and round-off errors;
- **TOT:** a kind of ARB. When the total running time of the system increases, its fault activation and/or error propagation rate will increase, but this type of bugs is not caused by the accumulation of internal error states.

2.3 Bug Report Classification Procedure

In this section, we perform classification of bug reports collected from TensorFlow, MXNET and PaddlePaddle. First of all, we extract the actual bugs from all the 3,555 bug reports collected. Then, we classify actual bugs based on the classification method introduced in Section 2.2, the detailed procedure are shown in Fig. 1. The classification is implemented manually by two authors who are familiar with developing deep learning projects. When encounter suspicious classified cases, a cross-check will be taken. We

TABLE 4
Examples of Root Causes

Project	Bug ID	Root Cause	Description
TensorFlow	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..."
MXNET	10341	Concurrency	"Deadlock during ThreadedEnginePerDevice destructor after CuDNNConvolutionOp<float>::SelectAlgo called."
PaddlePaddle	3537	Memory	"tensor didn't free memory at the end of the program"
TensorFlow	6171	Compatibility	"tfprof: Python3 incompatibility: ...This line in tfprof_logger.py uses dict.iteritems(), which breaks my Python 3 code..."
MXNET	9363	Semantic	"Incorrect weight_decay implementation in AdaGrad"

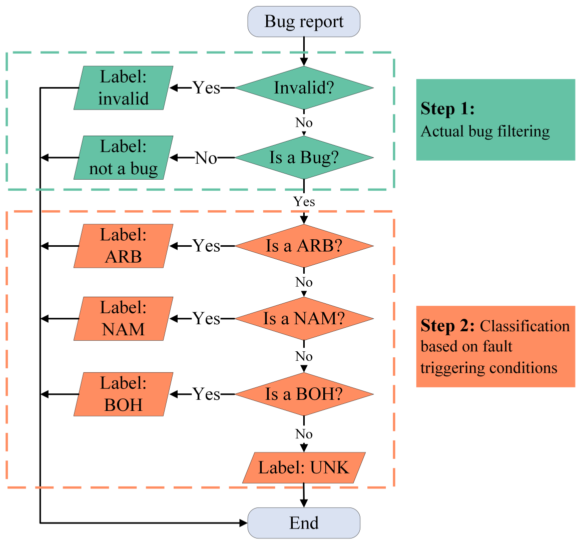


Fig. 1. Process of bug report filtering.

carefully check the information provided by reporters as well as the discussions between users and developers. In addition, we also consider the pull requests and commits provided in these reports for bug fixing. We have released our dataset online[4]. To clarify the classification, TABLE 3 shows some examples of BOHs and MANs, and their partial descriptions.

Step 1: Actual bug filtering. Even if we select bug reports with label "type:bug" in Section 2.1, there are still many invalid bug reports and reports that do not contain actual bugs (i.e., non-bugs). For invalid reports, we mean bug reports that contain too little information to determine if they are bugs or not. Furthermore, we also consider reports as non-bugs if they are related to 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 or duplicate reports.

Step 2: Classification based on fault triggering conditions.

We carefully check each bug report to find the activation conditions of each bug, including the operations or inputs before the bug is triggered; how the bug is propagated, for example, whether any parameter or state of the program is changed due to the bug, and how the changed parameters or states are propagated; what scene does the user observe when the failure occurres.

Based on the definition and characteristic of each bug type, we check whether a bug belongs to ARB, NAM, or BOH. For the classification of ARB subtypes, if a bug is classified as an ARB, but there is not enough information to determine which subtype it belongs to, it will be labeled as ARU. Similarly, a NAM will be labeled as NAU if there is not enough information to determine its subtype. Finally, if there is an actual bug, but there is insufficient information to classify it as one of ARB, NAM, or BOH, it will be labeled as UNK.

4. https://github.com/xiaotingdu/DLFrameworkFaultTriggers

2.4 Root Causes of Bohrbugs and Mandelbugs

After dividing bugs into BOHs, ARBs and NAMs, we aim to find the root cause of each bug. Through manually examining each bug report in TensorFlow, MXNET and PaddlePaddle, we summarize five root causes of Bohrbugs and Mandelbugs, including environment/configuration, memory, compatibility, concurrency and semantic. The examples of different root causes are listed in TABLE 4. Referring to the definitions in [26] and [30], we define the following five root causes we identified.

- **Environment/configuration:** There are errors in dependent libraries, underlying operating systems or non-codes that can adversely affect a system's functionality;
- **Concurrency:** In concurrent programs, there are synchronization problems with concurrent threads or processes;
- **Memory:** These errors are caused by incorrect handling of memory objects;
- **Compatibility:** The program cannot run normally on a specified CPU architecture, operating system, or web browser, etc.;
- **Semantic:** Inconsistent with the requirements or the programmers' intention, and do not belong to the above categories.

TABLE 5
Examples of Impacts

Project	Bug ID	Impact	Description
TensorFlow	5688	Crash/exception	"TensorFlow crashes when using large image with 3d convolutional network"
MXNET	9171	Hang/no response	"Using FusedRNNCell with its "bidirectional" flag turned True, can lead to hanging (i.e. infinite pause without progress/error/crash) of training run."
PaddlePaddle	8315	Operation failure	"After received enough batch barrier, listen_and_serv will execute listen_and_serv once again."
TensorFlow	2619	Wrong output	"tf.image.decode_png returns wrong values for uint16 images"
MXNET	10436	Warning style error	"FeedForward.scala and NDArrayIter.scala leak memory by not disposing of NDArrays. We see the below leak warnings occur running on MXNet 1.1.0."

2.5 Impacts of Bohrbugs and Mandelbugs

Impacts reflect the severe consequences caused by bugs [31], [32]. In this section, we investigate the impact of each bug to understand the severity of Bohrbugs and Mandelbugs. Through examination, five impacts are summarized, including crash/exception, hang/no response, operation failure, wrong output and warning style error. TABLE 5 presents some examples of different impacts. Note that, sometimes one bug may have multiple impact categories. For example, a bug may cause a crash and a operation failure at the same time. To avoid multiple counting, we give these impacts an order as shown in list below:

- **Crash/exception:** When the program stops and exits unexpectedly, a crash/exception occurs. When this situation happens, the program typically throws an error message.
- **Hang/no response:** When the program keeps running but has no response, a hang/no response occurs.
- **Operation failure:** Unexpected behaviors, such as build failure, incomplete processing or rejection of tasks, multiple processing of tasks and others.
- **Wrong output:** When the program generates a wrong result and presents it to users, a wrong output occurs.
- **Warning style error:** A warning style error indicating that the running of a program will not be disturbed, but the error still needs to be eliminated to improve code quality. Warning messages are usually displayed in this category. In addition, performance degradation and non-release of resources are considered as warning style errors.

2.6 Correlation Metric

A statistical metric $lift$ [31], [33] is used in this paper to indicate the correlation between two types of bugs. For instance, $lift(A_i, B_j)$ represents the $lift$ value of type A_i and type B_j. The formula to calculate $lift(A_i, B_j)$ is $P(A_iB_j)/(P(A_i) * P(B_j))$, in which $P(A_i)$ and $P(B_j)$ are the probability of A_i and B_j, respectively. And $P(A_iB_j)$ is the probability that a bug belongs to both category A_i and B_j. Take the correlation between BOHs and semantic bugs as an example. Suppose there are 100 actual bugs, 80 of them are classified as BOHs, 70 of them are semantic bugs, and the number of BOHs caused by semantic bugs is 60. The $lift$ correlation between BOHs and semantic bugs can be calculated as $lift(A_i, B_j) = P(A_iB_j)/(P(A_i) * P(B_j)) = (60/100)/((70/100) * (80/100)) = 1.07$. In the formula, A_i presents semantic bugs and B_j presents BOHs.

After obtaining the value of the $lift$ correlation, we can analyze the relationship between two bug types. If $lift(A_i, B_j)$ is equal to 1, it means that there is no correlation between types A_i and B_j. If $lift(A_i, B_j)$ is larger than 1, it means that categories A_i and B_j are positively correlated, i.e., a bug in type A_i is prone to belong to type B_j. In contrast, if $lift(A_i, B_j)$ is less than 1, it means that types A_i and B_j are negatively correlated, i.e., a bug in type A_i is unlikely to be of type B_j. In the above example, $lift(A_i, B_j)$ is 1.07, which means that BOHs are more likely caused by semantic bugs.

