Toward Understanding Deep Learning Framework Bugs

JUNJIE CHEN, Tianjin University, China
YIHUA LIANG*, Tianjin University, China
QINGCHAO SHEN*, Tianjin University, China
JIAJUN JIANG†, Tianjin University, China
SHUOCHUAN LI, Tianjin University, China

DL frameworks are the basis of constructing all DL programs and models, and thus their bugs could lead to the unexpected behaviors of any DL program or model relying on them. Such a wide effect demonstrates the necessity and importance of guaranteeing DL frameworks' quality. Understanding the characteristics of DL framework bugs is a fundamental step for this quality assurance task, facilitating designing effective bug detection and debugging approaches. Hence, in this work we conduct the most large-scale study on 1,000 bugs from four popular and diverse DL frameworks (i.e., TensorFlow, PyTorch, MXNet, and DL4J). By analyzing the root causes and symptoms of DL framework bugs associated with 5 components decomposed from DL frameworks, as well as measuring test coverage achieved by three state-of-the-art testing techniques, we obtain 12 major findings for the comprehensive understanding of DL framework bugs and the current status of existing DL framework testing practice, and then provide a series of actionable guidelines for better DL framework bug detection and debugging. Finally, based on the guidelines, we design and implement a prototype DL-framework testing tool, called TenFuzz, which is evaluated to be effective and finds 3 unknown bugs on the latest TensorFlow framework in a preliminary study, indicating the significance of our guidelines.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software defect analysis**; • **General and reference → Empirical studies**.

Additional Key Words and Phrases: Deep Learning Frameworks, Bug Analysis, Empirical Study, Deep Learning Testing

ACM Reference Format:
Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 31 pages. https://doi.org/10.1145/3587155

1 INTRODUCTION

In recent years, Deep Learning (DL) systems have become one of the most popular types of software systems and have been widely used in many domains, such as autonomous driving [16], aircraft

*Equal contribution.
†Corresponding author.

Authors' addresses: Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, junjiechen@tju.edu.cn; Yihua Liang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, liangyihua@tju.edu.cn; Qingchao Shen, School of New Media and Communication, Tianjin University, Tianjin, China, 300350, qingchao@tju.edu.cn; Jiajun Jiang, College of Intelligence and Computing, Tianjin University, Tianjin, China, jiangjiajun@tju.edu.cn; Shuochuan Li, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, lishuochuan@tju.edu.cn.

1049-331X/2023/1-ART1 $15.00
https://doi.org/10.1145/3587155

collision avoidance [35], and software engineering [21, 36, 58, 72]. However, like traditional software, DL systems also contain bugs, which could lead to huge economic losses or even threaten human lives. For example, in 2018, an Uber autonomous car killed a pedestrian in Arizona [6] and a Tesla Model S in autopilot mode crashed into a fire truck parked with light flashing on a California freeway [7]. Therefore, guaranteeing the quality of DL systems is critical.

A DL system typically involves three levels [71]: the production level (i.e., DL models), program level (i.e., DL programs used for training DL models), and framework level (i.e., DL frameworks, also called DL libraries in some existing work [64], used by developers for implementing DL programs). Bugs at any level could affect the overall quality of the DL system. Hence, it is necessary to ensure DL systems' quality at all these levels. Over the years, a lot of research has focused on the production level by designing various DL model testing metrics [38, 44, 46], proposing various adversarial input generation methods [23, 39, 51, 73], or prioritizing/selecting test inputs for improving DL model testing [18, 65, 81], as well as the program level by studying the characteristics of DL program bugs [31, 32, 61, 80] or designing bug detection and diagnosis methods [66, 71, 79]. However, there is less attention on the framework level. Actually, DL frameworks are the basis of constructing all DL programs and models, and thus their bugs could produce much wider effects than the bugs in a specific DL program or model. Therefore, it is very essential to put more effort in ensuring the quality of DL frameworks, and this work does focus on the framework level.

Indeed, DL frameworks' quality has begun to receive attention recently, and some DL framework testing techniques have been proposed [25, 50, 64, 78]. Although they have been demonstrated to be effective to detect some new bugs in their experiments, they tend to treat the DL framework under test as a black box and lack a comprehensive understanding of the DL framework bug characteristics (such as root causes and bug distribution). Such a lack could limit their performance and hinder the design of more effective bug detection techniques. Moreover, it could limit the development of DL framework bug diagnosis techniques since this kind of tasks require much more sufficient understanding of detected bugs. That is, understanding the characteristics of DL framework bugs comprehensively is the fundamental task in the area of DL framework quality assurance, which is also the goal of our work.

In the literature, some studies on investigating DL bug characteristics have been conducted [31, 32, 80], but almost all of them target DL program bugs rather than DL framework bugs. Due to the significant differences between DL programs and DL frameworks, their bug characteristics are different. Specifically, a DL program is to invoke the APIs provided by a DL framework for implementing the desired neural network structure, and thus DL program bugs actually refer to those caused by the incorrect usage of the DL framework rather than the bugs inside the DL framework code (which are DL framework bugs). Regarding DL framework bug characteristics, Jia et al. [33] made the only one attempt till now, but it is still not enough to comprehensively understand bugs in the family of DL frameworks due to its small scale and limited study points (e.g., studying only one DL framework from three aspects). More details on the differences between our work and these existing studies can be found in Section 7. Hence, in this work **we conduct a comprehensive study to facilitate the sufficient understanding of DL framework bugs**.

Specifically, we used four popular DL frameworks (in terms of the number of forks in their GitHub repositories) in the study, including TensorFlow [9] from Google, PyTorch [8] from Facebook, MXNet [5] from Apache, and Deeplearning4j (DL4J) [2] from Eclipse, as the experimental subjects. In particular, they have great diversity, e.g., involving both static and dynamic computational graphs, various programming languages for implementations, and different development organizations, which facilitates the generalizability of our conclusions. In total, we studied 1,000 real bugs collected from their bug repositories and manually labeled them according to a systematic process (to be presented in Section 3). To our best knowledge, our study is the most large-scale one for investigating

DL framework bugs. Based on the 1,000 bugs from the four DL frameworks, our study aims to address the following five research questions:

- **RQ1: What are the root causes of DL framework bugs and their distribution?** The root causes are helpful to understand the nature of DL framework bugs, which facilitates the detection, localization, and fixing of bugs. Also, it is interesting to investigate the root causes specific to DL framework bugs and explore whether the conclusions on prevalent root causes between DL framework bugs and other software bugs are consistent or not.
- **RQ2: What are the symptoms of DL framework bugs and their distribution?** The symptoms are helpful to understand the consequences of DL framework bugs, which facilitates to triage them and assess their impacts. Also, we analyzed at which stages of the DL pipeline we can observe these symptoms. The results can guide the improvement of test oracles for more effective testing of DL frameworks.
- **RQ3: What is the relationship between root causes and symptoms of DL framework bugs?** After investigating the root causes and symptoms of DL framework bugs individually, it can obtain more comprehensive information about the bugs by associating them to study which root cause is more likely to produce a specific bug symptom.
- **RQ4: Which levels in DL frameworks are more fragile to bugs?** In general, a DL framework consists of five levels (to be introduced in Section 2) and the fragility of different levels may be different for different kinds of DL framework bugs. Identifying bug-prone levels for each kind of bugs can make the testing and debugging practice more targeted.
- **RQ5: Do the bugs of different DL frameworks have commonality?** We investigated whether there is some relationship among the bugs of different DL frameworks. It is helpful to guide the design of more general testing and debugging techniques for DL frameworks. Also, it may improve the testing and debugging practice of a DL framework by drawing the experience from other DL frameworks.

In our study, we decomposed a DL framework into 5 levels, and identified 13 root causes and 6 symptoms of DL framework bugs through systematic manual analysis. By studying each aspect individually and associating different aspects together, we obtained 10 major findings. Inspired by these findings, we further conducted a preliminary experiment to investigate the state-of-the-art DL framework testing techniques in terms of test coverage on each level of DL frameworks, and obtained additional 2 findings about the current status of existing approaches. Based on the empirical results and all the findings, we further provided a series of actionable guidelines for future DL framework testing and debugging. Furthermore, to evaluate the usefulness of our guidelines, we have designed and developed a prototype DL-framework testing tool, called TenFuzz, and evaluated its effectiveness on the latest version of the TensorFlow framework. The results showed that it successfully detected 6 bugs, of which 3 bugs are previously unknown ones and have been confirmed by the maintainers, demonstrating the significance of our guidelines for future research.

To sum up, our work makes the following major contributions:

- We conduct the most large-scale comprehensive study on DL framework bugs based on 1,000 real bugs from four popular and diverse DL frameworks.
- We provide a classification of root causes and symptoms of DL framework bugs, and associate them with each other as well as each level of DL frameworks.
- We conduct a preliminary experiment to investigate the current status of the state-of-the-art DL framework testing techniques, further confirming the necessity of developing more effective testing approaches.
- We provide a series of actionable guidelines for future DL framework testing and debugging practice according to our findings.

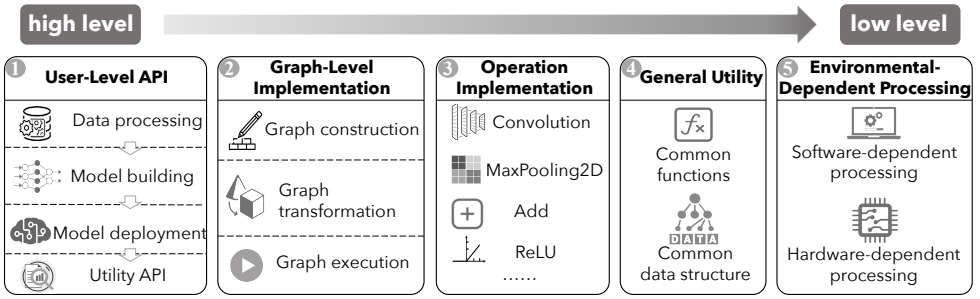


Fig. 1. Architecture of DL frameworks

- We design and implement a prototype DL-framework testing tool and conduct a preliminary study with it. The result demonstrates the significance of our findings and guidelines.

2 DEEP LEARNING FRAMEWORKS

DL frameworks are the basis for implementing DL programs and building DL models. To complete a prediction task, developers have to implement a DL program by invoking the APIs provided by a DL framework, and then a model can be built by executing the DL program with training data. The core functionalities of a DL program include determining the structure of a neural network (e.g., selecting proper layers and setting their order) and configuring the training process (e.g., setting the optimizer and loss function). All the detailed implementations under the invoked APIs for these DL functionalities are *inside the used DL framework*. Besides implementing various DL functionalities, DL frameworks also have the other two typical characteristics: On the one hand, DL frameworks are the bridge between DL functionalities and various hardware, and thus they also implement some strategies to support DL functionalities on different hardware. On the other hand, DL is still a fast-growing area and thus DL frameworks are frequently updated to incorporate the rapid advancement in DL algorithms. Therefore, DL frameworks are definitely important for DL development and very complicated especially compared with widely-studied DL programs.

By referring to the existing work [13] and understanding the functionality of DL frameworks, a DL framework can be decomposed into a general five-level architecture as shown in Figure 1. The five levels are **User-Level API**, **Graph-Level Implementation**, **Operation Implementation**, **General Utility**, and **Environment-Dependent Processing**, where User-Level API is the highest level that can be directly accessed by users to implement their DL programs while Environment-Dependent Processing is the lowest level that is related to the underlying infrastructure.

① **User-Level API**. This level contains a large number of high-level APIs, which aim to provide convenience for users to use DL frameworks to conduct their DL tasks. According to the workflow of DL, this level can be further divided into four components 1) *Data-Processing API* aims to process the input data to make them meet the corresponding requirement of a DL model, e.g., image resizing and text tokenization. 2) *Model-Building API* aims to construct a model structure and search for a group of optimal parameters for the model to make it well fit the training data via a given optimization target (e.g., a loss function). For example, APIs for various layers and loss functions, as well as various optimizers (e.g., Adam) belong to it. 3) *Model-Deployment API* aims to integrate a built DL model into an existing production environment to make practical prediction. Typically, it involves the processing (such as model quantization) that makes a DL model work in a specific environment. 4) *Utility API*: There are many utility APIs across the whole workflow of DL, which

provide some auxiliary functionalities to facilitate the DL process, e.g., model visualization and checkpointing.

