The Journal of Systems & Software 177 (2021) 110935

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The symptoms, causes, and repairs of bugs inside a deep learning library[☆,☆☆]

Li Jia[a], Hao Zhong[a,*], Xiaoyin Wang[b], Linpeng Huang[a], Xuansheng Lu[a]

[a] Shanghai Jiao Tong University, Shanghai 200240, China
[b] University of Texas at San Antonio, TX, USA

ARTICLE INFO

Article history:
Received 4 June 2020
Received in revised form 14 February 2021
Accepted 22 February 2021
Available online 26 March 2021

Keywords:
Deep learning
Bug analysis
TensorFlow
Empirical study

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 bugs in deep learning applications. 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 8 findings, and present our answers to 4 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 (*e.g.*, computer vision (Krizhevsky et al., 2012) and software analysis (Wang et al., 2018)). When implementing deep learning applications, instead of reinventing wheels, programmers often build their applications on mature libraries. Among these libraries, TensorFlow (Abadi et al., 2016) is the most popular, and a recent study (Zhang et al., 2018) 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. (2017) 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. (2018) conduct an empirical study to understand the bugs of TensorFlow applications. Here, an application of TensorFlow is a program that calls the APIs of TensorFlow. While Zhang et al. (2018) analyze only TensorFlow applications, Islam et al. (2019) analyze the applications of more deep learning libraries such as Caffe (Jia et al., 2014), Keras (Keras, 2019), Theano (Bergstra et al., 2011), and Torch (Collobert et al., 2002).

Although their results are useful to improve the quality of a specific application, 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 applications, 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 applications, but it is challenging to conduct the desirable empirical study, since TensorFlow implements many complicated algorithms and is written in multiple programming languages. In our prior work (Jia et al., 2020), we conducted the first empirical study to analyze the bugs inside TensorFlow. Compared with this work (Jia et al., 2020), our extended version has two additional contributions:

☆ This manuscript is an extended version of a paper (Jia et al., 2020) that is presented in the 25th International Conference on Database Systems for Advanced Applications (DASFAA), 2020.
☆☆ Editor: Shane McIntosh.
* Corresponding author.
E-mail address: zhonghao@sjtu.edu.cn (H. Zhong).

https://doi.org/10.1016/j.jss.2021.110935
0164-1212/© 2021 Elsevier Inc. All rights reserved.

1. Our prior work (Jia et al., 2020) analyzed only symptoms and causes of bugs, but in this extended version, we analyzed bug fixes and multiple language bugs.
2. We compared our identified symptoms, causes, and repair patterns with those that were reported by the prior studies (Islam et al., 2019; Zhang et al., 2018; Tan et al., 2014; Seaman et al., 2008; Thung et al., 2012; Kim et al., 2013; Le et al., 2016; Liu and Zhong, 2018) (see Section 5 for details). Based on the comparison, we find that TensorFlow has type confusions, which are not reported by the prior studies. In addition, we find that like deep learning applications, TensorFlow also has dimension mismatches.

Our research questions and their answers are as follows:

- **RQ1. What are the symptoms and causes of bugs?**
Motivation. The symptom and the cause of a bug are important to understand and to fix the bug. For deep learning bugs, the results of the prior studies Zhang et al. (2018), Islam et al. (2019) are incomplete, because they analyze only deep learning applications. As the prior studies do not analyze bugs inside a deep learning library, the answers to the above research question are still unknown.
Major results. In total, we identify six symptoms and eleven root causes. 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. How do the bugs spread across components?**
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 applications, 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 Zhang et al. (2018), Islam et al. (2019) do not explore this research question. To explore the bug characteristics in different library components, we analyze the impacts of TensorFlow bugs by their components.
Major results. 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.

- **RQ3. What are the repair patterns inside TensorFlow?**
Motivation. Researchers have conducted empirical studies to explore the repairing patterns of bugs (see Section 7 for details), but none of them have analyzed the repair patterns of deep libraries bugs. In this research question, we analyze such repair patterns. The results can be useful to determine to what degree can the prior tools repair deep learning bugs.
Major results. From TensorFlow bugs, we identify ten repair templates. Compared with the prior studies Kim et al. (2013), Le et al. (2016), Liu and Zhong (2018), besides confirming known templates, we find two new templates. Although it needs different expertise to fix TensorFlow bugs, from the viewpoint of modifying code, we find that fixing deep learning bugs requires largely the same repair actions with fixing bugs in other types of projects (Finding 6). The correlation of common repair patterns and their causes are also displayed (Finding 7).

- **RQ4. Which bugs involve multiple programming languages?**
Motivation. The implementation of TensorFlow concerns several types of programming languages. The prior study (Kochhar et al., 2016) shows that multiple languages in software can introduce more bugs. In our study, we also explore the interaction of different languages in TensorFlow.
Major results. We find that only 5% TensorFlow bugs involve multiple programming languages, and we classify them into two categories: (1) source files and configuration files can have related bugs, and (2) the core and its applications/test cases can have related bugs (Finding 8).

Instead of analyzing deep learning applications as the prior studies Zhang et al. (2018), Islam et al. (2019) did, for the first time, our study explores the bugs and their fixes inside deep learning libraries. Our findings are useful to improve the quality of deep learning libraries, and have positive impacts on downstream applications. We discuss this issue in Section 6.

2. Preliminary

2.1. The implementation of TensorFlow

TensorFlow uses dataflow graphs to define the computations and states of a machine learning algorithm. In a dataflow graph, each node represents an individual mathematical operator (*e.g.*, matrix multiplication), and each edge represents a data dependency. In each edge, a tensor (n-dimensional arrays) defines the data format of the information transferred between two nodes.

TensorFlow provides official APIs in different programming languages such as Python, C++, Java, JavaScript and Go. The Python interface is the most popular (TensorFlow Team, 2019a). As unofficial APIs, open source communities also provide APIs in other programming languages such as C#, Julia, Ruby, Rust, and Scala (TensorFlow Team, 2019a). TensorFlow is released under the Apache 2.0 license, and its documents are presented in its website (TensorFlow Team, 2019a). TensorFlow supports multiple client languages (*e.g.*, Python and C++) and they all need to use the corresponding foreign function interface (FFI) (Chakravarty et al., 2004) to call into a C API provided by TensorFlow to implement computational functionalities (TensorFlow Team, 2019b).

2.2. The repair process of TensorFlow bugs

The source code of TensorFlow is maintained on GitHub (TensorFlow Team, 2019c), where its issues are reported, and commits are recorded since November 2015. Typically, if a user encounters a problem (*e.g.*, a bug), she will submit an issue, which we call a bug report in this article. Such a report presents information to diagnose the problem. The bug report includes basic information such as the OS platform, buggy TensorFlow version, and the code snippets to reproduce the buggy behavior. Besides such information, the bug report also presents a description, which introduces bug briefly. Furthermore, the reporter may suggest a feasible way to repair the bug. After receiving the bug report, the developers of TensorFlow discuss the possible causes of the bug and how to repair it. Moreover, for a more complicated bug, developers can refer to related bug reports and pull requests in their discussion. Other open source communities have more advanced issue trackers (*e.g.*, Jira), where bug reports are marked as "resolved" or "fixed". However, GitHub provides a much simpler issue tracker, and its status is often not reliable. Meanwhile, programmers often submit pull requests without reporting their found bugs. When submitting a pull request, the submitter typically introduces the bug briefly, presents the corresponding bug report, the buggy behavior (a wrong error is thrown), submit commits to change the source code and explains the changes (fixing the bug and adding a test case). After that, reviewers need to assess the fix and communicate with the submitter about the modification. Finally, if the modification is confirmed correct, other developers will approve the changes and merge the commit.

