Investigating Failure Patterns in Machine Learning-based Object Detection Tasks in Software Development Courses

Ziyuan Wang[1], Jinwu Guo[1], Dexin Bu[1], Chongchong Shi[2]*

[1] School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, China
[2] Changzhou Xingyu Automotive Lighting Systems Co., Ltd, China
wangziyuan@njupt.edu.cn, guojinwu.0801@foxmail.com, bdexin@qq.com, qwerscc@163.com

Abstract

Object detection, one of the popular tasks in computer vision, is to find all objects of interest in an image and determine their category and location. When people use deep learning frameworks to implement object detection networks, defects are often caused by human-introduced faults. These defects may cause different types of failures. Exploring frequent failure patterns in object detection programs can help developers detect and fix defects more effectively and efficiently. Therefore, we conducted an empirical study on failure patterns in deep learning-based object detection programs submitted in university software development courses. By exploring 101 submissions of a Yolov4 object detection task completed by 104 students, we found the most frequent 13 failure patterns in these submissions and six types of root causes of these failures. To help students and entry-level software engineers avoid possible faults in object detection programs, 13 concrete suggestions that belong to six classes are given in this paper. These results can reveal some basic laws of failures and mistakes in the development of deep learning-based object detection programs and provide guidances to assist students and entry-level developers in improving their skills in developing object detection programs.

Keywords: Object detection, Empirical study, Education, Failure pattern, Root cause

1 Introduction

The deep learning (DL) technique has been widely utilized in many fields [1], and object detection is one of the most common applications of the DL technique [2]. Implementing object detection program by calling APIs provided by the DL framework is often regarded as coursework in university software development courses. Because the DL framework provides rich functional APIs, software engineers can quickly use these APIs to complete their customized object detection tasks. However, it isn't easy to correctly develop an object detection program by using these APIs provided by the DL framework. People often make mistakes, which may trigger failures of their DL-based programs. Collecting and analyzing frequent mistakes, classifying them into different failure patterns, investigating root causes of these failures, and giving suggestions on how to avoid them are of great significance for detecting and repairing mistakes in object detection programs and guaranteeing the correct running of these programs. It is also helpful for students and entry-level software engineers to understand the mistakes that are easy to make when implementing object detection networks, and to avoid possible failures induced by these mistakes.

There have been many empirical studies on the defects and failures in different open-source softwares. Some people conducted empirical studies on bugs in traditional softwares, including CXF [3], Camel [3], Blockchain [4], and others [5-7]; Some people conducted empirical studies on bugs in DL-related softwares, including Caffe [8], TensorFlow [8-10], Keras [8], Theano [8], Torch [8], Scikit-learn [11], Paddle [11], and others [12-14]. However, empirical studies on bugs in object detection programs are still lacking. Therefore, we conduct the first empirical study on bugs in DL-based object detection tasks in software development courses.

We collect 101 programs submitted by 104 students in university software development courses. Then we manually analyze mistakes in these programs, classify them into the most frequent 13 failure patterns, and find six types of root causes of these failures. It is found that failures that students and entry-level software engineers are prone to trigger when implementing object detection programs include Path Not Found, Type Mismatch, and GPU Out of Memory. And in order to avoid these types of failures, some suggestions for students and entry-level software engineers in the field of object detection are given. They should pay attention to the following issues when developing DL-based object detection programs, including the conversion of file paths between different systems, the type of Python parameters, and the setting of network training parameters.

The key contributions of this paper are as follows:

(1) Collect mistakes made by university students in developing the object detection program Yolov4, and conduct an empirical study on them;

(2) Classify failures caused by these mistakes into the most frequent 13 failure patterns within four categories, and illustrate cases of these failure patterns;

(3) Analyze the root cause of these failures and give suggestions for the students and entry-level developers to avoid potential risks due to these failures.

The rest of this paper is organized as follows. Section

*Corresponding Author: Ziyuan Wang; E-mail: wangziyuan@njupt.edu.cn
DOI: 10.53106/160792642023072404017

2 introduces the research questions, research subjects, and research design of the empirical study in detail. Section 3 introduces the types of failure; Section 4 analyzes the root causes of failures. Section 5 gives some suggestions on how to avoid failures. Section 6 describes the potential threats to validity. Section 7 briefly reviews related works. Section 8 gives a conclusion.