② **Graph-Level Implementation**. After implementing a DL program based on these user-level APIs, the follow-up process is mainly based on a static or dynamic computational graph. A computational graph is a directed graph, in which each node represents an operation (e.g., convolution operation). An operation can feed its outputs to another operation through an edge, and the values that flow along an edge are tensors. This level contains all the computational-graph-level implementations in DL frameworks. According to the functionalities on a computational graph, this level can be divided into three components: 1) *Graph Construction* aims to create a computational graph and obtain subgraphs via partitioning a graph for distributed execution (especially for a static graph). 2) *Graph Transformation* is responsible for graph optimization (e.g., common subexpression elimination and operation fusion) to improve computation performance, and graph conversion (e.g., converting to the ONNX format). 3) *Graph Execution* aims to execute the graph in a runtime environment, including local and distributed execution. For example, the execution process involves data propagation and gradient computation. Please note that the functionalities of these components are conducted in order for static computational graphs, but are mostly intertwined for dynamic computational graphs.

③ **Operation Implementation**. As presented above, each node in a graph is an operation. This level contains all the detailed implementations for these operations. An operation takes zero or more tensors as input and produces zero or more tensors as output. There are a large number of operations implemented in DL frameworks, such as convolution operations, pooling operations, batch normalization operations, mathematical operations (e.g., log), and array manipulation operations (e.g., shuffle).

④ **General Utility**. To facilitate the implementations of the above levels, there are many general utilities in DL frameworks, including common data structures and common functions (such as type conversion and padding functions). This level includes all these general utilities. Please note that this level is different from *Utility API* in User-Level API that focuses on facilitating the process of training and deployment for users.

⑤ **Environment-Dependent Processing**. This level is the lowest one, which aims to support the functionalities of DL frameworks in different environments. A typical example is the memory allocation strategies on different devices, which aim to achieve high efficiency on different devices by considering their corresponding characteristics. That is, this level contains all the implementations establishing connections between the functionalities of DL frameworks and environments, including both hardware environments (e.g., GPU) and software environments (e.g., operating systems).

3 METHODOLOGY

3.1 Data Collection

In the study, we selected four popular DL frameworks as our subjects, i.e., TensorFlow [9] from Google, PyTorch [8] from Facebook, MXNet [5] from Apache, and DL4J [2] from Eclipse. Following the existing work [17, 26], we used the number of forks and the number of projects using the DL framework in GitHub to measure the popularity of a DL framework, which is used for DL framework selection. We found that Top-6 DL frameworks are the same, i.e., TensorFlow, PyTorch, Keras, Caffe, MXNet, and DL4J, regardless of the used metrics. Although Caffe [11] and Keras [12] are more popular than MXNet and DL4J in terms of both metrics, Caffe has stopped its update for a relatively long time (i.e., more than 12 months before the accessing date in October, 2021) while Keras is only a front end and has to run on top of some other DL frameworks (e.g., TensorFlow). More specifically, Keras can be considered as a "User-Level API" in the general five-level architecture. In

Table 1. Statistical Information on Our Dataset

Framework	#SLOC	#PR	#Bug	Duration	#Fork	Language	Organization
TensorFlow	3,090,623	319	250	2020/08-2021/10	85.3k	C++, Python	Google
PyTorch	1,762,228	307	250	2018/06-2021/10	13.5k	C++, Python	Facebook
MXNet	455,363	359	250	2020/03-2021/10	6.9k	C++, Python	Apache
DL4J	1,041,927	265	250	2018/12-2021/10	4.9k	Java, C++	Eclipse
Total	6,320,141	1,250	1,000	2018/06-2021/10	110.6k	-	-

actual, our work studied more comprehensive DL frameworks in order to sufficiently understand DL framework bugs at various levels. Hence, we did not use them in our study. All the four DL frameworks are built with the above five-level architecture, but they are also diverse, e.g., involving different programming languages for implementations, different development organizations, and different types of computational graphs.

Since our study aims to investigate the characteristics of DL framework bugs, we collected *closed* and *merged* pull requests that are responsible to fix bugs from the corresponding GitHub repositories of the four DL frameworks following the existing work [22, 32, 52]. On the one hand, the bugs involved in these pull requests have been accepted and fixed by developers; On the other hand, these pull requests tend to contain more comprehensive information, e.g., code changes, links to related issues, and discussions among developers, which is helpful to understand the bugs. In fact, not all of such pull requests are responsible to fix bugs, e.g., some of them aim to add new features or update documents. Hence, we further identified *bug-fixing pull requests* through keyword searching in the tags and titles of pull requests. Following the existing work[22, 32, 52], we adopted several bug-relevant keywords, including *fix, defect, error, bug, issue, mistake, correct, fault*, and *flaw*.

It is quite time-consuming to manually analyze bugs, and thus it is unaffordable for us to collect all bugs for manual inspection. Following the existing studies [19, 32, 34], we collected bugs in a specific duration (i.e., 2018/06 - 2021/10 in our study). However, different DL frameworks have different numbers of bugs within the same period, which may affect the analysis and conclusions across different DL frameworks, and thus we further balanced the number of studied bugs for each DL framework by selecting the same number of bugs for them. Specifically, among the four DL frameworks, PyTorch has the smallest number of bug-fixing pull requests within this period, and thus we first manually analyzed all these pull requests and finally obtained 250 bugs for PyTorch. The detailed process for manual analysis will be presented in Section 3.2. Then, for each of the other three DL frameworks, we analyzed the bug-fixing pull requests in the reversed chronological order within this period like the existing study [49], until the same number of bugs as PyTorch (i.e., 250) were identified. That is, we did not manually analyze all the collected pull requests for each of the other three DL frameworks (except PyTorch), but analyzed a subset of bug-fixing pull requests corresponding to the identified 250 bugs. More recent bugs may be more relevant to the characteristics of the current versions of DL frameworks. Specifically, DL is a fast-growing area and thus the characteristics of DL frameworks tend to be frequently updated, such as incorporating the rapid advancement in DL. Therefore, paying more attention to recent bugs could be more helpful to improve the current DL frameworks. Meanwhile, the involved duration for the 250 bugs of each DL framework is at least 14 months (i.e., 2020/08 - 2021/10 for TensorFlow), which can also support the generalizability of our conclusions to a large extent. Please note that in our study each bug is uniquely different from the other 249 bugs for each DL framework. In general, developers merge a bug-fixing pull request after they carefully verify that the target bugs have been correctly fixed by

the pull request. Also, many bug-fixing pull requests have the related issue reports, and we have checked that all the related issue reports in our study are indeed different. We also regard it as a potential threat in our study since it is also possible that developers make mistakes when verifying the bugs fixed by a pull request, but this threat may be not serious according to the above analysis.

Table 1 shows the statistical information of our dataset, where each column presents the number of source lines of code (SLOC), the number of pull requests (PR) that were manually analyzed by us for obtaining the 250 bugs (rather than the total number of bug-fixing pull requests collected initially), the number of identified bugs (i.e., 250 for each DL framework), the duration for the identified bugs, the number of forks in the GitHub repository (accessed in October, 2021), the used major programming languages, and the development organization, respectively. In total, we collected 1,000 bugs from the four popular DL frameworks after manually analyzing 1,250 bug-fixing pull requests. The involved duration for the collected bugs ranges from 14 months to 40 months across different DL frameworks. To our best knowledge, our study is the most large-scale in this area. In particular, we have released our dataset at our project homepage: **https://github.com/DLFrameworkBug/DLFrameworkBugsData**, to facilitate the replication of our study and promote the future research in this area.

3.2 Classification and Labeling Process

In the study, for each bug, we labeled its *root cause*, the *symptom* that the bug exhibits, the *stage* of the DL pipeline (to be introduced in Section 4.2.2) in which the bug symptom is observed, and the *level* of the DL framework in which the bug occurs. To label the root cause and the symptom of each bug, we first adopted the general taxonomies of root causes and symptoms from the existing work [22, 31, 32, 55, 56] as the *initial* taxonomies, and then adapted them to DL framework bugs. Specifically, following the general open-coding scheme [43], two authors went through all the pull requests to determine the root-cause and symptom categories of our collected DL framework bugs based on the initial general taxonomies by adding DL-framework-specific categories (e.g., Environment Incompatibility) on demand and removing irrelevant categories.

Regarding the levels of DL frameworks, we have introduced them in Section 2. To label the level of the DL framework in which each bug occurs, we divided all the source files of each DL framework into the five levels by understanding the functionality of each source file based on the source code, comments, and documents for the source file and the corresponding folder. This task was conducted by two authors together. We have also released our classification results for the source files of each DL framework at our project homepage to facilitate the replication of our study.

As mentioned above, based on the prepared root-cause and symptom categories as well as the classification results for the source files of each DL framework, two authors *independently* labeled each pull request via an open-coding scheme following the existing studies [32, 52]. Specifically, to label the level of the DL framework in which a bug occurs, the two authors identified the bug-fixing code changes in the pull request and labeled it according to which source files the bug-fixing code changes lie in. To label the other aspects, the two authors carefully understood the bug-fixing code changes, the descriptions about the bugs in the related issues, and the discussions among developers in the pull request. During the labeling process, we adopted the Cohen's Kappa coefficient [59] to measure the inter-rater agreement between them following the existing work [32, 52]. The Cohen's Kappa coefficient was just nearly 35% for the first 5% of labeling results, and thus we conducted a training session about labeling. Then, the Cohen's Kappa coefficient reached 80% for the first 10% of labeling results (including the first 5%). Through further discussion on these inconsistencies, the Cohen's Kappa coefficients were over 95% in all the subsequent labeling studies (i.e., labeling 20%~100% of bugs with the interval of 10%). For the inconsistencies in each labeling

study (including the training session), the two authors discussed with the third author until all the bugs were labeled consistently.

In particular, we provided an example to illustrate how a bug was manually labeled in more detail at our project homepage. During the careful labeling process, we came across slight inaccuracy in the prepared root-cause and symptom categories, and thus we further improved them to obtain better classification results. Moreover, we filtered out the pull requests that are actually irrelevant to bug fixing. There are some pull requests where more than one bugs were fixed, and we treated each of them as an individual bug following the existing work [22, 52].

4 RESULTS AND ANALYSIS

4.1 RQ1: Root Causes

4.1.1 Root Cause Classification Results. Based on the above classification and labeling process, we identified the following 13 root causes of DL framework bugs. The first four root causes involve the characteristics of DL frameworks (meaning that there is at least one sub-category specific to the characteristics of DL frameworks in each of the four root causes), while the others are common categories. Besides the four root causes involving the characteristics of DL frameworks, we also newly added the root cause of Dependent Module Issue over the general taxonomy of traditional software. This is because the number of this category of bugs is non-negligible for DL frameworks, and thus we did not treat these bugs as Others. The remaining categories were adapted from the general taxonomy of traditional software. In the study, we not only discussed these root causes very relevant to DL frameworks, but also investigated whether these common root causes have different distributions and characteristics between DL frameworks and traditional software.

① **Type Issue**. This kind of bugs involves type-related problems, such as type conversion and type checking. Different from traditional software, tensors are quite widely-used in the development of a DL framework. A tensor is a multi-dimensional matrix consisting of elements with some data type. According to this characteristic of DL frameworks, we divide this root cause into two sub-categories. 1) *Tensor type issue*, which refers to the bugs caused by the data types of tensors. 2) *Traditional data type issue*, which refers to the bugs caused by the types of traditional variables. For example, Figure 2(a) shows the patch[1] for an example bug belonging to this category. By default, the function `torch.zeros()` returns a matrix with type `float32`, while the expected type of the returned matrix should be consistent with that of the fed `input`.

