An Empirical Study on TensorFlow Program Bugs[*]

Yuhao Zhang
Key Laboratory of High Confidence
Software Technologies, MoE
EECS, Peking University
Beijing, PR China
nathan_zhang@pku.edu.cn

Yifan Chen
Key Laboratory of High Confidence
Software Technologies, MoE
EECS, Peking University
Beijing, PR China
yf_chen@pku.edu.cn

Shing-Chi Cheung
Department of Computer Science and
Engineering, The Hong Kong
University of Science and Technology
Hong Kong, PR China
scc@cse.ust.hk

Yingfei Xiong
Key Laboratory of High Confidence
Software Technologies, MoE
EECS, Peking University
Beijing, PR China
xiongyf@pku.edu.cn

Lu Zhang
Key Laboratory of High Confidence
Software Technologies, MoE
EECS, Peking University
Beijing, PR China
zhanglucs@pku.edu.cn

ABSTRACT

Deep learning applications become increasingly popular in important domains such as self-driving systems and facial identity systems. Defective deep learning applications may lead to catastrophic consequences. Although recent research efforts were made on testing and debugging deep learning applications, the characteristics of deep learning defects have never been studied. To fill this gap, we studied deep learning applications built on top of TensorFlow and collected program bugs related to TensorFlow from StackOverflow QA pages and Github projects. We extracted information from QA pages, commit messages, pull request messages, and issue discussions to examine the root causes and symptoms of these bugs. We also studied the strategies deployed by TensorFlow users for bug detection and localization. These findings help researchers and TensorFlow users to gain a better understanding of coding defects in TensorFlow programs and point out a new direction for future research.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**;

KEYWORDS

TensorFlow Program Bug, Deep Learning , Empirical Study

ACM Reference Format:
Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213866

1 INTRODUCTION

We are entering an era of artificial intelligence with deep learning (DL) applications built on top of multi-layered neural networks. Various frameworks such as TensorFlow [9], Caffe [18], MXNet [5], PyTorch [27] and Theano [36] have been proposed to facilitate programming of such applications.

The programming paradigm of DL applications differs significantly from that of traditional applications. In traditional applications, programs are written to directly encode the model to solve a target problem. However, programs in DL applications do not encode the problem-solving model directly. Instead, the programs of a DL application encode the network structure of a desirable DL model and the process by which the problem-solving model is trained using a large corpus of data. Both the network structure and the training process are subject to the careful setting of hyper-parameters. The development of DL applications often faces tasks that are seldom encountered in developing their traditional counterparts, e.g., configuring a complex network structure (also known as computation graph) comprising layers of nodes. In addition, the training process involves intensive looping with computation sensitive to hyper-parameter tunings such as learning rate and dropout rate.

As DL is increasingly adopted for mission-critical applications, defective DL applications can lead to catastrophic consequences. For example, defective self-driving systems may lead to car accidents, and defective facial identity systems may lead to cracking of bank accounts. Various research efforts on the testing [28, 34, 38, 40] and debugging [17, 23] of DL applications were recently made. Despite these efforts, the characteristics of defects in DL applications have never been systematically studied. In particular, it is still unclear what new challenges the paradigm shift from traditional program languages to DL languages bring to fault detection and localization.

[*]The authors acknowledge Yang Liu at the Nanyang Technological University and the anonymous reviewers for the constructive comments. This work is supported by the National Key Research and Development Program under Grant No. 2017YFB1001803, National Natural Science Foundation of China under Grant No. 61672045, 61332010 and 61529201, and Hong Kong RGC/GRF grant. Yingfei Xiong is the corresponding author.

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5699-2/18/07.
https://doi.org/10.1145/3213846.3213866

For example, if there is a missing defect in constructing a DL model, how likely can it be caught when we train the model?

This paper presents the first empirical study on detecting and locating coding mistakes in DL applications programmed on top of TensorFlow (TF), which is the most popular DL framework on Github. There are 36,079 Github projects using TensorFlow, which is 4.8 times of those (7,485 projects) using PyTorch, the second most popular DL framework on Github. The study aims to provide a systematic understanding of the coding defects that TensorFlow users have made in programming DL applications. Please note that defects in a TensorFlow application may come from its training data, program code, execution environment or the TensorFlow framework. Our empirical study focuses on the defects in TensorFlow programs. To ease presentation, we refer to the defects in TF programs as *bugs*. We also refer to those who use TensorFlow to develop DL applications as TensorFlow users (or TF users).

Our study collected 175 TensorFlow coding bugs from GitHub issues and StackOverflow questions. We analyzed these bugs quantitatively and qualitatively, and reported (1) their symptoms and root causes, (2) the challenges in their detection, and (3) the challenges in their localization.

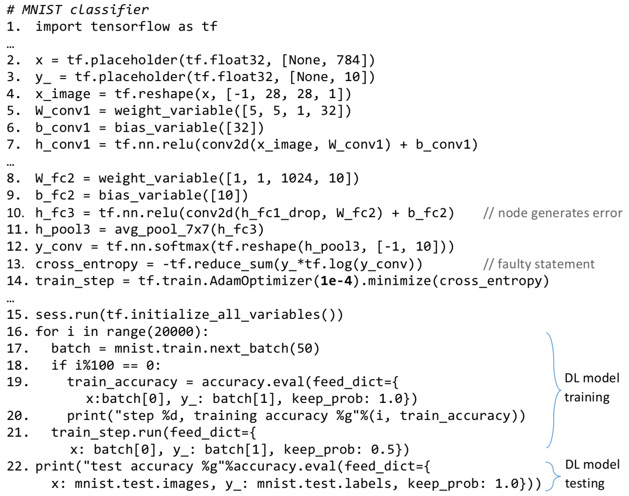
Our study has led to multiple findings. In particular, we identify four types of symptoms, seven types of root causes, five challenges in detection and fault localization, and five strategies that the TF users have adopted to address the challenges. We highlight the challenges below.

- Due to the stochastic nature of the learning process, the correctness criteria is probabilistic and TF users rely on statistical values to determine test results. New testing techniques are needed to support such tests.
- Due to the huge computation model of a neural network, coincidental correctness [7, 10, 16, 25, 31] occurs on a larger scale but less observable.
- Non-determinism is prevalent in the training process such that bug reproduction becomes difficult.
- Due to the densely inter-dependent of a neural network, traditional debugging techniques such as slicing [41] provide little help, and new research techniques for debugging is needed.
- Due to the black-box nature of neural networks, TF users often cannot examine the states at different program points, and rely on black-box techniques such as replacing parameters or switching training set.

These findings help researchers and TF users to gain a better understanding of deep learning defects and point out a new direction for future research.

To summarize, this paper makes the following contributions.