2 Methodology

This section introduces the methodology of empirical study, including research questions, research subjects, and research design.

2.1 Research Questions

We focus on the following three research questions.

RQ1 (Failure patterns): Which types of failures occur most frequently in object detection networks?

By answering this question, we can find failures occur most frequently in object detection networks developed by students or entry-level developers. This finding could be beneficial for them to prepare high-quality test cases that can trigger these failures effectively and efficiently before the implementation of object detection networks.

RQ2 (Root causes): What are the root causes of those frequent failures in object detection networks?

By answering this question, we can understand why those failures occur most frequently in object detection networks developed by students or entry-level developers. This finding could provide guidance for maintainers of object detection networks to debug programs, including both locating and fixing bugs.

RQ3 (How to avoid): How to avoid those frequent failures in object detection networks?

By answering this question, we can give reasonable suggestions on how to code DL-based object detection programs with fewer defects and failures. This finding could help students or entry-level developers improve their skills in implementing object detection networks.

2.2 Research Subject

In the software development course, we arranged an object detection task that requires students to use Python to train the object detection network based on Yolov4 [15] and make a training set according to the COCO2017 [16] data set. The prepared data set is used for network training. After the network training is completed, the test set is used for testing, and the mean average precision (mAP) of network detection is observed. In this task, the specific work that students need to complete includes: learning the API for the DL framework, e.g. the API for convolution (`conv1d()`) and pooling (`avg_ pool1d()`), loading training data and implementing network training by Python. Then, the prepared test set is inputted into the trained model for testing. In addition, students also need to improve the mAP of the network as much as possible to achieve higher scores.

The DL framework we chose is PyTorch, which is much easier to deploy. And students can complete their tasks more quickly.

We collect 101 programs of 104 students in three classes using Yolov4 to complete object detection tasks in software development courses and take the failure triggered in the program as the research subject. We collect all relevant information for each failure, including input data, source code, execution script, log, and runtime data for analysis. All failures in our study are caused by improper human operation, excluding caused by objective factors such as hardware or system problems.

2.3 Research Design

We manually analyze and classify failures. First, search the log for the keywords that cause the failure, such as `assert`, `wrong`, `error`, `exception`, `fault`, `failure`, `crash`, and `unsuccessful`, to locate the information that triggers the failure. Then, manually analyze all relevant defect logs to determine the failure and classify it. Finally, we summarize failure patterns into four dimensions: Object Detection, Execution Environment, Code Fault, and Data. For the controversial pattern, we invite the corresponding students to participate in the discussion until a consensus is reached. Each program's source code and logs are manually analyzed through discussion with students. For data-related failures, additional checks are performed on the input data, summarizing the root causes of failures.

3 Failure Classification

There are mainly 13 failures patterns collected from all the 101 submitted programs. These failures patterns could be classified into four categories that shown in Table 1.

3.1 Object Detection

Totally 18% of failures are related to object detection networks. In this dimension, GPU out of memory is the most frequent failure, accounting for 12% of all failures. When the model calculates that the memory used exceeds the available physical memory of the GPU, GPU out of memory will occur. CPU out of memory occurs when the program runs beyond the main memory. API misuse, mainly in violation of the provisions of the API interface document, such as `optimizer.zero_Grad()` and `model.zero_Grad()` is confused. The former is to clear the gradient of the parameters corresponding to the optimizer, and the latter is to clear the gradient of the entire network. Finally, `loss` becoming `Nan` indicates that the calculated loss value becomes uncertain or cannot be expressed. For example, in gradient explosion, loss after each iteration becomes larger with each iteration, and finally exceeds the range of floating-point representation, then becomes `Nan`.

3.2 Execution Environment

Execution environment related failures occur in the interaction with the platform rather than in the execution of code logic, accounting for 32.6% of the total failures. In this dimension, the path not found accounts for 26%. For example, the data set path not found triggers the failure that loads the training data for network training, and the output path not found triggers the failure that finds the written file.

Secondly, the dependencies of the program execution are not imported correctly, resulting in the failure to use the relevant libraries correctly. For example, "`no module named tools.Nnwrap`" will be reported when using `CONDA` install PyTorch [17] through the command line.

