CS 527 SP23 Course Project
FreeFuzz Extension for PaddlePaddle

Yiteng Hu*
Yuehao Shi*
yitengh2@illinois.edu
yuehaos2@illinois.edu
UIUC
Champaign, IL, USA

ABSTRACT

The article presents an extension of the FreeFuzz automated fuzz testing tool to test PaddlePaddle, an emerging open-source deep learning library. The team aims to improve the reliability of deep learning libraries by identifying and fixing potential bugs and vulnerabilities. The proposed solution involves four steps: code collection, instrumentation, mutation test, and oracle test.

To collect code snippets for our fuzz testing, we leveraged three distinct input sources: code snippets from the library's documentation, developer tests, and deep learning models in the wild. We then utilized code instrumentation for fuzz testing, which enabled FreeFuzz to trace the dynamic invocation information of APIs from all input sources. Additionally, we employed metamorphic testing to address the test oracle problem.

The team plans to use metrics such as covered APIs, the size of the value space, and line coverage to evaluate the effectiveness of the Freefuzz tests for PaddlePaddle. However, due to the limit of time, we failed to implement the code for evaluating the line coverage of our tests.

The team run 427 paddle api and found one potential bug caused by paddle.pow when using the precision oracle. The extensive study of FreeFuzz on paddle, a popular DL libraries, shows that FreeFuzz is able to automatically trace valid dynamic information for fuzzing 427 popular APIs.

Overall, our work demonstrates a promising approach of FreeFuzz to improving the reliability of deep learning libraries through automated fuzz testing. We believe that our methodology can be applied more broadly to other deep learning libraries and software systems beyond PaddlePaddle.

KEYWORDS

Fuzz testing, Deep learning libraries, PaddlePaddle, Automated testing, Mutation testing

*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Yiteng Hu and Yuehao Shi. 2018. CS 527 SP23 Course Project FreeFuzz Extension for PaddlePaddle. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 PROBLEM

Deep learning (DL) has become an indispensable tool for solving complex problems in various domains, including computer vision[1, 2], natural language processing[3, 4], and software engineering[5–8]. As DL models become more prevalent and essential, testing their reliability has become a critical issue, particularly in safety-critical applications. Many researchers have devoted significant efforts to testing DL models, mainly focusing on adversarial attacks[9–11], testing metrics[12–14], and specific applications[15–17].

However, there is limited work done for testing the underlying DL libraries that are crucial for building, training, optimizing, and deploying DL models. For example, for two famous DL libraries, PyTorch and TensorFlow, although existing work on testing DL libraries has shown promising results, they still suffer from several limitations, such as limited sources for test input generation, limited mutation techniques, and inefficiencies in model-level testing[18, 19].

FreeFuzz is a recent approach that utilizes open source mining to fuzz DL libraries[21]. This automated fuzz testing tool has been effectively employed to test TensorFlow and PyTorch libraries by generating a large number of test cases from code snippets in library documentation, developer tests, and DL models in the wild. FreeFuzz uses a line coverage metric to measure its code coverage.

PaddlePaddle is an emerging open-source DL library developed by Baidu[20]. It is widely used in many applications, such as image classification, object detection, and natural language processing. PaddlePaddle is also used by many companies, including Baidu, Xiaomi, and China Mobile. However, there is few existing work on testing PaddlePaddle library. To solve this problem, the project team aims to extend the existing fuzzing system to test PaddlePaddle library. The goal is to improve the reliability of PaddlePaddle library by discovering and fixing potential bugs and vulnerabilities.

In summary, our paper makes the following contributions:
Dimension. Our study introduces a novel dimension for carrying out fully automated fuzzing of PaddlePaddle library at the API level by extending FreeFuzz methodology. This is accomplished by mining data from real code and model executions in the wild.

Technique. We have adapted the effective fuzzing technique, FreeFuzz, for use in performing fuzzing tests on the PaddlePaddle library. Our technique leverages three distinct input sources: code snippets from the library's documentation, developer tests, and deep learning (DL) models in the wild. Code instrumentation is utilized for fuzz testing, which enables FreeFuzz to trace the dynamic invocation information of APIs from all input sources. Additionally, FreeFuzz employs metamorphic testing to address the test oracle problem. Throughout the fuzzing process, we provided detailed documentation and publicly shared our code, which can be utilized by other researchers interested in employing the FreeFuzz methodology to test the PaddlePaddle library.
Study. Our extensive study on PaddlePaddle library shows that FreeFuzz can successfully trace 511 out of 720 APIs.

