An Empirical Study on Bugs inside TensorFlow

Li Jia[1], Hao Zhong[1⋆], Xiaoyin Wang[2], Linpeng Huang[1], and Xuansheng Lu[1]

[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2] Department of Computer Science, University of Texas at San Antonio
{insanelung,zhonghao,huang-lp,luxuansheng}@sjtu.edu.cn,
xiaoyin.wang@utsa.edu

Abstract. In recent years, deep learning has become a hot research topic. Although it achieves incredible positive results in some scenarios, bugs inside deep learning software can introduce disastrous consequences, especially when the software is used in safety-critical applications. To understand the bug characteristic of deep learning software, researchers have conducted several empirical studies on deep learning bugs. Although these studies present useful findings, we notice that none of them analyze the bug characteristic inside a deep learning library like TensorFlow. We argue that some fundamental questions of bugs in deep learning libraries are still open. For example, what are the symptoms and the root causes of bugs inside TensorFlow, and where are they? As the underlying library of many deep learning projects, the answers to these questions are useful and important, since its bugs can have impacts on many deep learning projects. In this paper, we conduct the first empirical study to analyze the bugs inside a typical deep learning library, *i.e.*, TensorFlow. Based on our results, we summarize 5 findings, and present our answers to 2 research questions. For example, we find that the symptoms and root causes of TensorFlow bugs are more like ordinary projects (*e.g.*, Mozilla) than other machine learning libraries (*e.g.*, Lucene). As another example, we find that most TensorFlow bugs reside in its interfaces (26.24%), learning algorithms (11.79%), and how to compile (8.02%), deploy (7.55%), and install (4.72%) TensorFlow across platforms.

1 Introduction

In recent years, deep learning has been a hot research topic, and researchers have used deep learning techniques to solve the problems in various research fields such as computer vision [36], speech recognition [31], natural language processing [23,45], software analysis [50] and graph classification [35]. Specifically, in the research community of databases, deep learning techniques and database techniques have promoted each other in various perspectives. On one hand, deep learning techniques are employed to handle various database problems (*e.g.*, tuning database configurations with a deep reinforcement learning model [37]). On the other hand, database techniques are be used to improve deep learning systems (*e.g.*, optimizing neural networks [49]).

⋆ corresponding author

When implementing deep learning applications, instead of reinventing wheels, programmers often build their applications on mature libraries. Among these libraries, TensorFlow [20] is the most popular, and a recent study [53] shows that more than 36,000 applications of GitHub are built upon TensorFlow. As they are popular, one bug inside deep learning libraries can lead to bugs in many applications, and such bugs can lead to disastrous consequences. For example, Pei *et al.* [41] report that a Google self-driving car and a Tesla sedan crash, due to bugs in their deep learning software.

To better understand bugs of deep learning programs, researchers have conducted empirical studies on such bugs. In particular, Zhang *et al.* [53] conduct an empirical study to understand the bugs of TensorFlow clients. Here, a client of TensorFlow is a program that calls the APIs of TensorFlow. While Zhang *et al.* [53] analyze only TensorFlow clients, Islam *et al.* [33] analyze the clients of more deep learning libraries such as Caffe [34], Keras [19], Theano [24], and Torch [25]. Although their results are useful to improve the quality of a specific client, to the best of our knowledge, no prior studies have ever explored the bugs inside popular deep learning libraries. Although the bugs inside TensorFlow influence thousands of its clients, many questions on such bugs are still open. For example, what are the symptoms and the root causes of such bugs, and where are they? A better understanding on such bugs will improve the quality of many clients, but it is challenging to conduct the desirable empirical study, since TensorFlow implements many complicated algorithms and is written in multiple programming languages. In this paper, we conduct the first empirical study to analyze the bugs and their fixes inside TensorFlow, and we present our answers to the following research questions:

– **RQ1. What are the symptoms and causes of bugs in TensorFlow?**
Motivation. The symptom and the root cause of a bug are important to understand and to fix the bug. For deep learning bugs, the results of the prior studies [53,33] are incomplete, because they analyze only deep learning clients. As the prior studies do not analyze bugs inside a deep learning library like TensorFlow, the answers to the above research question are still unknown.
Major results. In total, we identify six symptoms and eleven root causes for the bugs inside TensorFlow. We find that root causes are more determinative than symptoms, since several root causes have dominated symptoms (Finding 1). In addition, we find that the symptoms and the root causes of TensorFlow bugs are more like those of ordinary projects (*e.g.*, Mozilla) than other machine learning libraries (Finding 2). For the symptoms, build failures have correlations with inconsistencies, configurations and referenced type errors, and warning-style bugs have correlation with inconsistencies, processing, and type confusions. For the root causes, dimension mismatches lead to functional errors, and type confusions have correlation with functional errors, crashes, and warning-style errors (Finding 3).