Table 1. Categories of failure patterns

Category	Pattern	Description	Ratio
Object detection	GPU out of memory	Insufficient GPU memory to continue the DL coputation	12%
	CPU out of memory	Insufficient main memory	2.9%
	API misuse	API usage violates framework assumptions	2.5%
	Loss NaN	Loss value is not a number	0.6%
	Subtotal		18%
Execution environment	Path not found	File or directory cannot be found	26%
	Module not fonud	Dependency not loaded	6.6%
	Subtotal		32.6%
Code fault	Type mismatch	Parameter type mismatch function	16%
	Attribute not found	Referencing a non-existent Python class field, function, etc.	15.1%
	Syntax defect	Violation of the grammatical rules	13.2%
	Illegal index	Accessing array elements with an out-of-range	1.2%
	Division by zero	Dividing a decimal value by zero	0.3%
	Subtotal		45.8%
Data	Corrupt data	Input erro data	2.8%
	Encoding mismatch	Data cannot be correctly encoded or decoded	1.2%
	Subtotal		3.6%
Total			100%

3.3 Code Fault

Code fault is the most frequent in all dimensions, accounting for 45.8% of all failures. Their types are similar to other Python programs, and 16% of failures are type mismatches. For example, when the constructor format is `__init__`, not `_init_`, "`Typeerror: this constructor takes no arguments`" will be reported when writing incorrectly. 15.1% are Attribute not found, most of which occur when starting Python scripts. E,g,, the nonexistent function "`attributeerror: type object 'add_dim' has no attribute 'dim_value' is called`". Since Python has strict space indentation, it is very easy to make syntax defects, such as "`indentation error: expected an indented block`". Incorrect index values, such as "`indexerror: list index out of range`". The divisor is zero, such as "`zerodivisionerror: division by zero`".

3.4 Data

Data accounts for 3.6% of all failures, divided into corrupt data and encoding mismatch. The corrupt data shows that data integrity is broken, such as training data label fault or training field loss. For encoding mismatch, for example, the interpreter uses UTF-8 to decode, but the file is in the encoding format of GBK.

4 Root Causes

To understand why these failures occured in the object detection programs, we analyze the root causes of these failures manually. According to the analysis results, we can classify the root causes of failures into six categories.

4.1 Network Training Parameters

Excluding hardware factors, the cause of GPU out of memory is the value of `batch_ size` setting is too large, and the `subdivisions` setting is too small. After modifying the value of parameters `batch_size` and `subdivisions`, the failure is fixed. Large batch sizes and small subdivisions will improve the training efficiency of the network; however, they significantly increase GPU memory consumption. Due to the lack of experience in deep neural network parameter optimization, GPU out of memory accounts for a high proportion. As shown in Figure 1, when the value of `batch_size` is 64, and the value of `subdivisions` is 8, Yolov4 cannot be trained normally, and GPU is out of memory. After adjusting the `batch_size` to 32 and the `subdivisions` to 16, it can train normally.

```
1   - batch = 64
2   + batch = 32
3   - subdivisions = 8
4   + subdivisions = 16
5   max_batch=50000
6   policy=steps
```

Figure 1. Adjust `batch_size` and `subdivisions`

When the `learning_rate` is set too high, the network training will not converge, and the loss value will become larger with each iteration and eventually exceed the range of the floating-point representation, resulting in `Nan`. As shown in Figure 2, when the `learning_rate` is 0.01, the

network has a gradient explosion, and `loss Nan` occurs. After reducing the `learning_rate` to 0.001, the network can train normally.

```
- learning_rate=0.01
+ learning_rate=0.001
burn_in=10000
policy=steps
steps=14000,16000
scales=.1,.1
```

Figure 2. Adjust `learning_rate`

4.2 Differences in Execution Environment

Due to the difference between the local environment and the network requirement environment, the script execution defect is caused, and the network cannot be started correctly. For the file path not found, due to the provisions of Linux and Windows on different symbols in the path, the corresponding path needs to be manually modified. Because the students rarely use Linux and do not understand this aspect, the data file cannot be loaded or written so that the path not found is more easily triggered. As shown in Figure 3, the format of the same path in different systems is inconsistent due to the different use of slashes in Linux and Windows systems.

```
import os


- data_file=open("C:\yolo_data\train_data.txt")
+ data_file=os.path.join(os.path.dirname(__file__),"train_data.txt")

def read_file(data_file):
    model.train(data_file)
```