2 SOLUTION

FreeFuzz performs four main steps for current DL libraries: 1) obtaining code/models from three different sources, 2) running the collected code/models with instrumentation to trace dynamic information for each covered API, 3) leveraging the traced dynamic information to perform fuzz testing for each covered API, and 4) resolving the test oracle problem with differential testing and metamorphic testing.

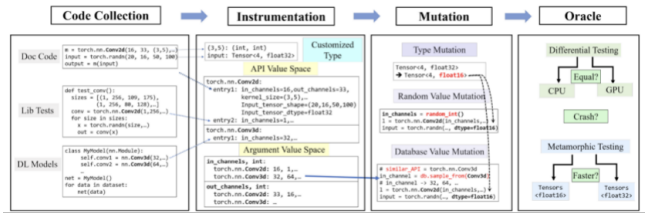

Figure 1: Work flow of Freefuzz

To extend FreeFuzz to test PaddlePaddle library, we will follow the same four steps. We will obtain code/models from three different sources: 1) code snippets from library documentation, 2) developer tests, and 3) DL models in the wild. We will then run the collected code/models with instrumentation to trace dynamic information for each covered API. Next, we will leverage the traced dynamic information to perform fuzz testing for each covered API. Finally, we will resolve the test oracle problem with differential testing and metamorphic testing.

To address the challenge of dynamic typing in Python, we will use type annotations to specify input parameter types for each API. By using type annotations, we can automatically determine API input parameter types and generate valid test cases.

By following the same four steps used in FreeFuzz paper and addressing challenges specific to PaddlePaddle library such as dynamic typing in Python using type annotations, we can extend FreeFuzz to test PaddlePaddle library and improve its quality and reliability. Additionally, we provided detailed documentation and publicly shared our code on Github (https://github.com/yuehaoshi/FreeFuzz), which can be utilized by other researchers interested in employing the FreeFuzz methodology to test the PaddlePaddle library.

2.1 Code Collection

The methodology for collecting code snippets for the PaddlePaddle framework was inspired by the FreeFuzz research paper. The team obtained code snippets from three primary sources, namely: (1) code snippets from the official documentation, (2) testing files from the PaddlePaddle library developer tests, and (3) deep learning models in the wild. The team also collected API executions of PaddlePaddle based on the same three sources.

The official website of PaddlePaddle was the primary source of code snippets, which provided all available API names, definitions, and execution examples for the latest version of the framework. To automate the process of parsing the documentation and obtaining the code snippets, the team utilized the bs4 Python package.

The team also collected testing files from the PaddlePaddle library developer tests. There are a total of 692 testing files in the latest version of PaddlePaddle, which the team obtained by cloning the latest PaddlePaddle official repository into their local machine.

Lastly, the team obtained code from deep learning models in the wild. The team collected publicly available PaddlePaddle models and utilized them as a source of real-world use cases for testing the PaddlePaddle libraries. For instance, the PaddleNLP package, which is an easy-to-use and powerful NLP library that leverages PaddlePaddle as a basic structure, provided numerous Paddle API calls that were suitable for collecting APIs for fuzzing. The team executed all testing files for this package to augment their API pool with this package.

2.2 instrumentation

Instrumentation is a crucial step in FreeFuzz that enables the collection of dynamic execution information for test-input generation. To extend FreeFuzz to test PaddlePaddle library, we will perform code instrumentation using the same approach as used in the original paper. The second stage of the Freefuzz adaptation process for PaddlePaddle involves dynamic tracing with instrumentation, which allows Freefuzz to intercept selected PaddlePaddle APIs and collect information about their execution, including input argument values and output values.

Next, we will insert additional code into this representation at specific points where an API is invoked to record its dynamic information such as name, input parameters and return value. Once all covered APIs have been instrumented, we will run all collected code/models with instrumentation enabled. During execution, whenever an instrumented API is called, its dynamic information is stored in MongoDB in JSON format to create the necessary type space, API value space, and argument value space for later fuzzing stages.