②**Tensor Shape Misalignment**. This kind of bug occurs due to tensor shape mismatching in shape-related operations, e.g., tensor shape inference and transformation. Specifically, tensor shape describes the number of elements in each dimension. For example, when computing the *cosine* value between two tensors, their dimensions (i.e., tensor shapes) should be broadcastable to a common shape, but the buggy code requires that their original dimensions must be the same. The patch[2] in Figure 2(b) fixes it by calculating the common size of the two tensors.

③**Incorrect Algorithm Implementation**. This kind of bugs is caused by the problematic implementation logic (rather than the lack of implementation) of an algorithm, which tends to involve a number of statements or blocks. According to the functionality of an algorithm, we divide this root cause into two sub-categories. 1) *Incorrect DL-specific algorithm implementation*: there are a large number of algorithms with DL-specific functionalities in a DL framework (such as operation fusion and gradient computation algorithms), and this sub-category of bugs occurs due to the incorrect implementation logic of these algorithms. 2) *Incorrect DL-irrelevant algorithm implementation*: This sub-category of bugs occurs due to the incorrect implementation logic of the

[1]https://github.com/pytorch/pytorch/pull/8230
[2]https://github.com/pytorch/pytorch/pull/66214

```
  def make_jacobian(input, num_out):
     ……
-  return torch.zeros(input.nelement(),
-            num_out)
+   return torch.zeros(input.nelement(),
+            num_out, dtype=input.dtype)
```

(a) Type Issue (PyTorch#8230)

```
+ auto common_size = at::infer_size_dimvector(
+      x1.sizes(), x2.sizes());
- Tensor x1_ = x1.to(commonDtype);
- Tensor x2_ = x2.to(commonDtype);
+ Tensor x1_ = x1.to(commonDtype).expand(
+      common_size);
+ Tensor x2_ = x2.to(commonDtype).expand(
+      common_size);
```

(b) Tensor Shape Misalignment (PyTorch#66214)

```
 static std::unordered_map<NodeKind,
      std::vector<size_t>> broadcast_positions={
           {onnx::Gemm, {2}}, ……}
+ std::vector<size_t> positions;
    ……
    auto iter = broadcast_positions.find(
      node->kind());
-   return iter->second;
+   for (size_t position : iter->second) {
+     if (position < node->inputs().size()) {
+       positions.emplace_back(position);} }
+   return positions;
```

(c) Incorrect Algorithm Impl. (PyTorch#35416)

```
- lib = ctypes.CDLL(lib_path[0],
-            ctypes.RTLD_LOCAL)
+ if sys.version_info >= (3, 8) \
+            and os.name == "nt":
+   lib = ctypes.CDLL(lib_path[0],
+            winmode=0x00000008)
+ else:
+   lib = ctypes.CDLL(lib_path[0],
+            ctypes.RTLD_LOCAL)
```

(d) Environment Incompatibility (MXNet#19236)

```
- using Vec = Vectorized<scalar_t>;
+ using Vec = Vec256<scalar_t>;
```

(e) API Incompatibility (PyTorch#59008)

```
  jpeg_read_header(&cinfo, TRUE);
- jpeg_start_decompress(&cinfo);
+ jpeg_calc_output_dimensions(&cinfo);
```

(f) API Misuse (TensorFlow#44066)

```
  auto best_it = batches_.end();
- double best_score;
+ double best_score = (
+      std::numeric_limits<double>::max)();
```

(g) Incorrect Assignment (TensorFlow#42032)

```
  throw ErrorReport(lc.range())
-    << “iterator expression is expected to be"
-            "a list, iterable, or range, found"
-    << (siv?siv->
-            getValue()->type()->python_str()
-            : siv->kind());
+    << "iterator expression is expected to be"
+            " a list, iterable, or range, found"
+    << sv->kind();
```

(h) Incorrect Exception Handling (PyTorch#27398)

```
+ if(UNIX AND NOT APPLE)
+   set(LD_LIBRARY_PATH "LD_LIBRARY_PATH")
+ elseif(APPLE)
+   set(LD_LIBRARY_PATH "DYLD_LIBRARY_PATH")
      ……
- LD_LIBRARY_PATH=${CMAKE_CURRENT_BINARY_DIR}:
-      ${CMAKE_CURRENT_BINARY_DIR}
-      /3rdparty/tvm:$ENV{LD_LIBRARY_PATH}
+ ${LD_LIBRARY_PATH}=${CMAKE_CURRENT_BINARY_DIR}:
+      ${CMAKE_CURRENT_BINARY_DIR}
+      /3rdparty/tvm:$ENV{${LD_LIBRARY_PATH}}
```

(i) Misconfiguration (MXNet#20570)

```
 for (int dim = ndim-1; dim >=0; dim--) {
-  if (begin[dim]<0)
-    begin[dim] = shape[dim] - begin[dim];
-  if (end[dim]<0)
-    end[dim] = shape[dim] - end[dim];
+  if (begin[dim]<0)
+    begin[dim] = shape[dim] + begin[dim];
+  if (end[dim]<0)
+    end[dim] = shape[dim] + end[dim];
```

(j) Numerical Issue (MXNet#17937)

```
+ mutex_lock lock(mu_);
    if(mem != alloc_mem) {
      QCHECK(mem_map_.insert(
             {mem, alloc_mem}).second);}
```

(k) Concurrency Issue(TensorFlow#50382)

```
+ import static org.junit.Assert.assertEquals;
+ import static org.junit.Assert.assertFalse;
+ import static org.junit.Assert.assertTrue;
+ ……
```

(l) Dependent Module Issue (DL4J#9113)

Fig. 2. Bug Examples of Different Root Causes

algorithms with DL-irrelevant functionalities (such as memory allocation algorithms). Figure 2(c) shows a patch[3] from the method `getBroadcastPositions()` in PyTorch to fix an incorrect DL-specific algorithm implementation for model transformation. Specifically, when transforming `Torch.mm` to `Gemm` in ONNX, the output of the method should satisfy certain constraints (i.e., `position < node→inputs().size()`), while the buggy code missed the checking and produced incorrect results.

④**Environment Incompatibility**. This kind of bugs occurs due to neglecting some characteristics (e.g., the endianness for an architecture) of a specific environment (e.g., hardware or operating systems). For example, the lack of implementation for supporting the hardware is one reason for this kind of bugs. This root cause is common in DL frameworks since DL frameworks are required to work on various environments, such as CPU and GPU with various architectures. Figure 2(d) shows an example patch[4], which fixes the compatibility issue when deploying MXNet on different versions of the system.

⑤**API Incompatibility**. This kind of bugs contains two sub-categories. 1) *Internal incompatibility* refers to the API compatibility issues within a DL framework caused by API evolution; 2) *External incompatibility* refers to the API compatibility issues between a DL framework and third-party libraries (such as NumPy and HIPIFY) caused by the update of the latter. Please note that we did not consider the bugs in the third-party libraries depended by DL frameworks, but consider the bugs caused by the incompatibility between DL frameworks and the third-party libraries. This is because this latter kind of bugs are fixed by modifying the DL-framework code (rather than the third-party library code) to replace the invoked incompatible API with a compatible one. Taking the patch[5] shown in Figure 2(e) as an example, the version 1.9 of PyTorch changed the class `Vectorized` to `Vec256`, while the use of this class was not consistently updated, causing an API incompatibility issue.

⑥**API Misuse**. This kind of bugs occurs due to the misunderstanding of the APIs invoked by the DL framework, which contains three main sub-categories. 1) *Condition missing/redundancy* means that developers miss to add (or redundantly add) a condition check for an API; 2) *API missing/redundancy* means that developers miss to use (or redundantly use) an API; 3) *Wrong API* means that developers use a wrong name/argument/receiver of an API. By taking an API call `a.b(x,y)` as an example, `a` is the receiver, `b` is the API name, and `x` and `y` are the arguments. For example, developers misused APIs `jpeg_start_decompress` and `jpeg_calc_output_dimensions` as shown in Figure 2(f)[6], where the former API may cause over 50x performance decline on certain images due to the *decompression* operation in it. In fact, the incorrect implementation of an "iteration" over API calls is also a potential reason for this kind of bugs, but it does not occur in our study. This conclusion is almost consistent with the existing work [15], which shows that only 2 among 165 API misuses for traditional software are caused by this reason.

⑦**Incorrect Assignment**. This kind of bugs occurs when a variable is incorrectly assigned or lacks initialization, such as the example[7] shown in Figure 2(g), where the variable `score` was uninitialized before use.

⑧**Incorrect Exception Handling**. This kind of bugs occurs due to incorrect exception handling, which includes three scenarios: 1) *Missing exception*, i.e., a DL framework does not throw/handle an exception when it should; 2) *Spurious exception*, i.e., a DL framework throws an exception

[3]https://github.com/pytorch/pytorch/pull/35416
[4]https://github.com/apache/incubator-mxnet/pull/19236
[5]https://github.com/pytorch/pytorch/pull/59008
[6]https://github.com/tensorflow/tensorflow/pull/44066
[7]https://github.com/tensorflow/tensorflow/pull/42032

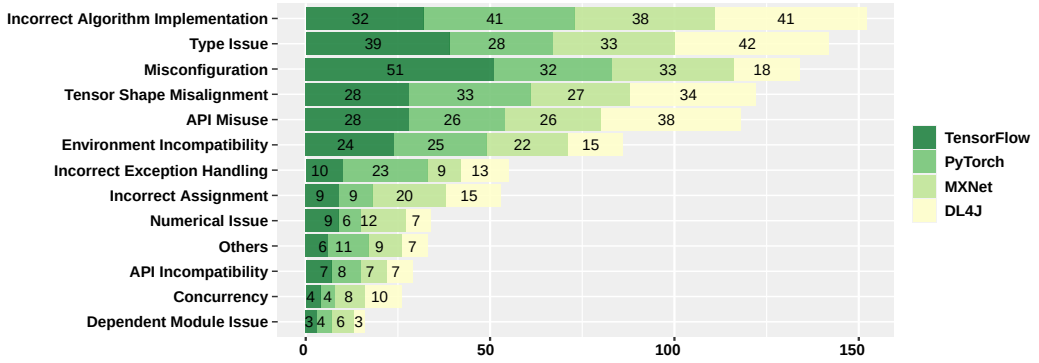


Fig. 3. Bug Distribution by Root Causes

when it should not; 3) *Wrong exception message*, i.e., a DL framework produces incorrect/imprecise exception messages for an exception, such as the example[8] shown in Figure 2(h).

⑨**Misconfiguration**. This kind of bugs is caused by incorrect configurations in a DL framework, such as configurations in Bazel files and various Shell configuration scripts. For example, the default path configuration of dynamic shared libraries is "LD_LIBRARY_PATH" for building MXNet, which works well on Linux systems. However, for MacOS, it will cause build failures since this path changes to "DYLD_LIBRARY_PATH". As a result, the configuration file should be updated for explicitly setting the path for different systems[9] (see Figure 2(i)).

⑩**Numerical Issue**. This kind of bugs is caused by incorrect numerical computations, such as dividing by 0, overflow/underflow, using wrong operators or operands e.g., a computation should use "+" but use "×" or a computation should be "i+1" but wrongly write as "i+2", and missing operands. For example, developers misused the operators "+" and "-" as shown in Figure 2(j), which caused the incorrect computation of batch size[10].

⑪**Concurrency Issue**. This kind of bugs is caused by incorrect operations on concurrency-oriented structures, such as threads, shared memory, and race conditions. For example, the variable `mem_map_` shown in Figure 2(k) records the allocated memory resources and can be updated by different threads[11]. Therefore, ensuring the mutually exclusive access to it is critical. To fix the bug, a `mutex_lock` was introduced.

⑫**Dependent Module Issue**. This kind of bugs occurs due to missing to import dependent modules or importing wrong modules, such as the example patch[12] for DL4J shown in Figure 2(l).

⑬**Others**. Each bug in this root cause is unusual and cannot be assigned to any other root causes.