Figure 3. Path not found

As shown in Figure 4, if the library on which the program depends is not imported correctly, the execution environment will also be different, and the network cannot be started correctly.

```
- import torch_xla.distributed.parallel_loader as xmp
+ import torch_xla.core.xla_model as xmp

for data, target in mp_decice_loader:
    optimizer.zero_grad()
    output = model(data)
    loss = loss_fn(output, target)

if __name__=='__main__':
    xmp.spawn(_mp_fn, args=())
```

Figure 4. Module defect

4.3 API Mismatch

First, the correct use of API in the interface document is not correctly understood because the current Chinese API documents are directly translated from English documents, and the interpretation of relevant interfaces may have translation errors or ambiguous semantics.

As shown in Figure 5, `optimizer.zero_ grad()` is to clear the parameter and gradient corresponding to the optimizer, and `model.zero_ grad()` is to clear the gradient of the entire network.

```
import torch_xla.core.xla_model as xmp

for data, target in mp_decice_loader:
    optimizer.zero_grad()
    model.zero_grad()

if __name__=='__main__':
    xmp.spawn(_mp_fn, args=())
```

Figure 5. API mismatch

Second, it may be that the third-party library update iteration speed is inconsistent, resulting in incompatibility between the interfaces of different libraries.

4.4 Coding Fault

Nearly half of the failures are coding faults, which can be quickly repaired through the log. The main reason is that students have little programming experience, and Python is a dynamic programming language with no strict requirements for parameter types and function types, so it is easy to make faults in the coding process. As shown in Figure 6, the function of `dumps()` requires the parameter type to be String, and the type of file passed in will cause a type defect.

```
import json

def write_data(self):

+   json.dumps("data.txt")
-   json.dumps(open("data.txt"))
```

Figure 6. Type defect

In addition, Python has strict regulations on code indentation. Some students may be used to other programming languages such as Java and ignore the indentation of Python, which may lead to coding defects. As shown in Figure 7, we need to extract the values in the dictionary for return. However, we failed to extract all the key_values due to code indentation, resulting in a numerical defect.

```
def extract_field(ori_dict, ori_list):

    new_dict = {}

    for value in ori_list:
        new_dict[value] = ori_dict[value]
-       return new_dict
+   return new_dict
```

Figure 7. Syntax defect

On the use of the default value of the function parameter, since the default value of the Python parameter is only used once, it will not be assigned after multiple calls, which will also cause parameter defects. For index defect, as shown in Figure 8, since the sequence number of the list starts from 0, the length should be subtracted by 1 when obtaining the last element of the list.

```
def read_list(self, list1):

    length=len(list1)

-   print(list1[length])
+   print(list1[length-1])
```

Figure 8. Index out of array

As shown in Figure 9, the float number needs to be converted into an integer type for processing in the program. Directly using `int()` for type conversion will throw it to the decimal part and trigger failure.

```
1   import math
2
3   if __name__ == "__main__":
4
5       min_num = 0.8
6       data_f = 8
7   -   data_num = data_f / int(min_num)
8   +   data_num = data_f / math.ceil(min_num)
```

Figure 9. Division by zero

To reduce coding defects, we should standardize code writing and improve the code review process. Maybe some static code review tools could be designed for such a specific requirement.

4.5 Data

Data fault accounts for 3.6% of all failures. For the object detection network, there may be label defect in making training sets by students, or the detection object is not strictly labeled, triggering network training failure. The encoding defect is caused by the inconsistent encoding format between the interpreter and the data file. As shown in Figure 10, the decoding type of the file is different from the encoding method. When loading the file, the garbled code is triggered, and the data cannot be loaded correctly.

```
1   data_file_name = "train_data.txt"
2
3   if __name__ == " __main__":
4
5   -   file_r = open(data_file_name, encoding = "GBK")
6   +   file_r = open(data_file_name, encoding = "UTF-8")
```

Figure 10. Encoding type defect

4.6 Others

For GPU and CPU out of memory, it is also possible that the students run other programs unrelated to the network simultaneously while training the network, resulting in the memory being unable to satisfy the network training needs, thus leading to the training stop.