3 Bug Classification

This section aims to answer RQ1. After examining and extracting the bug reports from GitHub repositories, as described in Section 2.1, we obtain 3,555 bug reports from TensorFlow, MXNET and PaddlePaddle. In this section, bugs are classified based on fault triggering conditions, and the distribution of bugs and the proportion evolution of BOHS and MANs are investigated.

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

First of all, we classify all bug reports we collected into actual bugs, non-bugs and invalid bug reports, the distribution results are shown in Fig. 2.

Finding #1: *The proportions of actual bugs in TensorFlow, MXNET and PaddlePaddle are 41.7%, 45.5% and 50.6%, respectively.*

Before classifying bug reports based on fault triggering conditions, we first filter out invalid reports and non-bugs from all the reports. It should be noted that a report is labeled as invalid if it contains too little information to determine whether it is an actual bug or not. Non-bugs are those reports related to (1) the requests of features or enhancements, (2) the descriptions of compile-times issues or documentation issues, or (3) duplicated reports.

After manual examination, we get the results in Fig. 2. The results indicate that almost half of reports do not contain actual bugs even though they are labeled as "type:bug"

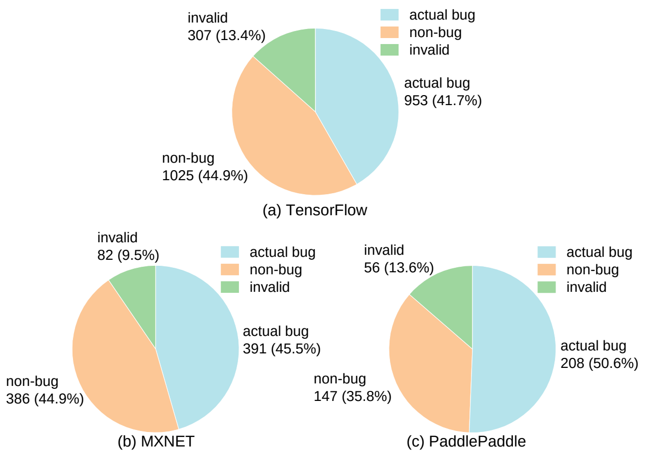


Fig. 2. Numbers and percentages of actual bugs, non-bugs and invalid bug reports.

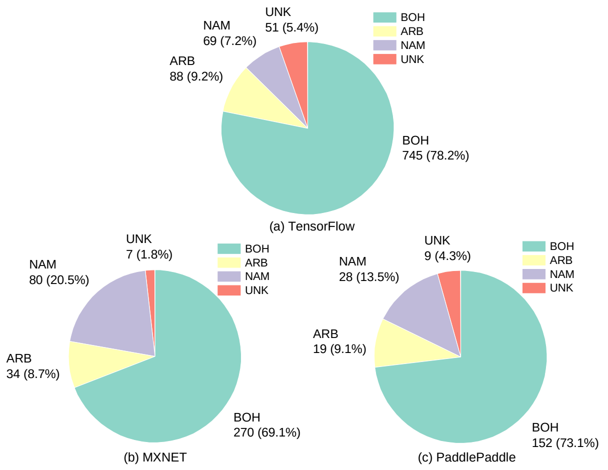


Fig. 3. Numbers and proportions of Bohbugs (BOHs), aging-related bugs (ARBs) and non-aging related Mandelbugs (NAMs).

by developers. According to our examination, there are several typical situations of invalid reports: (1) a report is submitted but no response is given for months even years, usually this kind of bug reports would be closed for the reason of inactivity; (2) a bug cannot be reproduced based on the information provided by reporter. Sometimes, reporters do not provide the complete reproduce information needed to reproduce a bug, such as code, dataset and specific running environment. As a result, developers have to close the bug report hence it cannot be reproduced; (3) a bug corresponding to deprecated features or out of the scope of deep learning frameworks we analyzed is labeled as invalid. The most common situations of non-bugs are build and link errors. For example, Bug ID-13918 in TensorFlow, a user tried to build TensorFlow for GPU but failed. Then, there are many feature requests and enhancements. For example, in Bug ID-30642 in TensorFlow, the reporter says "I hope tf.scatter_nd_update support string ref, and I really need this feature in my project", a feature is request in this report. In addition, there are other situations, such as document-relevant issues, performance issues and duplicate bug reports.

Implications: This finding indicates that amounts of non-bugs and invalid reports are submitted, which is a heavy burden for developers. To help developers deal with invalid reports and non-bugs, on the one hand, methods could be introduced to detect invalid bug reports to save developers' time [34]. On the other hand, tools could be integrated to detect duplicate bug reports. For example, a just-in-time duplicate detection method was proposed in [35], which can prevent duplicate bug reports before they are submitted by continuously querying. It not only helps users find solutions to their questions effectively, but also reduces workloads of developers.

3.2 Distribution of BOHs, ARBs and NAMs among All the Actual bugs

In this section, we classify actual bugs into Bohrbugs (BOHs), aging-related bugs (ARBs) and non-aging related Mandelbugs (NAMs). If there is not enough information to determine whether a bug is of type BOH, ARB, or NAM, then we label it as UNK.

Finding #2: *More than two-thirds of actual bugs are BOHs in all the three deep learning frameworks (i.e., TensorFlow, MXNET and PaddlePaddle).*

Fig. 3 illustrates the distribution of BOHs, ARBs, NAMs and UNKs. It can be observed that the percentage of BOHs is the highest in all three deep learning frameworks. Compared with traditional software projects, for example, Linux kernel (i.e., 55.82% [36]), Android (i.e., 65.2% [37]) and MySQL (i.e., 56.6% [21]), the proportions of BOHs in deep learning frameworks are higher. The proportions of ARBs in TensorFlow, MXNET and PaddlePaddle are similar, which are 9.2%, 8.7% and 9.1%, respectively. As for NAMs, the percentage of NAMs in MXNET is the highest among all the three frameworks, which is 20.5%, followed by PaddlePaddle (i.e., 13.5%) and TensorFlow (i.e., 7.2%). Compared to traditional software systems, the proportions of MANs, including ARBs and NAMs, in deep learning frameworks (i.e., 16.4% in TensorFlow, 29.2% in MXNET and 22.6% in PaddlePaddle) are lower than that of Linux kernel (i.e., 36.34% [36]), Android OS (i.e., 31.4% [37]), MySQL (i.e., 38% [21]) and space mission on-board software (i.e., 36.5% [38]).

One of the reasons why the proportions of BOHs in deep learning frameworks are higher than that of traditional software projects is that the testing of deep learning frameworks is more challenging [39] considering that it is difficult for developers to know the expected output of a given instance. Another reason is that the rapid development of deep learning has led to the continuous addition of a large number of new features, which introduces more BOHs at the same time. In addition, the fault activation and/or error propagation conditions of MANs are complicated, making MANs hard to discover. This is also one of the reasons for the low percentage of MANs.

Implication: Since BOHs account for more than two-thirds of the bugs we studied, mitigating BOHs should be the focus of work. Considering that debugging/testing software is the classic way to deal with BOHs, we suggest developers conducting sufficient testing before releasing

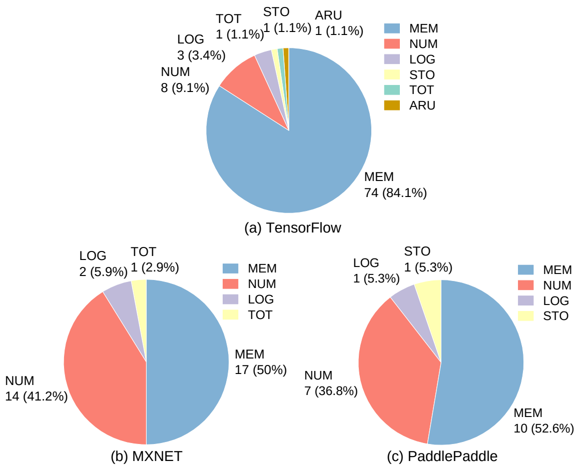


Fig. 4. Subtype distribution of aging-related bug (ARB).

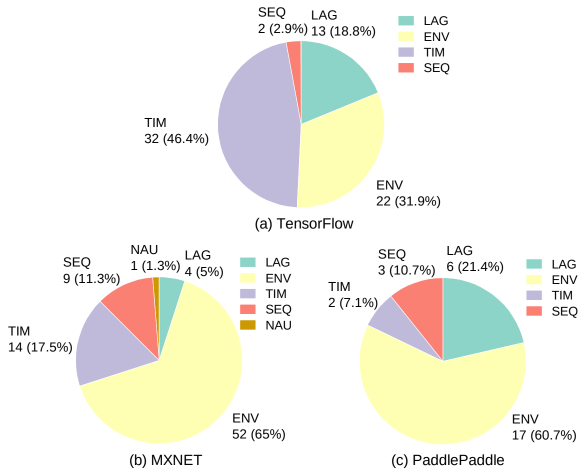


Fig. 5. Subtype distribution of non-aging related Mandelbug (NAM).

a version. For example, static program analysis [40], [41] could be used to detect bugs in code. As for MANs, a non-negligible fraction exists, software rejuvenation [42] is a proactive technique that can clean the internal state of the system and reset the system runtime, therefore reducing the failure rate and improving performance.