– **RQ2. Where are the bugs inside TensorFlow?**
Motivation. From the perspective of TensorFlow developers, the locations of its bugs are important to improve the quality of TensorFlow. From the perspectives of the programmers of TensorFlow clients, they can be more careful to call TensorFlow, if they know such locations. From the perspective of researchers, they can design better detection techniques for our identified bugs, after the locations of target bugs are known. The prior studies [53,33] do not explore this research question.
Major results. To explore the bug characteristics in different library components, we analyze TensorFlow bugs by location. We find that major reported bugs reside in deep learning algorithms (*kernel*, 11.79%) and their interfaces (*API*, 26.42%). The two categories of bugs are followed by bugs in the deployment such as compiling (*lib*, 8.02%), deploying (*platform*, 7.55%), and installing (*tools*, 4.72%). The other components such as *runtime* (3.77%), *framework* (0.94%) and *computation graph* (0.94%) have fewer bugs.

This paper presents an empirical research. The purpose of an empirical study is to answer open questions on important issues, which can enrich the knowledge and motivate the follow-up research. Empirical studies have been widely conducted in various research fields such as software error analysis [27,29], database management [39,21] and information security [28,26]. As open questions are too complicated to be automated, like ours, some empirical studies are conducted manually, especially for those on the bug characteristics [46,53].

2 Methodology

2.1 Dataset

We select TensorFlow as the subject of our study, since Zhang *et al.* [53] report that more than 36,000 GitHub projects call the APIs of TensorFlow. As a result, the bugs inside TensorFlow influence thousands of its clients. In total, we analyze 202 TensorFlow bug fixes repaired between December 2017 and March 2019, and 84 of them have corresponding bugs reports. The number is comparable to other empirical studies. For example, Thung *et al.* [47] analyze 500 bugs from machine learning projects such as Mahout, Lucene, and OpenNLP. For each project, they analyze no more than 200 bugs. As another example, Zhang *et al.* [53] analyze 175 bugs from TensorFlow clients. Indeed, for deep learning programs, libraries are typically much larger than clients. As a result, our analyzed bugs are much more complicated than the bugs in the prior studies [53,47].

We apply the following steps to extract bugs:
Step1. Filtering pull requests by labels. To avoid problems which have not been handled correctly and to collect accurate information about fixed bugs, we start with *closed* pull requests with label "*ready to pull*". We notice that finished pull requests before a specific date are not tagged, so we also collect cases from earlier closed pull requests by searching keywords as described in Step 2. We manually check each collected pull request to ensure that its commit is already approved by reviewers and is merged into the master branch.

Step2. Searching pull requests by keywords. From closed pull requests, we use the keywords such as "bug", "fix" and "error" to identify the ones that fix bugs. From bug fixes, we use the keywords such as "typo" and "doc" to remove the ones that fix superficial bugs. From the remaining bug fixes, we manually inspect them to select real ones, by reading their pull requests carefully. In total, we selected 202 bug fixes for latter analysis.
Step3. Extracting bug reports and code changes. For each one of the 202 bug fixes, we extract its bug report and code changes. The extracted results and corresponding pull requests are used to determine their symptoms, root causes (RQ1), and locations (RQ2). We introduce the detailed analysis in Section 2.2.

2.2 Manual Analysis

In our study, we employ two graduate students to manually inspect all bugs. The two students major in computer science, and both are familiar with deep learning algorithms. They have experience in developing deep learning applications (*e.g.*, mining on business data) based on TensorFlow. Following our protocols, the two students inspect the bugs independently, and compare the results. If we cannot reach a consensus on a TensorFlow bug, they discuss it on our group meetings.

Protocol of RQ1 When they build their own taxonomy of bug symptoms and their root causes, they refer to the taxonomies of the prior studies [22,46]. In particular, they add an existing category into their taxonomy, if they find a TensorFlow bug falls into this category. If a TensorFlow bug does not belong to an existing category, they try to modify a similar category of the prior studies [22,46]. If they fail to find such a similar category, they add a new one.

For bug classifying, if a pull request has a corresponding bug report, they first read its report to identify its symptoms and root causes. If a pull request does not provide a report, they manually identify its symptom and root cause from the description, bug-related discussion, code changes and comments of the pull request. For example, a bug fix without report [18] is titled "*Fix for stringpiece build failure*". Based on the title, they determine that the symptom of the bug is build failure. They notice that the only code modification of this bug fix is:

```
1 void Append(StringPiece s) {
2 -     key_.append(s.ToString());
3 +     key_.append(string(s));
4       key_.append(1, delimiter); }
```

The `ToString()` method that is called to build the key in the buggy version is removed. In the fixed version, the `string(StringPiece)` method should be called to build the correct key, but in the old location, the method call is not updated. Considering this, they determine that the root cause of the bug is the inconsistency introduced by API change.

After the symptoms and root causes of all the bugs are extracted, the two students further classify them into categories, and use the *lift* function [30] to

measure the correlations between symptoms and root causes. According to the definition, the *lift* between different categories (A and B) is computed as:

$$lift(A, B) = \frac{P(A \cap B)}{P(A) \cdot P(B)} \quad (1)$$

where $P(A)$, $P(B)$, $P(A \cap B)$ are the probabilities that a bug belongs to category A, category B, and both A and B. If a *lift* value is greater than one, category A and B are correlated, which means a symptom is correlated to a root cause. If it is equal to or less than one, a symptom is not correlated to a root cause.