2.3 Mutation Test

In this phase, We applies various mutation rules to mutate the arguments.[21]
Mutation Rules. The mutation rules for FreeFuzz are composed of two parts: type mutation and value mutation, shown in Tables 1 and 2, respectively. Type mutation strategies include Tensor Dim Mutation that mutates n1-dimensional tensors to n2-dimensional tensors, Tensor Dtype Mutation that mutates the data types of tensors without changing their shapes, Primitive Mutation that mutates one primitive type into another, as well as Tuple Mutation

and List Mutation that mutate the types of elements in collections of heterogeneous objects.[21]
Different from pytorch and tensorflow, paddle doesn't have the api to generate random complex values. Thus, the team created such function which is randomizing the real part and imaginary part of complex number respectively.

We collected different types for PaddlePaddle, and decide to extend the mutation strategies used in the original paper. To summarize, the team has selected a subset of APIs as a proof of concept to demonstrate the feasibility of applying the complete Freefuzz testing process to the PaddlePaddle package, as set out in the midterm goals.

2.4 Test Oracle

The original Freefuzz paper proposed leveraging multiple execution modes during the instrumentation process to detect wrong-computation results. By comparing the results obtained from different execution modes, potential DL library bugs can be detected. However, due to having only MacOS, on which Paddle does not support CUDA, the team was unable to perform CUDA testing for Paddle.

Therefore, the team focused on detecting crash bugs and performance bugs by employing mutation-based fuzzing techniques that modify existing inputs, modify data type, and generate new inputs from scratch. By modifying existing inputs in various ways, FreeFuzz can identify performance issues and crash bugs that traditional unit tests may have missed.

3 EVALUATION

3.1 Implementation

Code Collection. For code collected from Paddle official documentation, the resulting scripts and outputs for crawling can be accessed from the */src/reptiledirectory*, while the API execution code snippets are available in src/reptile/code_snippets. After filtering out invalid results from the reptile tool and several API execution codes that caused instrumentation to crash, the team obtained a total of 826 API names, 720 API definitions, and 1493 executable code snippets from the official PaddlePaddle documentation website.

For official test source, the team cloned Paddle source code from Github and executed all test_*.py files in Paddle/test folder using pytest as automation testing tool. After removing unsupported imports by removing invalid tests, there are 2741 tests left. After executing all 2741 tests, only one more collection shown on MongoDB, but the total size in database increased 2%, indicating significant API execution overlap between official documentation code snippets and testing documents. For wild Paddle project source, the team collected PaddleNLP package and ran testing files inside it.

Instrumentation. To implement this instrumentation stage, the team repurposed the code used for testing the PyTorch library, utilizing functions such as hijack, decorate_class, decorate_function, and write_fn. Different sources of code collection require different ways of implementing instrumentation. During the process, the team found that not all collected APIs are valid for instrumentation, and decided to focus on fuzzing Paddle functions, keeping 511 APIs over 720 defined APIs on the Paddle official website.

For the code collected from the official documentation, the team executed all 1493 code snippets, but several code snippets could not compile during instrumentation due to typo errors on the Paddle official documentation examples. After executing all code snippets and hijacking API execution information into MongoDB, there are a total of 420 collections in MongoDB after running all code snippets, where one collection means the execution information of one API.

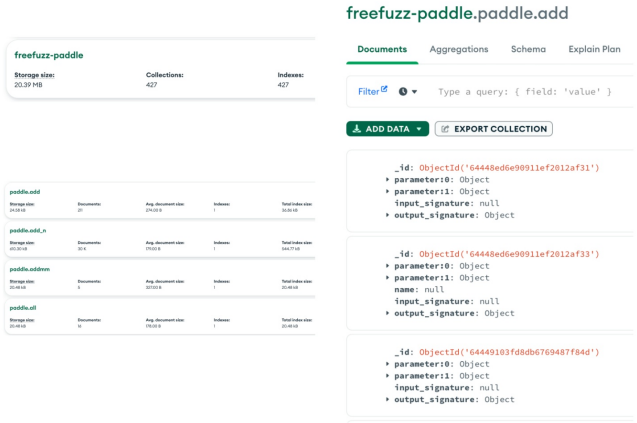

Figure 2: Database overview (left top), Database collections (left bottom), and Collection detail (right)

For the official test code, the team cloned Paddle source code from Github and executed all test_*.py files in Paddle/test folder using pytest as automation testing tool. After removing unsupported imports by removing invalid tests, there are 2741 tests left. After executing all 2741 tests, only one more collection shown on MongoDB, but the total size in database increased 2%, indicating significant API execution overlap between official documentation code snippets and testing documents.