5 Implication

To help students and entry-level software engineers avoid possible faults in object detection tasks, we give 13 concrete suggestions that belong to six classes. These suggestions correspond to those root causes one by one

5.1 Object Detection

5.1.1 GPU Out of Memory

The larger the value of `batch_size` and the smaller the value of `subdivisions`, the higher the network training efficiency, but it will also increase the GPU memory consumption. Therefore, the appropriate value of `batch_size` and `subdivisions` parameters should be set according to hardware to ensure normal network calculation.

5.1.2 Loss Nan

In order to avoid the failure of `loss Nan` caused by gradient explosion, an appropriate `learning_rate` should be set before network training. It is suggested to set a higher `learning_rate` before network training. If there is a `loss Nan`, the training efficiency and convergence speed can be achieved by continuously reducing the `learning_rate` until there is no `Nan`. Generally, it can be 1-10 times lower than the current `learning_rate`.

5.2 Execution Environment

5.2.1 Path Not Found

To be compatible with different representations of the same path in different environments, it is recommended to put the read file and the code execution file in the same directory and use the Python built-in function to load.

5.2.2 Module Not Found

For dependency defects, it is suggested that in the case of multiple third-party libraries with the same name. The third-party libraries should be strictly imported according to object detection network requirements to avoid failure due to differences in the execution environment caused by importing dependency defects.

5.3 API Mismatch

It is suggested that users carefully read the relevant API documents to understand the functions of different APIs. At the same time, the document maintainer should also update the documents in time.

5.4 Coding Fault

5.4.1 Attribute Not Found

When calling a function, carefully check whether the function name is correct, whether the called function exists in the class, and name the function strictly following the function naming rules.

5.4.2 Type Mismatch

The parameters should be passed in strict accordance with the parameter type of the calling function.

5.4.3 Syntax Defect

Strictly control the code indentation format to avoid failure caused by code indentation defects during coding with Python.

5.4.4 Illegal Index

When accessing list elements, remember that the initial index of the list is 0, and pay attention to the boundary of the list. Avoid trigger array out-of-range failures.

5.4.5 Division by Zero

If the integer part is 0, it should be avoided from becoming a divisor or using the function of ceil for processing.

5.5 Data

5.5.1 Input Data Defect

When making the object detection network training data, the category of the object to be detected should be carefully checked to avoid label defects.

5.5.2 Input Data Defect

The decoding type of the file should be the same as the encoding method of the interpreter. Otherwise, the garbled code will be triggered, and the data cannot be loaded correctly.

5.6 Others

During network training, other irrelevant programs should be closed to avoid network training stopping due to other programs occupying too many system resources.

6 Threats to Validity

The primary internal threat is that there is too much manual analysis and effort in our study. But we tried our best to minimize the subjectivity in the analysis. To reduce the threat, four authors analyzed the failures separately and discussed inconsistent results until at least an agreement was reached.

The external threats mainly involve two aspects. Firstly, it is possible that the research subjects (programs submitted by students) included in our study are not representative of the overall population. Therefore, we limit the suggestions provided by our empirical study to students and entry-level developers in the field of object detection. Secondly, our empirical study focused on the PyTorch-based Yolov4 object detection networks. For some other object detection networks, such as SSD [18] and fast-RCNN [19-20], the classification and the root cause of failure may be different. However, from our conclusion, most of the mistakes are not related to the structure of the DL framework, so the difference in the framework does not affect the correctness of our empirical results.

7 Relate Works

7.1 Empirical Study on Traditional Software

Yi et al. studied Bitcoin, Ethereum, Coin Monero, and Stellar programs on GitHub for blockchain, manually extracted the bugs in the program, and then analyzed the bugs at three levels. Finally, they divided these bugs into four categories and found 21 attack modes of blockchain, and proposed methods to avoid them according to different attack modes [4]. Zhao et al. collected the bugs generated in the process of software construction by studying open source projects: CXF, camel, and others, and comprehensively analyzed the bugs. By comparing the differences between build-process-bugs and other bugs in bug severity, fix time, and the number of files modified, the author found that the time spent on repeating a build-process-bugs was 2.03 times that of non-build bugs and the number of source files that need to be changed to repair an error in the build-process-bugs is 2.34 times that of non-build bugs, which indicates that the repair process of build-process-bug is more labor-intensive than other bugs [3]. Hirsch et al. collected 54,755 closed bug reports from 103 GitHub projects, created a benchmark dataset of 10,459 bug reports using heuristics, manually analyzed the root cause of bugs and classified them into three categories (semantic, memory, and concurrency), and based on this proposed a supervised machine learning approach for predicting the root cause of bugs [5]. Dalal et al. conducted root cause analysis on some historical severe software failures and some software failures that are still evolving, in addition to summarizing and comparing various approaches to defect root cause analysis [6].