4.1.2 Root Cause Distribution. Figure 3 shows the bug distribution by the identified root causes. From this figure, the four root causes involving the characteristics of DL frameworks (i.e., Incorrect Algorithm Implementation, Type Issue, Tensor Shape Misalignment, and Environment Incompatibility) are indeed prevalent, all of which are ranked within Top-6 (among 13 root causes) and account for 50.2% of bugs in total. Among all these root causes, Incorrect Algorithm Implementation is the

[8]https://github.com/pytorch/pytorch/pull/27398
[9]https://github.com/apache/incubator-mxnet/pull/20570
[10]https://github.com/apache/incubator-mxnet/pull/17937
[11]https://github.com/tensorflow/tensorflow/pull/50382
[12]https://github.com/deeplearning4j/deeplearning4j/pull/9113

most prevalent one. It accounts for 152 bugs in total, including 32 in TensorFlow, 41 in PyTorch, 38 in MXNet, and 41 in DL4J. The reason mainly lies in that deep learning is a fast-growing area and thus DL frameworks have to be frequently updated to incorporate the rapid advancement in DL algorithms. Moreover, hardware (especially DL-related hardware) is also rapidly developed and thus DL frameworks are required to provide the corresponding implementations to support these new features in hardware. Regardless of supporting advanced DL algorithms or new hardware features, the corresponding implementations in DL frameworks tend to involve complicated code logic, and thus it is very likely for them to incur various technical debts. Through further analysis, we found that about 79.61% of this kind of bugs (121 out of 152) occur in the implementations of DL-specific algorithms, significantly outnumbering the bugs in the implementations of DL-irrelevant algorithms.

Finding 1: Regarding the root causes involving DL framework characteristics, all of them are prevalent, accounting for 50.2% of bugs in total. Among them, the most prevalent root cause is Incorrect Algorithm Implementation (especially in DL-specific algorithms).

Type Issue is the second most prevalent one among all the root causes. It accounts for 142 bugs in total, including 39 in TensorFlow, 28 in PyTorch, 33 in MXNet, and 42 in DL4J. Through further investigation, nearly 70.42% of this kind of bugs (100 out of 142) are caused by tensor types rather than traditional data types. This is because all the DL operations depend on tensors, and meanwhile tensor type is an important property in a tensor and is usually involved in various operations. In particular, type conversion, especially implicit type conversion, tends to incur Type Issue bugs in DL frameworks, which deserves more attention in practice.

Finding 2: Type Issue is the second most prevalent root cause, which accounts for 14.20% of DL framework bugs and mainly occurs on tensor types.

In addition, there are common categories of root causes between DL frameworks and traditional software, and some of them are also notable. Besides the four root causes involving DL framework characteristics, the remaining two root causes ranked within Top-6 are Misconfiguration and API Misuse. In particular, Misconfiguration is the third most prevalent one among all the root causes, which accounts for 134 bugs in total. The phenomenon is different from the existing studies on traditional software bugs where Misconfiguration bugs either are ignored by them [54, 55] or account for only a small percentage among all the studied bugs [48, 56]. For example, as shown in the existing study [56], only 5.7% of bugs are caused by Misconfiguration in traditional machine learning systems, which is ranked at 9^{th} position among their identified 11 root causes.

The reason why DL frameworks contain many Misconfiguration bugs may lie in that, there are a large number of configuration files/options for compilation, installation, and ensuring compatibility of DL frameworks due to their complex implementations involving multiple programming languages as well as the large number of dependent third-party libraries and hardware/software environments. Through further analysis, we found that 22.39% (30 out of 134) bugs are caused by the configuration files/options for compilation, 41.04% (55 out of 134) are caused by the configuration files/options for installation, and 36.57% (49 out of 134) are caused by the configuration files/options for ensuring compatibility of DL frameworks.

API Misuse is another prevalent root cause for DL framework bugs. Indeed, this root cause is also common in traditional software, but it is unknown whether they manifest in the same way or not. To further investigate it, we then show its distribution in the three subcategories in

Table 2. Distribution of API Misuse Bugs

Framework	Condition Missing or Redundancy	API Missing or Redundancy	Wrong API				*Total*
			Receiver	Name	Args	*Sum*	
TensorFlow	3	4	4	7	10	21	28
PyTorch	3	5	2	9	7	18	26
MXNet	3	3	6	5	9	20	26
DL4J	4	9	4	13	8	25	38
Total	13	21	16	34	34	84	118

Table 2, following the existing work [15, 76]. From Table 2, 71.19% of API Misuse bugs (84 out of 118) are due to using wrong APIs, significantly outnumbering those caused by the other two subcategories. However, as demonstrated by the existing studies on MuBench [14, 15] (one of the most widely-studied benchmarks in the area of API misuse, including 90 API misuses from Java projects), API Missing/Redundancy is the most prevalent subcategory. That is, while API Misuse is a common root cause for both DL frameworks and traditional software, they manifest in a significantly different way. The result indicates that in DL frameworks, developers may usually confuse different API usage scenarios, especially for a set of similar APIs, calling for new API misuse detection methods that can clearly distinguish those similar APIs.

Finding 3: Regarding the common categories between DL frameworks and traditional software, Misconfiguration and API Misuse are two most notable root causes, but those bugs in DL frameworks have different characteristics and distributions with those in traditional software, calling for different bug detection strategies.

4.2 RQ2: Symptoms

4.2.1 Symptom Classification Results. According to the aforementioned classification and labeling process, we identified the following 6 symptoms of DL framework bugs.

①**Crash**. This symptom refers to that a DL framework terminates unexpectedly during running, such as terminating with an error message like "out of memory" or "null pointer". For example, the bug shown in Figure 2(b) made PyTorch crash during computing *cosine* similarity due to inconsistent tensor shapes.

②**Incorrect Functionality**. This symptom refers to that a DL framework behaves incorrectly but does not crash, such as producing unexpected prediction results, unexpected model structures, or incorrect intermediate states. Figure 4(a) presents such an example, which produces incorrect intermediate results due to an unsafe pointer conversion to `uint64_t` since GCC prevents using addresses at 0x80000000 or above[13].

③**Build Failure**. This symptom refers to that a DL framework fails to be installed, such as the previously introduced example shown in Figure 2(i).

④**Poor Performance**. This symptom refers to that the spent time or consumed resource (such as memory) is much larger than expectation during the usage of a DL framework. As mentioned above, the API misuse shown in Figure 2(f) may cause about 50x decline of running efficiency.

⑤**Hang**. This symptom refers to that a DL program written on top of a DL framework cannot terminate within a long period of time (due to a DL framework bug). Figure 4(b) presents such

[13]https://github.com/tensorflow/tensorflow/pull/46509

```
- base_addrs[num_tensors] =
-       reinterpret_cast<uint64_t>(
-             tensor->data.uint8);
+ base_addrs[num_tensors] =
+       static_cast<uint64_t>(
+             reinterpret_cast<uintptr_t>(
+                   tensor->data.uint8));
```

(a) Incorrect Functionality (TensorFlow#46509)

```
+ void TRTInt8Calibrator::setDone() {
+       done_ = true;}
    ……
    if (trt_builder->platformHasFastInt8()) {
        ……
    }else{
+       calibrator->setDone();
        calibrator = nullptr;}
```

(b) Hang (MXNet#19349)

Fig. 4. Bug Examples of Different Symptoms

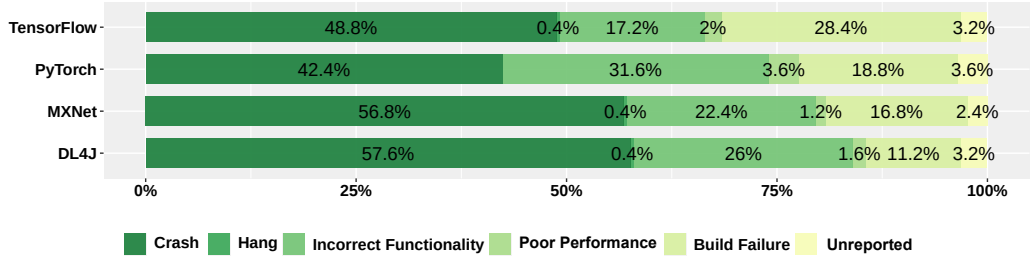


Fig. 5. Bug Distribution by Symptoms

an example from MXNet[14]. Specifically, before nullifying the pointer `calibrator`, the flag `done_` denoting the finish of the process was not properly set, causing that it cannot correctly terminate.

⑥**Unreported**. We cannot identify the symptoms for some bugs after carefully reading the corresponding pull requests, including the related issues, discussions, and code changes.

4.2.2 Symptom Distribution. Figure 5 shows the bug distribution of the identified symptoms. We found that Crash is the most common symptom. The number of bugs exhibiting this symptom is 122, 106, 142, and 144 for TensorFlow, PyTorch, MXNet, and DL4J respectively, and the total number is 514. The detection of this kind of bugs has an explicit test oracle, and thus automated test input generation (that tends to suffer from the test oracle problem but does not here) has a great potential to facilitate the detection of the large percentage of Crash bugs. Also, Crash bugs occur with error messages, which can provide hints for the bugs, and thus designing effective debugging techniques based on those informative messages is beneficial for such a large percentage of Crash bugs.

> **Finding 4**: Crash is the most common symptom for DL framework bugs, which accounts for 51.40% of bugs.

Incorrect Functionality takes the second place, which accounts for 243 bugs in total, including 43 in TensorFlow, 79 in PyTorch, 56 in MXNet, and 65 in DL4J. The major challenge for detecting this kind of bugs lies in the test oracle problem since the symptom is not as obvious as Crash. Specifically, a DL framework is used by developers for implementing a DL program and then building a DL model, but it is difficult to determine the correctness of the DL program/model due to its complexity. Hence, the adverse impact of Incorrect Functionality bugs is severe due to the weak observability. Through

[14]https://github.com/apache/incubator-mxnet/pull/19349

Table 3. Bug Distribution by Symptoms in Each Stage.

Symptoms \ Stages	Installation	Preprocessing	Training	Deployment	Utility Operation	*Total*
Crash	2	20	350	65	77	514
Incorrect Functionality	4	9	154	28	48	243
Build Failure	166	0	13	5	4	188
Poor Performance	1	2	14	2	2	21
Hang	0	0	2	1	0	3
Unreported	0	1	18	9	3	31
Total	173	32	551	110	134	1,000

analyzing the large number of studied Incorrect Functionality bugs, we found that they were often detected by checking whether the prediction results of the built model, the model structure, or some intermediate states (e.g., the calculation results of some operations) are as expected. More specifically, among the 243 bugs, 115 of them produced incorrect intermediate states, 105 of them produced incorrect prediction results, and 23 of them caused incorrect model structures. The 115 bugs producing incorrect intermediate states include 18 in TensorFlow, 46 in PyTorch, 17 in MXNet, and 34 in DL4J. We further investigated the possible reason why PyTorch produces more incorrect intermediate state bugs, and found that most of such bugs were introduced for optimizing the ONNX-related component in PyTorch, which involves many parameters and thus may be easy to be incorrectly modified. Since most of Incorrect Functionality bugs in PyTorch produce incorrect intermediate states, this also causes that PyTorch has more Incorrect Functionality bugs than the other three DL frameworks. Therefore, deciding which information should be observed and how to determine its expected result are important but indeed challenging to effectively detect this kind of bugs.

Finding 5: Incorrect Functionality is the second most common symptom for DL framework bugs, accounting for 24.30% of bugs. Defining effective test oracles deserves much more attention for the detection of this kind of bugs.

In addition, both Poor Performance and Hang are rare for DL framework bugs, indicating that functional bugs are generally more common than performance-related bugs for DL frameworks. It is also consistent with the existing studies on bugs of other software [19, 22, 32, 52]. Besides, there are some DL framework bugs whose symptoms are not provided in the pull requests (including the related issue reports). This is also as expected, because different developers may have different styles to describe bugs.