Protocol of RQ2 In this research question, the two students analyze the locations of bugs. As an open source project, TensorFlow does not officially list its components, but like other projects, TensorFlow puts its source files into different directories, by their functionalities. When determining their functionalities, they refer to various sources such as official documents, TensorFlow tutorials, and forum discussions. Their identified components are as following:
1. Kernel. The kernel implements the core deep learning algorithms (*e.g.*, the `conv2d` algorithm), and its source files are located in the `core/kernels` directory.
2. Computation graph. TensorFlow uses computation graphs to define and to manage its computation tasks. The graph implements the definition, construction, partition, optimization, operation, and execution of computations. Most source files of this component are located in the `core/graph` directory; its data operations are located in the `core/ops` directory; and its optimization-related source files are located in the `core/grappler` directory.
3. API. TensorFlow provides APIs in various programming languages (*e.g.*, Python, C++, Java), which are located in the `python`, `c`, `cc` and `java` directories.
4. Runtime. The runtime implements the management of sessions, thread pools, and executors. TensorFlow has a common runtime (`core/common_runtime`) and a distribution runtime (`core/distributed_runtime`). Common runtime supports the executions on a local machine, and distribution runtime allows to deploy TensorFlow on distributed ones. For simplicity, we merge them into one component.
5. Framework. The framework implements basic functionalities (*e.g.*, logging, memory, and files). Most source files of this component are located in `core/framework` directory, and the serialization is located in `core/protobuf` directory.
6. Tool. The tool implements utilities. For example, `tools/git` and `tools/pip_package` directories implement the utilities to install TensorFlow; the `core/debug` directory provides a tool to debug TensorFlow clients; and the `core/profile` directory provides a tool to profile the execution of TensorFlow and its clients.
7. Platform. The platform allows to deploy TensorFlow on various platforms. The `core/platform` directory contains the source files to handle hardware issues (*e.g.*, CPU and GPU); the `core/tpu` directory allows executing on TPU; the `lite` directory allows executing TensorFlow on mobile devices; and the `compiler` directory allows compiling to native code for various architectures.
8. Contribution. The `contrib` directory contains extensions that are often implemented by outside contributors. For example, the `contrib/seq2seq` directory

contains a sequence-to-sequence model that is widely used in neural translation. After they become mature, they can be merged into other directories. In our study, we define a component for this directory.
9. Library. The library includes the API libraries. Most libraries are located in the `third-party` directory, and some libraries are located in other directories (*e.g*, `core/lib`, `core/util` and some files under the root directory of `tensorflow`).
10. Documentation. The documentation includes samples, which are located in the `examples` and `core/example` directories. It also includes other types of documents. For example, the `security` directory stores security guidelines.

We use the *lift* metric as defined in Equation 1 to measure the correlation between a bug location and a symptom or a root cause. Here, if a bug involves more than one directory, we count them once for each directory to ensure that each location does not lose a symptom and a root cause.

3 Empirical Result

This section presents the results of our study. More details are listed on our anonymized project website: https://github.com/fordataupload/tfbugdata/

3.1 RQ1. Symptoms and Root Causes

The categories of symptoms :
1. Functional error (35.64%). If a program does not function as designed, we call it a functional error. For example, we find that a bug report [2] complains the functionality of the `tf.Print` method:

> If you print a tensor of shape [n, 4] with tf.Print, by default (summarize=3 is the default value), you get: [[9 21 55]...], which wrongly looks like your tensor is of shape [n, 3]. The correct output should be: [[9 21 55...]...].

The method is designed to print the details of tensors. The bug report complains that it prints incorrect output, when the shape is `[n, 4]`. As the result is not as expected, it is a functional error.
2. Crash (26.73%). A crash occurs, when a program stops and exits irregularly. When it happens, the program often throws an error message. For example, a bug report [4] describes a crash caused by an unsupported operand type:

> Using a TimeFreqLSTMCell in a dynamic_rnn without providing optional parameter frequency_skip results in an exception: TypeError: unsupported operand type(s) for /: 'int' and 'NoneType'.

3. Hang (1.49%). A hang occurs, when a program keeps running without stopping or responding. A bug report [17] provides description as below:

> When running the above commands (Inception V3 synchronized data parallelism training with 2 workers and 1 external ps), the tf_cnn_benchmarks application hangs forever after some iterations (usually in warm up).

4. Performance degradation (1.49%). A performance degradation occurs, when a program does not return results in expected time. For example, we find a performance degradation in a bug report [16]:

> There is a performance regression for TF 1.6 comparing to TF 1.5 for cifar 10.

5. Build failure (23.76%). A build failure occurs in the compiling process. For example, we find that a bug report [3] describes a build failure, which is caused by a missing header file:

Build failing due to missing header files "tensorflow/contrib/tpu/proto/tpu_embedding_config.pb.h".