3.3 Subtype Distribution of Aging-related Bugs (ARBs)

Finding #3: *MEM is the major subtype of ARB. The proportions of MEMs in TensorFlow, MXNET and PaddlePaddle are 84.1%, 50% and 52.6%, respectively.*

Fig. 4 shows the numbers and percentages of ARB's subtypes, including MEM, NUM, LOG, STO and TOT. If there is not enough information to determine which subtype an ARB belongs to, it will be labeled as ARU. From the results, we can see that in all the three deep learning frameworks we analyzed, the proportions of MEMs have an absolute advantage in ARBs. In TensorFlow, the proportion of MEMs is 84.1%. Compared with traditional software projects, it is higher than that of Linux kernel (i.e., 68.78% [36]) and Android (i.e., 76.2% [37]). In MXNET and PaddlePaddle, the proportions of MEMs are 50% and 52.6%, respectively, which are lower than those of Linux kernel and Android. MEM is a critical bug type in deep learning frameworks. It is because memory is one of the biggest challenges when developing deep learning models. Deep learning frameworks are often used to process millions of images or neural networks with very deep layers, all of these tasks consume large amounts of storage and rely heavily on memory devices [43]. For example, a typical MEM was reported in Bug ID-14181 in TensorFlow "...with the increasing time the whole process starts consuming more and more RAM although it should clean it up...". As time increases, the process consumes more and more RAM instead of cleaning it up.

Implication: To deal with MEMs and mitigate the use of memory, we suggest that developers (1) work on improvement of memory management, such as garbage collection [44]; (2) optimize memory efficiency, for example, ZeRO [45], a solution proposed towards training trillion parameter models, which can optimize memory and vastly improve training speed; (3) to deal with memory leak bugs, memory monitoring tools and memory detectors could be used [46], [47]. For example, a static bug detector [48], which could be used to detect memory leaks in C programs.

3.4 Subtype Distribution of Non-aging Related Mandelbugs (NAMs)

Finding #4: *In MXNET and PaddlePaddle, the major subtype of NAM is ENV, the proportions are 65% and 60.7%, respectively. In TensorFlow, TIM is the major subtype of NAM, accounting for 46.4%.*

Fig. 5 illustrates the distribution of NAM's subtypes. It can be observed that ENVs account for more than half of NAMs in both MXNET and PaddlePaddle. Compared with traditional software projects, the proportions of ENV in MXNET and PaddlePaddle are higher than those of Linux kernel (i.e., 36.51% [36]) and Andrioid (i.e., 55.0% [37]). TIM is the major subtype of NAM in TensorFlow, accounting for 46.4%, higher than that of Linux kernel (i.e., 37.23%) and Android (i.e., 1.6%). The reason why ENV and TIM are two major subtypes of NAM can be explained by features of deep learning frameworks. First, the complexity and diversity of the operating environment lead to a high proportion of ENVs. Deep learning frameworks support a variety of platforms, from mobile devices such as phones to large scale training systems running on hundreds of machines and thousands of computational devices (such as GPU cards). In addition, deep learning frameworks allow clients to express various kinds of parallelism through replication and parallel execution, which are highly prone to containing concurrency bugs [49], such as deadlocks. A deadlock occurs when two or more threads attempt to access shared resources owned by another thread and neither is willing to give it up [50]. The occurrence of deadlock is high-related to the timing of operation. For example, a deadlock occurs in Bug ID-932 in TensorFlow: "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...".

Implication: To deal with NAMs in deep learning frameworks, we recommend that developers pay more atten-

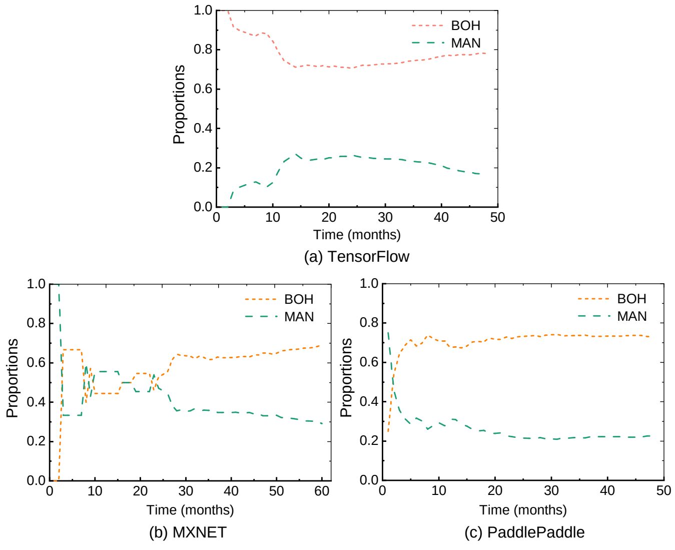


Fig. 6. The evolution of proportions of Bohrbugs (BOHs) and Mandelbugs (MANs) among all the valid bug reports.

tion to TIMs and ENVs. On the one hand, concurrency bug detectors could be used to improve the reliability of frameworks. For example, programmer RaceFuzzer and DeadlockFuzzer [51] could be used to detect data races and deadlocks in concurrent programs. On the other hand, since the environment is unpredictable and uncontrollable, fault-tolerance strategies, such as recovery [52], could be used to deal with ENVs.

3.5 Evolution of Proportions of Bohrbugs (BOHs) and Mandelbugs (MANs) over Time

In the following, we analyze the evolution trend of proportions of BOHs and MANs over time, as shown in Fig. 6.

Finding #5: *The proportions of BOHs in MXNET and PaddlePaddle tend to grow over time, while the proportion of MANs tends to decrease. In TensorFlow, the proportion of BOHs tends to shrink in the first year around, and then increases slowly afterwards. On the contrary, the ratio of MANs in TensorFlow starts to increase in a period of time and then goes down slowly.*

Fig. 6 shows the evolution of the ratios of BOHs and MANs. In the beginning, as time goes on, the proportions of BOHs in MXNET and PaddlePaddle increase rapidly. However, in TensorFlow, the proportion of BOHs has a significant trend for decreasing. As the development of deep learning frameworks becomes more mature, the proportions of BOHs appear to increase slowly in all three frameworks. Compared with BOHs, the proportions of MANs in MXNET and PaddlePaddle drop significantly at the beginning of time while it tends to increase in TensorFlow. After a period of time, the proportions of MANs in these three frameworks eventually decline slowly. It should be noted that the evolution trends of all the proportions mentioned above are tested through the Mann-Kendall test [53] and results are presented in TABLE 6. It can be seen that for a significance level of $alpha = 0.05$, the above conclusions are statistically significant. The evolution trends of BOHs and MANs can be explained as follows. With the dramatic development of deep learning frameworks, their functions keep increasing and expanding, which may introduce more BOHs into newly released frameworks. As a result, the percentage of BOHs gradually increases, which in turn causes the percentage of MANs to decrease accordingly. After our further investigation, we find that the increase of MANs in TensorFlow at the very beginning may be related to the development history of TensorFlow. In April 2016 and June 2016, TensorFlow starts to support distributed operations and multi-platforms, respectively. This leads to an increase in the MANs' ratio and causes a decrease in the BOHs' proportion.

TABLE 6
Results of Mann-Kendall Trend Detection for Fig. 6

Project	Time frame	Type	p value	Trend
TensorFlow	Nov.26 2015 - Nov.26 2017	BOH	<0.001	decreasing
		MAN	<0.001	increasing
	Nov.27 2017 - Nov.26 2019	BOH	<0.001	increasing
		MAN	<0.001	decreasing
MXNET	Sep.11 2015 - Sep.11 2020	BOH	<0.001	increasing
		MAN	<0.001	decreasing
PaddlePaddle	Aug.31 2016 - Aug.31 2020	BOH	<0.001	increasing
		MAN	<0.001	decreasing

Implication: Even if BOHs are easy to reproduce and debug, there are still a large number of BOHs in deep learning frameworks. The proportion of BOHs continues to increase as the framework evolves. This is due to that deep learning frameworks are hard to test or ineffective in testing activities compared to traditional software systems. We recommend that developers adopt specific strategies developed for deep learning framework testing. For example, CRADLE, a method developed to detect and localize bugs in deep learning libraries [54].