For open-source projects, the team chose the PaddleNLP package and ran testing files inside it. After executing all testing files in PaddleNLP, there are 6 more collections, and 22% more MB data increased in MongoDB, meaning that PaddleNLP contributes additional API execution data in total data. PaddleNLP focuses on a specific area of APIs and generates more in-depth API calls compared to Paddle official tests, which have a large proportion of unexecutable tests and false testing files that hamper their contribution to the total data.

Mutation. We use the Mutation Algorithm proposed in [21]. The implementation details can be found in our project repository.

Test Oracle. Due to the lack of CUDA support on MacOS, we are unable to perform CUDA testing for Paddle. Since we can only run tests on CPU, we cannot do the differential testing. Meanwhile, the implementation of metamorphic testing is to wrap the invocation of APIs with code for timing.[21]

Table 1: Type Mutation

Mutation Strategies	T_1	T_2
Tensor Dim Mutation	tensor<n1,DT>	tensor<n2,DT>
Tensor Dtype Mutation	tensor<n,DT_1>	tensor<n,DT_2>
Primitive Mutation	T_1 = int|bool|float|str	T_2
Tuple Mutation	$(T_i^{i\in 1\dots n})$	$(typemutate(T_i^{i\in 1\dots n}))$
List Mutation	$[T_i^{i\in 1\dots n}]$	$[typemutate(T_i^{i\in 1\dots n})]$

Table 2: Value Mutation

Mutation Strategies	T	V
Random Tensor Shape	tensor<n,DT>	tensor(shape=[randint()],dtype=DT)
Random Tensor Value	v: tensor<n,DT>	tensor(shape=v.shape,dtype=DT).rand()
Random Primitive	int|bool|float|str	rand(int|bool|float|str)
Random Complex	real + imag	rand(real) + rand(imag)
Random Tuple	$(T_i^{i\in 1\dots n})$	$(value\_mutate(T_i^{i\in 1\dots n}))$
Random List	$[T_i^{i\in 1\dots n}]$	$[value\_mutate(T_i^{i\in 1\dots n})]$

3.2 Metrics

The team will use a set of metrics to evaluate the effectiveness of Freefuzz tests for PaddlePaddle. These metrics include the number of covered APIs, the size of the value space, and line coverage.
Number of Covered APIs. Given the extensive range of APIs available in DL libraries, it is essential to demonstrate the total number of APIs covered as a critical indicator of the thoroughness of testing.

Size of Value Space. Each API invocation will create a new entry in the API value space. Thus, we utilize the total size of the value space for all APIs as a metric to compare and analyze different input sources. In this project, we count the total amount of data size in the API value space, excluding any duplicates. It is important to note that this metric is primarily used to illustrate the scale of the traced data and should not be considered as a measure of fuzzing effectiveness.

Line Coverage. Code coverage is widely recognized as a crucial metric for evaluating the effectiveness of a test suite and its ability to provide thorough testing. As part of this project, our aim was to use line coverage as a metric to assess the effectiveness of Freefuzz tests for PaddlePaddle. To do this, we utilized the python coverage package during the instrumentation process. Unfortunately, we encountered some issues that prevented us from obtaining reliable line coverage metrics. As a result, we shifted our focus towards analyzing input source metrics and identifying failures and bugs.

Number of Detected Bugs. The effectiveness of Freefuzz tests for PaddlePaddle will be evaluated by the team based on the number of bugs detected, such as potential bugs, the ratio of failures to successes, and any other bugs that may be discovered during the project implementation process.

4 RESULT ANALYSIS

As shown in table.3, we run the tests on 427 collected paddle APIs from different sources and do freefuzz tests those code.

4.1 Input Source Study

The team evaluated the effectiveness of FreeFuzz for testing PaddlePaddle library by analyzing different input sources. We explored three different sources: 1) code snippets from library documentation, 2) developer tests, and 3) DL models in the wild.

Our results showed that FreeFuzz was able to automatically trace valid dynamic information for fuzzing 427 popular APIs using each input source individually, including 420 from official documentation, 1 from official tests, and 6 from wild project such as PaddleNLP.