6. Warning-style error (10.89%). Warning-style error means the running of a program is not disturbed, but modifications are still needed to get rid of risk or improve code quality, including interfaces to be deprecated, redundant code and bad code style. Most bugs in this category are shown by warning messages, while a few others do not provide visible messages which are found by code review or other events. For example, we find a bug in such category [11], since it calls a method with a deprecated argument:

According to tf.argmax, dimension argument was deprecated, it will be removed in a future version.

The categories of root causes :

1. Dimension mismatch (3.96%). We put a bug into this category if it is caused by dimension mismatch in tensor computations and transformations. A bug fix [14] describes the cause of a bug in this category as:

Wrongly "+1" for output shape, that will cause CopyFrom failure in MklToTf op because of tensor size and shape mismatch.

The buggy code sets the dimension of an output tensor:

```
output_tf_shape.AddDim((output_pd->get_size() / sizeof(T)) + 1);
```

The fixed code sets the correct dimension:

```
output_tf_shape.AddDim((output_pd->get_size() / sizeof(T)));
```

2. Type confusion (12.38%). Type confusions are caused by the mismatches of types. A sample report [12] is as below:

CRF decode can fail when default type of "0" (as viewed by math_ops.maximum) does not match the type of sequence_length.

After the bug was fixed, programmers modified a test case to ensure that the method accepts more types of input values:

```
 np.array(3, dtype=np.int32),
-         np.array(1, dtype=np.int32)
+         np.array(1, dtype=np.int64)
```

3. Processing (22.28%). We put a bug into this category, if it is caused by wrong assignment or initialization of variables, wrong formats of variables, or other wrong usages that are related to data processing. For example, we find a bug report [8] in such category as follow:

ConvNDLSTMCell class in tensorflow.contrib.rnn cannot pass the name attribute correctly when created, because of the missing parameter in constructor.

The constructor of `ConvNDLSTMCell` has no parameters to define their names:

```
super(Conv1DLSTMCell, self).__init__(conv_ndims=1, **kwargs)
```

The bug is fixed in a latter version:

```
super(Conv1DLSTMCell, self).__init__(conv_ndims=1, name=name, **
    kwargs)
```

4. Inconsistency (16.83%). We put a bug into this category, if it is caused by incompatibility due to API change or version update. For example, a bug report [7] complains that a removed `ops` is called:

NotFoundError: Op type not registered 'KafkaDataset' in binary. is returned from kafka ops. The issue was that the inclusion of kafka ops was removed due to the conflict merge from the other PR.

The above compilation error was caused by a conflict merge of two commits. One removed `kafka ops`, but the other added a call to the operator.

5. Algorithm (2.97%). We put a bug into the algorithm category, if it is caused by wrong logic in algorithms. For example, a bug report [5] complains that a method returns wrong values:

Input labels = tf.constant([[0., 0.5, 1.]]), predictions = tf.constant([[1., 1., 1.]]), the result of tf.losses.mean_pairwise_squared_error(labels, predictions) should be $[(0-0.5)^2+(0-1)^2+(0.5-1)^2]/3 = 0.5$, but TensorFlow returns different value 0.333333.

According to the code document, the mean pairwise squared error is incorrectly calculated. In the process of deduction, the denominators of two intermediate variables are wrong. A developer replaces an assignment and changes a method with corresponding parameters to fix denominators as below:

```
-     num_present_per_batch)
+     num_present_per_batch-1)
...
+     math_ops.square(num_present_per_batch))
-     math_ops.multiply(num_present_per_batch, num_present_per_batch-1)
      )
```

6. Corner case (15.35%). We put a bug into this category, if it is caused by erroneous handling of corner cases. A bug of this kind is reported [15] as:

When batch_size is 0, max pooling operation seems to produce an unhandled cudaError_t status. It may cause subsequent operations fail with odd error message.

As the reporter says, a crash happens when `batch_size` of the input is 0, which belongs to corner cases.

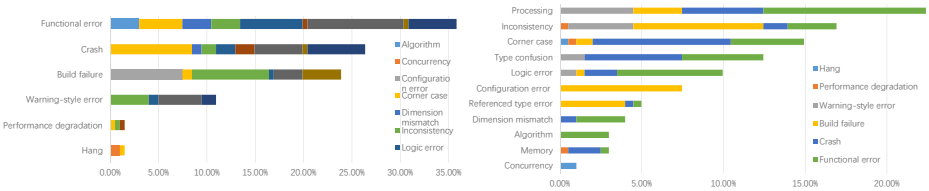
7. Logic error (9.90%). We put a bug into this category, if mistakes happen in the logic of a program. A logic error can be an incorrect program flow or a wrong order of actions. A bug report [10] provides the description as:

When a kernel Variable is shared by two Conv2Ds, ... there will be only one Conv2D getting the quantized kernel.

TensorFlow implements a mechanism called quantization to shrink tensors. The reporter complains that when a tensor shares two Conv2D, the second one cannot obtain the right quantized kernel. The logic of the code is flawed, in that the program in complex flow does not behave as expected.

8. Configuration error (7.43%). We put a bug into this category, if it is caused by wrong configuration. A sample bug [6] is as follow:

Linking of rule '...toco' fails because LD_LIBRARY_PATH is not configured.