Generally, the status of a bug is not tagged by labels on bug reports, while a pull request marked as "ready to pull" indicates that the bug is solved and ready to be merged. The label helps us to filter pull requests with fixed bug simply. Some pull requests contain references to their bug reports, which is straightforward to help us identify bugs. However, some pull requests are not submitted by users, and have no corresponding bug reports. We use keywords (*e.g.*, bug) searching to identify bug fixes from such pull requests (Section 3.1).

3. Methodology

3.1. Dataset

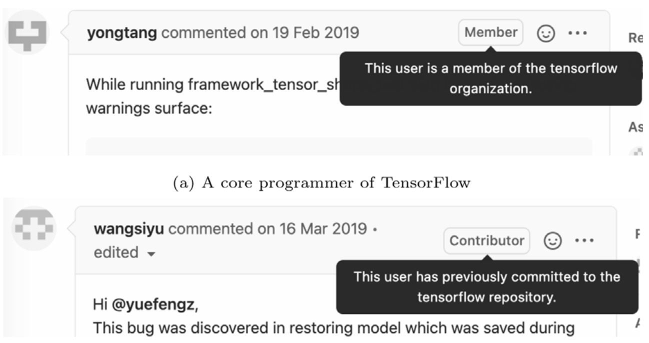
We select TensorFlow as the subject of our study, since Zhang et al. (2018) report that more than 36,000 GitHub projects call the APIs of TensorFlow. As a result, the bugs inside TensorFlow influence thousands of its applications. We apply the following steps to extract approved pull requests:

Step1. Filtering pull requests by labels. To avoid superficial 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. In this step, we collected 1367 pull requests whose labels are "*ready to pull*" and 700 *closed* pull requests without label. These pull requests are submitted between December 2017 and March 2019,

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 removed 1858 pull requests, and selected 209 bug fixes for latter analysis.

Step3. Extracting bug reports and code changes. For each one of our selected bug fixes, we extract its bug report and code changes from the posts and submitted commits. The extracted results and corresponding pull requests are used to determine their symptoms, root causes (RQ1), and locations (RQ2). We introduce the details in Section 3.3. In this step, we abandoned 7 pull requests because we cannot infer their symptoms or causes. The percentage of such cases is low, and its influence is minor.

In total, we collected 202 TensorFlow bug fixes, and 84 of them have corresponding bugs reports. The number is comparable to other empirical studies. For example, Thung et al. (2012) analyze 500 bugs from machine learning projects such as Mahout,


yongtang commented on 19 Feb 2019 Member

This user is a member of the tensorflow organization.

While running framework_tensor_sh

warnings surface:

(a) A core programmer of TensorFlow

wangsiyu commented on 16 Mar 2019 • edited Contributor

This user has previously committed to the tensorflow repository.

Hi @yuefengz,

This bug was discovered in restoring model which was saved during

(b) A user/outsider of TensorFlow

Fig. 1. The marks for core members and contributors.

Lucene, and OpenNLP. For each project, they analyze no more than 200 bugs. As another example, Zhang et al. (2018) analyze 175 bugs from TensorFlow applications. Indeed, for deep learning programs, libraries are typically much larger than applications. As a result, our analyzed bugs are much more complicated than the bugs in the prior studies (Zhang et al., 2018; Thung et al., 2012). We did not analyzed bugs before 2017 because we had collected sufficient bugs for analysis, and those old bugs may not reflect the recent characteristics of deep learning bugs.

3.2. Pull request and commit

Core programmers have the authority to submit commits directly, and we call such commits as direct commits. In our study, we did not analyze direct commits for the following considerations:

1. Pull requests reveal the important bugs from users. While direct commits are mostly submitted by core programmers, pull requests are submitted by users (Bertoncello et al., 2020). Pull requests often fix annoying bug symptoms, and users would rather change the code of TensorFlow by themselves. Although they are considered as users, we notice that bugs in some pull requests are critical. For example, TensorFlow is built upon Intel MKL (Intel MKL Team, 2021) to achieve the best performance on Intel CPUs. Compared with the core programmers of TensorFlow, the programmers of Intel MKL (considering as users) have more expertise in detecting bugs that are related to calling Intel MKL. As it takes much time to train a real deep learning model, such bugs typically are important to many users. We notice that the programmers of Intel MKL also submitted bugs through pull requests. For example, they submitted a pull request (Bhuiyan, 2017) to fix a performance degradation on a wide-used image classification dataset (CIFAR-10) (Krizhevsky, 2017).

2. Some core programmers would submit bug fixes as pull requests. Even if they have the privilege to directly commit their changes, to collect the feedback from others, some core programmers prefer to submit pull requests (Gulsoy, 2018; Zhu, 2019). As shown in Fig. 1, GitHub provides a label, *i.e.*, *Member*, to denote the core programmers of a project. By checking the labels of programmers, we find that 29.7% pull requests in our collected data are submitted by the core programmers of TensorFlow.

3. Pull requests are reviewed and approved by core programmers. According to the guideline of TensorFlow (Warden, 2017), pull requests must be reviewed by core programmers. Although some pull requests are not submitted by core programmers, we can analyze the discussions from core programmers and understand their opinions on bugs from pull requests. For example, in the discussions of a bug report (Takahara, 2017), a core programmer and a user discussed how an unexpected graph is generated, and they determined that this bug is caused by inconsistent values of two variables. From such discussions, we can infer that its symptom is an unexpected graph and its cause is the inconsistency of two values.

4. Pull requests contain more informational details than direct commits. An approved pull request contains details (*e.g.*, discussions among programmers), and some pull requests are linked to their bug reports. Alternatively, the core programmers of deep learning libraries can bypass pull requests, and submit their changes directly to code repositories. A direct code change has only a short message, and they seldom describe error messages and bug symptoms. Indeed, if a code change has no bug reports or pull requests, it is even difficult to determine whether the code change is a bug fix (Tian et al., 2012).

To compare code changes in pull requests with those in direct commits, we manually identified 104 direct bug fixes. To ensure that the comparison is comprehensive, we selected a code metric called the maintainability index (Coleman et al., 1994). This metric combines several metrics such as Halstead's Volume (HV) (Shen et al., 1983), McCabe's cyclomatic complexity (CC) (Watson et al., 1996), lines of code (LOC), and percentage of comments (COM). We randomly collected 104 direct bug fixes from direct commits. To show the differences between these bug fixes and those in our dataset, we used one-way ANOVA (Lowry, 2014) and compared their maintainability index. We find that all the differences are insignificant. As a result, although we agree that it can enrich our findings if the core programmers of TensorFlow are invited to analyze direct commits, as the differences between direct commits and pull requests are insignificant, the new findings over our current ones can be minor.