4 Fixing Time of Bugs

To answer the RQ2, we analyze the relationship between bug types and bug fixing time in this section. In a bug report, the time of submitting and closing the report are recorded. Therefore, we calculate the difference between submission time and the last closing time of the report to obtain the fixing time of each bug.

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

Based on the results in TABLE 7, we can conclude that both the average and median time to close invalid reports are longer than the time to fix actual bugs. In PaddlePaddle, it takes an average of 237.5 days to close an invalid report, which is almost as four times long as the time used to fix an actual bug (i.e., 64.7 days). The median time to close an invalid report is 214.6 days, which is more than 23 times longer than the time to fix an actual bug (i.e., 8.9 days). For MXNET, it takes an average of 187.1 days to close an invalid report, while the average time to fix an actual bug is 101.2 days. And the median fixing time is 104.7 days, which is around three times more than that of the time for fixing the actual bug (i.e., 30.8 days). In TensorFlow, the average time

TABLE 7
Fixing Time of Actual Bugs, Non-bugs and Invalid Reports

Project	Fixing time (day)	Actual bug	Non-bug	Invalid
TensorFlow	Average	96.4	76.4	120.2
	Median	37.3	27.3	63.2
MXNET	Average	101.2	83.8	187.1
	Median	30.8	18.1	104.7
PaddlePaddle	Average	64.7	53.5	237.5
	Median	8.9	4.9	214.6

TABLE 8
Fixing Time of Bohrbugs (BOHs) and Mandelbugs (MANs)

Project	Fixing time (day)	Bohrbug	Mandelbug
TensorFlow	Average	91.0	129.1
	Median	33.6	69.2
MXNET	Average	97.2	105.8
	Median	27.4	42.2
PaddlePaddle	Average	56.5	96
	Median	7.2	37.8

to close an invalid report is 120.2 days and the median time is 63.2 days, both of which are one month longer than the time required to fix an actual bug.

There are two typical reasons why it takes more time to close an invalid report. One of them is that developers or users do not respond to bug reports in a timely manner. Thus, the time interval between two comments is very long. As a result, these reports have expired and been closed. Another reason is that some reporters do not provide detailed information to reproduce the bug, and developers fail to reproduce the bug after a long period of debugging, and eventually have to close it.

Compared with the results in the existing study, for example, the authors in [55] calculated the fixing time of bugs in three machine learning software systems, including Apache Mahout, Lucene and OpenNLP. According to the results, over two-thirds of bug reports were closed within a month. In [26], the authors analyzed 329 bugs from Scikit-learn, PaddlePaddle and Caffe. In their study, 68.39% of bugs were fixed within one month. Compared with the results in [26], it takes more time to close bug reports in TensorFlow and MXNET. On the basis of our analysis, 47.22% and 51.80% of bugs in TensorFlow and PaddlePaddle were fixed within a month, respectively. The median time to close bug reports in TensorFlow and MXNET is 34 and 27 days respectively, which is three times and twice as long as the time to fix bug reports in PaddlePaddle.

Implication: It will save a lot of time if reporters can provide high-quality reports, containing as detailed information as possible, including running environments, error messages and reproducible examples. For example, in TensorFlow, a reporter submitted two reports (Bug ID-4651 and Bug ID-5394). However, neither of them provided instructions to reproduce the bug. After a long time of discussion, they were eventually closed due to unreproducible. In addition, developers should always respond to bug reports assigned to them. For reports that are mistakenly assigned to them, they should be reassigned to other relevant developers as soon as possible.

Finding #7: *It takes more time to fix a Mandelbug (MAN) than fix a Bohrbug (BOH).*

From TABLE 8, we can see that in all three deep learning frameworks we analyzed, it takes more time to fix a MAN than fix a BOH. In TensorFlow, it takes an average of 129.1 days to fix a MAN, which is nearly a month longer than the time required to fix a BOH (i.e., 91.0 days). Similarly, the median fixing time of MANs in TensorFlow is 69.2 days, which is more than twice the median fixing time of BOHs (i.e., 33.6 days). In MXNET, the average fixing time of MANs is 105.8 days, which is 8.6 days longer than the average fixing time of BOHs (i.e., 97.2 days). And the median fixing time of MANs is 42.2 days, which is 14.8 days longer than the median fixing time of BOHs (i.e., 27.4 days). In PaddlePaddle, the average fixing time of MANs is 96 days, which is almost 40 days longer than the average fixing time of BOHs (i.e., 56.5 days). As for the median fixing time, the time to fix a MAN is 37.8 days, which is over four times longer than the time to fix a BOH (i.e., 7.2 days).

The results are consistent with traditional software systems [21], [33], [37]. In [21], the authors performed their study on four traditional software systems, including Linux, HTTPD, MySQL and AXIS. They found that in Linux, HTTPD and AXIS, the fixing time of MANs tends to be longer than that of BOHs. Xiao et al. [33] confirmed the results obtained in [21] by researching on 5,741 bug reports for the Linux kernel. According to their analysis, the average time taken to fix a MAN is 254.22 days, while it takes an average of 218.63 days to fix a BOH. In addition, Qin et al. [37] performed an empirical study of bugs in Android. From their results, the average time to fix a BOH is 63.0 days and the average time to fix a MAN is 71.4 days. The conclusion they obtained is consistent with ours, which states that it takes more time to fix MANs than to fix BOHs.

The reason for the long fixing time of MANs can be explained by the characteristics of MANs. Before fixing a MAN, developers need to reproduce them, which requires users to provide sufficient information, including not only the operating environments and source code, but also each step of operations they performed. Furthermore, MANs require a strict reproduce environment, and the construction of the environment also takes a lot of time. Finally, some MANs appear to occur in a non-deterministic manner, and it may need to run the code multiple times or run for a really long time to trigger them.

Implication: The nondeterministic behavior of MANs makes it impossible to deal with them in the same way as it is used to deal with BOHs. Specific strategies could be used to deal with MANs. For example, mitigation methods such as fault tolerance [52] and software rejuvenation [56] could be used to mitigate the adverse effects of MANs.

TABLE 9
Distribution of Root Causes Among Different Bug Types

Root Cause	TensorFlow			MXNET			PaddlePaddle		
	BOH	ARB	NAM	BOH	ARB	NAM	BOH	ARB	NAM
Environment/configuration	0	0	19	0	0	50	7	0	17
Memory	3	74	5	2	17	2	6	10	0
Compatibility	5	0	0	2	0	0	10	1	1
Concurrency	0	0	27	0	0	12	0	0	2
Semantic	737	13	15	263	17	15	119	7	7
UNK	0	1	3	3	0	1	10	1	1
Total	745	88	69	270	34	80	152	19	28

TABLE 10
Correlation Between Bug Types and Root Causes

Root Cause	TensorFlow			MXNET			PaddlePaddle		
	BOH	ARB	NAM	BOH	ARB	NAM	BOH	ARB	NAM
Environment/configuration	0	0	**13.07**	0	0	**4.80**	0.38	0	**5.03**
Memory	0.04	**9.25**	0.80	0.14	**9.14**	0.46	0.49	**6.55**	0
Compatibility	**1.21**	0	0	**1.42**	0	0	**1.09**	0.87	0.59
Concurrency	0	0	**13.07**	0	0	**4.80**	0	0	**7.11**
Semantic	**1.17**	0.17	0.26	**1.27**	0.65	0.34	**14.84**	0.55	0.37

5 Root causes of Bohrbugs (BOHs) and Mandelbugs (MANs)

To answer RQ3, we analyze each BOH and MAN to identify its root causes, which is important for understanding and fixing bugs. Through manually checking, we identified five root causes of BOHs and MANs in TensorFlow, MXNET and PaddlePaddle, including environment/configuration, memory, compatibility, concurrency and semantic. We have described the definitions of these root causes in Section 2.4. The results are shown in TABLE 9 and TABLE 10, and findings are listed as follows.

Finding #8: *The major root cause of BOHs is semantic bugs and the primary root cause of ARBs is memory bugs.*

TABLE 9 presents the root cause distribution of BOHs, ARBs and NAMs in TensorFlow, MXNET and PaddlePaddle. According to the results, semantic is the main root cause of BOHs. In TensorFlow, 98.93% (737 out of 745) of BOHs are caused by semantic bugs, while in MXNET and PaddlePaddle, 97.41% (263 out of 270) and 78.29% (119 out of 152) of BOHs are caused by semantic bugs, respectively. The reason why semantic bugs are more likely to cause BOHs is that semantic bug is a kind of bug corresponding to the inconsistencies with requirements or the programmer's attention. Most of semantic bugs are improper functionality implementation or simple typo errors, which are prone to induce BOHs. For example, Bug ID-12179 in MXNET, "Spelling mistake in "mxnet/symbol/image.py" in which "gen_image" is written to be "gen_iamge"". As described by the reporter, a typo appeared in "mxnet/symbol/image.py", it is obviously a BOH caused by semanitc bug.