Based on the symptoms, we then analyzed when we can observe these bugs. From *the view of DL framework users*, we classified the DL pipeline into five stages following the existing work [20, 28, 32]: ①**Installation**: the stage of installing the DL framework; ②**Preprocessing**: the stage of preprocessing the dataset used for model building; ③**Training**: the stage of training and validating a model; ④**Deployment**: the stage of deploying the built model to a device; ⑤**Utility Operation**: the stage of conducting auxiliary operations, e.g., model visualization. To determine the stage in which a bug can be observed, we carefully understood the description about the bug in the related issue, the discussion among developers in the pull request, and the bug-fixing code changes. Table 3 shows the bug distribution according to the stage in which the bugs with each symptom were observed. From the view of DL framework users, DL framework bugs are mainly observed at the stage of Training (i.e., 55.10%). The Training stage tends to be time-consuming due to heavy numerical computation based on a large amount of training data. As presented in the existing

study [71], the typical training time ranges from a few minutes to several days. Hence, the bugs observed at this stage, especially those Incorrect Functionality bugs (account for 27.95% of bugs observed at this stage), may be manifested after hours or even days into the training process. This is very harmful to the efficiency of both testing and debugging. In particular, the training process for exposing the bugs has to be repeated several times to validate whether a fix is correct, and meanwhile the training process involves randomness that further aggravates the difficulty of bug reproduction [71]. Hence, the large percentage of bugs observed at the Training stage suggests the urgent need of speeding up the process of exposing bugs at this stage.

Finding 6: About 55.10% of DL framework bugs are observed at the Training stage. It could lead to lengthy testing and debugging for them, especially the large number of Incorrect Functionality bugs without halfway crashes, due to the costly and non-deterministic training process.

4.3 RQ3: Relationship between Root Causes and Symptoms

Table 4 presents the number of each kind of bugs (including each sub-category of a root cause) exhibiting each symptom. Here, Crash and Incorrect Functionality are the most common symptoms for all the root causes (except Misconfiguration for both, and Environment Incompatibility, API Incompatibility, Dependent Module Issue for the latter). The result indicates designing effective test oracles targeting the two symptoms is helpful to detect a wide variety of DL framework bugs. As presented before, Crash has an explicit test oracle, while the test oracle problem is the major challenge for detecting Incorrect Functionality bugs. Currently, differential testing has been adopted as the test oracle for the latter [50, 64], but it could lead to false positives and false negatives due to the *randomness* in DL (which is different from traditional software). Hence, more precise test oracles are still desirable.

Intuitively, Type Issue and Tensor Shape Misalignment usually lead to Crash rather than Incorrect Functionality. However, from Table 4, the two root causes can also lead to many Incorrect Functionality bugs. For better understanding this phenomenon, we thus present two examples. The first one is a TensorFlow bug caused by Type Issue[15] (shown in Figure 4(a)). This bug occurs due to unsafe type conversion from pointer to `uint64_t` in Ethos-U kernel, since GCC prevents using addresses at 0x80000000 or above. Such unsafe conversion causes that the value becomes a wrong value, and thus leads to Incorrect Functionality. The patch is to first convert the pointer to `uintptr_t` and then convert `uintptr_t` to `uint64_t`. The second example is a PyTorch bug caused by Tensor Shape Misalignment[16]. This bug occurs due to performing Conv2d non-zero padding in wrong dimensions, leading to wrong data values for subsequent calculation. Hence, it also leads to Incorrect Functionality.

Regarding the symptoms of Build Failure and Poor Performance, they can be produced by some specific root causes. Specifically, among the 188 bugs exhibiting the symptom of Build Failure, 65.43% are produced by Misconfiguration and 13.30% are produced by Environment Incompatibility. Among the 21 bugs exhibiting the symptom of Poor Performance, 52.38% are produced by Incorrect Algorithm Implementation or API Misuse. Among the 6 Poor Performance bugs caused by API Misuse, all of them are caused by *Wrong API*. In particular, Figure 2(f) shows an example for further illustrating how Wrong API leads to Poor Performance. Therefore, when a bug occurs with the two

[15]https://github.com/tensorflow/tensorflow/pull/46509
[16]https://github.com/pytorch/pytorch/pull/38583

Table 4. Bug Distribution by Symptom for Each Sub-Root-Cause

Root Cause \ Symptom		Crash	Incorrect Functionality	Build Failure	Poor Performance	Hang	Unreported	Total
IAI	DL-related	75	32	3	4	0	7	152
	DL-unrelated	19	8	0	1	0	3	
TI	Tensor Type Issue	68	23	1	2	0	6	142
	Conventional Type Issue	27	9	3	1	2	0	
MC	Misconfiguration	8	3	123	0	0	0	134
TSM	Tensor Shape Misalignment	80	39	1	0	0	2	122
AM	Condition M/R	6	5	2	0	0	0	118
	API M/R	13	7	0	0	0	1	
	Wrong API Args	18	9	1	1	0	5	
	Wrong API Name	14	11	3	3	0	3	
	Wrong API Receiver	9	4	1	2	0	0	
EI	Environment Incompatibility	49	8	25	1	1	2	86
IEH	Missing Exception	12	9	0	0	0	0	55
	Spurious Exception	2	1	0	0	0	0	
	Wrong Exception Message	18	13	0	0	0	0	
IA	Incorrect Assignment	27	22	3	1	0	0	53
NI	Numerical Issue	11	21	1	1	0	0	34
Others	Others	14	9	8	1	0	1	33
AI	External Incompatibility	10	3	7	1	0	0	29
	Internal Incompatibility	7	1	0	0	0	0	
CI	Concurrency Issue	17	6	0	2	0	1	26
DMI	Dependent Module Issue	10	0	6	0	0	0	16
Total		514	243	188	21	3	31	1000

IAI: Incorrect Algorithm Implementation; **TI**: Type Issue; **MC**: Misconfiguration; **TSM**: Tensor Shape Misalignment; **AM**: API Misuse; **Condition M/R**: Condition missing or redundancy; **API M/R**: API missing or redundancy; **EI**: Environment Incompatibility; **IEH**: Incorrect Exception Handling; **IA**: Incorrect Assignment; **NI**: Numerical Issue; **AI**: API Incompatibility; **CI**: Concurrency Issue; **DMI**: Dependent Module Issue

symptoms, developers can first check these highly relevant root causes to speed up the debugging process.

> **Finding 7**: The symptom of Build Failure is highly relevant to the root causes of Misconfiguration and Environment Incompatibility, while the symptom of Poor Performance is highly relevant to the root causes of Incorrect Algorithm Implementation and API Misuse.

4.4 RQ4: Bug-Occurring Levels

We analyzed the distribution of each kind of bugs over different levels of DL frameworks in Table 5. From this table, the number of bugs occurring at the level of Operation Implementation (i.e., 260) is the largest. It is reasonable since this level includes hundreds or even thousands of algorithms, which implement the complicated functionalities of DL models (e.g., gradient calculation), and always involve a significant amount of source code. In contrast, the number of bugs occurring at the level of Environment-Dependent Processing is the smallest since this level involves the least amount of code. Also, among the five levels, Graph-Level Implementation and Operation Implementation are more core than the other three, since both the training process and the inference process in DL are based on a static or dynamic computational graph and each node in the graph is an operation. From Table 5, 435 bugs occur at the two levels. The results indicate that it is urgent to conduct more extensive testing on the level of Operation Implementation (such as improving its test coverage) in terms of bug detection, in order to sufficiently improve the reliability of DL frameworks.

> **Finding 8**: The level of Operation Implementation contains the most bugs, accounting for 30.77% of the bugs, while the level of Environment-Dependent Processing contains the fewest bugs.

Table 5. Bug Distribution by Root Causes in Levels

Components \ Root Causes	IAI	TI	TSM	AM	EI	IEH	IA	NI	Others	AI	CI	DMI	*Total*
User-Level API	29	35	16	41	7	17	17	4	8	8	6	5	193
Graph-Level API	51	21	31	20	15	9	7	4	5	3	7	2	175
Operation Implementation	50	43	47	27	18	14	19	16	11	3	8	4	260
General Implementation	18	37	28	24	8	14	7	9	3	5	4	1	158
Environment-Dependent Processing	4	6	0	6	34	1	3	0	3	1	1	0	59

* **IAI**: Incorrect Algorithm Implementation; **TI**: Type Issue; **TSM**: Tensor Shape Misalignment; **AM**: API Misuse; **EI**: Environment Incompatibility; **IEH**: Incorrect Exception Handling; **IA**: Incorrect Assignment; **NI**: Numerical Issue; **AI**: API Incompatibility; **CI**: Concurrency Issue; **DMI**: Dependent Module Issue

* We excluded Misconfiguration bugs and the bugs caused by external configuration files in this table, since our five-level architecture of DL frameworks does not contain configuration files and almost all the Misconfiguration bugs occur in configuration files.

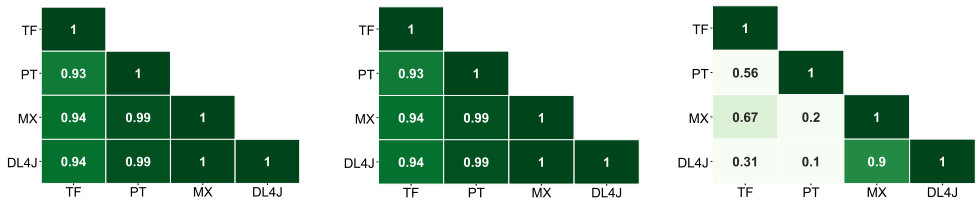
* We added the underline for the root causes involving DL framework characteristics.

Further, the bugs caused by the four root causes involving DL framework characteristics (i.e., IAI, TI, TSM and EI in Table 5) are chiefly distributed at the level of Operation Implementation, while the common root-cause categories of bugs are chiefly distributed at the level of User-Level API. The results indicate that the level of User-Level API is more similar to traditional software and the level of Operation Implementation is more specific to DL frameworks. Therefore, it is likely to apply existing testing and debugging techniques to the former level, which may facilitate to ensure its quality to a large degree, while new testing and debugging techniques targeting DL framework characteristics are desirable for the latter level. Regarding the level of Graph-Level Implementation, Incorrect Algorithm Implementation (IAI) and Tensor Shape Misalignment (TSM) are two major causes, since it involves many DL-specific algorithms (such as various graph transformation algorithms) and takes tensors as the basic elements of computational graphs. In particular, 46.85% (82 out of 175) bugs in this level occur in Graph Transformation. As expected, Environment Incompatibility is the major cause for the bugs at the level of Environment-Dependent Processing. Such different bug distribution characteristics in terms of root causes call for different testing and debugging techniques for different levels.

Finding 9: Different levels of DL frameworks involve different major root causes. DL-specific bugs are chiefly distributed at the level of Operation Implementation, while traditional categories of bugs are chiefly distributed at the level of User-Level API.

4.5 RQ5: Bug Commonality

In order to measure the bug commonality across different DL frameworks, following the existing work [41, 82], we calculated the Spearman correlation between each pair of DL frameworks in terms of bug distributions from the perspectives of root causes, symptoms and components respectively. Spearman's correlation coefficient is a statistical measure of the strength of a monotonic relationship between two paired variables [74]. Figure 6 shows the correlation results, where [0.8,1.0] indicates the very strong correlation, [0.6,0.8) indicates the strong correlation, [0.4, 0.6) indicates the moderate correlation, while [0, 0.4) indicates the weak or no correlation. From this figure, when considering the bug distributions over different root causes and symptoms, all the correlation coefficients are larger than 0.8. The results show that regardless of root causes or symptoms, the four DL frameworks share a high degree of the commonality, demonstrating the generality of our findings in the study and the potential of developing general testing and debugging techniques for various



(a) Symptom (b) Root Cause (c) Component

Fig. 6. Spearman Correlation across DL Frameworks

DL frameworks. For example, from Figure 3, Top-5 root causes for all the four DL frameworks are the same, i.e., Incorrect Algorithm Implementation, Type Issue, Misconfiguration, Tensor Shape Misalignment, and API Misuse. Also, from Figure 5, Crash and Incorrect Functionality are more common than the other three symptoms for all the four DL frameworks. However, these frameworks tend to display diverse bug distributions over different components as shown in the figure. The major reason lies in the different designs and implementations of DL frameworks. For example, DL4J implements many User-Level APIs to bridge the gap between different frameworks (e.g., it can load models trained in Keras) so as to be compatible with the others, while PyTorch takes many efforts for constructing a uniform model structure (e.g., ONNX) for convenient optimization. Please note that it does not contradict with the conclusions in Findings 9 and 10. Through a more fine-grained analysis and comparison, although the four DL frameworks do not consistently share the same bug distribution over different components, the bugs in Operation Implementation level still take a significantly large portion for each individual DL framework. Particularly, MXNet shares considerable commonalities (with at least moderate correlation) with other frameworks except for PyTorch. In summary, the results suggest that future research aiming to detect DL-framework bugs should consider the commonalities of bug symptoms and root causes so as to make the approaches have better generalizability over different frameworks.