- A dataset of TensorFlow bugs collected from StackOverflow and GitHub.
- A study of the symptoms and root causes of the bugs, which could assist future studies on TensorFlow application testing and debugging techniques.
- A study of the new challenges in detecting and localizing the bugs and the current strategies to address them, which opens new problems for future research.



```
# MNIST classifier
1.  import tensorflow as tf
…
2.  x = tf.placeholder(tf.float32, [None, 784])
3.  y_ = tf.placeholder(tf.float32, [None, 10])
4.  x_image = tf.reshape(x, [-1, 28, 28, 1])
5.  W_conv1 = weight_variable([5, 5, 1, 32])
6.  b_conv1 = bias_variable([32])
7.  h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
…
8.  W_fc2 = weight_variable([1, 1, 1024, 10])
9.  b_fc2 = bias_variable([10])
10. h_fc3 = tf.nn.relu(conv2d(h_fc1_drop, W_fc2) + b_fc2)    // node generates error
11. h_pool3 = avg_pool_7x7(h_fc3)
12. y_conv = tf.nn.softmax(tf.reshape(h_pool3, [-1, 10]))
13. cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))        // faulty statement
14. train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
…
15. sess.run(tf.initialize_all_variables())
16. for i in range(20000):
17.   batch = mnist.train.next_batch(50)
18.   if i%100 == 0:
19.     train_accuracy = accuracy.eval(feed_dict={
          x:batch[0], y_: batch[1], keep_prob: 1.0})
20.     print("step %d, training accuracy %g"%(i, train_accuracy))
21.   train_step.run(feed_dict={
        x: batch[0], y_: batch[1], keep_prob: 0.5})
22. print("test accuracy %g"%accuracy.eval(feed_dict={
      x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Figure 1: A faulty TensorFlow example extracted from StackOverflow #33699174

The rest of the paper is organized as follows. In Section 2, we provide a background of programming over the TensorFlow framework. In Section 3, we propose three research questions. In Section 4, we present how we collected our data. In Section 5, 6, and 7, we answer these three research questions respectively.

2 BACKGROUND

Deep learning (DL) is an artificial intelligence computational paradigm that makes classification based on hierarchical layers of neurons that are interconnected to form a neural network. Each neuron is a simple processing unit that accepts inputs from neurons in the preceding layer, applies a non-linear activation function to these inputs, and passes the resulting value to other connected neurons in the succeeding layer. Each connection edge in the neural network is augmented by a weight parameter (W_i) that characterizes its connection strength. A DL model is often trained by gradient descent using back-propagation before deployment [12], usually in a non-deterministic way. The training is designed to search for these weight parameters' values that collectively minimize a cost function over the dataset. In supervised learning, the cost function quantifies the error, known as "loss"[1], between the labeled values from the training data and the classified values outputted by the model [12]. For example, a cost function can be stated to calculate the cross entropy between the two sets of values. In unsupervised learning, the cost function can quantify the distance between the encoded and decoded representations in the underlying autoencoder neural network.

Figure 1 gives an example of a TensorFlow program[2]. The program consists of two major phases: construction and execution. First, a computation graph is configured at the construction phase

[1] https://en.wikipedia.org/wiki/Loss_function. We use "loss" instead of the cost function to ease presentation.

[2] https://github.com/loliverhennigh/All-Convnet-TensorFlow-MNIST-Tutorial/blob/master/all_conv_mnist.py

(Lines 2-14). Second, a session object is created to launch the constructed computation graph and build a neural network. The execution phase can be further divided into two sub-phases: training and testing. In this training phase (Lines 16-21), a set of labeled samples are used to train the neural network, minimizing the model loss by means of cross entropy. A gradient descent algorithm is often deployed to carry out the minimization. In the training phase, the network will be trained for numerous iterations. After a model is trained, in the testing phase, it can be applied to classify samples in a dataset (Line 22).

3 RESEARCH QUESTIONS

Our study aims to answer the following three research questions.

- RQ1: What are the symptoms and root causes of the bugs?
- RQ2: What new challenges exist to detect the bugs and how do TF users handle them?
- RQ3: What new challenges exist to localize the bugs and how do TF users handle them?

The first research question concerns the characteristics of the bugs. The symptoms help us understand the consequences of the bugs and are useful in designing detection method. The root causes help us understand the nature of the bugs and the connections between root causes and symptoms are useful in designing fault localization methods. The second and third research questions concern the new challenges imposed by the paradigm shift from traditional program to TF programs, with an emphasis on fault detection and localization. When answering these questions about challenges, we are also concerned about the solutions currently used by TF users. Understanding these solutions helps the development of new fault detection and localization techniques.

4 DATA COLLECTION

We collected TensorFlow bugs from two sources: StackOverflow pages and GitHub commits. StackOverflow pages contain bugs that might be difficult to debug: at least the TF user could not resolve the bug quickly and has to ask a question for assistance. On the other hand, GitHub commits contain bugs that might be difficult to detect: at least the TF user did not discover it at the first place and committed into the project. Putting the two sources together, we have a dataset of interest: the bugs those cause problems to the TF users and those are worth studying.

To collect bugs from StackOverflow pages, we used a search term "*tensorflow answers:1 -how -install -build*" in StackOverflow's search engine. The parameter "*answers:1*" ensures that only questions with at least one answer were considered. And other parameters "*-how -install -build*" were used to filter out discussions about installment and building of TensorFlow which we do not concern about. Then we manually reviewed top 500 question returned by StackOverflow and found 87 questions related to TensorFlow application bugs. Please note that StackOverflow may contain both novices' and experts' posts, and we believe both are important and should be included in the study. The statistics of the QA pages can be found in Table 1.

To collect bugs from GitHub commits, we searched for projects with keyword "*tensorflow*" in GitHub's search engine. Among the search results, we selected 11 target projects that are well-maintained with the highest numbers of commits and stars for further examination. The statistics of these projects are shown in Table 2. We take into consideration commits between start date and end date to collect bugs in each project. Then we searched commit messages with keywords "*bug, fix, wrong, error, nan, inf, issue, fault, fail, crash*" in each project. In addition, we filtered out "*typo*" and merged pull requests to eliminate irrelevant and duplicate commits. We manually inspected the source code, commit messages, pull request messages, and issue messages to identify coding bugs. As a result, we found 82 commits which contain 88 bugs related to TensorFlow application bugs on GitHub. For each commit, we read the commit and pull request message to see if there were any associated issues, and included the discussion thread of the issue into consideration.

The subjects were collected between July 2017 and May 2018. We have calculated the time spending from posting the issues until its resolving on Github issues and StackOverflow QA pages. In Github issues, the mean is 27,845 minutes and the median is 5,122 minutes. In StackOverflow QA pages, the mean is 33,312 minutes and the median is 177 minutes. When manual inspections are involved, two authors performed the inspection separately and discussed inconsistent issues until agreement. During the process, one StackOverflow bug and eight GitHub bugs identified by one author were removed from the discussion.

Putting together, we got a dataset[3] of 175 bugs, including 87 collected from StackOverflow and 88 collected from GitHub. The scale of our dataset is similar to other existing studies that require manual inspection, e.g., Jin et al. conducted a study of performance bugs and inspected 109 performance bugs [19], and Nasehi et al. conducted a study on what makes a good code example and analyzed 163 StackOverflow QA pages [26].

5 RQ1: SYMPTOMS AND ROOT CAUSES

5.1 Information Sources for Analysis

To answer the first research question, we analyzed each bug in our dataset to identify its root causes and symptoms. For GitHub bugs, the root causes can be identified by the changes made in the commits. We identified the symptoms of bugs by reading the commit message, pull request messages and the associated issues. For StackOverflow bugs, we learnt the root causes of bugs by reading the answers that provide a solution. We identified these bugs' symptoms from the question description. Besides, we also tried to reproduce the bugs to further understand their symptoms. We were able to reproduce 75 out of 88 Github bugs and 76 out of 87 StackOverflow bugs. The rest of the bugs were not reproducible because of dead links, missing datasets, or the requirement of specific hardware. We summarized the common root causes and symptoms of collected bugs into major categories and classified each bug accordingly. Two authors performed classification separately, no disagreement was found on StackOverflow bugs and five Github bugs were classified differently.

5.2 Results

The statistics of the symptoms (rows) and root causes (columns) that we found from our analysis are given in Table 3. We identified

[3] Our dataset is available at https://github.com/ForeverZyh/TensorFlow-Program-Bugs.

Table 1: Statistics of QA pages from StackOverflow

Bug Count	$answers_{min}$	$answers_{max}$	$answers_{mean}$	$answers_{median}$
87	1	7	1.53	1

Table 2: Statistics of Projects from Github

Projects	Start Date: End Date[1]	LOC (Python)	Commits	Issues	Id'd Bugs	Assocd Issues
tensorflow/models	2016-02-05: 2018-01-27	112553	1678	1974	40	30
davidsandberg/facenet	2017-01-02: 2018-05-12	6699	566	673	5	1
google/seq2seq	2017-03-02: 2018-01-27	5955	880	228	7	0
chiphuyen/stanford-tensorflow-tutorials	2016-11-05: 2018-05-12	4799	106	74	1	0
mfigurnov/sact	2017-03-23: 2018-01-27	2783	60	5	1	0
blackecho/Deep-Learning-TensorFlow	2015-08-17: 2018-01-27	2630	245	54	5	1
aymericdamien/TensorFlow-Examples	2015-11-11: 2018-01-27	1620	197	133	1	1
Conchylicultor/DeepQA	2016-07-07: 2018-01-27	1372	181	139	1	1
dennybritz/reinforcement-learning	2016-08-24: 2018-01-27	1181	205	89	5	1
carpedm20/DCGAN-tensorflow	2015-12-11: 2018-01-27	901	265	192	17	17
bamos/dcgan-completion.tensorflow	2016-08-09: 2018-01-27	646	68	44	5	0
Total		141139	4451	3605	88	52

[1] LOC, commits, and issues are counted at End Date.

Table 3: Bug Causes and Symptoms

	IPS[1]		UT		CCM		APIC		APIM		SI		Others		Total	
Error	4[2]	0	12	9	3	0	8	36	22	5	1	0	2	10	52	60
Low Effectiveness	13	10	3	0	9	1	0	0	1	1	0	0	1	1	27	13
Low Efficiency	0	0	0	0	1	1	0	0	4	0	1	1	1	0	7	2
Unknown	0	11	0	0	1	1	0	0	0	0	0	0	0	1	1	13
Total	17	21	15	9	14	3	8	36	27	6	2	1	4	12	87	88

[1] The abbreviation of each bug type can be seen in RQ1. For instance, **IPS** stands for **Incorrect Model Parameter or Structure**.

[2] The first and second columns of each bug type denote the number of issues from StackOverflow and Github, respectively.

three common types of symptoms exhibited by 161 (92%) of our collected bugs. We grouped the remaining 14 (8%) ones that do not have observable symptoms under "Unknown".

Symptom 1: Error. A TensorFlow error is analogous to exceptions or crashes in conventional applications, such as NaN errors. Errors can be raised by the TensorFlow framework at the construction or the execution phase.

Symptom 2: Low Effectiveness. The program exhibits extraordinarily poor accuracy, loss, or other unexpected outputs during the execution phase.

Symptom 3: Low Efficiency. The program executes slowly or even infinitely during the construction or the execution phase.

Symptom 4: Unknown. There is no indication in discussions about the consequences of a bug. And we were not able to reproduce their failures. In particular, a number of bugs are detected by code review, and their symptoms remain unknown.

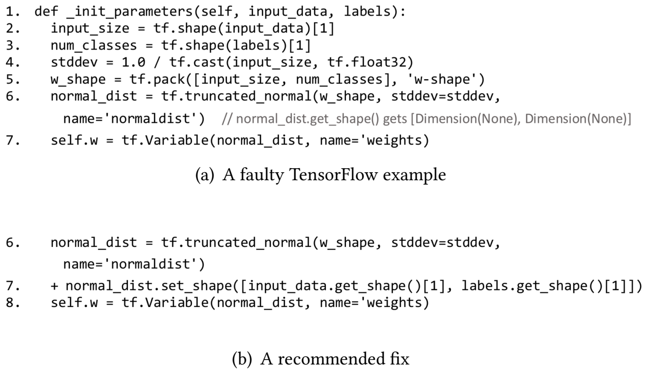
Among the bug-inducing root causes, 159 (90.9%) of them are related to TensorFlow and can be categorized into six major causes. The remaining 16 (9.1%) ones are not able to be classified and categorized as "Others".