The main root cause of ARBs is memory bugs. As shown in TABLE 9, 74 out of 88 ARBs in TensorFlow are caused by memory bugs, accounting for 84.09%. In MXNET, half of ARBs are caused by memory bugs. In PaddlePaddle, 10 out of 19 ARBs are caused by memory bugs, accounting for 52.63%. It is reasonable that most ARBs are memory bugs. Because the most common situations of ARBs are memory leaks and out of memory errors, they are usually caused by improper handling of memory objects that can lead to the accumulation of memory costs. For example, Bug ID-6111 in TensorFlow, "Flawed memory management: allow_growth=True consumes more memory, causing out-of-memory". This is an ARB caused by a memory bug. Due to the improper memory management, an out of memory error occurs as the memory consumption grows. It should be noted that not all memory bugs will lead to ARBs, only memory bugs that cause accumulation of memory consumption are ARBs.

NAMs are mainly caused by concurrency bugs and environment/configuration bugs. In TensorFlow, concurrency bug is the major cause of NAMs, while in MXNET and PaddlePaddle, environment/configuration bug is the ma-

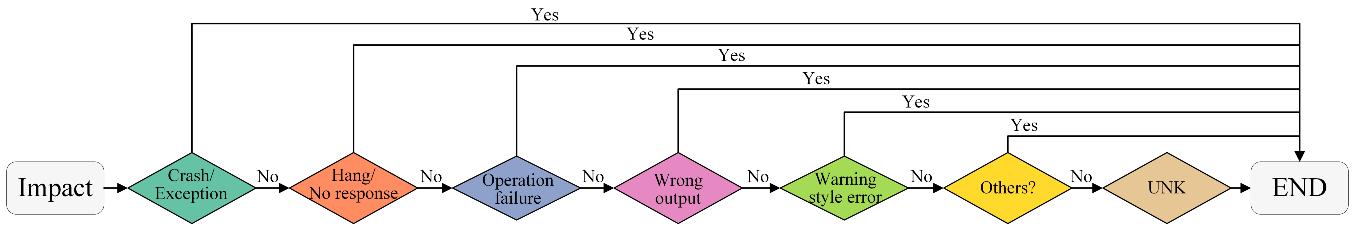


Fig. 7. The Flow Chart of Impact Classification.

jor cause of NAMs. However, concurrency and environment/configuration bugs hardly cause BOHs and ARBs. In order to further understand the correlation between root causes and bug types, we show the $lift$ correlation (defined in Section 2.6) between bug types and root causes in TABLE 10. In TABLE 10, numbers equal to 1 indicate that two categories are not related, numbers greater than 1 indicate positive correlation and are shown in bold, and numbers less than 1 indicate negative correlation.

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

As shown in TABLE 10, the results of the three deep learning frameworks we analyzed are similar. BOHs are more likely to be caused by compatibility bugs or semantic bugs. In TensorFlow, MXNET and PaddlePaddle, the $lift$ correlation between BOHs and the compatibility bugs are 1.21, 1.42 and 1.09, respectively. And the $lift$ correlation between BOHs and the semantic bugs are 1.17, 1.27 and 14.84 in TensorFlow, MXNET and PaddlePaddle, respectively. It is because that usually semantic bugs and compatibility bugs can always be reproduced under certain conditions. Among them, semantic bugs refer to incorrect implementations of functions or conflicts between features and corresponding requirements. For example, Bug ID-9363 in MXNET, "Incorrect weight_decay implementation in AdaGrad". This kind of bug can be continuously triggered every time the program is run. Compatibility bug is a kind of bug that causes software to fail on a particular CPU architecture, an operating system, or a web browser, etc. For example, Bug ID-33767 in TensorFlow is reported that "The dependencies in TF 2.0 pull the latest version of gast, which breaks compatibility at 0.3". This report shows a compatibility bug, which can be repeatedly reproduced using the gast version 0.3. Once downgrade the gast's version from 0.3 to 0.2.2, the bug will disappear.

ARBs tend to be memory bugs. The $lift$ correlation between ARBs and memory bugs in TensorFlow, MXNET and PaddlePaddle are 9.25, 9.14 and 6.55, respectively. The reason why ARBs are positively related to memory bugs is that one of ARB's subtypes is MEM, which is a kind of bug caused by the accumulation of errors due to improper memory management. If there is a memory bug that consumes more and more memory as the program runs, it is an ARB. A NAM is more likely to be an environment/configuration bug or a concurrency bug. According to the results, the $lift$ correlation between NAMs and environment/configuration bugs, as well as the correlation between NAMs and concurrency bugs are all greater than 1 in TensorFlow, MXNET and PaddlePaddle. It is because ENV and TIM are the two major subtypes of NAMs. Among them, ENVs are mainly caused by environment bugs, such as errors in dependent libraries, underlying operating systems, or non-code that affects functionality. TIM is a kind of bug whose activation is affected by the timing of inputs and operations. For example, deadlock and data race, which are typical concurrency bugs, are highly dependent on the timing of operations. As a result, NAMs have a strong correlation with concurrency bugs and environment/configuration bugs.

Compatibility and memory bugs are also typical root causes in other frameworks. According to the results in [26], for Scikit-learn, PaddlePaddle and Caffe, 22.49% of bugs are compatibility bugs. Incompatible bugs are mainly caused by incompatible operating systems, incompatible versions of Python, incompatible backward versions of the algorithm model and conflicts with hardware. In addition to compatibility bugs, the authors also found that 2.74% of bugs are memory overflow bugs.

Implication: To deal with BOHs, more efforts should be put into semantic bugs and compatibility bugs since semantic bugs and compatibility bugs are positively correlated with BOHs. Since memory bugs are the major root cause of ARBs, developers could refer to the solutions used to solve memory bugs to avoid ARBs. For NAMs, attention should be paid to environment/configuration bugs and concurrency bugs.

6 Impacts of Bohrbugs (BOHs) and Mandelbugs (MANs)

In this section, we present the results of RQ4. We analyze the impacts of bugs to understand how severe a BOH or a MAN is. According to our analysis, the failures caused by BOHs and MANs in TensorFlow, MXNET and PaddlePaddle include crash/exception, hang/no response, wrong output, operation failure and warning style error. The definitions of these impacts have been described in Section 2.5. If the impact of a bug does not belong to any of the five impacts mentioned above, we will label the impact as others. If the impact of a bug is not given or discussed in a bug report, we will label the impact as UNK (i.e., unknown).

In some bug reports, according to the reporters' description, a bug may have more than one type of impact. For example, for Bug-ID 7353 in TensorFlow, it is said that "...one of my machines was something wrong and caused almost all session run timeout, eventually memory reached 80G and been killed. This also caused worker0 process failed to save model, saver.save() stuck forever." In this sentence, the "session run timeout" is a crash/exception

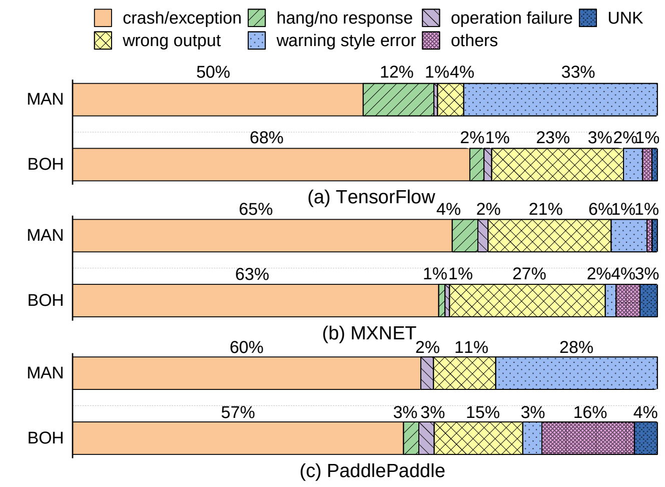


Fig. 8. Impacts of Bohrbugs (BOHs) and Mandelbugs (MANs).

impact and "saver.save() stuck forever" is a hang/no response type of impact. Another example is Bug-ID 1135 in TensorFlow. It is said that "In 0.7, there are different errors about Saver, first Warnings in serialization...and then, it throws an OS Error". In this example, an error and a warning were reported in the same report. To avoid multiple counting, we sort these impacts according to the severity of the consequences: crash/exception; hang/no response; operation failure; wrong output; and warning style error. The classification procedure is shown in Fig. 7. Given a bug report, we consider in turn whether its impact belongs to crash/exception, hang/no response, operation failure, wrong output, or warning style error. If the impact of a bug does not belong to any of the above five impacts, we will label it as others. If the bug's impact is not discussed in the bug report, we will label the impact as UNK.