Finding 10: There is a significant commonality among the four DL frameworks in both root causes and symptoms.

5 IMPLICATIONS AND THE APPLICATION

In this section, we explain the implications we have learned from our study to facilitate future research on DL framework bugs. In order to provide a more intuitive an targeted analysis, we have conducted an empirical study to investigate the status of existing testing techniques. In the end, based on the implications, we have developed a prototype testing tool aiming at finding DL framework bugs, and evaluated its effectiveness in a preliminary experiment.

5.1 Status of Existing Testing Techniques

To investigate the status of existing testing techniques, we analyzed them in terms of test coverage on each DL-framework component. Here, we studied three typical DL framework testing techniques, i.e., CRADLE [50], LEMON [64], Audee [25]. There are also some other DL framework testing techniques and more details about them can be found in Section 7. Also, the selection of studied techniques may be a threat in this experiment, which will be discussed in Section 6. All of the

studied techniques adopt differential testing as the test oracle. Their main difference lies in the used test inputs: CRADLE is the first technique, which takes real-world pre-trained DL models as test inputs; LEMON and Audee adopt different search-based mutation strategies to generate mutated models based on pre-trained models as test inputs, where the former proposes to mutate the layers, neurons, and weights of pre-trained models and the latter proposes to mutate the parameters of layers, weights, and inputs (e.g., images).

Here, we used 8 pre-trained models widely-used in the existing studies [25, 64], involving different model structures and different sets of input data. They are LeNet-5 trained on MNIST, LeNet-5 trained on Fashion-MNIST, AlexNet trained on Cifar10, MobileNetV2, ResNet-50, and VGG-16 trained on ImageNet, and two LSTM models trained on Sinewave and Price. CRADLE uses the 8 models as test inputs directly, while LEMON and Audee produced 100 mutated models based on each pre-trained model respectively and the latter also produced mutated input data for each mutated model. In total, there are 800 mutated models as test inputs for LEMON and Audee, respectively. We measured the achieved test coverage (i.e., line, branch, and function coverage) by the three techniques respectively via *Gcov* (for C code coverage collection) [3] and *Coverage.py* (for Python code coverage collection) [1]. Since collecting DL framework coverage is costly, we used MXNet and PyTorch as the representatives in this experiment. MXNet, TensorFlow, and DL4J share a significant bug commonality as presented in Section 4.5, and thus we used one of them (i.e., MXNet) as the representative[17]. To ensure the generality of conclusions, we also used PyTorch as another subject in this experiment. In particular, we also ran the equipped test suites in MXNet (version 1.9.0) and PyTorch (version 1.9.0) and collected the achieved coverage to facilitate analysis.

We first measured the coverage results achieved by the three testing techniques together, and also compared them with the coverage result achieved by the equipped test suite, whose results are shown in Table 6. We found that the line, branch, and function coverage achieved by these testing techniques are only 24.19%, 4.58% , 23.38% on MXNet and 10.12%, 1.30%, 7.11% on PyTorch respectively, which are significantly smaller than those achieved by the equipped test suite (i.e., 70.51%, 11.88%, , 38.99% on MXNet and 64.34%, 23.60%, 49.81% on PyTorch). That is, the studied testing techniques suffer from the low test coverage issue. It is very harmful to the testing performance since test coverage is the first condition of bug detection according to the PIE theory [60]. Hence, improving test coverage is an important direction of designing new DL framework testing techniques. Also, from Table 6, the studied testing techniques tend to achieve relatively high test coverage on the components of User-Level API and Graph-Level Implementation (especially function coverage on the User-Level API component) compared with the remaining three components. The results suggest that focusing on the remaining three components could be more helpful to improve test coverage.

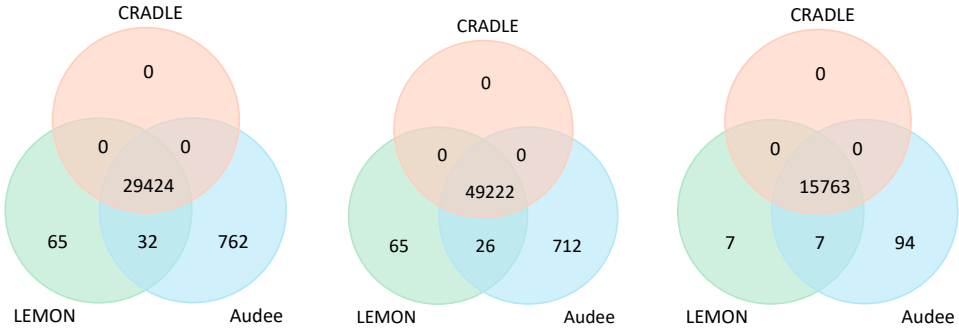
Finding 11: The studied DL framework testing techniques suffer from the low test coverage issue, especially on the components of Operation Implementation, General Utility, and Environment-Dependent Processing.

We then compared the test coverage achieved by each of the three testing techniques. Figure 7 and Figure 8 show the Venn diagrams to analyze the overlaps of their covered lines, branches, and functions. We found that the number of unique lines, branches, and functions covered by

[17]Handling various environment/configuration/dependency issues for collecting code coverage of DL frameworks is challenging. We tried our best to handle these issues for collecting TensorFlow's code coverage in order to make the experiments in Sections 5.1 and 5.3 use the same DL framework (i.e., TensorFlow) as the subject, but unfortunately failed due to a version incompatibility bug regarding Bazel [10] and Gcov. In the future, we will try to solve this problem to obtain TensorFlow code coverage results for more sufficient evaluation.

Table 6. Coverage of Existing Testing Techniques and Equipped Test Suite

Framework	Component		User-Level API	Graph-Level Impl.	Operation Impl.	General Utility	Env.-Dep. Processing	*Overall*
MXNet	Test Suite	Line	72.78%	68.75%	72.56%	65.58%	39.42%	70.51%
		Branch	59.07%	29.72%	10.57%	17.11%	19.11%	11.88%
		Function	93.87%	62.23%	34.47%	51.54%	46.09%	38.99%
	CRADLE+ LEMON+ Audee	Line	30.22%	38.37%	20.29%	17.79%	13.00%	24.19%
		Branch	8.65%	18.00%	4.05%	3.81%	7.57%	4.58%
		Function	91.51%	37.54%	18.65%	20.56%	15.33%	23.38%
PyTorch	Test Suite	Line	64.43%	67.06%	70.51%	55.97%	48.25%	64.34%
		Branch	54.86%	21.82%	25.17%	24.00%	13.20%	23.60%
		Function	53.37%	69.32%	42.85%	44.16%	42.57%	49.81%
	CRADLE+ LEMON+ Audee	Line	15.04%	7.71%	4.39%	15.34%	2.96%	10.12%
		Branch	7.00%	0.90%	0.50%	3.43%	0.55%	1.30%
		Function	30.00%	5.95%	1.85%	5.03%	2.61%	7.11%



(a) Line coverage (b) Branch coverage (c) Function coverage

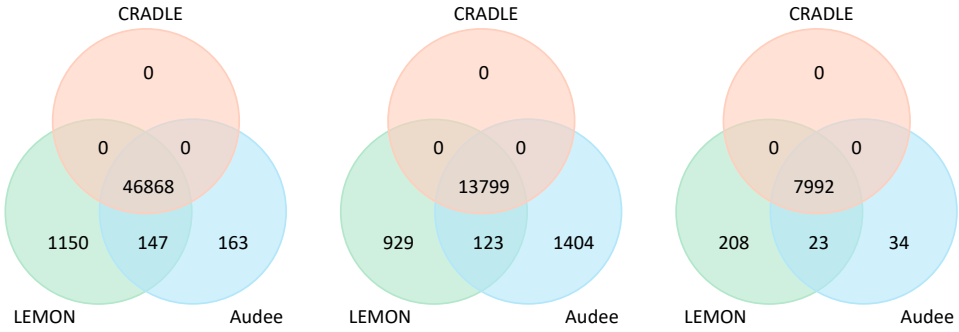
Fig. 7. Number of overlapping elements covered by different approaches on MXNet

each technique is small, especially compared with those that can be covered by all of them. The results indicate that these techniques have the significant commonality in terms of test coverage. In particular, both LEMON and Audee are on the basis of CRADLE, and according to Figure 7 and Figure 8 we found that their achieved test coverage mainly depends on the used pre-trained models and the coverage increments achieved by the mutated models by both LEMON and Audee are small. That means the diversity of the mutated models from the same pre-trained model is limited, and using more pre-trained models could increase the test diversity as well as the test coverage of the studied testing techniques. Moreover, it is necessary to design new techniques with great diversity compared with these existing ones.

Finding 12: The studied DL framework testing techniques share a significant commonality in terms of test coverage and their coverage mainly depends on the used pre-trained models rather than mutated models by LEMON and Audee.

5.2 Implications

According to the performance of existing state-of-the-art testing techniques (Section 5.1), there is still much room for improvement. By further combining our findings, in this section we provide a



(a) Line coverage (b) Branch coverage (c) Function coverage

Fig. 8. Number of overlapping elements covered by different approaches on PyTorch

series of actionable guidelines for future research on the detection and debugging of DL framework bugs.

New mutation operators. Based on Findings 1 and 2, DL-involving root causes can result in a large percentage of bugs, and thus defining new mutation operators specific to their characteristics is helpful to efficiently explore whether DL frameworks can handle various cases involving them correctly. New mutation operators can include: 1) *type mutation*: many Type Confusion bugs are caused by type conversion, especially implicit type conversion, and thus we can *add typecast for tensors* so that implicit type conversion may be triggered in tensor computation; 2) *shape mutation*: we can create the scenarios, in which various tensor shapes can be involved to check whether they match, by *inserting new layers with diverse shapes into different contexts*; 3) *environment mutation*: we can *put a DL program into various environments for model building*, to test whether the used DL framework can stably support the training process.

Test oracle improvement. Based on Findings 4, 5, 7, Crash and Incorrect Functionality are two most common symptoms for DL framework bugs, and thus designing effective test oracles with regard to them can cover a large percentage of bugs. Regarding Crash, it has an explicit test oracle with error messages, but we still found many bug reporters complained that the error messages are ambiguous, which could affect the follow-up debugging process. For example, among 55 bugs caused by Incorrect Exception Handling, 56.36% are due to *wrong exception messages*. Hence, it is necessary for developers to refine error messages to make them precise and informative. Regarding Incorrect Functionality, although differential testing on multiple DL frameworks has been adopted in the existing DL framework testing techniques by pre-defining a threshold for determining an inconsistency, it still cannot precisely identify Incorrect Functionality bugs due to inherent non-determinism in DL. To reduce false positives and false negatives, a voting mechanism can be incorporated by integrating several test oracles, including differential testing on multiple versions of one DL framework as well as multiple environments, and metamorphic testing by constructing a group of equivalent tests. Although integrating various test oracles may relieve the test oracle problem to some degree, new test oracles specific to such non-determinism definitely deserve more attention from the software engineering community.