(a) The distribution of symptoms

(b) The distribution of root causes

Fig. 1: Distribution of bug symptoms and root causes

To repair the bug, in a configuration file, programmers add the following statement to initiate `LD_LIBRARY_PATH`:

```
if 'LD_LIBRARY_PATH' in environ_cp and environ_cp.get('
    LD_LIBRARY_PATH')!='1':
  write_action_env_to_bazelrc('LD_LIBRARY_PATH', ...)
```

9. Referenced types error (4.95%). We put a bug into this category, if it is caused by missing or adding unnecessary `include` or `import` statements. A bug [13] triggers the following error message:

The compiler couldn't find std::function, because header file #include <functional> is missing.

Programmers forget to add the said `include` statement, which causes the bug.

10. Memory (2.97%). We put a bug into the memory category, if it is caused by incorrect memory usages. For example, a bug report [9] describes a possible memory leak, which can be triggered by an exception, because of missing deconstruction operation.

11. Concurrency (0.99%). We put a bug into this category, if it is caused by synchronization problems. A bug report [1] describes a deadlock as follow:

notify_one was used to notify inserters and removers waiting to insert and remove elements into Staging Areas. This could result in deadlock when many removers were waiting for different keys.

As the reporter says, when multiple removers wait for keys but `notify_one` only notifies one of them, a deadlock may occur.

Distribution Figure 1a shows the distribution of symptoms. Its horizontal axis shows symptom categories, and its vertical axis shows the percentage of corresponding symptom. For each symptom, we refine its bugs by their root causes. Tan *et al.* [46] report the distributions of Mozilla, Apache, and the Linux kernel. We find that the distribution of TensorFlow is close to their distributions. Figure 1a shows that functional errors account for 39%, which are the most common bugs of TensorFlow. Tan *et al.* [46] show that in Mozilla, Apache, and the Linux kernel, function errors vary from 50% to 70%. We find that crashes account for 26.5% TensorFlow bugs, which are close to Linux (27.2%), and hangs account for 1% bugs, which are close to Mozilla (2.1%).

Figure 1b shows the distribution of root causes. Its horizontal axis shows cause categories, and its vertical axis shows the percentage of corresponding causes. For each root cause, we refine its bugs by symptoms. We find that all the symptoms have multiple and evenly distributed root causes, but the distribution of root causes are not so evenly. The distributions lead to our first finding:

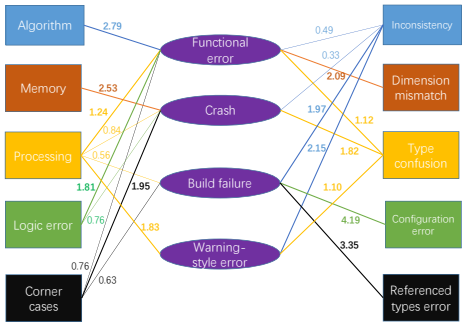


Fig. 2: Correlation between symptoms and root causes

Finding 1. Compared to symptoms, root causes are more determinative, since several root causes have dominated symptoms.

Tan *et al.* [46] show that in Mozilla, Apache, and the Linux kernel, the dominant root cause is semantic (80%). In our taxonomy, memory, configuration and referenced types errors belong to semantic bugs (85%), which are close to Tan *et al.* Thung *et al.* [47] show that in machine learning systems, algorithm errors are the most common bugs (22.6%). The above observations lead to another finding:

Finding 2. The symptoms and causes of TensorFlow are more like an ordinary software system (*e.g.*, Mozilla) than a machine learning system (*e.g.*, Lucene).

A machine learning system typically provide many algorithms for users to invoke. For example, although Lucene is also large (554,036 lines of code), the symptoms and root causes of its bugs are more different from TensorFlow than ordinary software systems like Mozilla. We find that Lucene provides numerous APIs to handle natural language texts in different ways (*e.g.*, tokenization). In the contrast, TensorFlow provides much fewer interfaces to invoke, which is more like a traditional software system.

Correlation of bug categories Figure 2 shows the correlation of bug categories. The rectangles on the left side denote symptoms, the ovals on the right side denote root causes. We choose different colors to distinguish the correlations, and the root causes of the same color are not related. We ignore categories whose bugs are fewer than three, since they are statistically insignificant. For example, we ignore hangs, since only two bugs are hangs. The lines denote correlations, and we highlight correlations whose values are greater than one.