```
13. - cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
13. + cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv+1e-9))
```

Figure 2: A recommended fix

As shown in Table 3, the most common symptom is Error in our dataset and the most common root cause is APIC. The most common root causes for every symptoms are APIC, IPS, APIM, and IPS respectively. Besides, the only bug symptom of APIC is Error, while bugs of CCM and Others cover all the symptoms.

Cause 1: Incorrect Model Parameter or Structure (IPS). Bugs related to modeling mistakes arose from either an inappropriate model parameter like learning rate or an incorrect model structure like missing nodes or layers. This kind of modeling bugs is



```
1.  def _init_parameters(self, input_data, labels):
2.    input_size = tf.shape(input_data)[1]
3.    num_classes = tf.shape(labels)[1]
4.    stddev = 1.0 / tf.cast(input_size, tf.float32)
5.    w_shape = tf.pack([input_size, num_classes], 'w-shape')
6.    normal_dist = tf.truncated_normal(w_shape, stddev=stddev,
        name='normaldist')   // normal_dist.get_shape() gets [Dimension(None), Dimension(None)]
7.    self.w = tf.Variable(normal_dist, name='weights')
```

(a) A faulty TensorFlow example

```
6.    normal_dist = tf.truncated_normal(w_shape, stddev=stddev,
        name='normaldist')
7.    + normal_dist.set_shape([input_data.get_shape()[1], labels.get_shape()[1]])
8.    self.w = tf.Variable(normal_dist, name='weights')
```

(b) A recommended fix

Figure 3: A faulty TensorFlow example extracted from StackOverflow #34079787 and a recommended fix

a distinctive type in TF programs, leading to anomalous behaviors at execution phase. The major symptom of this cause is Low Effectiveness such as low accuracy and a huge loss.

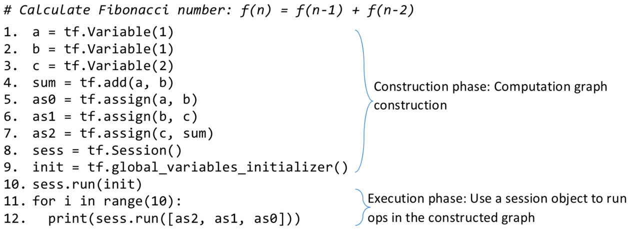
We found these three observations hold for most of the bugs in this root cause.

(1) It usually requires many training cycles to catch the failure, and the number of cycles required depends on hyperparameter settings. Although the faulty statement is reached in most program executions many times, failures occurred infrequently.
(2) In our dataset, failures usually occur at the training stage. Testing a DL model after it has been trained is unlikely to catch these failures.
(3) When its symptom is Error, the error message is usually confusing. The stacktrace reporting the crash in node does not pinpoint the faulty code. StackOverflow discussions suggest that fault determination for TF programs is non-trivial even for a small program.

Figure 1 shows the example extracted from a program written as a programming tutorial by a TensorFlow expert on Github. The program ran smoothly with the original dataset, but a TF user found the runtime error when running it using another dataset and initiated the discussion on StackOverflow. For some datasets, the program crashed with a *"ReluGrad input is not finite"* error after around 8,000 training iterations. The crash occurred much sooner when the value of a hyper-parameter e^{-4} was replaced by e^{-3} (Line 14). The error message reporting the crash in node *h_fc3* (Line 10) does not pinpoint the faulty code in the expression (Line 13). When *y_* equals to 0 and the value of *y_conv* approaches 0, the expression is evaluated as $0 \log 0$.

Figure 2 shows the fix recommended in StackOverflow. The fix involves a structural change in the model's computation graph by inserting a node adding a constant value e^{-9} to the node of *y_conv*.

Cause 2: Unaligned Tensor (UT). A bug spotted in computation graph construction phase when the shape of the input tensor does not match what it is expected is called an unaligned tensor bug. The major symptom of this cause is Error, since TensorFlow has assertions in API to check the shape of input tensors. Compared with conventional bugs such as unmatched array size, some bugs involved with TensorFlow tensors are quite unique, since TensorFlow tensors are allowed to have a dynamic shape which can vary from one iteration to another.