7.2 Empirical Study on Deep Learning Software

Sun et al. collected 329 closed bugs in three popular machine learning (ML) projects on Github through artificial inspection, divided these bugs into seven categories, and summed up 12 repair modes. After that, an in-depth exploration was carried out from the time of repair and the software maintenance cycle where the bug was located. It was found that 70% of ML bugs were repaired within a month, and In the software maintenance cycle, the highest proportion is corrective maintenance [11]. Islam collected 2716 posts and 500 fixed bug commits on deep learning frameworks such as Caffe, TensorFlow, Keras, Thean, and Torch in StackOverflow and GitHub for empirical study. After analyzing the types of bugs, the root causes of bugs, the impact of bugs, and the stages in which bugs are prone to occur, the authors found that data errors and logic errors are the most severe types of errors in deep learning software, occurring more than 48% of the time. They found the same antipatterns that cause bugs in different deep learning frameworks [8]. Zhang et al. conducted a comprehensive empirical study on the failure of deep learning. After collecting the failure programs in the 4960 Microsoft deep learning platform Philly, they were divided into 20 categories by manually checking the failure messages. In addition, they also extracted 400 sample failures that were summarized to summarize common failure causes and solutions. Finally, the author also proposed tools for deep learning platforms based on the research content to avoid potential risks [12]. Zhang et al. pulled StackOverflow, and the program for building TensorFlow on the Github QA page collected 175 related bugs and made a quantitative analysis of these bugs, summed up the characteristics of common TensorFlow bugs and the root causes of bugs. Then, research the strategies of TensorFlow users to detect and locate bugs and summarize five kinds of positioning and repair strategies [9]. Han J take the first step to perform an exploratory study on the dependency networks of deep learning libraries. The study unveils some commonalities in various aspects of deep learning libraries and reveals some discrepancies as for the update behaviors, update reasons, and the version distributions. The findings highlight some directions for researchers and also provide suggestions for deep learning developers and users [21]. Tambon extracted closed issues related to Keras from the TensorFlow GitHub repository, categorized the bugs based on the effects on the users' programs and the components where the issues occurred, using information from the issue reports. They then derived a threat level for each of the issues, based on the impact they had on the users' programs and provided a set of guidelines to facilitate safeguarding against such bugs in DL frameworks [22].

8 Conclusion

Compared with traditional software, the development of DL-based software involves fewer lines of code but larger-scaled deep neural networks. For a developer of DL-based software, the risk of introducing bugs into the software still exists. As object detection is one of the most popular

application scenarios of deep learning techniques, it's helpful to pay attention to the bugs in DL-based object detection programs. To understand which types of failures occur most frequently in object detection programs and what are the root causes of them, we conduct an empirical study to investigate failure patterns in DL-based object detection programs. By exploring 101 submissions of a Yolov4 object detection task developed by 104 students in software development courses, we found the most frequent 13 failure patterns, six types of root causes of these failures, and 13 concrete suggestions to avoid these failures. Our empirical results can reveal basic laws of mistakes in the development of object detection programs, provide useful guidance to assist students and entry-level developers in improving skills in developing object detection programs, and assist teachers in performing high-quality teaching tasks of the development of object detection programs in university software development courses.

References

[1] R.-B. Abdessalem, A. Panichella, S. Nejati, L.-C. Briand, T. Stifter, Testing Autonomous Cars for Feature Interaction Failures Using Many-objective Search, *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, Montpellier, France, 2018, pp. 143-154.

[2] Y. Lecun, Deep Learning & Convolutional Networks, *IEEE Hot Chips 27 Symposium (HCS)*, Cupertino, CA, USA, 2015, pp. 1-95.

[3] X. Zhao, X. Xia, P.-S. Kochhar, D. Lo, S. Li, An Empirical Study of Bugs in Build Process, *29th Annual ACM Symposium on Applied Computing (SAC)*, Gyeongju, Korea, 2014, pp. 1187-1189.