Both Tan *et al.* [46] and we find that crashes have correlations with memory bugs and corner cases. Tan *et al.* [46] find that crashes also have correlations with concurrency, but we do not consider it, since only two of our analyzed bugs are concurrency. Instead, our study shows that crashes of TensorFlow have correlations with type confusions, which are not identified by Tan *et al.* In addition, Tan *et al.* [46] and we find that function errors have correlations with processing and logic errors. Tan *et al.* [46] find that function errors have correlations with

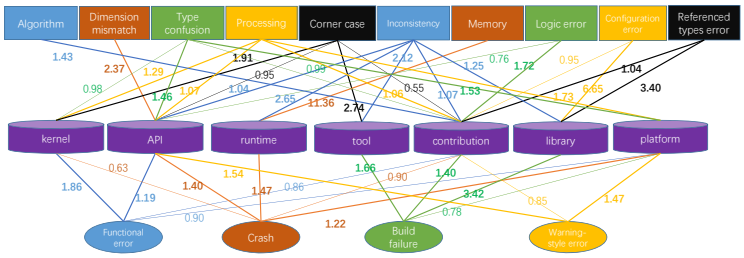


Fig. 3: Correlation between locations

missing features by defining a missing feature as a feature is not implemented yet. As we find that TensorFlow programmers seldom write their unimplemented features in their code, we eliminate this subcategory. We find that build failures have correlation with inconsistencies, configurations and referenced type errors, and warning-style bugs have correlation with inconsistencies, processing, and type confusions. We believe that other open source projects (*e.g.*, Mozilla) also have the two types of symptoms, but are ignored by Tan *et al.* [46]. We identify the correlations of build failures and warning-style bugs, complementing the study of Tan *et al.* [46]. For our identified root causes and symptoms of TensorFlow, our observations lead to the following finding:

Finding 3. For symptom of TensorFlow bugs, build failures have correlation with inconsistencies, configurations and referenced type errors, and warning-style bugs have correlation with inconsistencies, processing, and type confusions. For the root causes of TensorFlow bugs, dimension mismatches lead to functional errors, and type confusions have correlation with functional errors, crashes, and warning-style errors.

3.2 RQ2. Bug Locations

Distribution As it contains immature implementations, it is not surprising that *contribution* is the buggies component (34.91%). The next two buggy components are *kernel* (11.79%), and *API* (26.42%). The two components implement the main functionalities of TensorFlow. In particular, kernel implements deep learning algorithms, and API implements their interfaces. Following them, the next three buggy components are *library* (8.02%), *platform* (7.55%), and *tool* (4.72%). The three components implement features to support the compilation, the execution, and the installation on different platforms. The other components are less buggy. Even though computation graphs define the process of TensorFlow computing, we find that only 0.94% bugs locate in this component. Our observations lead to a finding:

Finding 4. In TensorFlow, the major reported bugs are in deep learning algorithms and their interfaces, and the bugs in compiling, deploying, and installing TensorFlow on different platforms occupy a smaller proportion.

Correlation of bug categories Figure 3 shows the correlations among symptoms, root causes, and bug locations. In this figure, the rectangles denote root causes; the ovals denote symptoms; and the cylinders denote bug locations. We ignore bug locations, if their bugs are fewer than three. The lines denote correlations, and we highlight correlations whose values are greater than one.

For root causes, we find that inconsistencies are popular, and for symptoms, crashes and build failures are popular among the components. From the perspective of components, we find that *kernel* has strong correlation with functional errors and corner cases, which indicates semantic bugs are dominant in this component. Meanwhile, we find that *API* has strong correlation with root causes related to tensor computations such as dimension mismatches and type confusions. For *library* and *tool*, their symptoms have strong correlations with build failures, and their root causes have strong correlations with inconsistences. The above observations lead to a finding:

Finding 5. Crashes and build failures are popular symptoms, and inconsistencies are a popular root cause among components. For those buggiest components, we find that *kernel* contains many sematic bugs, and *API* contains root causes related to tensor computations such as dimension mismatches and type confusions. In the related components such as *library* and *tool*, build failures are popular, and most bugs are caused by inconsistencies.

3.3 Threat to validity

The internal threats to validity include the possible errors of our manual inspection. To reduce the threat, we ask two students to inspect our bugs. When they encounter controversial cases, they discuss them with others on our group meeting, until they reach an agreement. The threat can be mitigated with more researchers, so we release our inspection results on our website. The internal threats to external validity include our subject, since we analyzed the bugs inside only TensorFlow. Although our analyzed bugs are comparable with the prior studies and other studies (*e.g.* [53]) also analyzed only TensorFlow bugs, they are limited. The threat can be reduced by analyzing more libraries in future.

4 The Significance of Our Findings

Improving the quality of deep learning libraries. For every root cause of TensorFlow bug, we find several major symptoms occupy a large proportion (Finding 1), and the correlations between root cause and symptom can also suggest possible links (Finding 3), which can help developers to diagnose the cause of a bug according to its symptom. Since TensorFlow bug characteristics show strong similarity to traditional software (Finding 2), the experience and tools of bug repairing in other software can also be transferred to TensorFlow. Since the proportion of bugs in different component varies obviously (Finding 4), developers should pay more attention to safety check and test case design,

when adding new features or making modifications to bug-prone components. Moreover, as the integration of libraries is common in deep learning software, the connection of different libraries should obtain higher priority in development. To overcome this problem, developing unified APIs can be helpful.