```
# Calculate Fibonacci number: f(n) = f(n-1) + f(n-2)
a = tf.Variable(1)
b = tf.Variable(1)
c = tf.Variable(2)
sum = tf.add(a, b)
as0 = tf.assign(a, b)
as1 = tf.assign(b, c)
as2 = tf.assign(c, sum)
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
for i in range(10):
  print(sess.run([as2, as1, as0]))
```

Construction phase: Computation graph construction
Execution phase: Use a session object to run ops in the constructed graph

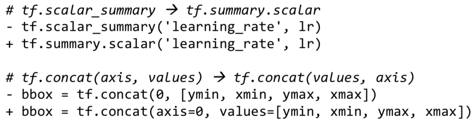
Figure 4: A faulty TensorFlow example extracted from StackOverflow #44676248

We present an example shown in Figure 3 extracted from a question from StackOverflow (#34079787): The program raised a *"ValueError: initial_value must have a shape specified"* error at Line 7. Tensor *normal_dist* has a partially-defined static shape, which means the real shape of *normal_dist* can be known after *w_shape* executed in the execution phase. In the construction phase, TensorFlow can only infer *normal_dist*'s shape is *[Dimension(None), Dimension(None)]*, which means a 2-dimensional matrix with an unknown number of rows and columns. The fix is shown in Figure 3(b) provides TensorFlow with a static shape of *normal_dist*. In this way, the variable *self.w* can be constructed correctly.

Cause 3: Confusion with TensorFlow Computation Model (CCM). Bugs arise when TF users are not familiar with the underlying computation model assumed by TensorFlow. A typical case is that TF users incorrectly constructed TensorFlow computation graphs using control-flow instead of data-flow semantics. Another typical case is the confusion between the graph construction and evaluation phases. The major symptom of this kind of bug is the poor results in accuracy and loss, which is classified under the category of "Low Effectiveness". When TF users make confusion on the TensorFlow computational semantics, the program does not encode a valid DL model. As such, the training process based on an invalid DL model is ineffective even though it does not necessarily result in a TensorFlow error. This explains the poor performance of the trained model in terms of accuracy and loss.

Figure 4 gives an example of this bug type. The user intended to calculate a Fibonacci sequence using *tf.assign*. The program constructed nodes *as*0, *as*1, and *as*2. Since TensorFlow imposes no sequential order on their computation, they can be computed in an arbitrary order. However, the questioner mistook that they follow the conventional control-flow semantics and would be computed sequentially.

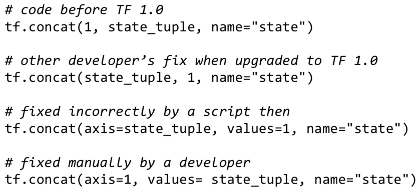
This type of bugs is more commonly found in StackOverflow discussions than Github projects. Since the deviated computation can mostly be observed when these bugs are triggered, TF users have usually solved them before committing the code. Indeed, they can be common mistakes made by TF users and discussed at StackOverflow seeking advice.



```
# tf.scalar_summary → tf.summary.scalar
- tf.scalar_summary('learning_rate', lr)
+ tf.summary.scalar('learning_rate', lr)

# tf.concat(axis, values) → tf.concat(values, axis)
- bbox = tf.concat(0, [ymin, xmin, ymax, xmax])
+ bbox = tf.concat(axis=0, values=[ymin, xmin, ymax, xmax])
```

Figure 5: Two fixes from Github when upgraded to TF 1.0



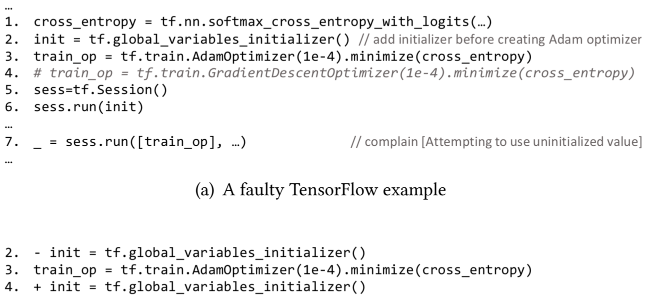
```
# code before TF 1.0
tf.concat(1, state_tuple, name="state")

# other developer's fix when upgraded to TF 1.0
tf.concat(state_tuple, 1, name="state")

# fixed incorrectly by a script then
tf.concat(axis=state_tuple, values=1, name="state")

# fixed manually by a developer
tf.concat(axis=1, values= state_tuple, name="state")
```

Figure 6: Three fixes on one changed API because of the incorrect script



```
…
1.  cross_entropy = tf.nn.softmax_cross_entropy_with_logits(…)
2.  init = tf.global_variables_initializer() // add initializer before creating Adam optimizer
3.  train_op = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
4.  # train_op = tf.train.GradientDescentOptimizer(1e-4).minimize(cross_entropy)
5.  sess=tf.Session()
6.  sess.run(init)
…
7.  _ = sess.run([train_op], …)              // complain [Attempting to use uninitialized value]
…
```

(a) A faulty TensorFlow example

