BugsInDLLs : A Database of Reproducible Bugs in Deep Learning Libraries to Enable Systematic Evaluation of Testing Techniques

M M Abid Naziri
NC State University, Raleigh, USA
mnaziri@ncsu.edu

Aman Kumar Singh
Amrita Vishwa Vidyapeetham, Coimbatore, India
amanks@am.amrita.edu

Feiran Qin
NC State University, Raleigh, USA
fqin2@ncsu.edu

Benjamin Wu
Purdue University, West Lafayette, USA
wu2059@purdue.edu

Saikat Dutta
Cornell University, Ithaca, USA
saikatd@cornell.edu

Marcelo d'Amorim
NC State University, Raleigh, USA
mdamori@ncsu.edu

Abstract

AI-enabled applications are prolific today. Deep Learning (DL) libraries, such as PyTorch and Tensorflow, provide the building blocks for the AI components of these applications. As any piece of software, these libraries can be buggy. An impressive number of bug-finding techniques to address this problem have been proposed, but the lack of a curated set of reproducible bugs in DL libraries hinders credible evaluation of these techniques. We present BugsInDLLs, a database of curated reproducible bugs to fill that gap. Unique challenges exist in this context, such as installing drivers of specific CUDA versions to reproduce certain GPU-related bugs. Our dataset currently consists of 112 environments to reproduce bugs across three popular DL libraries, namely, JAX, Tensorflow, and PyTorch.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Deep learning libraries, testing, benchmarking

ACM Reference Format:
M M Abid Naziri, Aman Kumar Singh, Feiran Qin, Benjamin Wu, Saikat Dutta, and Marcelo d'Amorim. 2025. BugsInDLLs : A Database of Reproducible Bugs in Deep Learning Libraries to Enable Systematic Evaluation of Testing Techniques. In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3713081.3731739

1 Introduction

Several application domains (e.g., transportation and medicine) use AI as part of their solutions. Deep Learning (DL) libraries, such as JAX, PyTorch, and Tensorflow, provide the building blocks for the AI components of these applications. Unfortunately, as any piece of software, these libraries contain bugs. An impressive number of techniques have been recently proposed to find bugs in these libraries [1–3, 6, 7, 10, 11, 18, 19, 23, 24, 26], however we observe these techniques propose an independent evaluation methodology. These techniques do *not* use a reference database of reproducible bugs to evaluate their effectiveness. The lack of an evaluation standard is a serious obstacle to a fair and rigorous comparison of techniques and hinders research progress.

Although many datasets of reproducible bugs have been proposed in the literature, there is a lack of solutions satisfying the following criteria:
(1) ability to write test scripts in Python;
(2) handlin nightly builds and specific CUDA versions (§ 3.2);
(3) ability to integrate fuzzing tools in the framework.
Note that DL libraries are written in Python. Existing datasets of reproducible bugs exist in Python (e.g., BugInPy [25] and Tests4Py [20]), but they fail to satisfy the second or the third requirement. For example, BugsInPy and Tests4Py do not support artifacts in Docker, which are necessary to modify CUDA drivers in the guest OS to reproduce specific GPU-related bugs.[1] More importantly, a dataset of Deep Learning Libraries needs to support the integration of new fuzzing tools. The central purpose of such dataset is to *enable the systematic evaluation of testing techniques*.

We present BugsInDLLs, a database of curated reproducible bugs to enable credible evaluation of DL library testing techniques. BugsInDLLs is equipped with a command-line interface to enable researchers to analyze and reproduce bug instances. BugsInDLLs has been under active development since March 21, 2024, the day of the first commit in its GitHub repository. Two UROP students, two PhD students, and two faculty were involved in the work during this period. Section 2 details tool availability.

2 Tool Availability

BugsInDLLs is publicly available [15]. A video demonstrating BugsInDLLs is also available [16].

3 Objects and Methods

This section describes the criteria for selecting libraries and bugs (Section 3.1), the challenges for bug reproduction (Section 3.2), and the method we followed to create bug instances (Section 3.3).

[1]BugsInPy and Tests4Py use python virtual environments, e.g., venv and pyenv.

ISSTA Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1474-0/2025/06
https://doi.org/10.1145/3713081.3731739

Table 1: Characterization of bugs from BugsInDLLs.

	#	build release-nightly	enviroment conda-docker
JAX	46	45-1	45-1
PyTorch	37	18-19	21-16
Tensorflow	29	27-2	29-0
Σ	112	90-22	95-17

BugsInDLLs contains 112 instances of reproducible bugs representing three popular DL libraries, namely, JAX, Tensorflow, and PyTorch. Table 1 shows the list of bug instances for each supported library. Column "#" shows the number of bug instances, column "build" shows the breakdown of instances for each kind of build (release or nightly), and column "environment" shows the breakdown of instances for each kind of environment to reproduce the bug (conda enviroment or docker). Co-incidentally, all bugs that require a GPU are reproduced using docker containers, i.e. the column conda-docker matches what would have been observed with a column CPU-GPU, to indicate if the bug can be reproduced with a CPU or if it requires a GPU. Table 2 shows the breakdown of error types for the bugs in our dataset. The row "Incorrect Output" is listed first as it requires a distinct kind of oracle comparing consistency of the outputs of test runs on a CPU and on a GPU. In total, we have 18 different types of bug manifestations across the three libraries.