3.3. Manual analysis

In our study, we invite two graduate students to manually inspect all bugs. The two students are major in computer science, and both are familiar with deep learning algorithms. In the past two years, they have developed at least two deep learning application projects (*e.g.*, mining on business data) on TensorFlow. Following our protocols, the two students inspect the bugs independently, and compare the results. If they cannot reach a consensus on a TensorFlow bug, they discuss it on our weekly group meetings. Our initial agreement rate is 92.57%. Here, the initial agreement rate is defined as the consistent cases over the total cases.

3.3.1. 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 (Avizienis et al., 2004; Tan et al., 2014). 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 (Avizienis et al., 2004; Tan et al., 2014). 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, the pull request of #21956[1] without report 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:

```
void Append(StringPiece s) {
-     key_.append(s.ToString());
+     key_.append(string(s));
      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 (Han et al., 2011) to measure the correlations between symptoms and root causes. The *lift* between two categories (A and B) is defined as follows:

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

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, respectively. If a *lift* value is greater than one, a symptom is correlated to a root cause; otherwise, it is not.

3.3.2. 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 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, 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. We merge them into one component.

5. Framework. The framework implements basic functionalities (*e.g.*, logging). 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 applications; and the core/profiler directory provides a tool to profile the execution of TensorFlow and its applications.

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

[1] In the following paragraphs, the numbers denote the ids of pull requests. Their urls can be constructed by adding the url of Tensorflow (*e.g.*, https://github.com/tensorflow/tensorflow/pull/21956).

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 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 Eq. (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.

3.3.3. Protocol of RQ3

We find that some bug fixes are repetitive, *i.e.*, appearing at least twice, so we follow next steps to analyze such fixes:

1. Inspecting symptoms and root causes. We inspect the symptom, root cause and location information obtained from previous sections to outline the general situation of a bug.

2. Locating related code modifications. We determine the fix scale of a bug from the code changes including the number of related files, changed lines and commit frequency. If a commit contains modifications that are irrelevant to repair bugs (*e.g.*, test case modifications), we ignore such modifications. If a pull request fixes more than one bug, we consider them as individual bugs, and analyze them respectively, but such cases are rare in our observation. Generally, it is easier to find specific templates in fixes with small scale.

3. Analyzing the characteristics of modifications. We focus on several characteristics of a bug fix to describe the repair process in detail including scope of buggy code (in a method, in a constructor or global), modified code elements (variables, methods or classes), and modification intention (*e.g.*, changing a value, and modifying conditions of if-statements).

4. Extracting fix templates. We suppose that bug fixes with similar characteristics mentioned above are possible to share the same repair template. We define repair patterns according to these characteristics and extract instances appearing multiple times as templates.

When we design our protocol, we refer to the ones used in the prior studies (Kim et al., 2013; Le et al., 2016; Liu and Zhong, 2018). When we analyze repair patterns in TensorFlow, if a pattern is not identified by the prior studies Kim et al. (2013), Le et al. (2016), Liu and Zhong (2018), we then create a new category to define its pattern.

3.3.4. Protocol of RQ4

For simplicity, we use build script languages to denote the languages of configuration files, batch files, and build files.

To study the impact of language overlap, we investigate all bugs which concern multiple files belonging to different languages, which can be identified by their extensions. For bugs with both programming and build script languages, we check their symptoms and root causes to make sure whether they contain configuration errors and extra defects. For bugs with only programming languages, we further inspect relative reports and fixes to determine their fix objectives in corresponding files to summarize main pattern of these bugs.

4. Empirical result

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

4.1. RQ1. Symptoms and root causes

4.1.1. The categories of symptoms

Our identified symptoms are as follows:

1. Functional error (35.64%). If a program does not function as designed, we call it a functional error. For example, we find that the bug report of #20751 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 exits irregularly. When it happens, the program often throws an error message. For example, the bug report of #16100 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. The bug report of #11725 is an example:

> 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 the bug report of #17605:

> 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 the bug report of #16262 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 in the pull request of #18558, since it calls a method with a deprecated argument:

> According to tf.argmax, dimension argument was deprecated, it will be removed...

4.1.2. The categories of root causes

Our identified causes are as follows:

1. Dimension mismatch (3.96%). We put a bug into this category if it is caused by dimension mismatch in tensor computations and transformations. The pull request of #22822 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. The pull request of #21371 is a sample 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 in such category reported in the pull request of #17345 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, the pull request of #17418 complains that a removed `ops` is called:

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, the pull request of #16433 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 in the pull request of #21338 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 it occurs in the logic of a program. A logic error indicates an incorrect program flow or a wrong order of actions. The pull request of #19894 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 a wrong configuration. A pull request #16130 is an example:

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

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 in the pull request of #21017 triggers the following error message:

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

Programmers forget to add the `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, the pull request of #21950 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. The pull request of #13684 describes a deadlock:

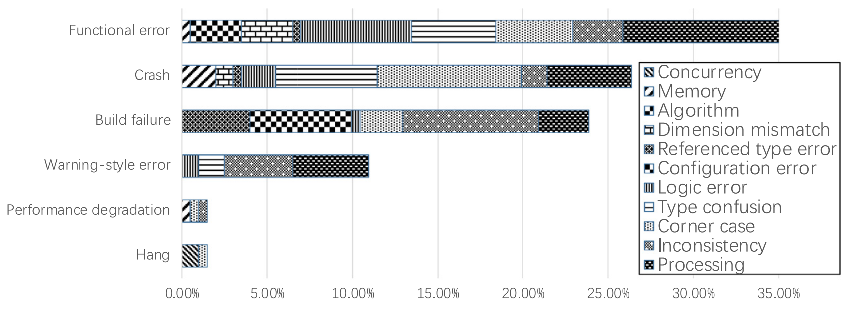
notify_one was used to notify inserters and removers waiting to insert and remove elements into Staging Areas. This could result in deadlock when many removers were waiting for different keys.

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

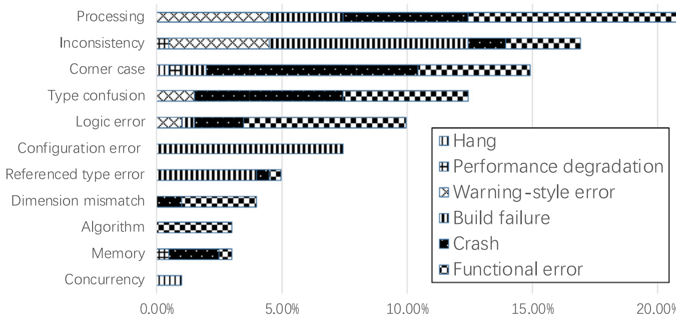
4.1.3. Distribution

Fig. 2(a) shows the distribution of symptoms. Its vertical axis shows symptom categories, and its horizontal axis shows the percentage of corresponding symptom. For each symptom, we refine its bugs by their root causes. Tan et al. (2014) report the distributions of Mozilla, Apache, and the Linux kernel. We find that the distribution of TensorFlow is close to their distributions. Fig. 2(a) shows that functional errors account for 39%, which are the most common bugs of TensorFlow. Tan et al. (2014) 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%).

Fig. 2(b) shows the distribution of root causes. Its vertical axis shows cause categories, and its horizontal axis shows the percentage of corresponding causes. For each root cause, we refine its bugs by symptoms. All the symptoms have multiple and evenly distributed root causes, but the distribution of root causes are not so evenly. For example, as shown in Fig. 2(b), the bugs in processing mainly cause the symptoms such as warning-style errors, build failures, crashes and functional errors, but as shown in Fig. 2(a), a symptom typically has more fragmented causes.



(a) The distribution of symptoms



(b) The distribution of root causes

Fig. 2. The distribution of bug symptoms and root causes.

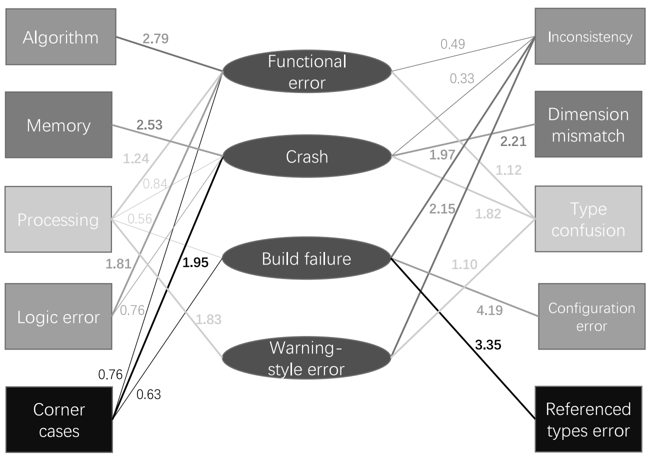


Fig. 3. The correlation between symptoms and root causes.

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

Tan et al. (2014) 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. Meanwhile, Thung et al. (2012) show that in machine learning systems, algorithm errors are the most common bugs (22.6%). The above observations lead to a 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 has 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.

4.1.4. Correlation of bug categories

Fig. 3 shows the correlation of bug categories. The rectangles denote symptoms, the ovals denote root causes. We ignore categories whose bugs are fewer than three, since they are statistically insignificant (*e.g.*, hangs). The lines denote correlations, and we highlight correlations whose values are greater than one.

Both Tan et al. (2014) and we find that crashes have correlations with memory bugs and corner cases. Tan et al. (2014) find that crashes also have correlations with concurrency, but we do not consider it, since only two of our analyzed bugs are related to 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. (2014) and we find that function errors have correlations with processing and logic errors. Tan et al. (2014) find that function errors have correlations with 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. (2014). We identify the correlations of build failures and warning-style bugs, complementing the study of Tan et al. (2014). For our identified root causes and symptoms of TensorFlow, our observations lead to the following finding:

Finding 3. Build failures have correlation with inconsistencies, configurations and referenced type errors. Warning-style bugs have correlation with inconsistencies, processing, and type confusions. Dimension mismatches lead to crashes, and type confusions lead to functional errors, crashes and warning-style errors.

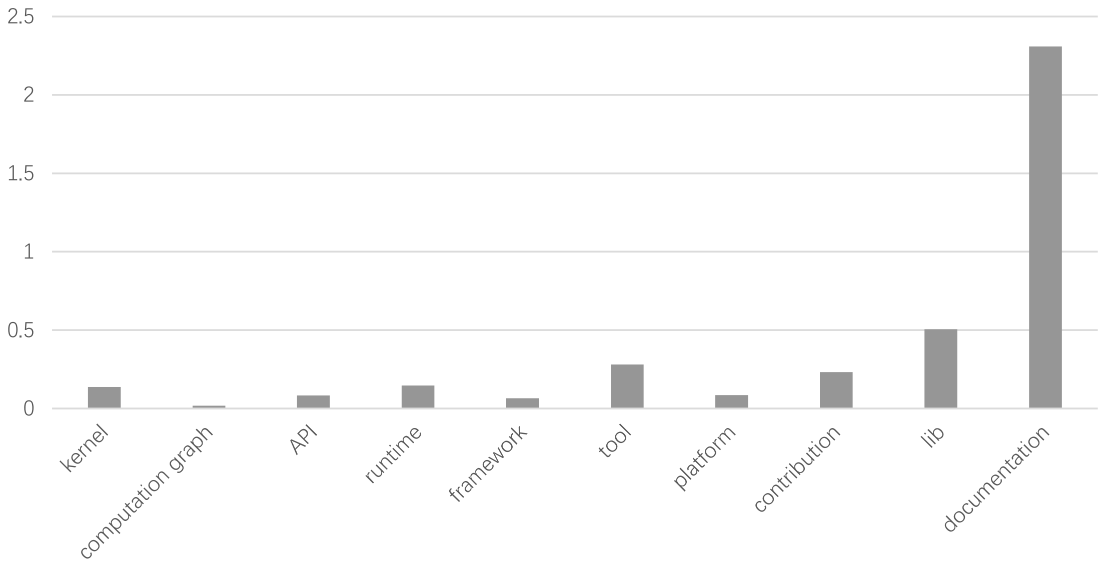


Fig. 4. The bug density of different locations.

In summary, our correlations between symptoms and root causes are largely consistent with those of Tan et al. (2014). Additionally, we also discover correlations between symptoms and root causes, which are not reported by the prior study.

4.2. RQ2. Bug locations

4.2.1. Distribution

Fig. 4 shows the distribution of bug locations. Some components have more bugs because they are larger. To reduce the bias, we define the bug density as the number of bugs per 1000 lines of code (LoC). The densities of *documentation* is much larger than others. As described in Section 3.1, we have ignored superficial bugs (*e.g.*, textual errors in documents). However, the documentation module contains illustrative code samples, and their modifications appear in Fig. 4. When programmers fix bugs, they often modify the corresponding samples, which can partially explain its high ratio in Fig. 4.

Finding 4. The documentation of TensorFlow is the most frequently modified component.

4.2.2. Correlation of bug categories

Fig. 5 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 inconsistencies. The above observations lead to a finding:

Finding 5. As core components, *kernel* contains many semantic bugs, and *API* bugs are often caused by tensor computation problems such as dimension mismatches and type confusions. In *library* and *tool*, build failures are popular, and most bugs are caused by inconsistencies.

In summary, most TensorFlow bugs reside in deep learning algorithms, API interfaces, and platform-related components. Furthermore, the correlations between their locations and symptoms or root causes follow specific patterns.

4.3. RQ3. Repair patterns

4.3.1. The categories of repair patterns

We find several recurring repair templates as following:

1. Parameter modifier (21.85%). This repair pattern adds, removes, or replaces a parameter input. The pull request #18674 is an example:

```
− vs.get_variable(opaque_kernel, ...)
+ vs.get_variable(opaque_kernel, dtype=self._plain_dtype, ...)
```

For `float16` data type, reusing `opaque_kernel` in CudnnLSTM throws a ValueError. The issue is fixed by passing the data type.

2. Method replacer (16.81%). This repair pattern replaces a method with another method whose parameters and return type are compatible. A pull request #25427 is an example:

```
−    return log(1 + exp(−y_wx)) * example_weight;
+    return log1p(exp(−y_wx)) * example_weight;
```

The `log(1+x)` method is replaced with the `log1p(x)` method, since the latter is more precise.

3. Value checker (14.29%). This repair pattern checks the value of a variable. A sample fix in the pull request of #16051 is as follows:

```
+ if(logits_in.dim_size(0)>0) {...
```

This a crash caused by corner cases. According to the fix description, if the first dimension of `logits_in` is 0, a crash on GPU will be triggered. Adding a value check fixes this issue.

4. Type replacer (11.76%). This repair pattern replaces the type of a variable. A fix in the pull request of #17148 in this pattern is shown below:

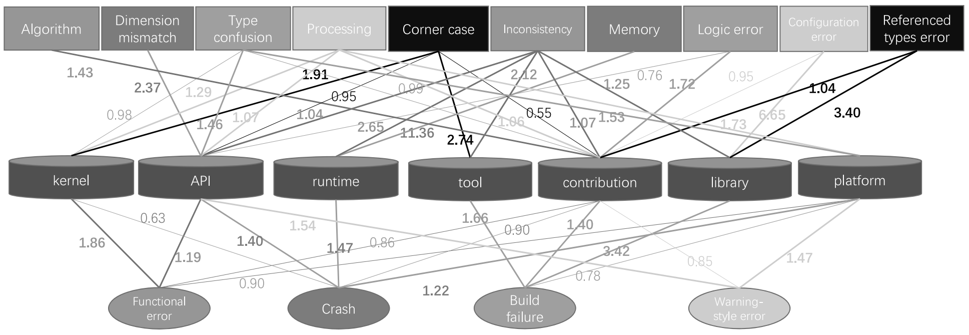


Fig. 5. The correlation between locations.

```
−on_value=0.,
+on_value=ops.convert_to_tensor(0.,  dtype=probs.dtype),
```

This is a crash caused by a type error. When `float16` values are fed into the method, it crashes, since the data type of `on_value` is inferred from value `0.`, which is `float32`. As a result, converting “`0.`” to a tensor fixes this bug.

5. Referenced type modifier (11.76%). This pattern adds, removes, or replaces referenced types. The pull request of #21017 shows this pattern:

```
+   #include <functional>
```

This is a build failure caused by referenced type errors. As the reporter says, the compiler cannot find `std::function`, and including `<functional>` fixes it.

6. Initializer modifier (6.72%). This repair pattern modifies the initial value of a variable. A fix in the pull request of #25909 in this pattern is provided:

```
−   int64 new_size;
+   int64 new_size = −1;
```

This a warning-style error caused by processing problem. A warning message complains that the `new_size` variable is not initialized, and adding an initializer fix this problem.

7. Variable replacer (5.88%). This repair pattern replaces a variable with a compatible one. For example, to fix a functional error, the pull request of #16081 is as follows:

```
−   ... concat_dim = N + concat_dim;
+   ... concat_dim = expected_dims + concat_dim;
```

If `concat_dim` is negative, its value is wrongly updated. To repair the bug, `N` is replaced with `expected_dim`.

8. Format checker (5.04%). This repair pattern checks the data format of a variable. The pull request of #18481 shows this repair pattern:

```
+if isinstance(type_value, (type, np.dtype)):  for key ...
```

This is a crash caused by type error. Crash happens when an invalid dtype (*e.g.*, `[,]`) is given, and adding a format checker can fix this.

9. Condition replacer (2.52%). This repair pattern replaces the predicate of a branch with a compatible one. A fix example in the pull request of #18183 is shown as follows:

```
−    if not module or 'tensorflow.' not in module.__name__:
+    if (not module or not hasattr(module, "__name__") or '
     tensorflow.' not in module.__name__):
```

This is a crash caused by corner cases. An object in program does not have `__name__` attribute and leads to crash, so a `hasattr()` checker is added to check whether the attribute is contained.

10. Exception adder (1.68%). This repair pattern handles exceptions. A sample in the pull request of #20479 is as follow:

```
context_t = CondContext(pred, pivot_1, branch=1)
+try:
...
 orig_res_t, res_t = context_t.BuildCondBranch
 (true_fn)
 if orig_res_t is None: raise ValueError("true_fn must have a
      return value.")
 context_t.ExitResult(res_t)
+finally:
+ context_t.Exit()
```

This is a crash caused by corner case. If an exception occurred in `tf.cond()`, `CondContext` is left uncleaned, which will be passed to `context_t`, then causes crash. So a way to fix this bug is adding a try statement to catch exception.

11. Syntax modifier (1.68%). This repair patternremoves syntax errors. A fix in the pull request of #25962 of this pattern is shown as below:

```
−EIGEN_STATIC_ASSERT((nr==4), YOU_MADE_A_PROGRAMMING_MISTAKE);
+EIGEN_STATIC_ASSERT((nr==4), YOU_MADE_A_PROGRAMMING_MISTAKE)
```

A warning message complains that there is an unnecessary semicolon at the end of the code, and removing it fixes this issue.

Our found fix patterns are largely overlapped with the prior ones (Kim et al., 2013; Le et al., 2016; Liu and Zhong, 2018). This observation lead to a finding:

Finding 6. From the viewpoint of modifying code, fixing TensorFlow bugs is largely consistent with fixing bugs in other types of projects.

4.3.2. Correlation of bug categories

Fig. 6 shows the correlations between root causes and repair patterns. In this figure, the rectangles denote root causes, and the rounded rectangle denote repair patterns. We ignore repair patterns, if their bugs are fewer than three. The lines denote correlations, and we highlight correlations whose values are greater than one. Since not all bug fixes can be classified by a repair pattern, we exclude the isolated fixes. From Fig. 6, we find the following correlations:

Finding 7. Parameter modifiers, method replacers, value checkers, type replacers and referenced type modifiers are common repair patterns in TensorFlow. They are often introduced to fix bugs caused by inconsistency, type confusion and corner cases.

In summary, we find ten repair templates from our collected fixes. Compared with the prior studies, we find two new templates, but the majority of our found templates are overlapped with existing ones.

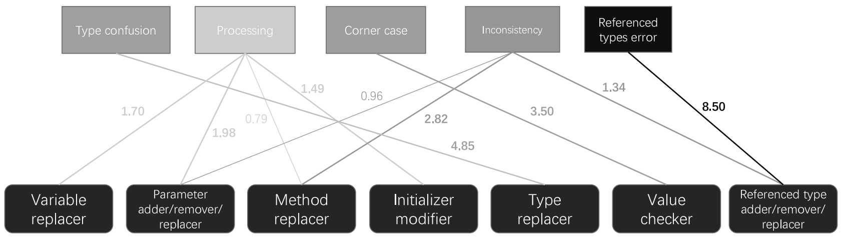


Fig. 6. The correlation between repair patterns.

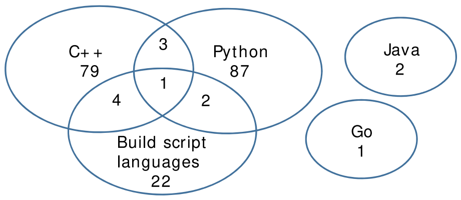


Fig. 7. The distribution of programming languages.

4.4. RQ4. Multi-language programming

Fig. 7 shows the distribution. In total, we find ten multiple-language bugs, and we classify them into two categories:

1. In total, six bugs are configuration bugs. For example, a pull request #17005 says that the `Cmake` file does not work on MacOS. This file is a build configuration file. To fix the bug, programmers modified the `Cmake` file:

```
−set (pywrap_tensorflow_lib "\${CMAKE_CURRENT_BINARY_DIR}/
      libpywrap_tensorflow_internal.so")
+set (pywrap_tensorflow_lib "${CMAKE_CURRENT_BINARY_DIR}/
      libpywrap_tensorflow_internal${CMAKE_SHARED_LIBRARY_SUFFIX
      }")
```

However, modifying this file alone does not fully repair the bug. Programmers also modified a `C` header file as follows:

```
−  #include <malloc.h>
+  #include <stdlib.h>
```

2. The other four bugs modify test cases in other languages. For example, the pull request of #16168 complains that the Python interface wrongly converts a unicode string. To fix the bug, programmers modified the `py_func.cc` file in C, and a test case in Python.

The above observations lead to a finding:

Finding 8. Only ten out of 202 bugs involve multiple languages, and their reasons are simple: (1) source files and configuration files can have related bugs, and (2) the core and its applications/test cases can have related bugs.

In summary, we find that only ten TensorFlow bugs involve multiple languages, and their reasons are simple.

4.5. Threats to validity

The internal threats to validity include the possible errors of our manual inspection. To reduce the threat, we asked two students to inspect our bugs. When they encountered controversial cases, we discussed in our group meeting, until we reach an agreement. The threat can be mitigated with more researchers, so we release our inspection results on our website. The threats to external validity include our subject, since we analyzed the bugs of only TensorFlow. Although our analyzed bugs are comparable with the prior studies and other studies (*e.g.* Zhang et al. (2018)) also analyzed only TensorFlow bugs, they are limited. This thread can be reduced by inspecting more bugs.

5. The comparison with prior studies

Table 1 summarizes the subjects, protocols, and findings of the prior studies (Tan et al., 2014; Zhang et al., 2018; Thung et al., 2012; Islam et al., 2019; Kim et al., 2013). We next introduce their details and our new findings.

5.1. Symptoms

Tan et al. (2014) classify bugs in open source projects by their symptoms as shown in Table 1. Comparing with their taxonomy, we do not find data corruptions as they did. As shown in their example, data corruptions are related to databases. As TensorFlow does not use databases, we do not find such bugs. Meanwhile, we find build failures and warning-style errors, which are not reported by Tan et al. They did not find build failures, since they focus on runtime bugs. They also ignore warning-style errors, possibly because their symptoms are trivial. In the study of Thung et al. (2012), they did not classify bugs by their symptoms.

5.2. Root causes

Zhang et al. (2018) and Islam et al. (2019) analyze deep learning applications and their main findings are shown in Table 1. Some of their found bugs do not exist or are rare in deep learning libraries. For example, they find incorrect model parameters and structure inefficiency, but such bugs appear only in TensorFlow applications. As another example, Islam et al. (2019) report that in TensorFlow applications, 92% bugs are crashes and the other 8% bugs are performance bugs. Comparing with the distribution, we find that the bugs inside TensorFlow are more diverse.

Our taxonomy has some minor differences from Tan et al. (2014). For example, Tan et al. put exception-handling bugs in one category, but we refine them into smaller categories by their root causes. As another example, Tan et al. classify memory bugs into subcategories, but we do not, since only 3% bugs in TensorFlow are memory bugs.

Comparing with the taxonomy of Seaman et al. (2008), we do not identify "external interface" (*i.e.*, UI) bugs, since TensorFlow has no UI. We identify several categories of bugs (*e.g.*, crashes), that are not reported by Seaman et al. (2008).

The taxonomy of Thung et al. (2012) is similar to Seaman et al. but Thung et al. identify configuration bugs. We refine such bugs into configuration errors and referenced type errors.

Seaman et al. (2008) proposes a bug categorization scheme in which they stress algorithm bug to describe bugs caused by incorrect algorithm implementation in computation. Such bugs are frequently seen in DL systems, so we also import this factor into our taxology. Besides, they introduce logic error is to describe bugs caused by incorrect expressions in conditional statements or loop blocks, which is similar to subcategory wrong control flow in result of Tan et al. We also utilize this concept and by extending its scope. Moreover, bugs caused by internal interfaces have also been noticed by Seaman et al. which refer to errors in the connections between different components of a system.

Table 1
The comparison to the prior studies.

Tan et al. (2014)	Subject: 2060 bugs that were collected from the issue trackers and the NVD of Mozilla, Apache, and Linux. Protocol: They read bug reports to identify bugs, and use the existing categories of NVD as a reference. Finding: They identified six symptoms and three causes, and they further refined them into more subcategories. They find that incorrect functionality is the dominant symptom, and semantic bugs are the dominant causes.
Thung et al. (2012)	Subject: 200 Mahout bugs, 200 Lucene bugs, and 100 OpenNLP bugs that were collected from issue trackers. Protocol: They read bug reports to classify bugs, and use the categories proposed by Seaman et al. (2008) as a reference. Finding: The most bugs are categorized as algorithm/method, followed by non-functional and assignment/initialization.
Zhang et al. (2018)	Subject: 175 TensorFlow application bugs were collected from GitHub commits and Stack Ovrerflow threads. Protocol: Then they manually inspect bugs to classify them. Finding: They find seven causes and four symptoms. Among them, incorrect model parameters/structures, API breaking changes and API misuses are the dominant causes, and crashes and exceptions are the dominant symptoms.
Islam et al. (2019)	Subject: 415 Stack Overflow discussions and 555 bugs from GitHub commits of Caffe, Keras, Tensorflow, Theano, and Torch applications. Protocol: They formulate a set of classification criteria, and use Zhang et al. (2018) as a reference. Finding: They find ten causes and six symptoms. Among them, incorrect model parameters/structures are the most common root cause, followed by structure inefficiency and unaligned tensors, and for symptoms, crashes are the most common, followed by bad performance and incorrect functionality.
Kim et al. (2013)	Subject: 62,656 human-written patches were collected from Eclipse JDT. Protocol: They implement a tool to build graphs from patches, but deeper analyses are still manual. Finding: They derived ten repair patterns, but did not present the percentage of repair patterns.

To make the definition more precise, we expand and improve this concept and present inconsistency category. Other categories such as checking, non-functional defects and optimization are merged into existing classes.

5.3. Repair patterns

With their support tool, Kim et al. (2013) manually inspected more than 60,000 human-written patches and derived ten repair patterns. Because their patterns were derived through an empirical study, we list their work in Table 1. Among their ten patterns, nine are overlapped with ours. For example, they derived a parameter replacer that replaces a parameter input with its compatible variable, and this repair pattern is identical to our parameter modifier. However, we do not find their null pointer checker. As null pointers lead to crashes and easy to be fixed, their fixes appear in commits other than pull requests. We do not list Le et al. (2016) and Liu and Zhong (2018) in Table 1, because their repair patterns were mined from fixes or derived from their programming experiences. Le et al. (2016) list 12 repair patterns. All their repair patterns are overlapped with ours. For example, they use insert/delete type cast operators to resolve type conversion errors, and this repair pattern is identical to our type replacer. Liu and Zhong (2018) list 12 repair patterns. Compare with theirs, 9 of the 12 repair patterns are overlapped. For example, they use a variable replacer to replace a variable to another, and this pattern is identical to our variable replacer. We do not identify their binary operator replacer, because pull requests often resolve complicated fixes but such bugs are easy to be fixed. Besides the overlapped patterns, we find new repair patterns (see Section 5.4 for details).