```
2.  - init = tf.global_variables_initializer()
3.  train_op = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
4.  + init = tf.global_variables_initializer()
```

(b) A recommended fix

Figure 7: A faulty TensorFlow example extracted from StackOverflow #33788989 and a recommended fix

Cause 4: TensorFlow API Change (APIC). Anomalies can be exhibited by a TF program upon a new release of TensorFlow libraries. In our dataset, these anomalies always appear in the form of TensorFlow Errors analogous to exceptions in conventional applications. The kind of bugs arises when the new API version is not backward compatible with its previous version. As compared with conventional applications, this problem is more serious, accounting for 36 (40.9%) issues in Github projects.

Studying the code commits of the fixing version, we found that the bugs can be usually fixed by replacing the name of changed APIs and/or changing the order of arguments. An example is shown in Figure 5. Nevertheless, the fix may spread across many files, and thus it is important to ensure the changes are applied consistently. Figure 6 shows an issue collected from Github when the fixes to a changed TensorFlow API were not consistently applied.

Cause 5: TensorFlow API Misuse (APIM). Like conventional application framework, TensorFlow supports a rich set of APIs. Bugs were introduced by TF users who did not fully understand the assumptions made by the APIs. When these APIs are used without fulfilling the assumptions, the APIs cannot be successfully executed, leading to a TensorFlow error, which is the major symptom of this type of bugs. To support the learning-based DL algorithms, the assumptions made by TensorFlow APIs can be uncommon to those occurring in traditional API libraries.

Let us illustrate it using a code snippet extracted from StackOverflow (shown in Figure 7): the program could work smoothly using GD optimizer (commented out at Line 4). However, if the TF user replaced the GD optimizer with Adam optimizer (Line 3), the program would crash with an error *"Attempting to use uninitialized value Variable_21/Adam"* (Line 7). Notice that an initializer is added (Line 2), which will add initializing operators to variables in the computation graph. The program worked smoothly previously since the GD optimizer will not add additional variables to the computation graph. As for Adam-like optimizers (such as Adam, Adagrad, Adadelta optimizer), it will add additional variables called "slots", which will not be initialized (Line 6) and cause the crash (Line 7). A recommended fix is shown in Figure 7(b): moving the initializer to the line right after Adam optimizer.

Cause 6: Structure Inefficiency (SI). A major difference between SI and IPS is that SI leads to performance inefficiency while the IPS leads to functional incorrectness.

The small number of performance inefficiency issues suggests either performance issues rarely occur or these issues are difficult to detect.

Cause 7: Others. Other bugs that cannot be classified are included in this type. These bugs are usually programming mistakes unrelated to TensorFlow, such as Python programming errors including undefined variables, or data preprocessing. As such, they are not commonly discussed at StackOverflow.

We observed that the last category of bug-inducing causes, some of which are unrelated to TensorFlow, are only accountable for 12 (13.6%) of real issues we found in Github projects. This suggests that TF-related issues are the main reason for bugs in TF applications, calling for new testing and debugging techniques to specifically address TF-related bugs.

6 RQ2: CHALLENGES ON FAULT DETECTION

6.1 Setup

To understand the challenges in bug detection, we first classified the bugs by answering two questions: (1) whether the bugs will be always triggered by any input or not; (2) whether the bugs will always lead to a crash or not. The first question concerns how certain it is to trigger the bug, and the second question concerns how certain it is to capture the fault. If the answers to both questions are "yes", the bug can be certainly detected. Then, we manually investigated the bugs where the answer to at least one question is "no", and tried to identify new challenges that are different from detecting bugs in traditional programs. For each challenge identified, we further read the issues and QA pages to find out what strategies have already been used by TF users to deal with these challenges. Finally, we analyzed the distribution of the challenges among the six common bug-inducing causes.

6.2 Results

Table 4 shows the distribution of the answers to the two questions. As we can see from the table, 82 (46.9%) bugs always lead to program crash and are certain to be detected.

Table 4: Distribution of the Two Questions

Bug Count	Crash	Not Crash
Always Trigger	82	31
Not Always Trigger	30	32

Table 5: Statistics of Challenges

Bug Count	Challenge 1	Challenge 2	Challenge 3
StackOverflow	11	1	24
Github	11	3	18

We have identified three new challenges in the detection of the rest 53.1% of the bugs. The numbers of bugs exhibiting these challenges are shown in Table 5. We describe the three challenges one by one below.

Challenge 1: Probabilistic Correctness.

In traditional programs, their input/output relation is usually well defined. The relation accounts for the program correctness. Given an input, if the program produces a wrong output, the program is considered buggy. However, in TF programs, a correctness with observable effects cannot be deterministically defined. Given a trained model and an input, if the program produces an incorrect classification, it does not necessarily mean the program contains a bug, as TF programs cannot guarantee 100% correct classifications. Instead, the correctness is often interpreted as a probability: given an input, the TF program is expected to produce a correct output with a probability. The traditional testing framework does not work well in handling probabilistic correctness. In most testing frameworks, a test is defined as a test input to be passed to the program and a test oracle to determine the correctness of the output, and an unexpected output of a test indicates a bug. However, when probabilistic correctness is involved, we cannot determine the existence of a bug by observing an unexpected output of a single test.

By studying the issues and QA pages, we found that the TF users mainly relied on statistics to address these issues. Though it is difficult to determine the probability from a single output, it is usually a good approximation to perform statistics on many pairs of input and output. In particular, we found the TF users mainly relied on two statistical values: the accuracy and the loss of the program on a training set or on a testing set. Two strategies for determining correctness are used based on accuracy and loss.

Strategy 1: Comparing overall accuracy and loss with fixed thresholds

When the trained model does not achieve the expected accuracy or loss on the training set or on the testing set, the model is considered to have a bug.

Strategy 2: Comparing the relative changes of accuracy and loss between iterations

While the previous strategy is useful in determining the correctness of a model, the test execution may be expensive. The complete training process of a TensorFlow model usually takes days to perform, and only after the training can we know the accuracy and loss of the model. On the other hand, the training phase consists of many iterations, and TensorFlow can report the intermediate accuracy and loss after each iteration. As a result, we observe that some TF users use the changes in these intermediate values between iterations to determine correctness. In general, the accuracy is expected to show an increasing trend across iterations and the loss is expected to a decreasing trend. If no clear trend of increasing or decreasing is observed among several iterations, the TF users consider the model as buggy.

These challenges and strategies call for new testing techniques and framework. First, most testing frameworks do not support statistical correctness, and mechanisms for determining statistical correctness should be developed. Second, traditional test generation techniques are designed for absolute correctness, and how to efficiently trigger bugs characterized by statistical correctness remains unknown. Third, statistical correctness is only an approximation of probabilistic correctness, but how we can measure the confidence remains unknown. New theories can be developed to measure confidence in testing TF programs.

Challenge 2: Coincidental Correctness.

Coincidental correctness indicates the situation where a test execution triggers a bug, but by coincidence, no failure is detected. We also observe coincidental correctness in our dataset. Though bugs are triggered during the training process, the trained model still achieves desirable accuracy and loss on the testing set. The bugs were finally discovered by code review.

Since coincidental correctness already exists in traditional programs, it is not a new challenge in bug detection. However, we observed that coincidental correctness can be a new challenge in terms of scale. A TF program's computation is driven by tensors, which are usually modeled by multi-dimensional arrays with large size. After iterations of computation, the value of an individual element in an array makes a small contribution to the final classification results such as whether there is an obstacle before a car. In addition, most computation adopts a non-linear activation function, whose output is insensitive to certain input ranges. As such, a computation mistake is more likely to have unobservable effects on the final results. In other words, TF programs tend to be more tolerant to computational mistakes. So, coincidental correctness occurs on a larger scale for TF programs as compared with their traditional counterparts.

However, this does not necessarily mean that coincidental correctness is not an important issue. The effects of computation mistakes falling in the transition range of an activation function can be greatly amplified. Such amplified effects can induce incorrect classification, which can cause serious consequences in mission-critical applications. Such effects are analogous to adversarial attacks [12, 35] in DL models. Since the transition range for some activation functions such as sigmoid is narrow, finding training data that are resistant to coincidental correctness is challenging. In our dataset, we observe cases that the bugs were identified by code review, the TF users decided to fix them though they have not found incorrect results[4].

Challenge 3: Stochastic Execution.

Given the stochastic nature of the training phase, it is possible that two executions exhibit different behavior, which makes it difficult to reproduce the bugs. An example is the bug presented in

[4] For example, see in https://github.com/dennybritz/reinforcement-learning/pull/39.

Figure 1. During training, TF users will evaluate their model in many iterations to optimize their loss function. The TF user reported on StackOverflow that the program crashed with an error after around 8,000 iterations for some datasets. However, which iteration will raise an error is non-deterministic because of the stochastic nature such as gradient descending algorithm. In one run, the error may occur at the 7900th iteration. And in another run, the error may occur at the 8100th iteration. If the TF user sets the model to iterate 8000 times, he may detect the bug in the first run, but could not detect it in the second run.

Strictly speaking, non-determinism is not new: many traditional programs also exhibit non-determinism. However, on TF applications the problem become much more serious because almost any execution is affected by non-determinism. More studies are needed to deal with non-determinisms in TF applications.

7 RQ3: CHALLENGES ON FAULT LOCALIZATION

7.1 Setup

Based on our classification, there are three main types of symptoms. Unlike the other two types, the symptom of "Error" provides additional information for debugging, including a line number to indicate where the error occurs and an error message to describe the reason of the fault. To understand the challenges on fault localization, we use different methods to analyze bugs in the "Error" type and bugs in the other two types.

For bugs in the "Error" type, we used trace dependency distance to measure the difficulty of fault localization quantitatively. We define an execution trace as a sequence of statements executed during execution. A trace dependency graph is a graph where the nodes are instances of statement executions in the execution trace and the edges are dynamic data or control dependency between the statement execution instances. *Trace dependency distance* is the smallest number of nodes on the trace dependency graph from the reported error location to the root cause of the bug, and was suggested by prior studies to measure the difficulty of fault localization [30].

Furthermore, to complement the quantitative analysis, we also read the error messages qualitatively to judge the difficulty of localizing the bugs. Please note that not all bugs we collected have error messages in our dataset, and we focus on only the bugs that have error messages. We analyzed 79 bugs which have error messages. If the execution trace involves multiple cycles, we will refer to additional information in discussions about which cycle raised the error.

For bugs in the other two types, we analyzed these bugs qualitatively to understand how difficult the localization is. Finally, for those bugs that we considered difficult to localize, we tried to find out how the TF users localized these bugs and summarized them into strategies.

7.2 Results

Figure 8 shows dependency distance distribution of bugs collected from StackOverflow and Github. Except for 2 bugs whose distance is more than 8000 (our motivating example in Figure 1 shows one of them), it can be seen that the rest of distances are less than 8. And the faulty executions of these two bugs both involve the execution phase. For the rest of bugs, the mean is 0.99 and the median is 0.

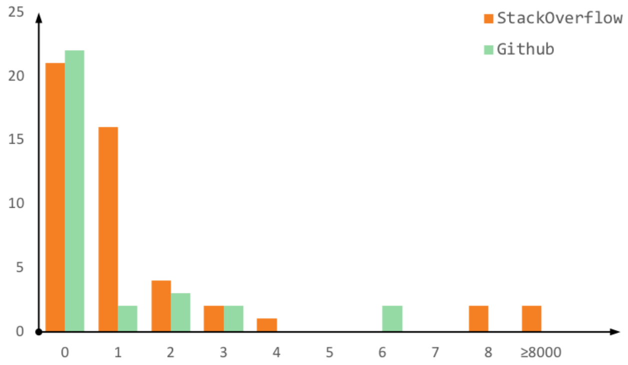


Figure 8: Dependency Distance Distribution

Combining this figure with our qualitative analysis of the error messages and the other two types of bugs, we have the following observation.

(1) When an error is raised during the construction phase, the bug can usually be localized with certainty. In such cases, trace dependency distances are short as compared with those within the execution phase that involves massive iterative and probabilistic computation. Besides, the information embedded by error messages help localize the bugs. One can examine the program from the faulty statement provided by error messages backward via trace dependency.

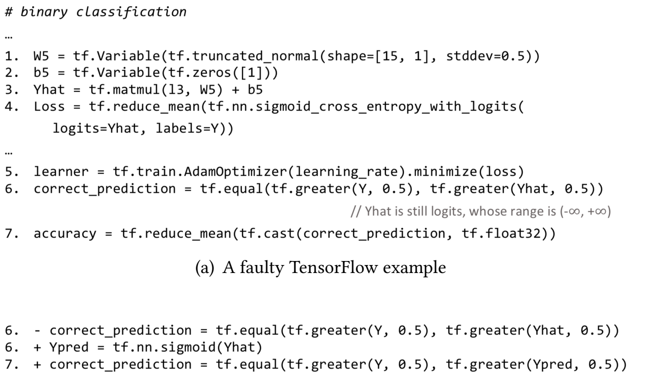
(2) When the faulty execution involves the execution phase, the buggy behavior becomes stochastic, dramatically increasing the fault localization effort. Compared with traditional programs, we identified the following two major challenges in localizing these bugs.

Challenge 4: The densely inter-dependent neural network.

Elements in a traditional program are often loosely dependent on each other. If we dynamically slice from the point where errors or incorrect outputs occur, the slicing result often contains only a small portion of coding entities in the program. However, in a neural network, typically nodes in the current layer mostly depends on the nodes in the previous layer. Furthermore, during the training phase, the dependency becomes bidirectional because of backpropagation. As a result, slicing could provide little assistance in the sense that the slice usually contains all nodes in the neural network and does not help debug.

Challenge 5: The unknown behavior of neural networks.

A typical way of debugging traditional program is to examine the program state at a specific program point, by comparing the values of variables with their expected values. However, in neural networks, since the program behavior is sensitive to the hyperparameters assigned during the training process, it is difficult for programmers to predict the expected value at a certain program point. As a result, though we can access the intermediate states of a neural network during the training process, it is often difficult for TF users to judge its correctness based on these states.