As for the size of the value space, we found that the official documentation code snippets contributed the most to the total API execution data, with 80% of the total data, followed by the wild project PaddleNLP with 18% of the total data, and the official tests with 1.7% of the total data. In summary, our Input Source Study demonstrates that FreeFuzz is effective in generating test cases for PaddlePaddle library using different input sources such as code snippets from library documentation, developer tests and DL models in the wild. The proportion of API value size and API number distribution from different sources are shown in figure 3.

However, due to large portion of deprecated tests in official test and limited resources of large Paddle project in wild, official documentation code snippets contribute the most to the total API execution data and the other two sources are unable to contribute more APIs types, indicating significant API execution overlap between official documentation code snippets and testing documents.

4.2 Failed tests

Among 427 api tests, 94 tests failed. majority of them are either type error or value error. It is understandable that our mutation strategies will unavoidably generate some invalid inputs.

Table 3: FreeFuzz tests on paddle api

oracle	potential bug	fail	success	time-out
crash-oracle	0	94	333	0
precision-oracle	1	112	301	13
cuda-oracle(no device supported)	0	0	0	0

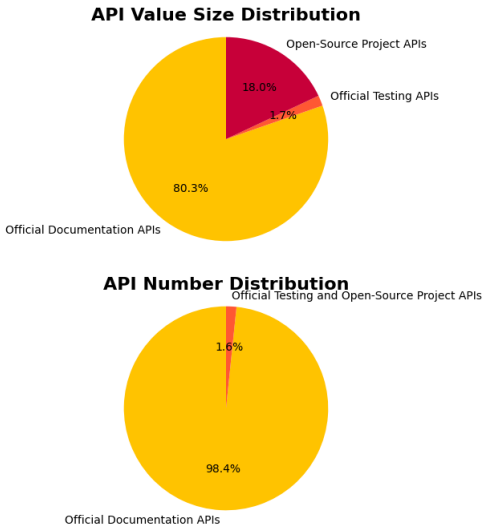

Figure 3: API Value Size and API Number distribution from different sources

4.3 Potential bugs

Although we only found one potential bug as shown below, it still has value for discussion. Let's first take a look at the criteria for judging whether the code will produce a potential bug. For precision oracle, we run two tests for one specific api. According to [21], we have this fact when we run the tests on hardware M.

$$precision(DT1) < precision(DT2) \Rightarrow cost(M, API_I, args, tensor\langle n, DT1\rangle) < cost(M, API_I, args, tensor\langle n, DT2\rangle) \quad (1)$$

Thus, when time of tests on low precision datatype is 10 times longer then the time of tests on high precision datatype, we will report a potential bug. In this case, low time value is 0.03082s while high time value is 0.001367s. Obviously, low time is 22.5 times longer than high time, which is larger than 10 times. However, due to the limitation of time, we have not found the intrinsic reason for this result. Thus, we can not confirm whether this is a real bug or not.

Listing 1: Potential bug when excuting precision oracle

```
import paddle
import time
results = dict()
arg_1_tensor = paddle.rand([1], dtype=paddle.float32)
arg_1 = arg_1_tensor.clone()
arg_2 = 4.0
start = time.time()
results["time_low"] = paddle.pow(arg_1,arg_2,)
results["time_low"] = time.time() - start
arg_1 = arg_1_tensor.clone().astype(paddle.float32)
start = time.time()
results["time_high"] = paddle.pow(arg_1,arg_2,)
results["time_high"] = time.time() - start
```

4.4 Other types of errors

4.4.1 paddle official documentation. During code collection and instrumentation part, the team collected code snippets from Paddle 2.4 version official website and collected approximately 1500 code snippets. After cleaning unexecutable snippets during wrong html tag of official website, there are several official code examples that would result in compile errors. After careful inspection, the team found four typos on Paddle official website examples such as redundant parenthesis and illegal syntax. The team then report those findings by opening several Paddle github issues as feedback[23, 24].

4.4.2 paddle official test files. To execute all official test files, the team initially cloned the Paddle source code from Github. They attempted to execute all testing files using the CMakeList file provided in the testing folder, but encountered several compile errors during the process. After conducting a thorough examination, the team discovered incompatibility issues between the CMakeList file and the current operating system. Consequently, the team opted to employ pytest as an automation testing tool to execute all testing files. During the pytest process, hundreds of import errors arose due to deprecated APIs and invalid tests.