Combining the results of the prior studies. From two different perspectives of deep learning software, the prior studies [53,33] analyze the bugs of deep learning clients, but our study analyzes the bugs of deep learning libraries. The bugs inside deep learning libraries can have impacts on the bugs of their clients. For example, Islam *et al.* [33] find that 11% percentage of TensorFlow client bugs are caused by incorrect usages of deep learning APIs. From the perspective of deep learning libraries, such bugs can be caused by the inconsistency bugs in our study. As another example, the prior studies [53,33] show that unaligned tensors and the absences of type checking are common causes of deep learning client bugs. We suspect that such bugs are related to dimension mismatches and type confusions, which are found in our study. In future work, we plan to combine the results of the prior studies and ours and explore more advanced techniques to detect deep learning bugs.

The inspiration to databases and their applications. We notice that some bugs in deep learning libraries are common in database systems (*e.g.*, memory bugs 2.97% and concurrency bugs 0.99%), and detecting such bugs has been a hot research topic in the research community of databases [38,43,44]. As advocated by Wang *et al.* [51], the existing techniques in the database community can be tailored to handle similar bugs in deep learning libraries [51]. Additionally, as more and more deep learning techniques and frameworks are applied to solve database problems [37,52], our revealed bugs in side such libraries are also important for database researchers and programmers.

5 Related Work

Empirical studies on bug characteristics. There has been a number of recent studies studying bugs from open source repositories. Tan *et al.* [46] analyze the bug characteristics of open source projects such as the Linux kernel and Mozilla. Thung *et al.* [47] analyze the bugs of machine learning systems such as Mahout, Lucene, and OpenNLP. Zhang *et al.* [53] analyze the client code that calls TensorFlow. Islam *et al.* [33] analyze the clients of more deep learning bugs. Humbatova *et al.* [32] introduce a taxonomy of faults in deep learning systems. Compared with all existing works, we analyze bugs inside a representative deep-learning library *i.e.*, TensorFlow, which is a different angle from theirs.

Detecting deep learning bugs. Pei *et al.* [41] propose a whitebox framework to test real-world deep learning systems. Ma *et al.* [40] propose a set of multi-granularity criteria to measure the quality of test cases prepared for deep learning systems. Tian *et al.* [48] and Pham *et al.* [42] introduce differential testing to discover bugs in deep learning software. Our empirical study reals new types of bugs, which cannot be effectively detected by the above approaches. Our findings are useful for researchers, when they design detection approaches for such bugs.

6 Conclusion and Future Work

Although researchers have conducted empirical studies to understand deep learning bugs, these studies focus on bugs of its clients, and the nature of bugs inside a deep library is still largely unknown. To deepen the understanding of such bugs, we analyze 202 bugs inside TensorFlow. Our results show that (1) its root causes are more determinative than its symptoms; (2) bugs in traditional software and TensorFlow share various common characteristics; and (3) inappropriate data formatting (dimension and type) is bug prone and popular in API implements while inconsistent bugs are common in other supporting components. In future work, we will analyze bugs from more deep-learning libraries to obtain a more comprehensive understanding of bugs in deep learning frameworks, and we plan to design automatic tools to detect bugs in deep-learning libraries.

Acknowledgement

We appreciate the anonymous reviewers for their insightful comments. This work is sponsored by the National Key R&D Program of China No. 2018YFC083050.

References

1. Fix deadlocks in staging areas. https://github.com/tensorflow/tensorflow/pull/13684 (2017)
2. Bug in tf.print summarized formatting. https://github.com/tensorflow/tensorflow/issues/20751 (2018)
3. Cannot opened include file "tensorflow/contrib/tpu/proto/tpu_embedding_config.pb.h": no such file or directory. https://github.com/tensorflow/tensorflow/issues/16262 (2018)
4. Exception when not providing optional parameter frequency_skip in timefreqlstmcell. https://github.com/tensorflow/tensorflow/issues/16100 (2018)
5. Fix an imperfect implementation of tf.losses.mean_pairwise_squared_error. https://github.com/tensorflow/tensorflow/pull/16433 (2018)
6. Fix broken python3 build. https://github.com/tensorflow/tensorflow/pull/16130 (2018)
7. Fix build issue with kafkadataset. https://github.com/tensorflow/tensorflow/pull/17418 (2018)
8. Fix error: Convndlstmcell does not pass name parameter. https://github.com/tensorflow/tensorflow/pull/17345 (2018)
9. Fix possible memory leak. https://github.com/tensorflow/tensorflow/pull/21950 (2018)
10. Fix routing of quantized tensors. https://github.com/tensorflow/tensorflow/pull/19894 (2018)
11. Fix tf.argmax warnings on dimension argument by using axis instead. https://github.com/tensorflow/tensorflow/pull/18558 (2018)
12. Fix var type issue which breaks crf_decode. https://github.com/tensorflow/tensorflow/pull/21371 (2018)