Component-targeted testing. In general, it is challenging to design a general testing technique that can effectively detect bugs occurring at various components, which can be demonstrated by Finding 11 to some degree (i.e., all these general testing techniques suffer from the low test

Table 7. Mutation Operators supported in TenFuzz.

Mutation Operator	Brief Description
tensor type mutation	replacing the type of a tensor with another compatible type supported in TensorFlow.
tensor shape mutation	reshaping a tensor while keeping the data unchanged, e.g., changing the shape of a tensor from 3×4 to 2×6.
tensor structure mutation	changing the structure of a tensor, e.g., changing the `tensor` to `sparse_tensor` (that stores the non-zero values of the tensor and the corresponding coordinates of them) or `ragged_tensor` (that is the TensorFlow equivalent of nested variable-length lists).
tensor rotating	rotating a tensor with a random angle θ, where $\theta \in [30°, 270°]$ with the interval of 30°. Please note that we consider the tensors with the dimension no more than three.
parameter mutation	changing the value of an API parameter to a special value, including the negation of the parameter value, `0`, `NaN`, and the maximum/minimum of the parameter.

coverage issue, especially on some components). Hence, conducting component-targeted testing could be more practical. According to Findings 9 and 11, we can assign the component of Operation Implementation the highest priority for designing targeted testing techniques, because this component involves the largest number of bugs but has little test coverage regardless of using existing testing techniques or the equipped test suite. Further analysis shows that only a small portion of operators are indeed covered by the existing approaches. However, since there are hundreds and even thousands of operators in a DL framework, the current testing is definitely insufficient. Testing tools targeting this component are urgently desired due to its core importance in the framework. To achieve the targeted testing for the component of Operation Implementation, it is better to construct tests on the computational graph level, since it can more directly invoke and operate various operations to achieve high coverage compared with the widely-used model level by the existing testing techniques.

Efficient reproduction. Based on Finding 6, over 55.10% of bugs occur at the training process. Since the training process involves heavy numerical computation based on a large amount of training data and inherent non-determinism, bug reproduction is unstable and time-consuming. Therefore, efficient bug reproduction deserves much more attention. One promising direction may be to shorten the training process by simplifying the model structure and reducing the amount of training data, which can still trigger the bug but with higher efficiency. Here, adapting the idea of delta debugging [75] to both model and training data may be effective to achieve this goal.

Build failure fixing. Based on Findings 3 and 8, Build Failure is common and has two highly relevant root causes, i.e., Misconfiguration and Environment Incompatibility. Thus, there are hints for fixing building failures, making the design of automated methods feasible. Indeed, there are some automated build failure fixing methods proposed for traditional software [29, 41], but these methods tend to target the *gradle* build framework [4], which is different from the one depended by DL frameworks. Moreover, as shown in Finding 3, Misconfiguration bugs in DL frameworks have different characteristics with those in traditional software (i.e., there are a large number of configuration files/options for compilation, installation, and ensuring compatibility of DL frameworks due to their complex implementations involving multiple programming languages as well as the large number of dependent third-party libraries and hardware/software environments, leading to different proportions of Misconfiguration bugs). Hence, not only investigating whether the existing methods still work on build failures of DL frameworks is valuable, but also designing new methods specific to the characteristics of DL frameworks is necessary.

5.3 TenFuzz: A Preliminary Application

In this section, we demonstrate the usefulness of our findings with a preliminary proof-of-concept application TenFuzz, which aims to generate tests for TensorFlow by mutating its equipped unit tests. In the preliminary application, we selected TensorFlow as the representative due to its popularity. TenFuzz is designed based on two major findings: (1) Bugs caused by tensors (especially tensor types and tensor shapes) are common; Specifically, the number of Tensor Type bugs is 100 (24 in TensorFlow, 19 in PyTorch, 22 in MXNet, 35 in DL4J) and the number of Tensor Shape Misalignment bugs is 122 (28 in TensorFlow, 33 in PyTorch, 27 in MXNet, 34 in DL4J). (2) The studied DL framework testing techniques (by generating DL models) achieve less coverage than the equipped unit tests on all the five components. Hence, TenFuzz takes the equipped unit tests in TensorFlow as the initial pool of tests and designs five mutation operators on tensors (as presented in Table 7). These mutation operators can usually produce the exceptional cases of tensor type incompatibility or tensor shape misalignment, which can be helpful to more sufficiently test the ability of TensorFlow for handling these exceptional cases. Please note that for a given test, not all the mutation operators are applicable. This is because a test may not contain the elements required by each mutation operator (e.g., the mutation operator of negating the value of an API parameter cannot be applied when the test does not contain the APIs with numerical parameters).

During the testing process with TenFuzz, it first randomly selects a test from the pool and then performs static analysis to determine a set of applicable mutation operators for this test. Further, it randomly selects an applicable mutation operator and then applies it to the test for generating a new test. That is, a new test is a mutant from the original test. If the new test does not produce different outputs after normal executions on different versions of TensorFlow (we used differential testing on different versions of TensorFlow as the test oracle, which will be presented later), it will be put into the test pool for supporting high-order mutation. The testing process will terminate until the given time budget is reached. Indeed, it would be better for TenFuzz to perform deeper analysis to improve the possibility of generating bug-triggering tests. For example, it is more likely for the mutation operator of replacing the parameter value with 0 to generate a bug-triggering test by analyzing whether there is division operation inside the API. In the future, we will incorporate more analysis to further improve TenFuzz. Please note that TenFuzz generates tests based on the existing unit tests, while unit tests do not distinguish different stages and can test the APIs for various stages (including the training stage). Therefore, All the stages tested by the existing unit tests, can be also tested by TenFuzz.

Procedure: We conducted an experiment on TensorFlow to evaluate the effectiveness of TenFuzz. During the testing process, we applied each generated test by TenFuzz to test four TensorFlow versions, i.e., v2.5.0, v2.6.0, v2.7.0, and v2.8.0, respectively. That is, TenFuzz adopts cross-version differential testing to determine whether the test detects a bug. Initially, we collected the tests from the *python* folder in TensorFlow as the initial test pool for TenFuzz. In particular, some of these initial tests can produce inconsistent results on the four versions due to the version incompatibility, which can incur noise to the testing process with TenFuzz. Specifically, for each initial test, we ran it on the four versions of TensorFlow and recorded the results (i.e., the values of the variables observed by the assertions in the test), respectively. If the results from the four versions are different or some versions crash on the test, we regarded that the test triggers an inconsistency. Hence, we discarded them from the initial test pool. In total, there are 509 tests in the initial test pool. Here, we set the time budget as 24 hours for running TenFuzz. When a test makes some of these versions under test crash or produces different results on these versions, we regard it as a potential bug and report it to the developers for manual investigation. In particular, when the results from the four

versions are different, we determined the buggy version through the voting mechanism, which assumes that most of versions can produce correct results for the same test.

Results: In total, TenFuzz reported 9 tests that triggered potential bugs (producing inconsistent results on different versions). After manual inspection for identifying duplicates, TenFuzz detected 6 unique bugs, of which 3 bugs exist in the early releases and have been fixed in the latest version, while the other 3 bugs still exist in the latest version. Specifically, 4 bugs were triggered on all the four versions, one bug was triggered on v2.5.0, and one bug was triggered on v2.5.0 and v2.6.0. We have reported them to the TensorFlow developers on GitHub, all of which have been successfully reproduced and confirmed by the developers. Figure 9 shows a test generated by TenFuzz, which triggers a bug in the latest version of TensorFlow. Specifically, TenFuzz mutates the type of the variable `grads`, which represents the step size during gradient calculation with the Adadelta optimizer (i.e., `adadelta_opt` in the example). By replacing the type `float32` with `float16` as shown in Line 6, the TensorFlow crashed with the message of "Aborted (core dumped)" when it ran into Line 13 due to the exception-handling issue for the type incompatibility. This bug was confirmed once we submitted it to the TensorFlow developers. As expected before, TenFuzz can effectively detect exception-handling bugs due to the designed mutation operators. In the future, we will improve TenFuzz to detect bugs with more diverse root causes by designing more mutation operators or incorporating advanced static analysis to avoid the cases of tensor type incompatibility and tensor shape misalignment during test generation. We have also released all the detected bugs in our project homepage. In summary, the preliminary experimental results demonstrate the effectiveness of TenFuzz and the value of our implications. In addition, the results also indicate that more effective bug detection techniques are desired.

In particular, the idea of TenFuzz is general since (1) its designed mutation operators are applied to tensors or API parameters, which also exist in other DL frameworks; (2) it treats the equipped unit tests as the initial pool of tests, and indeed other DL frameworks also contain equipped unit tests. However, the current TenFuzz tool cannot be directly applied to other DL frameworks. This is because the equipped tests in different DL frameworks have different data structures and different formats, causing that the mutation operators implemented in TenFuzz (specific to the equipped unit tests in TensorFlow) cannot be used to mutate the equipped unit tests in other DL frameworks. Therefore, before applying TenFuzz to other DL frameworks, we need to modify the implementation of the mutation operators to make them applicable to the equipped unit tests in other DL frameworks according to the corresponding data structures and formats. Furthermore, we have reported the detected bugs by TenFuzz to the TensorFlow developers on GitHub, all of which have been successfully reproduced and confirmed by the developers. This indicates that these bugs were not detected by the existing testing tools before to some degree. Indeed, the exception-handling bugs detected by TenFuzz cannot be detected by the existing testing tools studied in Section 5.1 (i.e., CRADLE, LEMON, and Audee), since they cannot produce exceptional cases of tensor type incompatibility or tensor shape misalignment. In the future, we will try more testing tools to better understand the effectiveness difference among various testing tools.

6 THREATS TO VALIDITY

The *external threats to validity* mainly lie in our used data. We systematically collected 1000 bugs of four DL frameworks as our study data, including collecting closed and merged pull requests, identifying bug-fixing pull requests via keyword searching, and conducting manual investigation following the existing work [32, 52, 80]. Hence, a big confidence can be obtained regarding the high quality of our data. As the most large-scale study on DL framework bugs, the generalizability of our study can be demonstrated to a large extent. Please note that same as many existing studies [32, 52, 80], we cannot consider the importance of each studied bug, since the information

```
num_updates = 4
for grad in [0.2,0.1,0.01]:
  for lr in [1.0,0.5,0.1]:
    var0 = variables.Variable([1.0,2.0], dtype=dtypes.float32)
    var1 = variables.Variable([3.0,4.0], dtype=dtypes.float32)
    grads = constant_op.constant([grad,grad], dtype=dtypes.float16)  # mutate "float32" to "float16"
    adadelta_opt = adadelta.AdadeltaOptimizer(learning_rate=lr,rho=0.95,epsilon=1e-08)
    if (not context.executing_eagerly()):
      adadelta_update = adadelta_opt.apply_gradients(zip([grads,grads], [var0,var1]))
      slot = ([None]*2)
      slot_update = ([None]*2)
    for step in range(num_updates):
       adadelta_opt.apply_gradients(zip([grads,grads], [var0,var1]))   # Running crashed here
```

Fig. 9. An example test case generated by TenFuzz that triggers a bug

about bug importance is not provided in the corresponding GitHub repositories. In the future, we may study the DL framework bugs with the importance information to reduce this threat.

The *internal threats to validity* mainly lie in our manual labeling process. To mitigate the inaccuracy and subjectivity of each individual developer, two authors with over 4-year developing experience conducted the labeling process independently through the general open-coding scheme [37] following many of the existing studies [32, 52, 80]. We leveraged the Cohen's Kappa coefficient to measure the inter-rater agreement between them, where a coefficient as high as 95% is reached, indicating a high agreement between them. Besides, when coming across the inconsistencies between two authors in each labeling study, the two authors discussed with *the third author (a senior developer)* until the bugs were labeled consistently, which can help further improve the reliability of the labels. Indeed, during the training session, we also involved the third person for the discussion of inconsistencies, which can help reduce the risk that the two authors agree on the same (but wrong) thing.