Following the removal of unsupported imports by eliminating invalid tests, only 2741 tests remained, but a majority of them generated failure results. The considerable proportion of unexecutable and failed tests impeded their contribution to the overall data. Based on the instrumentation process of the project, the team discovered several compile errors during the process. After careful inspection, the team found several incompatibility issues between the CMakeList file and the current operating system.

Therefore, the team suggests that the Paddle development team should better maintain the testing files to prevent compile errors and import errors during the testing process. Additionally, they recommend providing instructions on the testing files regarding

the prerequisites for executing them on various systems, if possible.

5 FUTURE WORK

In the current implementation of extending FreeFuzz to the PaddlePaddle library, several limitations exist. Firstly, the team has used MacOS as the testing environment, which does not support CUDA for PaddlePaddle[22]. Consequently, testing of CUDA related APIs in PaddlePaddle is not possible. To test CUDA related APIs, future developers are advised to use an operating system other than MacOS as the testing environment.

Secondly, due to limited time and resources, the team conducted the fuzzing process only a few times, resulting in the discovery of only one potential bug. To increase the probability of finding potential bugs, it is recommended that future developers run the fuzzing process multiple times. As for the potential bug captured so far, the team targeted the reason for the bug, but failed to find the root cause, thus this would also be a future work to dig into.

In addition, the research team endeavored to employ the Python coverage tool in order to gauge the code coverage of the testing process. Regrettably, the team encountered unresolved difficulties pertaining to the utilization of coverage tools, which impeded the successful implementation of code coverage measurement. Notwithstanding that code coverage measurement is not the exclusive metric for assessing the efficacy of the testing process, and that an elevated code coverage does not inevitably signify a superior quality of the testing process, it remains a valuable metric for appraising and refining the testing process. Consequently, forthcoming efforts ought to be directed towards the realization of code coverage measurement for fuzzing the PaddlePaddle library. Finally, the team only used PaddleNLP as the wild project to test FreeFuzz. To improve the robustness of FreeFuzz, future developers are encouraged to use multiple wild projects for testing purposes.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016.

[2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[3] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. Neural computation, 12(10):2451–2471, 2000.

[4] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing, pages 6645–6649. Ieee, 2013.

[5] J. Chen, H. Ma, and L. Zhang. Enhanced compiler bug isolation via memoized search. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pages 78–89, 2020.

[6] X. Li, W. Li, Y. Zhang, and L. Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 169–180, 2019.

[7] Y. Yang, X. Xia, D. Lo, and J. Grundy. A survey on deep learning for software engineering. arXiv preprint arXiv:2011.14597, 2020.

[8] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang. Deep just-in-time defect prediction: how far are we? In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 427–438, 2021.

[9] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. Ieee Access, 6:14410–14430, 2018.

[10] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. arXiv preprint arXiv:1902.06705, 2019

[11] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572, 2014.

[12] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 851–862, 2020.

[13] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1039–1049. IEEE, 2019.

[14] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 120–131, 2018.

[15] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neuralnetwork-driven autonomous cars. In Proceedings of the 40th international conference on software engineering, pages 303–314, 2018.

[16] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 132–142. IEEE, 2018.

[17] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pages 347–358. IEEE, 2020.

[18] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1027–1038, 2019.

[19] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang. Deep learning library testing via effective model generation. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 788–799, 2020.

[20] Ma, Y., Yu, D., Wu, T. and Wang, H., 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. Frontiers of Data and Domputing, 1(1), pp.105-115.

[21] Wei, A., Deng, Y., Yang, C., Zhang, L. (2022, May). Free lunch for testing: Fuzzing deep-learning libraries from open source. In Proceedings of the 44th International Conference on Software Engineering (pp. 995-1007).

[22] PaddlePaddle. *Installation Guide*. PaddlePaddle. Accessed May 4, 2023. https://www.paddlepaddle.org.cn/documentation/docs/en/install/index_en.html.

[23] PaddlePaddle, GitHub. Issue 53437, https://github.com/PaddlePaddle/Paddle/issues/53437, Accessed May 4, 2023.

[24] PaddlePaddle, GitHub. Issue 53437, https://github.com/PaddlePaddle/Paddle/issues/53438, Accessed May 4, 2023.