5.4. Our new findings

1. TensorFlow has type confusions, and besides deep learning applications, as a library, TensorFlow also has dimension mismatches. We identified that TensorFlow has type confusions, which is not reported by the prior studies. The prior studies Zhang et al. (2018, 2019), Islam et al. (2019) report that deep learning applications have unaligned tensors and shape inconsistencies. We find that as a deep learning library, TensorFlow also has similar bugs, and we call such bugs as dimension mismatches.

2. In TensorFlow, the main symptoms are functional errors, crashes, and build failures, and its main causes are processing and inconsistencies. As shown in Figs. 2(a) and 2(b), for the first time, we present the distributions of symptoms and root causes of TensorFlow bugs. For example, we find that the functional errors (39.5%), crashes (26%) and build failures (24%) are the main symptoms, and processing errors (26.5%), inconsistencies (23.5%), and corner and missing cases (19.5%) are the main causes of TensorFlow bugs.

3. Referenced type modifiers and syntax modifiers can be useful to repair bugs inside TensorFlow. Compared with the prior studies Kim et al. (2013), Le et al. (2016), Liu and Zhong (2018), we find two additional templates such as *referenced type modifier* and *syntax modifier*. We notice that Martinez and Monperrus (Martinez and Monperrus, 2016) implement a tool that is able to automatically build `import` statements for Java code. It shall be feasible to extend their tool, so it can repair reference type errors. In addition, we find that TensorFlow programmers take effort to repair warnings, which are ignored by previous repair tools. Although they are trivial, it can be interesting to implement a tool to automate this type of repairs.

6. The significance of our findings

Developing high-quality 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 components 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 Zhang et al. (2018), Islam et al. (2019) analyze the bugs of deep learning applications, but our study analyzes the bugs of deep learning libraries. The bugs inside deep learning libraries can have impacts on the bugs of their applications. For example, Islam et al. (2019) find that 11% percentage of TensorFlow application 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 Zhang et al. (2018), Islam et al. (2019) show that unaligned tensors and the absences of type checking are common causes of deep learning application 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.

Detecting deep learning bugs. We notice that some bugs in deep learning libraries reside in other software projects such as database systems (*e.g.*, memory bugs and concurrency bugs), and detecting such bugs has been a hot research topic (Lin et al., 2018; Ren et al., 2015; van Renen et al., 2018). As advocated by Wang et al. (2016), the prior detection techniques can be tailored to handle similar bugs in deep learning libraries. Meanwhile, as deep learning techniques and frameworks are applied to solve many software problems (Li et al., 2019; Xu et al., 2019), more advanced detection techniques are also critical for users of deep learning libraries.

7. Related work

Empirical studies on bug characteristics. There has been a number of recent studies studying bugs from open source repositories. Tan et al. (2014) analyze the bug characteristics of open source projects such as the Linux kernel and Mozilla. Thung et al. (2012) analyze the bugs of machine learning systems such as Mahout, Lucene, and OpenNLP. Zhang et al. (2018) analyze the application code that calls TensorFlow. Islam et al. (2019) analyze the applications of more deep learning bugs. Humbatova et al. (2020) 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.

Detecting deep learning bugs. Pei et al. (2017) propose a whitebox framework to test deep learning systems. Ma et al. (2018) propose a set of multi-granularity criteria to measure the quality of test cases prepared for deep learning systems. Tian et al. (2018) and Pham et al. (2019) introduce differential testing to discover bugs in deep learning software. Our empirical study reveals 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.

Mining repair histories. Kim et al. (2013) mine repair templates by analyzing thousands of existing human-written patches. Le et al. (2016) utilize historical bug fixes and mines bug fix patterns in previous works (Offutt et al., 1996; Le Goues et al., 2012) to develop program repair techniques. Liu and Zhong (2018) mine thirteen repair patterns from code samples of Stack Overflow. Zhong and Mei (2019) mine a classification model to predict buggy locations of a source file. In this work, we investigate the fixes of bugs inside TensorFlow and summarize several repair templates to make comparison with other templates. We found new templates that are not reported by the prior studies, and our findings may improve fixing deep learning bugs.

Empirical studies on multi-language programming. Our study shows that several bugs in TensorFlow are about multi-language programming, so our work is also related to existing studies in this area. Mayer and Bauer (2015) analyze the distribution and association of multiple languages. Kochhar et al. (2016) analyze the regression relations between multiple languages and software quality, and their results show that multiple languages can lead to more bugs. Lin et al. (2017) present a benchmark suite for evaluating automated multi-language program repair tools. In our study, we find that the causes of TensorFlow bugs involving multiple programming languages. As a result, the above approach can be useful to detect bugs in deep learning libraries.

8. Conclusion and future work

Although researchers have conducted empirical studies to understand deep learning bugs, these studies focus on bugs of its applications, 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. In addition, when analyzing repair patterns, we focus on source files, and ignore configuration files. Hassan and Wang (2018) show that the repairs on such files also follow specific patterns, and we leave the analysis to our future work. As the prior studies Zhang et al. (2018), Islam et al. (2019) did, we also read bug reports and their repairs to understand deep learning bugs. Some runtime behaviors of bugs may not be hidden, and are difficult to be discovered through static analysis. In future work, we will introduce dynamic analysis to explore the runtime behaviors of deep learning bugs.

CRediT authorship contribution statement

Li Jia: Methodology, Software, Formal analysis, Investigation, Data curation, Writing - original draft, Visualization. **Hao Zhong:** Conceptualization, Methodology, Resources, Writing - review & editing. **Xiaoyin Wang:** Conceptualization, Writing - review & editing. **Linpeng Huang:** Resources, Writing - review & editing, Supervision. **Xuansheng Lu:** Writing - review & editing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

References

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., 2016. TensorFlow: A system for large-scale machine learning. In: Proc. OSDI, pp. 265–283.

Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1 (1), 11–33.

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Wardefarley, D., Goodfellow, I., Bergeron, A., 2011. Theano: Deep learning on GPUs with Python. In: Proc. Nips, BigLearning Workshop.

Bertoncello, M.V., Pinto, G., Wiese, I.S., Steinmacher, I., 2020. Pull requests or commits? Which method should we use to study contributors' behavior? In: Proc. SANER, pp. 592–601.