13. Fixed build error on gcc-7. https://github.com/tensorflow/tensorflow/pull/21017 (2018)
14. [INTEL MKL] fix bug in mklslice op when allocating output tensor. https://github.com/tensorflow/tensorflow/pull/22822 (2018)
15. Max pooling cause error on empty batch. https://github.com/tensorflow/tensorflow/issues/21338 (2018)
16. Mkl dnn: fix the tf1.6 speed issue by fixing mkl dnn lrn taking the optimum path. https://github.com/tensorflow/tensorflow/pull/17605 (2018)
17. tf_cnn_benchmarks.py stuck when running with multiple gpus and imagenet data with protocol grpc+verbs. https://github.com/tensorflow/tensorflow/issues/11725 (2018)
18. Fix for stringpiece build failure. https://github.com/tensorflow/tensorflow/pull/21956 (2019)
19. Keras. https://keras.io. (2019)
20. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P.A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: A system for large-scale machine learning. In: Proc. OSDI. pp. 265–283 (2016)
21. Anand, S.S., Bell, D.A., Hughes, J.G.: An empirical performance study of the ingres search accelerator for a large property management database system. In: Proc. VLDB. pp. 676–685 (1994)
22. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. **1**(1), 11–33 (2004)
23. Bengio, Y., Ducharme, R., Vincent, P., Janvin, C.: A neural probabilistic language model. Journal of Machine Learning Research **3**, 1137–1155 (2003)
24. Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Wardefarley, D., Goodfellow, I., Bergeron, A.: Theano: Deep learning on gpus with python. In: Proc. Nips, BigLearning Workshop (2011)
25. Collobert, R., Bengio, S., Marithoz, J.: Torch: A modular machine learning software library (2002)
26. Derr, E., Bugiel, S., Fahl, S., Acar, Y., Backes, M.: Keep me updated: An empirical study of third-party library updatability on android. In: Proc. CCS. pp. 2187–2200 (2017)
27. Endres, A.: An analysis of errors and their causes in system programs. IEEE Trans. Software Eng. **1**(2), 140–149 (1975)
28. Florêncio, D.A.F., Herley, C.: A large-scale study of web password habits. In: Proc. WWW. pp. 657–666 (2007)
29. Glass, R.L.: Persistent software errors. IEEE Trans. Software Eng. **7**(2), 162–168 (1981)
30. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers (2011)
31. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Processing Magazine **29**(6), 82–97 (2012)
32. Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P.: Taxonomy of real faults in deep learning systems. In: Proc. ICSE. p. to appear (2020)

33. Islam, M.J., Nguyen, G., Pan, R., Rajan, H.: A comprehensive study on deep learning bug characteristics. In: Pro. ESEC/FSE. pp. 510–520 (2019)
34. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R.B., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proc. MM. pp. 675–678 (2014)
35. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: Proc. ICLR (2017)
36. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proc. NIPS. pp. 1106–1114 (2012)
37. Li, G., Zhou, X., Li, S., Gao, B.: Qtune: A query-aware database tuning system with deep reinforcement learning. PVLDB **12**(12), 2118–2130 (2019)
38. Lin, Q., Chen, G., Zhang, M.: On the design of adaptive and speculative concurrency control in distributed databases. In: Proc. ICDE. pp. 1376–1379 (2018)
39. Lockemann, P.C., Nagel, H., Walter, I.M.: Databases for knowledge bases: empirical study of a knowledge base management system for a semantic network. Data and Knowledge Engineering **7**, 115–154 (1991)
40. Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., Zhao, J., Wang, Y.: Deepgauge: Multi-granularity testing criteria for deep learning systems. In: Proc. ASE. pp. 120–131 (2018)
41. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proc. SOSP. pp. 1–18 (2017)
42. Pham, H.V., Lutellier, T., Qi, W., Tan, L.: CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In: Proc. ICSE. pp. 1027–1038 (2019)
43. Ren, K., Thomson, A., Abadi, D.J.: VLL: a lock manager redesign for main memory database systems. VLDB J. **24**(5), 681–705 (2015)
44. van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., Sato, M.: Managing non-volatile memory in database systems. In: Proc. SIGMOD. pp. 1541–1555 (2018)
45. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Proc. NIPS. pp. 3104–3112 (2014)
46. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical Software Engineering **19**(6), 1665–1705 (2014)
47. Thung, F., Wang, S., Lo, D., Jiang, L.: An empirical study of bugs in machine learning systems. In: Proc. ISSRE. pp. 271–280 (2012)
48. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proc. ICSE. pp. 303–314 (2018)
49. Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S.L., Xu, Z., Kraska, T.: Superneurons: dynamic GPU memory management for training deep neural networks. In: Proc. PPoPP. pp. 41–53 (2018)
50. Wang, S., Liu, T., Nam, J., Tan, L.: Deep semantic feature learning for software defect prediction. IEEE Transactions on Software Engineering (2018)
51. Wang, W., Zhang, M., Chen, G., Jagadish, H.V., Ooi, B.C., Tan, K.: Database meets deep learning: Challenges and opportunities. SIGMOD Record **45**(2), 17–22 (2016)
52. Xu, B., Cai, R., Zhang, Z., Yang, X., Hao, Z., Li, Z., Liang, Z.: NADAQ: natural language database querying based on deep learning. IEEE Access **7** (2019)
53. Zhang, Y., Chen, Y., Cheung, S., Xiong, Y., Zhang, L.: An empirical study on TensorFlow program bugs. In: Proc. ISSTA. pp. 129–140 (2018)