```
# binary classification
…
1.  W5 = tf.Variable(tf.truncated_normal(shape=[15, 1], stddev=0.5))
2.  b5 = tf.Variable(tf.zeros([1]))
3.  Yhat = tf.matmul(l3, W5) + b5
4.  Loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
       logits=Yhat, labels=Y))
…
5.  learner = tf.train.AdamOptimizer(learning_rate).minimize(loss)
6.  correct_prediction = tf.equal(tf.greater(Y, 0.5), tf.greater(Yhat, 0.5))
                                          // Yhat is still logits, whose range is (-∞, +∞)
7.  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

(a) A faulty TensorFlow example

```
6.  - correct_prediction = tf.equal(tf.greater(Y, 0.5), tf.greater(Yhat, 0.5))
6.  + Ypred = tf.nn.sigmoid(Yhat)
7.  + correct_prediction = tf.equal(tf.greater(Y, 0.5), tf.greater(Ypred, 0.5))
```

(b) A recommended fix

Figure 9: A faulty TensorFlow example extracted from StackOverflow #42821125 and a recommended fix

Since many bugs are difficult to localize, it would be interesting to know how TF users deal with them. Our study identified three strategies that the TF users used to localize the bugs as follows.

Strategy 3: Replacing hyper-parameters in a network.

StackOverflow question #42821125 (Figure 9) describes an IPS bug. This task is a binary classification (0-1 classification), and the input contains about 69% zeros, so it is really strange that the accuracy of the model could not go beyond 69%. Thus, the questioner suspected *Yhat* converged to all zeros.