Bhuiyan, M., 2017. MKL DNN: fix the TF1.6 speed issue by fixing MKL DNN LRN taking the optimum path. https://github.com/tensorflow/tensorflow/pull/17605.

Chakravarty, M., Finne, S., Henderson, F., Kowalczyk, M., Leijen, D., Marlow, S., Meijer, E., Panne, S., Jones, S.P., Reid, A., Wallace, M., Weber, M., 2004. The haskell 98 foreign function interface 1.0: an addendum to the haskell 98 report. http://web.archive.org/web/20180702051235/www.cse.unsw.edu.au/~chak/haskell/ffi/.

Coleman, D.M., Ash, D., Lowther, B., Oman, P.W., 1994. Using metrics to evaluate software system maintainability. Computer 27 (8), 44–49.

Collobert, R., Bengio, S., Marithoz, J., 2002. Torch: A modular machine learning software library.

Gulsoy, G., 2018. Make unused variable warning an error during TF builds. https://github.com/tensorflow/tensorflow/pull/15762.

Han, J., Kamber, M., Pei, J., 2011. Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers.

Hassan, F., Wang, X., 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In: Proc. ICSE, pp. 1078–1089.

Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P., 2020. Taxonomy of real faults in deep learning systems. In: Proc. ICSE, pp. 1110–1121.

Intel MKL Team, 2021. Intel® oneapi math kernel library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

Islam, M.J., Nguyen, G., Pan, R., Rajan, H., 2019. A comprehensive study on deep learning bug characteristics. In: Pro. ESEC/FSE, pp. 510–520.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R.B., Guadarrama, S., Darrell, T., 2014. Caffe: Convolutional architecture for fast feature embedding. In: Proc. MM, pp. 675–678.

Jia, L., Zhong, H., Wang, X., Huang, L., Lu, X., 2020. An empirical study on bugs inside tensorflow, in: Proc. DASFAA, pp. 604–620.

Keras, 2019. https://keras.io.

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: Proc. ICSE, pp. 802–811.

Kochhar, P.S., Wijedasa, D., Lo, D., 2016. A large scale study of multiple programming languages and code quality. In: Proc. SANER, pp. 563–573.

Krizhevsky, A., 2017. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. ImageNet classification with deep convolutional neural networks. In: Proc. NIPS, pp. 1106–1114.

Le, X.D., Lo, D., Le Goues, C., 2016. History driven program repair. In: Proc. SANER, pp. 213–224.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012. GenProg: A generic method for automatic software repair. IEEE Trans. Softw. Eng. 38 (1), 54–72.

Li, G., Zhou, X., Li, S., Gao, B., 2019. QTune: A query-aware database tuning system with deep reinforcement learning. PVLDB 12 (12), 2118–2130.

Lin, Q., Chen, G., Zhang, M., 2018. On the design of adaptive and speculative concurrency control in distributed databases. In: Proc. ICDE, pp. 1376–1379.

Lin, D., Koppel, J., Chen, A., Solar-Lezama, A., 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In: Proc. SPLASH, pp. 55–56.

Liu, X., Zhong, H., 2018. Mining stackoverflow for program repair. In: Proc. SANER, pp. 118–129.

Lowry, R., 2014. Concepts and applications of inferential statistics. http://vassarstats.net/textbook/.

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., 2018. DeepGauge: Multi-granularity testing criteria for deep learning systems. In: Proc. ASE, pp. 120–131.

Martinez, M., Monperrus, M., 2016. ASTOR: A program repair library for Java. In: Pro. ISSTA, pp. 441–444.

Mayer, P., Bauer, A., 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In: Proc. EASE, pp. 4:1–4:10.

Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. 5 (2), 99–118.

Pei, K., Cao, Y., Yang, J., Jana, S., 2017. DeepXplore: Automated whitebox testing of deep learning systems. In: Proc. SOSP, pp. 1–18.

Pham, H.V., Lutellier, T., Qi, W., Tan, L., 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In: Proc. ICSE, pp. 1027–1038.

Ren, K., Thomson, A., Abadi, D.J., 2015. VLL: a lock manager redesign for main memory database systems. VLDB J. 24 (5), 681–705.

van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., Sato, M., 2018. Managing non-volatile memory in database systems. In: Proc. SIGMOD, pp. 1541–1555.

Seaman, C.B., Shull, F., Regardie, M., Elbert, D., Feldmann, R.L., Guo, Y., Godfrey, S., 2008. Defect categorization: making use of a decade of widely varying historical data. In: Proc. ESEM, pp. 149–157.

Shen, V.Y., Conte, S.D., Dunsmore, H.E., 1983. Software science revisited: A critical analysis of the theory and its empirical support. IEEE Trans. Softw. Eng. 9 (2), 155–165.

Takahara, K., 2017. Name/variable scopes of tensorflow.python.layers.base.layer. https://github.com/tensorflow/tensorflow/issues/13429.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Empir. Softw. Eng. 19 (6), 1665–1705.

TensorFlow Team, 2019a. Tensorflow API documentation. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf.

TensorFlow Team, 2019b. TensorFlow In other languages. https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/bindings.md.

TensorFlow Team, 2019c. TensorFlow repo on Github. https://github.com/tensorflow/.

Thung, F., Wang, S., Lo, D., Jiang, L., 2012. An empirical study of bugs in machine learning systems. In: Proc. ISSRE, pp. 271–280.

Tian, Y., Lawall, J., Lo, D., 2012. Identifying linux bug fixing patches. In: Proc. ICSE, pp. 386–396.

Tian, Y., Pei, K., Jana, S., Ray, B., 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In: Proc. ICSE, pp. 303–314.

Wang, S., Liu, T., Nam, J., Tan, L., 2018. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng.

Wang, W., Zhang, M., Chen, G., Jagadish, H.V., Ooi, B.C., Tan, K., 2016. Database meets deep learning: Challenges and opportunities. SIGMOD Rec. 45 (2), 17–22.

Warden, P., 2017. How the tensorflow team handles open source support. https://www.oreilly.com/content/how-the-tensorflow-team-handles-open-source-support/.

Watson, A.H., Mccabe, T.J., Wallace, D.R., 1996. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. U.S. Department of Commerce/National Institute of Standards and Technology.

Xu, B., Cai, R., Zhang, Z., Yang, X., Hao, Z., Li, Z., Liang, Z., 2019. NADAQ: natural language database querying based on deep learning. IEEE Access 7.

Zhang, Y., Chen, Y., Cheung, S., Xiong, Y., Zhang, L., 2018. An empirical study on TensorFlow program bugs. In: Proc. ISSTA, pp. 129–140.

Zhang, T., Gao, C., Ma, L., Lyu, M.R., Kim, M., 2019. An empirical study of common challenges in developing deep learning applications. In: ISSRE. pp. 104–115.

Zhong, H., Mei, H., 2019. Learning a graph-based classifier for fault localization. Sci. China Inf. Sci. 1–22.

Zhu, Q., 2019. Fix the performance regression for model.predict for non-tpu strategy. https://github.com/tensorflow/tensorflow/pull/34325.

Li Jia is currently a Ph.D. student at Shanghai Jiao Tong University. He received his master's degree from Texas A&M University in 2013. His research interests include data mining and artificial intelligence on software engineering.

Hao Zhong received his Ph.D. degree from Peking University in 2009. He worked as an assistant professor at Institute of Software, Chinese Academy of Sciences, and was promoted as an associate professor in 2012. From 2013 to 2014, he was a visiting scholar at University of California, Davis. Since 2014, he had been an associate professor at Shanghai Jiao Tong University. His research interest is the area of software engineering and mining software repositories. He is a recipient of ACM SIGSOFT Distinguished Paper Award 2009, the best paper award of ASE 2009, and the best paper award of APSEC 2008.

Xiaoyin Wang is an associate professor in the University of Texas at San Antonio. He received his Ph.D. from Peking University in Jan. 2012 and did postDoc in UC Berkeley until Aug. 2013. His research focuses on software analysis, mining and security, especially for open source and mobile projects. He received MSRA fellow in 2009 and NSF CAREER Award in 2019.

Linpeng Huang received his M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests lie in the area of distributed systems and in-memory computing.

Xuansheng Lu is currently a master student of Shanghai Jiaotong University. He received his bachelor's degree from Shanghai Jiaotong University in 2018. His main research interests lie in recommender systems.