The *construct threats* to validity mainly lie in the selection of our studied DL framework testing techniques in Section 5.1. Specifically, the findings (Findings 11 and 12) obtained based on the three studied techniques may not represent the other DL framework testing techniques. In the future, we will evaluate more techniques to further reduce this threat.

7 RELATED WORK

The most related work to ours is the empirical study on TensorFlow bugs [33, 34]. This study is the only one on investigating DL framework bugs, but it is not enough to comprehensively understand bugs in the family of DL frameworks: 1) It investigates the bugs in only one DL framework (i.e., TensorFlow), while our study analyzed 1000 bugs of four popular and diverse DL frameworks. That shows that our study is indeed large-scale and general (e.g., obtaining more general root-cause and symptom taxonomies), and facilitates the understanding of bugs across different DL frameworks. 2) It directly uses the folders organizing TensorFlow code as the component categories, which cannot be generalized to other DL frameworks. However, our work proposes a general top-down five-level architecture for DL frameworks, and analyzes root cause distribution on each component, which facilitates the more fine-grained understanding of DL framework bugs. 3) Our study involves more study points, including studying bugs from some individual aspects (e.g., root causes) as well as associating different aspects for comprehensive analysis (e.g., associating root causes with DL framework components). In particular, our study further associates our identified bug characteristics with existing testing and debugging practice for DL framework bugs, in order to dissect the current status in testing and debugging DL frameworks and then guide the direction of improving them.

Therefore, we believe that our study makes significantly novel contributions to understanding DL framework bugs comprehensively and further ensuring DL frameworks' quality by providing insightful guidelines for repairing DL related bugs [70] .

There are also some studies on investigating DL program bugs [31, 32, 45, 57, 69, 80]. As explained in Sections 1 and 2, DL programs refer to the programs used for training DL models, which are implemented by invoking the APIs provided by DL frameworks, while DL frameworks implement the functionalities of those APIs. Therefore, DL program bugs can be caused by incorrectly using the APIs provided by DL frameworks when implementing the DL programs, rather than the bugs inside the DL frameworks. The latter is the target of our work. Hence, in our study, there is no bug caused due to using incorrect versions of a DL framework. However, since we analyzed 250 bugs for each DL framework, these bugs can involve different versions of the DL framework. In particular, the distribution and characteristics of DL framework bugs and DL program bugs are largely different. For example, there are many DL framework bugs being triggered during the installation process. On the contrary, there is no such kind of bugs in DL programs. Similarly, the bug category of *Incorrect Model Parameter* in DL programs [80] does not appear in DL frameworks either. Furthermore, Garcia et al. [22] studied the bugs of autonomous vehicles, which is a kind of DL-based applications and lies in the production level. Shen et al. [52] conducted an empirical study on DL compilers (e.g., TVM). Nejadgholi and Yang [47] studied the oracle approximation assertions implemented in DL libraries. Different from them, our work conducted a comprehensive study on DL framework bugs by investigating 1000 bugs from four DL frameworks.

Besides the studies on investigating DL (program and framework) bugs, there are also many studies focusing on traditional software bugs in the literature [19, 27, 30, 40, 42, 48, 56, 63]. For example, Ocariza et al. [48] conducted a study on client-side JavaScript bugs. Lu et al. [42] investigated the characteristics of concurrency bugs, while Li et al. [40] conducted a large-scale study on API misuses in traditional Java programs. Different from them, our work targets DL framework bugs, which not only investigates the bug characteristics specific to DL frameworks, but also analyze the difference for the common bug characteristics (such as some common root causes) between DL frameworks and traditional software. In particular, a DL framework has the following typical characteristics, leading to four unique root causes compared with traditional software bugs. First, it is the fundamental infrastructure of DL, which supports the construction and usage of DL models. Therefore, many of DL framework bugs involve tensor types and tensor shapes (almost all the DL operations depend on tensors). Second, it is the bridge between DL functionalities and various hardware, which implements some strategies to support DL functionalities on different hardware. Therefore, many of DL framework bugs involve the environment incompatibility. Third, DL is still a fast-growing area and thus DL frameworks have to be frequently updated to incorporate the rapid advancement in DL algorithms. Therefore, many of DL framework bugs occur in the implementations of DL-specific algorithms.

Recently, many testing techniques for DL frameworks have been proposed, which can be mainly divided into two categories based on the data format during the generation of test cases. They are graph-level [24, 25, 50, 53, 62, 64] and operator-level (API-level) test generation [67, 77, 78] techniques. In the first category, Pham et al.[50] proposed CRADLE to detect DL framework bugs via differential testing. After that, LEMON [64] and Audee [25] were proposed and both of them generated tests via a set of mutation rules, but their mutation targets were different. Similarly, the latest work EAGLE [62] defined a set of equivalence transformation rules to facilitate this testing process. Besides, Muffin [24] aims at detecting DL framework bugs related to model training. It used a DAG-based algorithm to generate diverse DL models and measured the inconsistencies in training phase with multiple metrics. For the second category, Predoo [78], FreeFuzz [67], and DocTer[68] are three representative techniques. The major difference is the data used for test generation. Predoo

mutates the input tensor values in the original program, FreeFuzz leverages code snippets from different sources to aid the generation, while DocTer depends on the API constraints extracted from corresponding documents. In this paper, we conducted a preliminary study over the three typical DL framework testing techniques (i.e., CRADLE, LEMON, and AUDEE), based on which we provide some guidance for future research and propose a simple DL framework testing technique. In the future, we plan to perform a more systematic review and comparison over all the existing techniques, and further improve the performance of DL framework testing techniques.

8 CONCLUSION

In this work, we conducted the most large-scale empirical study on the characteristics (e.g., root causes, symptoms, and their correlations with DL-framework components) of DL framework bugs, where we manually analyzed 1,000 bugs from four popular DL frameworks. Through the comprehensive study and further analyzing the current status of existing DL framework testing techniques, we summarized 12 major findings, based on which we provided a series of actionable implications for future studies on the detection and debugging of DL framework bugs. Finally, on the basis of those implications, we have developed a prototype DL-framework testing tool, called TenFuzz, which was evaluated to be effective to find unknown DL framework bugs in a preliminary study, indicating the significance of those implications to guide feature research.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62002256, 62232001, and 62202324.

REFERENCES

[1] Accessed: 2021. Coverage.py. https://coverage.readthedocs.io/.
[2] Accessed: 2021. Deeplearning4J. https://deeplearning4j.org/.
[3] Accessed: 2021. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[4] Accessed: 2021. Gradle. https://gradle.org/.
[5] Accessed: 2021. MXNet. https://mxnet.apache.org.
[6] Accessed: 2021. News. https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash.
[7] Accessed: 2021. News. https://www.newsweek.com/autonomous-tesla-crashes-parked-fire-truck-california-freeway-789177.
[8] Accessed: 2021. PyTorch. https://pytorch.org.
[9] Accessed: 2021. TensorFlow. https://www.tensorflow.org.
[10] Accessed: 2022. Bazel. https://bazel.build/.
[11] Accessed: 2022. Caffe. https://github.com/intel/caffe.
[12] Accessed: 2022. Keras. https://github.com/keras-team/keras.
[13] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[14] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 464–467.
[15] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.
[16] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.
[17] Fangwei Chen, Lei Li, Jing Jiang, and Li Zhang. 2014. Predicting the Number of Forks for Open Source Software Project. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies* (Nanjing, China) *(EAST 2014)*. Association for Computing Machinery, New York, NY, USA, 40–47. https://doi.org/10.1145/2627508.2627515

[18] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical accuracy estimation for efficient deep neural network testing. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 1–35.
[19] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering*. 509–519.
[20] Mengnan Du, Fan Yang, Na Zou, and Xia Hu. 2020. Fairness in deep learning: A computational perspective. *IEEE Intelligent Systems* (2020).
[21] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Software engineering meets deep learning: A literature review. *arXiv e-prints* (2019), arXiv–1909.
[22] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396.
[23] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations*.
[24] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. *arXiv preprint arXiv:2204.08734* (2022).
[25] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering*. 486–498.
[26] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. 2020. An empirical study of the dependency networks of deep learning libraries. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 868–878.
[27] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 23:1–23:10.
[28] Hannes Hapke and Catherine Nelson. 2020. *Building Machine Learning Pipelines*. O'Reilly Media.
[29] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.
[30] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.
[31] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.
[32] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.
[33] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *International Conference on Database Systems for Advanced Applications*. 604–620.
[34] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.
[35] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference*. 1–10.
[36] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. Apirecx: Cross-library api recommendation via pre-trained language model. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 3425–3436.
[37] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23 (2009), 2009.
[38] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*. 1039–1049.
[39] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *5th International Conference on Learning Representations*.
[40] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 241–252. https://doi.org/10.1109/ICST49551.2021.00034
[41] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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*. 329–339.

[43] Howard Lune and Bruce L Berg. 2017. *Qualitative research methods for the social sciences*. Pearson.

[44] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[45] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[46] Lei Ma, Fuyuan Zhang, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. Combinatorial Testing for Deep Learning Systems. arXiv:1806.07723 [cs.SE]

[47] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 785–796. https://doi.org/10.1109/ASE.2019.00078

[48] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 55–64.

[49] Alexandre Perez, Rui Abreu, and Marcelo D'Amorim. 2017. Prevalence of Single-Fault Fixes and Its Impact on Fault Localization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 12–22.

[50] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering*. 1027–1038.

[51] Qingchao Shen, Junjie Chen, Jie M Zhang, Haoyu Wang, Shuang Liu, and Menghan Tian. 2022. Natural Test Generation for Precise Testing of Question Answering Software. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[52] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. to appear.

[53] Xiangzhong Shen, Jieyi Zhang, Xiaonan Wang, Hongfang Yu, and Gang Sun. 2021. Deep Learning Framework Fuzzing Based on Model Mutation. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. 375–380. https://doi.org/10.1109/DSC53577.2021.00059

[54] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305.

[55] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

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

[57] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[58] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to Construct Better Mutation Faults. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[59] Susana M Vieira, Uzay Kaymak, and João MC Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *Proceedings of International Conference on Fuzzy Systems*. 1–8.

[60] J.M. Voas. 1992. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering* 18, 8 (1992), 717–727. https://doi.org/10.1109/32.153381

[61] Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. 2022. An Empirical Study on Numerical Bugs in Deep Learning Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[62] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. (2022).

[63] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. 2020. An empirical study on regular expression bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 103–113.

[64] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. 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*. 788–799.

[65] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 397–409.

[66] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. 251–262.

[67] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. *arXiv preprint arXiv:2201.06589* (2022).

[68] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.

[69] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[70] Yingfei Xiong, Yongqiang Tian, Yepang Liu, and Shing-Chi Cheung. 2022. Toward Actionable Testing of Deep Learning Models. *Science China Information Sciences* (2022).

[71] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638.

[72] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 1448–1460.

[73] Hanmo You, Zan Wang, Junjie Chen, Shuang Liu, and Shuochuan Li. 2023. Regression Fuzzing for Deep Learning Systems. In *45th International Conference on Software Engineering*. to appear.

[74] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of biostatistics* 7 (2005).

[75] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[76] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *Proceedings of 40th IEEE/ACM International Conference on Software Engineering*. 886–896.

[77] Xufan Zhang, Jiawei Liu, Ning Sun, Chunrong Fang, Jia Liu, Jiang Wang, Dong Chai, and Zhenyu Chen. 2021. Duo: Differential Fuzzing for Deep Learning Operators. *IEEE Transactions on Reliability* 70, 4 (2021), 1671–1685. https://doi.org/10.1109/TR.2021.3107165

[78] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–412.

[79] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *43rd IEEE/ACM International Conference on Software Engineering*. 359–371.

[80] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

[81] Yingyi Zhang, Zan Wang, Jiajun Jiang, Hanmo You, and Junjie Chen. 2022. Toward Improving the Robustness of Deep Learning Models via Model Transformation. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 104:1–104:13.

[82] Ziyuan Zhong, Yuchi Tian, and Baishakhi Ray. 2021. Understanding local robustness of deep neural networks under natural variations. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 313–337.