Finding #10: *More than half of BOHs and MANs would result in crashes/exceptions.*

According to the results in Fig. 8, more than half of the studied BOHs and MANs would cause crashes or exceptions. For example, Bug ID-1961 in PaddlePaddle is a floating point exception. Furthermore, a large number of bugs can generate and present wrong results to users, especially BOHs. For example, Bug ID-7725 in MXNET, "accuracy of cpp example is a constant value when training, no matter how many epochs trained", incorrect accuracy is output during the training process. For MANs, the major impacts are wrong outputs and warning style errors. For example, Bug ID-16152 in TensorFlow, "DeprecationWarning from inspect.getargspec()". According to the description, a warning message keeps filling up the test output.

The results are consistent with those obtained from other frameworks. In [11], the authors studied deep learning applications built on TensorFlow. According to the results, the most common symptom is "Error", which is analogous to crash or exception under the given definition. They discovered that 46.9% of bugs always lead to program crashes. In [13], the authors studied client software built on 5 deep learning libraries, including Caffe, Keras, TensorFlow, Theano and Torch. According to the results, crash is the top impact of bugs in all the libraries, ranging from 40% to 77%.

TABLE 11
Numbers and Proportions of Regression Bugs and Non-regression Bugs.

Project	Regression	Non-regression	Total
TensorFlow	86 (9.02%)	867 (90.98%)	953 (100%)
MXNET	53 (13.55%)	338 (86.45%)	391 (100%)
PaddlePaddle	10 (4.81%)	198 (95.19%)	208 (100%)

Implication: Amounts of studied bugs in deep learning frameworks can lead to crashes or exceptions. Failing to catch and handle these bugs properly causes negative end-user experiences. To improve the robustness of deep learning frameworks, exception handling strategies could be introduced to separate the source code that deals with unusual situations from the code that supports normal processing [57], [58]. In addition, crash reports could be used to help with the bug localization. For example, Wu et al. [59] propose a method called CrashLocator to locate faulty functions using information contained in crash reports.

7 Regression Bugs in Deep Learning Frameworks

In this section, we present the answer to RQ5. A regression bug is a type of bug that causes software features which worked normally to stop behaving as intended after a certain event [60]. For example, a submitted commit to fix a bug or implement a new feature may disrupt the original normally running system functions. In this section, we aim to study the proportions of regression bugs in deep learning frameworks, and the distribution of different bug types among regression bugs [61]. We carefully read the information contained in each bug report, including the description of the reporter and comments submitted to discuss the bug, to determine if it is a regression bug or not.

Finding #11: *The proportions of regression bugs in TensorFlow, MXNET and PaddlePaddle are 9.02%, 13.55% and 4.81%, respectively.*

TABLE 11 presents the numbers and proportions of regression bugs and non-regression bugs in TensorFlow, MXNET and PaddlePaddle. From the results, we can observe that among 391 actual bugs in MXNET, there are 53 (i.e., 13.55%) regression bugs and 338 (i.e., 86.45%) non-regression bugs. The percentage of regression bugs in MXNET is the highest among all three deep learning frameworks. In TensorFlow, there are 86 regression bugs and 867 non-regression bugs, accounting for 9.02% and 90.98% of 953 actual bugs, respectively. And among all the 208 actual bugs in PaddlePaddle, there are 10 regression bugs and 198 non-regression bugs, accounting for 4.81% and 95.19%, respectively. One of the most common scenarios that regression bugs appear is that some functions work normally in the previous versions but stop working after upgrading to a newly released version. For example, in TensorFlow's Bug ID-9708, the reporter says that "tf.random crop exception after upgrading to tf1.1 from tf1.0". The reporter discovered that after upgrading TensorFlow from version 1.0 to version

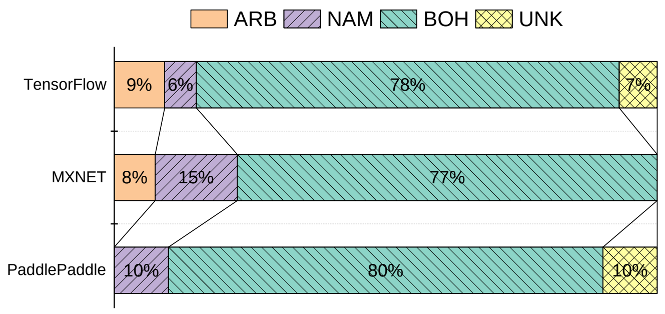


Fig. 9. Bug type distribution among regression bugs.

1.1, an exception is triggered in tf.random function. Another situation is that a commit to fix an existing bug will fail the original normally running function. For example, TensorFlow's Bug ID-3035, is described as "The following fails with UnboundLocalError after b80a4a8", that is, the commit b80a4a8 used to deal with the exception handling problem introducing a regression bug at the same time.

Compared to traditional software projects, the proportions of regression bugs in deep learning frameworks are lower than those of Linux kernel (i.e., 50.1% [61]) and Google Chromium (i.e., 51.09% [60]). The reason is that version upgrade is one of the most common ways to discover regression bugs. However, deep learning frameworks have only been developed in recent years, and the number of released versions is small. For example, over the past 25 years, Linux has put out more than 1300 releases ranging from version 1.0 to version 4.14. Compared with Linux, there are only dozens of versions in deep learning frameworks. As a result, in the process of continuously upgrading the system, users in Linux discovered more regression bugs.

Finding #12: *Among all regression bugs in TensorFlow, MXNET and PaddlePaddle, BOHs account for 78%, 77% and 80%, respectively, and MANs account for 15%, 23% and 10%, respectively.*

Fig. 9 gives the distribution of bug types in regression bugs. Among all 86 regression bugs in TensorFlow, BOHs account for 78%, ARBs and NAMs account for 9% and 6%, respectively. In MXNET, the proportions of BOHs, ARBs and NAMs among regression bugs are 77%, 8% and 15%. In PaddlePaddle, BOHs account for 80% of regression bugs, NAMs account for 10% and no ARB belongs to regression bug. In addition, we calculate the $lift$ correlation between bug types and regression bugs to further determine the type of bug that is more likely to be regression bug. The results are shown in TABLE 12. According to the results, the $lift$ correlation between non-regression bugs and MANs in TensorFlow, MXNET and PaddlePaddle are greater than 1, which means non-regression bugs are prone to be MANs. In addition, except for TensorFlow, the $lift$ correlation between regression bugs and BOHs are greater than 1 in MXNET and PaddlePaddle, which means regression bugs tend to be BOHs.

Implication: It is annoying to encounter regression bugs for both users and developers. It would make users confused and waste lots of time to debug. As a result, users might lose confidence in the new released version and refuse to upgrade. For developers, regression bugs are also painful and costly to deal with. By the time a regression bug is identified and reported, lots of changes have been made to the source code, which makes it difficult for developers to find the change that inducing the bug. We recommend developers conduct sufficient regression testing before releasing a new version. Regression testing is performed to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change [62], [63]. In addition, for regression bugs that have occurred, tools could be used to localize and predict them. For example, CodePsychologist [64] is a tool developed to assist the programmers in locating the lines of code that caused a given regression bug, and BCT [65] is a tool helping with the prediction of regressions.

TABLE 12
Correlation Between Bug Types and Regression Bugs.

Project	Correlation	BOH	MAN
TensorFlow	Regression	1	0.92
	Non-regression	1	**1.01**
MXNET	Regression	**1.12**	0.78
	Non-regression	0.98	**1.04**
PaddlePaddle	Regression	**1.09**	0.44
	Non-regression	1.00	**1.03**

8 Threats to Validity

Similar to other empirical studies, our study is naturally subject to validity problems. We identify potential threats to the validity of our study as follows:

Threats to Construct Validity. This study focuses on actual bug information provided by users and developers, that is, only bug reports tagged as "type:bug" are considered. There may be other bugs but are not labeled as "type:bug". In addition, we only analyze closed bug reports because unclosed bug reports may still be under discussion and may not have enough information to determine their type. If an unclosed bug report are considered, the distribution of bug types may differ. Different time frames are also one of the threats to construct validity. Although we have considered a great number of bug reports, there are still some bug reports that are beyond the scope of our analysis. If all bug reports are taken into account, it may lead to different results.