The questioner said that *I've tried a lot of things like different optimizers, loss functions, batch sizes, etc.. but no matter what I do it converges to 69% and never goes over. I'm guessing there's a more fundamental problem with what I'm doing but I can't seem to find it.*

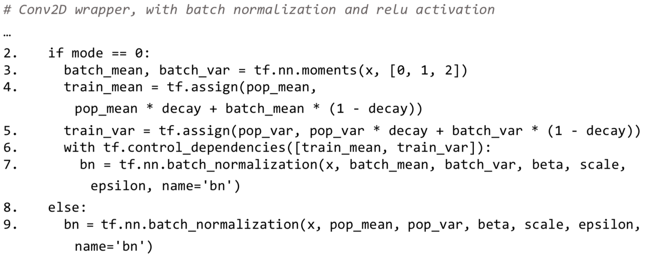
It can be inferred that the questioner replaced some parameters of the network and hoped to seek a better result. If TF users get an unexpected output from their networks, the very first thing they will do is to replace some parameters like different optimizers, loss functions, batch sizes, learning rates, etc..

The questioner had an expectation of how loss and accuracy would change corresponding to different replacements. Nevertheless, the output was not influenced by these replacements, leading to questioner's suspicion that there was a more fundamental problem in the model structure rather than these model parameters. He was supposed to use predicates (applying *sigmoid* on logits to convert its range from $(-\infty, +\infty)$ to $(0, 1)$) to calculate *correct_prediction* as shown in Figure 9(b).

Strategy 4: Examining the distribution of variable values.

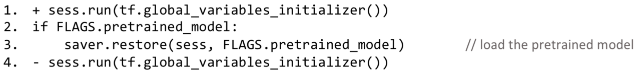
StackOverflow question #40166236 (Figure. 10) describes a CCM bug. The questioner intended to use different batch normalization parameters based on the tensor *mode*. It was later found out that the questioner should use *tf.cond* rather than *if* (at Line 2) to choose the desired branch. Using *if*, the branch will be chosen during graph building and will not change.

The questioner localized this fault by examining the distribution of the variables *pop_mean* and *pop_var* using TensorFlow visualizing tool and found the values of the two variables have never changed. Thus, though the questioner cannot predict the exact value of variables, some metamorphic relations between the values of the variables in different iterations could be specified.



```
# Conv2D wrapper, with batch normalization and relu activation
…
2.    if mode == 0:
3.      batch_mean, batch_var = tf.nn.moments(x, [0, 1, 2])
4.      train_mean = tf.assign(pop_mean,
          pop_mean * decay + batch_mean * (1 - decay))
5.      train_var = tf.assign(pop_var, pop_var * decay + batch_var * (1 - decay))
6.      with tf.control_dependencies([train_mean, train_var]):
7.        bn = tf.nn.batch_normalization(x, batch_mean, batch_var, beta, scale,
            epsilon, name='bn')
8.    else:
9.      bn = tf.nn.batch_normalization(x, pop_mean, pop_var, beta, scale, epsilon,
          name='bn')
```

Figure 10: A faulty TensorFlow example extracted from StackOverflow #40166236



```
1.  + sess.run(tf.global_variables_initializer())
2.  if FLAGS.pretrained_model:
3.      saver.restore(sess, FLAGS.pretrained_model)        // load the pretrained model
4.  - sess.run(tf.global_variables_initializer())
```

Figure 11: A fix commit on a faulty TensorFlow code snippet extracted from Github

Strategy 5: Switching the training dataset.

In a commit[5] found at Github (shown in Figure 11), the bug was caused by the incorrect order of initializer. The TF user wrote the initializer after the model loading the pre-trained model. It means that all the loaded data will be initialized to random values.

Since it is a video prediction model, it does not concern about accuracy and is only evaluated by its loss. Another (second) TF user posted an issue[6] reporting heavily blurred predicted images and poor loss performance.

The second TF user commented that *I used validation data for test prediction network. And since I got a similar validation loss curve, I assume the training process is correct.(I didn't make any changes in codes).*

It can be seen that the second TF user switched the dataset to find out where the bug is. The training loss curves of two datasets were similar to each other. Thus, the second TF user suspected that the bug was not triggered in the training process. Based on the patch (commit) for a later program version, the suspect was correct.

8 THREATS TO VALIDITY

First, our study involves manual inspections on bugs. These subjective steps can be biased due to our inference of the code's intention in the lack of documentation. In order to reduce this threat, two authors analyzed the bugs separately and discussed inconsistent issues until an agreement was reached. Second, our study investigated 175 bugs from StackOverflow and Github, and it is not clear how much our findings generalize beyond the dataset, especially considering the fact that TF is growing fast. However, it is not easy to expand this dataset. First, since TensorFlow is an emerging framework, there were not many well-maintained popular Github projects at the time we conducted this empirical study. Second, the manual efforts required to analyze the bugs were large. To collect

[5] https://github.com/tensorflow/models/commit/34af79db12577f2039c4f88bfae50734d8ddd2c6

[6] https://github.com/tensorflow/models/issues/670

and analyze the bugs, we spent approximately 400 person-hours, leading to an average 2.3 person-hours per bug.

9 DISCUSSION

Common Fixing Patterns. Our paper focuses on bug detection and localization, and leave bug repair for future work. Nevertheless, we also performed a small pilot analysis on the fixing patterns of the Tensorflow bugs from the patches in GitHub projects and recommended fixes in StackOverflow QA pages. In general, we found the fixing patterns are strongly correlated to the root causes of the bug. For example, APIC and APIM are both related to API calls, and the common fixing patterns are changing the parameter orders (Figures 5 and 6) and changing API calling sequences (Figure 7(b)). On the other hand, the common fixing pattern for IPS is to change the model structure (Figure 2). This finding suggests that analyzing the root causes could be useful for further developing automated repair approaches.

Bugs in Other Parts. As mentioned in the introduction, defects in a TF application may come from its training data, program code, execution environment or the TensorFlow framework. Our study focuses on bugs in TF programs but not other types because these types are different in nature and it is not easy to study in a unified way. Bugs in the training data are related to the problem of data quality [14, 33] and require methods such as data cleansing [15, 24, 29] and data augmentation [28, 38] to deal with. Bugs in the TensorFlow framework is a type of compiler bugs and often requires specific compile testing techniques [3, 4] to deal with. Bugs in the execution environment are often not controllable and require fault tolerance techniques [21] to deal with. These types of bugs are not common in StackOverflow pages or GitHub commits of TF programs, and require other data sources such as the history of training data sets or the commits for the Tensorflow framework.

10 RELATED WORK

Empirical Study: Thung et al. [37] surveyed three machine learning systems, Apache Mahout, Lucene, and OpenNLP. They analyzed a sample of their bugs and fixes and labeled bugs into various categories. They also studied bug severities, the time and effort needed to fix the bugs, and bug impacts. Different from them, our study focus on bugs of deep learning applications built on top of TensorFlow, which are based on hierarchical layers of neurons that are interconnected to form a neural network.

Seaman et al. [32] investigated 81 projects with NASA and constructed a new set of defect categories at a slightly higher level of abstraction than the historical ones. Thung et al. made use of defect categories provided by Seaman et al. in their empirical study on machine learning system. Different from their categories, we proposed categories on deep learning application bugs at a lower level of abstraction.