[4] X. Yi, D. Wu, L. Jiang, K. Zhang, W. Zhang, *Diving Into Blockchain's Weaknesses: An Empirical Study of Blockchain System Vulnerabilities*, October, 2021, https://arxiv.org/abs/2110.12162.

[5] T. Hirsch, B. Hoffer, Root Cause Prediction Based on Bug Reports, *31st IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Coimbra, Portugal, 2020, pp. 171-176.

[6] S. Dalal, R.-S. Chhillar, Empirical Study of Root Cause Analysis of Software Failure, *ACM SIGSOFT Software Engineering Notes*, Vol. 38, No. 4, pp. 1-7, July, 2013.

[7] P. Bhattacharya, L. Ulanova, I. Neamtiu, S.-C. Koduru, An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps, *17th European Conference on Software Maintenance and Reengineering (CSMR)*, Genova, Italy, 2013, pp. 133-143.

[8] M.-J. Islam, G. Nguyen, R. Pan, H. Rajan, A Comprehensive Study on Deep Learning Bug Characteristics, *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, 2019, pp. 510-520.

[9] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, L. Zhang, An Empirical Study on TensorFlow Program Bugs, *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, The Netherlands, 2018, pp. 129-140.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.-S. Corrado, A. Davis, J. Dean, M. Devin, X. Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, March, 2016, https://arxiv.org/abs/1603.04467.

[11] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, B. Li, An Empirical Study on Real Bugs for Machine Learning Programs, *24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, 2017, pp. 348-357.

[12] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, M. Yang, An Empirical Study on Program Failures of Deep Learning Jobs, *42nd International Conference on Software Engineering (ICSE)*, Seoul, South Korea, 2020, pp. 1159-1170.

[13] T. Chappelly, C. Cifuentes, P. Krishnan, S. Gevay, Machine Learning for Finding Bugs: An Initial Report, *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Klagenfurt, Austria, 2017, pp. 21-26.

[14] F. Thung, S. Wang, D. Lo, L. Jiang, An Empirical Study of Bugs in Machine Learning Systems, *23rd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Dallas, Texas, USA, 2012, pp. 271-280.

[15] A. Bochkovskiy, C.-Y. Wang, H. Liao, *YOLOv4: Optimal Speed and Accuracy of Object Detection*, April, 2020, https://arxiv.org/abs/2004.10934.

[16] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, C.-L. Zitnick, Microsoft COCO: Common objects in context, *European conference on computer vision. Springer (ECCV)*, Zurich, Switzerland, 2014, pp. 740-755.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library, *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Vancouver, Canada, 2019, pp. 8024-8035.

[18] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A.-C. Berg, SSD: Single Shot MultiBox Detector, *European Conference on Computer Vision (ECCV)*, Amsterdam, Netherlands, 2016, pp. 21-37.

[19] R. Girshick, Fast R-CNN, *IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 2015, pp. 1440-1448.

[20] Z. Cai, N. Vasconcelos, Cascade R-CNN: Delving into High Quality Object Detection, *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, Salt Lake City, UT, USA, 2018, pp. 6154-6162.

[21] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, X. Xia, An Empirical Study of the Dependency Networks of Deep Learning Libraries, *International Conference on Software Maintenance (ICSME)*, Adelaide, Australia,

2020, pp. 868-878.
[22] F. Tambon, A. Nikanjam, L. An, F. Khomh, G. Antoniol, *Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow*, December, 2021, https://arxiv.org/abs/2112.13314.

Biographies

Ziyuan Wang received the Ph.D. degrees in computer science from Southeast University in 2009. He is currently an associate professor in computer science with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, China. His research interests mainly include software testing and programming language.

Jinwu Guo received the B.S. degree in management engineering from Nanjing University of Posts and Telecommunications in 2019. He is currently pursuing the M.S. degree in software engineering at Nanjing University of Posts and Telecommunications, China.

Dexin Bu received the B.S.degree in the internet of things engineering from Anhui Polytechnic University in 2020. She is currently pursuing the M.S. degree in software engineering at Nanjing University of Posts and Telecommunications, China.

Chongchong Shi received the B.S. degree in automation from Changzhou Institute of Technology in 2017. She currently works as a hardware engineer at Changzhou Xingyu Automotive Lighting Systems Co., Ltd, China.