Threats to Internal Validity. The possibility of classification mistakes is a threat to internal validity. To mitigate this threat, four authors are all involved in manual classification, and all of them are experienced developers. First, two authors separately manually classified all the bug reports. Initially, we sampled a small set of bug reports and checked these reports together to determine their bug types according to the definitions in order to calibrate our classification work. In the process of manual classification, we carefully checked all the information contained in bug reports, including report descriptions, forum comments, attached files (e.g., patches applied for correcting the bug) and the external links that were attached for providing further information (e.g., Git commit IDs). Then, a cross-check was performed, and conflicting cases were eliminated through discussions among all four authors to reach a consensus. However, no matter how meticulous the authors were, we admit that there is still the possibility of misclassification, which cannot be completely avoided.

Threats to External Validity. The bug reports studied in this paper are collected from three representative deep learning frameworks (i.e., TensorFlow, MXNET and PaddlePaddle). Some findings and implications may not applicable to deep learning applications or other deep learning frameworks. In order to reduce this threat, we try not to extend the conclusions that only apply to TensorFlow, MXNET and PaddlePaddle to applications or other deep learning frameworks.

9 Discussion

According to the classification results, MANs occupy a non-negligible proportion of the total bugs in the DL frameworks. However, the classic approaches for handling software bugs, such as debugging and testing, are not applicable to MANs [66]. MANs are triggered by complex conditions [67], [68], such as interactions with hardware and other software systems and the timing or sequence of events. It is difficult to detect these bugs using traditional testing techniques because it can be challenging to control their complex triggering conditions in a testing environment [69]. Therefore, it is necessary to adopt specific verification and/or fault-tolerance strategies to deal with them in a cost-effective way [70]. This section focuses on discussing the mitigation strategies for MANs from both the developer and user perspectives.

The four most common methods for resuming system operations after failures caused by MANs are restart, reconfiguration, reboot and hot fix [52], [71]. Users could adopt software recovery strategies to deal with MANs. For example, Trivedi et al. [52] constructed a flowchart to describe the recovery process implemented in software systems, and they proposed a closed-form expression of the average recovery time from MANs. In addition, researchers have also investigated models for recovery from MANs [71]. For developers, MAN prediction methods could be used to locate MANs in software systems [72]. As a result, developers can concentrate verification and validation activities and fault tolerance mechanisms in those modules where MANs are most likely to exist. If there is a serious failure that cannot be solved by any of these four methods, then developers can perform thorough testing and code debugging, like regular bug fixes.

In addition to MAN, we find 8.7%-9.2% of bugs in TensorFlow, MXNET and PaddlePaddle are ARBs. ARB is a special type of MAN, as the software system runs for a long time, it will cause performance degradation or increased failure rate [19]. For example, performance degradation can cause an increase in response time or a reduction in the number of served requests per second [73]. For safety- and security-critical software systems, this can be fatal [38]. Taking autonomous driving software as an example, slower response may cause the car to fail to recognize objects ahead or brake in time, resulting in disastrous consequences. ARBs can be mitigated by proactive methods applied before failures occur, namely software rejuvenation [74], [75]. Software rejuvenation is a concept of gracefully terminating an application and immediately restarting it in a clean internal state. In [76], models were developed for analyzing software rejuvenation in continuously running applications. In [77], a set of software complexity metrics for ARBs were collected as predictor variables, and bug prediction models were built to predict the location of ARBs.

10 Related Work

Over the past decades, deep learning has achieved an enormous breakthrough in artificial intelligence and gained great popularity in various applications [78]. Deep learning frameworks play an important role to bridge the deep learning theory to the realization of deep learning software by providing high-level APIs to support deep learning models and runtime training configurations [79]. In recent years, the rapid development of deep learning frameworks also gain attentions on the reliability of these frameworks. Some of them focus on investigating the characteristics of bugs inside frameworks. The others perform studies on applications built upon these frameworks.

Zhang et al. [11], Islam et al. [13] and Humbatova et al. [80] studied the characteristics of bugs in deep learning applications, i.e., clients built on top of deep learning frameworks. Zhang et al. [11] performed an empirical study on deep learning applications programmed on the TensorFlow framework. They collected program bugs related to TensorFlow from Stack Overflow Q&A pages and GitHub projects, and 175 bugs were obtained. They analyzed the symptoms and root causes of bugs and investigated the challenges in bug detection and localization. While only TensorFlow clients were analyzed in [11], client software built on more frameworks were studied in [13], including Caffe, Keras, Theano and Torch. In [13], the authors studied 2716 posts from Stack Overflow and 500 bug fix commits from GitHub to identify the bug types, root causes of bugs and effects of bugs in the usage of deep learning. They categorized bugs into 11 bug types, 10 root causes and 7 impacts. Humbatova et al. [80] analyzed 1059 artefacts, including 477 Stack Overflow discussion, 271 issues and pull requests, and 311 commits, which were collected from projects using TensorFlow, Keras and PyTorch. They identified the root cause behind each problem by manual analysis. While these works studied deep learning clients, this paper focuses on deep learning frameworks.

The study in [81] analyzed 715 questions in Stack Overflow related to three deep learning frameworks, including TensorFlow, PyTorch and Deeplearning4j. Through manually inspecting these questions, they identified seven types of frequently asked questions. According to their findings, program crashes and model migration are the two most frequently asked topics. In addition, they built a classification model to quantify the distribution of different types of deep learning questions. Compared to [81], which extracts questions in Stack Overflow as research data, our research data are actual bugs collected from GitHub repositories.

Li et al. [12] conducted the empirical study to analyze the bugs inside TensorFlow. They analyzed 202 TensorFlow bug fixes repaired between December 2017 and March 2019, and 84 of them have corresponding bug reports. Root causes and symptoms of bugs were analyzed in this work, as well as the proportions of bugs inside different library components. Instead of TensorFlow, Sun et al. [26] conducted an empirical study on three other frameworks, including Scikit-learn, PaddlePaddle and Caffe. Based on the occurring reasons of bugs, they classified 329 bugs into 7 categories and 12 fixing patterns. Compare to [26], instead of analyzing the reasons and fixing patterns, we perform our research from the perspective of fault triggering conditions.

11 Conclusions and Future Work

This paper conducts a large-scale research on actual bugs in three widely-used deep learning frameworks, including TensorFlow, MXNET and PaddlePaddle in terms of fault triggers. We manually analyzed 3,555 bug reports collected from Github, which were actual bug information submitted by users and developers. Our analyses are conducted from five dimensions: bug types of deep learning frameworks and the proportion evolution of bugs over time; fixing time of bugs; root causes of Bohrbugs and Mandelbugs; impacts of Bohrbugs and Mandelbugs; and features of regression bugs in deep learning frameworks. Our study found that more than two-thirds of bugs are Bohrbugs, and MEM is the major subtype of aging-related bugs. It takes more time to close a Mandelbug than a Bohrbug. There are five root causes of Bohrbugs and Mandelbugs, including environment/configuration, memory, compatibility, concurrency and semantic. A BOH is more likely caused by a semantic bug or a compatibility bug, an ARB is prone to be a memory bug, and a NAM is most likely to be an environment/configuration bug or a concurrency bug. There are five impacts by Bohrbugs and Mandelbugs, and more than half of Bohrbugs and Mandelbugs would cause crashes/exceptions. The proportions of regression bugs in deep learning frameworks are lower than that of Linux kernel. Finally, some practical implications are given for both users and developers.

In addition to classifying bugs based on fault triggers and root causes, we can also identify bugs according to the different phases in which they appear, such as training and inference. Focusing on bugs related to model training and reference, we performed a preliminary analysis. First, we used the keywords "train" and "inference" to filter all the bug reports and obtained 261 and 44 bug reports, respectively. Then, we manually examined these 301 bug reports and removed the irrelevant bug reports. Finally, 84 training-related bugs and 11 reference-related bugs were obtained. Training-related bugs involve three different aspects, including the quality of the training data, the running of the training process and the training results obtained. Among all the 84 training-related bugs, most (88.09%) occurred during the training process, including the crash of the distributed training process, exceptions during multi-GPU training, memory leaks when training complex models and incorrectly transferred parameters or variables. 9.52% of them led to incorrect training results. For example, the training results always appeared to be NaN (Not-a-Number) or constant values, and quite different results were obtained with the same parameter set. A total of 4.76% of them were related to training data, including broken training data, insufficient training data and improper data processing.