Some empirical studies focused on certain types of bugs. Jin et al. [19] and Zaman et al. [42, 43] conducted studies of performance bugs. Gunawi et al. [13] conducted a study of development and deployment issues of cloud systems. Xiao et al. [39] studied non-commutative reduce functions bugs in MapReduce programs. Chen et al. [6] studied dormant bugs from Apache foundation software systems. A number of studies [2, 8, 20, 22] focused on bugs from API changes. Our study focused on TF bugs, which are different from most the above bugs. The only exception is the bugs from API changes, which appear both in TF programs and traditional programs. Our observation on these bugs is consistent with existing studies and we did not observe new challenges in detecting and localizing these bugs.

Machine Learning Testing: Xie et al. [40] proposed a technique based on metamorphic testing to address the test oracle problem for the implementations of machine learning classification algorithms: k-nearest neighbors and Naive Bayes Classifier.

DeepXplore designed by Pei et al. [28] is a whitebox differential testing system that can find inputs that can trigger inconsistencies between multiple DNNs and identify erroneous behaviors. They introduced neuron coverage as a systematic metric for measuring how much of the internal logic of a DNNs have been tested.

Tian et al. [38] proposed DeepTest, a tool for automated testing of DNN-driven autonomous cars. DeepTest can use test images that generated by different realistic transformations to maximize the neuron coverage of a DNN. They leveraged domain-specific metamorphic relations to find erroneous behaviors of the DNN.

Srisakaokul et al. [34] proposed an approach of multiple implementation testing for supervised learning softwares: k-nearest neighbor and Naive Bayes.

Big data Debugging: Interlandi et al. [17] built Titian, a data provenance library that integrates directly with the Spark runtime and programming interface. Ma et al. [23] proposed LAMP, a data provenance computation technique for graph-based machine learning algorithms.

Probabilistic Testing: Barr et al. [1] studied test oracles in software testing including probabilistic test oracle. Gerhold et al. [11] designed an executable model-based testing framework for probabilistic systems with non-determinism.

Our work differs from these studies on testing and debugging approaches in that it is the first empirical study on coding mistakes in DL programs built on TensorFlow by collecting related discussions at StackOverflow and bugs that have been fixed in Github projects.

11 CONCLUSION AND IMPLICATION

We studied 175 TensorFlow application bugs collected from StackOverflow QA pages and Github projects. We examined the root causes and symptoms of these bugs according to QA pages, commit messages, pull request messages, and issue discussions. We also studied the strategies deployed by TF users for bug detection and localization.

Two groups of people can benefit from this study. For TF users, we summarized five strategies used by other TF users to detect and debug the bugs in TF programs. For software engineering researchers, we pointed out five new challenges which call for more research efforts. Our classification of causes and symptoms offers both TF users and software engineering researchers a better understanding of deep learning program bugs.

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[2] Gabriele Bavota, Mario Linares Vásquez, Carlos Eduardo Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Trans. Software Eng.* 41, 4 (2015), 384–407. https://doi.org/10.1109/TSE.2014.2367027

[3] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 700–711. https://doi.org/10.1109/ICSE.2017.70

[4] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 180–190. https://doi.org/10.1145/2884781.2884878

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[6] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. 2014. An empirical study of dormant bugs. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India.* 82–91. https://doi.org/10.1145/2597073.2597108

[7] Muriel Daran and Pascale Thévenod-Fosse. 1996. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996.* 158–171. https://doi.org/10.1145/229000.226313

[8] Danny Dig and Ralph E. Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance* 18, 2 (2006), 83–107. https://doi.org/10.1002/smr.328

[9] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[10] István Forgács and Antonia Bertolino. 2002. Preventing untestedness in data-flow based testing. *Softw. Test., Verif. Reliab.* 12, 1 (2002), 29–58. https://doi.org/10.1002/stvr.234

[11] Marcus Gerhold and Mariëlle Stoelinga. 2018. Model-based testing of probabilistic systems. *Formal Asp. Comput.* 30, 1 (2018), 77–106. https://doi.org/10.1007/s00165-017-0440-4

[12] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org/

[13] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patanaanake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014.* 7:1–7:14. https://doi.org/10.1145/2670979.2670986

[14] David J. Hand. 2007. Principles of Data Mining. *Drug Safety* 30, 7 (01 Jul 2007), 621–622. https://doi.org/10.2165/00002018-200730070-00010

[15] Mauricio A. Hernández and Salvatore J. Stolfo. 1998. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Min. Knowl. Discov.* 2, 1 (1998), 9–37. https://doi.org/10.1023/A:1009761603038

[16] Robert M. Hierons. 2006. Avoiding coincidental correctness in boundary value analysis. *ACM Trans. Softw. Eng. Methodol.* 15, 3 (2006), 227–241. https://doi.org/10.1145/1151695.1151696

[17] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *PVLDB* 9, 3 (2015), 216–227. http://www.vldb.org/pvldb/vol9/p216-interlandi.pdf

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[19] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012.* 77–88. https://doi.org/10.1145/2254064.2254075

[20] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. 2018. An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information & Software Technology* 93 (2018), 186–199. https://doi.org/10.1016/j.infsof.2017.09.007

[21] Peter Alan Lee and Thomas Anderson. 1990. *Fault Tolerance.* Springer Vienna, Vienna, 51–77. https://doi.org/10.1007/978-3-7091-8990-0_3

[22] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013.* 300–307. https://doi.org/10.1109/ICWS.2013.48

[23] Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. 2017. LAMP: data provenance for graph based machine learning algorithms through derivative computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 786–797. https://doi.org/10.1145/3106237.3106291

[24] Jonathan I. Maletic and Andrian Marcus. 2000. Data Cleansing: Beyond Integrity Analysis. In *Fifth Conference on Information Quality (IQ 2000).* 200–209.

[25] Brian Marick. 1991. The Weak Mutation Hypothesis. In *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991.* 190–199. https://doi.org/10.1145/120807.120825

[26] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012.* 25–34. https://doi.org/10.1109/ICSM.2012.6405249

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[28] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* 1–18. https://doi.org/10.1145/3132747.3132785

[29] Erhard Rahm and Hong Hai Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13. http://sites.computer.org/debull/A00DEC-CD.pdf

[30] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada.* 30–39. https://doi.org/10.1109/ASE.2003.1240292

[31] Debra J. Richardson and Margaret C. Thompson. 1993. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Trans. Software Eng.* 19, 6 (1993), 533–553. https://doi.org/10.1109/32.232020

[32] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany.* 149–157. https://doi.org/10.1145/1414004.1414030

[33] Victor S. Sheng, Foster Provost, and Panagiotis G. Ipeirotis. 2008. Get Another Label? Improving Data Quality and Data Mining Using Multiple, Noisy Labelers. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08).* ACM, New York, NY, USA, 614–622. https://doi.org/10.1145/1401890.1401965

[34] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software.. In *Proceedings of the AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS 2018), co-located with AAAI 2018, New Orleans, LA, Feburary 2018.*

[35] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2013). arXiv:1312.6199 http://arxiv.org/abs/1312.6199

[36] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[37] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012.* 271–280. https://doi.org/10.1109/ISSRE.2012.22

[38] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2017. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. *CoRR* abs/1708.08559 (2017). arXiv:1708.08559 http://arxiv.org/abs/1708.08559

[39] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. 2014. Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014.* 44–53. https://doi.org/10.1145/2591062.2591177

[40] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920

[41] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36. https://doi.org/10.1145/1050849.1050865

[42] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2011. Security versus performance bugs: a case study on Firefox. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings.* 93–102. https:

//doi.org/10.1145/1985441.1985457

[43] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A qualitative study on performance bugs. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. 199–208. https://doi.org/10.1109/MSR.2012.6224281