For inference-related bugs, there are two aspects, including the running of the inference process and the reference results. Among 11 inference-related bugs, 7 (66.63%) occurred during the inference process, including failed inference when the inference code was incorrect, freezing inference when dealing with a larger size of data, parallel inference failures due to API bugs and memory leaks when performing inference. Four (33.36%) of them resulted in incorrect inference results. For example, different inference results were obtained while performing inference multiple times or running with multiple GPUs. In the future, we will check all 3,555 bug reports studied in this paper to determine the phase at which each bug occurred. In addition to the training and reference phases, we will also study bugs in other phases, such as the data preparation phase before training the model and the model deployment process after reference.

References

[1] Y. Sun, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," *Advances in neural information processing systems*, vol. 27, 2014.

[2] E. Menasalvas and C. Gonzalo-Martin, *Challenges of Medical Text and Image Processing: Machine Learning Approaches*. Springer International Publishing, 2016.

[3] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, and K. a. Macherey, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[4] X. Sun, X. Liu, B. Li, Y. Duan, and J. Hu, "Exploring topic models in software engineering data analysis: A survey," in *IEEE/ACIS International Conference on Software Engineering*, 2016.

[5] 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 (ICSE-C)*, 2017.

[6] B. Sravyapranati, D. Suma, C. Manjulatha, and S. Putheti, "Large-scale video classification with convolutional neural networks," in *IEEE conference on Computer Vision and Pattern Recognition*, 2020.

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

[8] None, "Uber self-driving car fatality," *New Scientist*, vol. 237, no. 3170, pp. 7–7, 2018.

[9] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[10] T. W. Cai, D. Feng, S. Liu, S. Liu, R. Kikinis, and S. Pujol, "Early diagnosis of alzheimer's disease with deep learning," in *IEEE 11th international symposium on biomedical imaging*, 2014.

[11] 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.

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

[13] 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.

[14] 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.

[15] M. Lux and M. Bertini, "Open source column: deep learning with keras," *ACM SIGMultimedia Records*, vol. 10, no. 4, pp. 7–7, 2019.

[16] 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.

[17] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

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

[19] 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.

[20] ——, "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.

[21] 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.

[22] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with tensorflow: A review," *Journal of Educational and Behavioral Statistics*, vol. 45, 2020.

[23] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *Stats*, 2015.

[24] Y. Ma, D. Yu, T. Wu, and H. Wang, "Paddlepaddle: An open-source deep learning platform from industrial practice," *Frontiers of Data and Domputing*, vol. 1, no. 1, pp. 105–115, 2019.

[25] X. Du, G. Xiao, and Y. Sui, "Fault triggers in the tensorflow framework: An experience report," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[26] 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.

[27] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 788–799.

[28] G. Jahangirova, N. Humbatova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," *arXiv preprint arXiv:1910.11015*, 2019.

[29] A. Shatnawi, G. Al-Bdour, R. Al-Qurran, and M. Al-Ayyoub, "A comparative study of open source deep learning frameworks," in *2018 9th international conference on information and communication systems (icics)*. IEEE, 2018, pp. 72–77.

[30] 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.

[31] 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.

[32] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node.js," *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[33] 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.

[34] A. Di Sorbo, J. Spillner, G. Canfora, and S. Panichella, "" won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on github," *arXiv preprint arXiv:1904.02414*, 2019.

[35] A. Hindle and C. Onuczko, "Preventing duplicate bug reports by continuously querying bug reports," *Empirical Software Engineering*, vol. 24, no. 2, pp. 902–936, 2019.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] K. S. Trivedi and G. E. Andrade, "Software fault mitigation and availability assurance techniques," *International Journal of Systems Assurance Engineering&Management*, 2010.

[43] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, and M. Bernstein, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[44] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–35, 2018.

[45] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimization towards training a trillion parameter models," *arXiv preprint arXiv:1910.02054*, 2019.

[46] 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.

[47] M. Xie, L. Lei, Y. Hao, C. Wu, and H. Geng, "Sysmon: Monitoring memory behaviors via os approach," *International Workshop on Advanced Parallel Processing Technologies*, 2017.

[48] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[49] M. Cavage, "There is no getting around it: you are building a distributed system," *Communications of the Acm*, vol. 56, no. 6, pp. 63–70, 2013.

[50] 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.

[51] W. U. Zhendong, L. U. Kai, and X. Wang, "Surveying concurrency bug detectors based on types of detected bugs," *Science China*, no. 03, pp. 5–31, 2017.

[52] 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.

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

[54] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[55] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 271–280.

[56] 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.

[57] C. Marinescu, "Should we beware the exceptions? an empirical study on the eclipse project," in *International Symposium on Symbolic & Numeric Algorithms for Scientific Computing*, 2013.

[58] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis, "The exception handling riddle: An empirical study on the android api," *Journal of Systems and Software*, vol. 142, no. aug., pp. 248–270, 2018.

[59] M. Medeiros, U. Kulesza, R. Bonifacio, E. Adachi, and R. Coelho, "Improving bug localization by mining crash reports: An industrial study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[60] 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.

[61] 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.

[62] A. Bajaj and O. P. Sangwan, "A survey on regression testing using nature-inspired approaches," in *ICCCA-2018: 4th IEEE International Conference on Computing Communication and Automation*, 2019.

[63] S. Nayak, C. Kumar, S. Tripathi, N. Mohanty, and V. Baral, "Regression test optimization and prioritization using honey bee optimization algorithm with fuzzy rule base," *Soft Computing*, no. 2, pp. 1–18, 2020.

[64] 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.

[65] A. Tarvo, "Mining software history to improve software maintenance quality: A case study," *IEEE Software*, vol. 26, no. 1, pp. 34–40, 2009.

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

[67] S. Russo, D. Cotroneo, R. Pietrantuono, and K. Trivedi, "How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation," *Journal of Systems & Software*, vol. 113, pp. 27–43, 2016.

[68] X. Du, Z. Zheng, G. Xiao, Z. Zhou, and K. S. Trivedi, "Deepsim: Deep semantic information-based automatic mandelbug classification," *IEEE Transactions on Reliability*, 2021.

[69] R. Chillarege, "Understanding bohr-mandel bugs through odc triggers and a case study with empirical estimations of their field proportion," in *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*. IEEE, 2011, pp. 7–13.

[70] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *International Journal of System Assurance Engineering and Management*, vol. 1, no. 4, pp. 340–350, 2010.

[71] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery from software failures caused by mandelbugs," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 70–87, 2015.

[72] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Analysis and prediction of mandelbugs in an industrial software system," in *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE, 2013, pp. 262–271.

[73] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, "Workload characterization for software aging analysis," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 240–249.

[74] N. A. Valentim, A. Macedo, and R. Matias, "A systematic mapping review of the first 20 years of software aging and rejuvenation research," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016, pp. 57–63.

[75] K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and analysis of software aging and rejuvenation," in *Proceedings 33rd annual simulation symposium (SS 2000)*. IEEE, 2000, pp. 270–279.

[76] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*. IEEE, 1995, pp. 381–390.

[77] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting aging-related bugs using software complexity metrics," *Performance Evaluation*, vol. 70, no. 3, pp. 163–178, 2013.

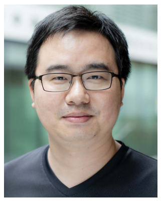
[78] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[79] Q. Guo, X. Xie, L. Ma, Q. Hu, R. Feng, L. Li, Y. Liu, J. Zhao, and X. Li, "An orchestrated empirical study on deep learning frameworks and platforms," *arXiv preprint arXiv:1811.05187*, 2018.

[80] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE '20: 42nd International Conference on Software Engineering*, 2020.

[81] 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.

Xiaoting Du received her M.Sc. from Beihang University in China. She is now a Ph.D. Student with the Depintelligence Software Lab, School of Automation Science and Electrical Engineering, Beihang University. Her research interest focuses on intelligent software reliability engineering.



Yulei Sui is a Senior Lecturer at School of Computer Science, Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). His research focuses on building fundamental static and dynamic analysis techniques and tools to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages and machine learning.

Zhihao Liu received his bachelor degree from SHENYUAN Honors College of Beihang University in 2016. He is currently a master student in School of Automation Science and Electrical Engineering of Beihang University. His research interests include software reliability and machine learning testing.

Jun Ai received the Ph.D degree in Beihang University, Beijing, China, in 2006. He is associate professor and vice-president in School of Reliability and Systems Engineering, Beihang University. His current research interests include software reliability, software testing and software fault prediction.