3.1 Selection Criteria

Libraries. PyTorch and Tensorflow are popular DL libraries intensively used in the literature. JAX is a newer library that has been rapidly gaining popularity. All these libraries are open source with active communities supporting their maintenance. From these three libraries, we selected issue reports from specific periods in their respective issue trackers. For PyTorch, we selected issues from the period between December 1, 2023 and August 16, 2024. For Tensorflow, we selected issues from the period between February 1, 2021 and July 31, 2024. For JAX, we selected issues from the period between August 1, 2023 and July 10, 2024. We have chosen different periods for each library depending on the number of issues reported in the issue tracker in that period. For example, Tensorflow has a much longer period compared to the others because the frequency of issues reported in the issue tracker is lower.

Bugs. We focus on issue reports that have been accepted by the developer community as true bugs. We determine this by checking whether the issue reports have the label "bug" and have been fixed with pull requests linked to them. We then create a filter that incorporates the periods mentioned above, the label (bug), the issue status being closed and the presence of a linked pull request. We found 725 issues for PyTorch, 217 issues for Tensorflow, and 111 issues for JAX after filtering the issues. PyTorch has the highest number of issues due to a lack of a label for bugs in the issue tracker. Then, we manually inspect the filtered issues to ensure that they contain enough information to reproduce the bug, i.e., code to reproduce the bug, dependencies required for the bug reproduction, clear description of the desired behavior, etc. We discard cases that

Table 2: Different bug manifestation types from BugsInDLLs.

Bug manifestation type	Jax	Pytorch	Tensorflow	Total
Incorrect Output	14	11	5	30
Internal Exception	11	9	4	24
Value Error	7	0	4	11
Type Error	0	1	8	9
Attribute Error	3	2	3	8
Index Error	0	7	1	8
Runtime Error	2	0	2	4
Memory Error	2	1	0	3
Invalid Argument Error	0	2	1	3
Floating Point Error	1	1	0	2
Error Not Raised	1	0	1	2
Parse Error	0	2	0	2
Runtime Warning	1	0	0	1
Not Implemented Error	1	0	0	1
Assertion Error	1	0	0	1
Segmentation Fault	1	0	0	1
Floating Point Exception	1	0	0	1
Aborted	0	1	0	1
Σ	46	37	29	112

are not in the core functionality of the library e.g. documentation issues, feature requests, etc. After this stage of manual inspection, we end up with 143 issues for PyTorch, 102 issues for Tensorflow, and 80 issues for JAX. Finally, we do a deeper analysis on these bugs. We find some cases that are related to building/installation/problematic unit tests instead of bugs in the core API functionality, as well as some cases that are simply API misuses, not bugs. These bugs can be detected and discarded at this stage. We assess the reproducibility of a bug by creating a virtual environment that contains the necessary dependencies to reproduce the bug. Some bugs require very specific hardware (e.g. TPU) or can not be reproduced due to requiring a debug build that only the developers can access. We successfully reproduce the bug if none of these issues are present by creating the enviroment detailed in the report, and we include the corresponding artifacts in our dataset. We use a virtual enviroment (conda [4]) when the bug can be reproduced on a CPU, and we use docker containers [12] with appropriate CUDA drivers when the bug requires a GPU. Throughout this process, we obtain **46**, **37**, and **29** bug instances for JAX, PyTorch, and Tensorflow, respectively.

3.2 Challenges

Handling bugs from nightly builds requires saving wheels. Many of the reproducible bugs rely on the libraries' nightly builds, which are available for a limited amount of time. To ensure that bugs reported on these builds remain reproducible, it is necessary to save Python wheel files (.whl) [13] for the correspondind builds.

Handling bugs in specific GPU-CUDA versions requires OS changes: Some GPU-related bug instances can only be reproduced with specific versions of CUDA [17], NVIDIA's platform and API for programming GPUs. These instances require specific CUDA drivers to be installed on the system. Python virtual environments are not designed to enable changing OS drivers. We use Docker containers [12] for the bug instances that require changes in drivers.

3.3 Method

The following steps show the method we use to create bug instances:

(1) Select an issue following the criteria defined in Section 3.1;
(2) Identify the version of the library used to report the bug along with other dependencies;
(3) Write a `requirements.txt` file with the list of dependencies to reproduce the bug;
(4) Create a file showing the code that causes the bug (for documentation);
(5) Create a file with a `pytest` test case that passes if the code triggers the buggy behavior documented in the issue;
(6) If the bug depends on a CUDA-specific build, write a Dockerfile to create a container including the needed CUDA-driver and the other artifacts mentioned above (e.g., test file and requirements);
(7) Write a script that creates the virtual environment (either Python's `Conda Enviroment` [4] or Docker container [12]) and runs the test on it;
(8) Create a directory `<lib>/<bug-id>` containing all the artifacts mentioned above, where `lib` is the affected library and `bug-id` is the issue number of the bug report.

Test Oracles. The bug reproduction code includes python tests (pytest). The test oracle is satisfied when the bug is reproduced successfully. For bugs that throw an unexpected exception, the test catches the exception and reports the exception details along with a pass status. For bugs that result in incorrect output value, assertions are placed in the code to expect the incorrect output value, hence passing the test when the bug occurs. For bugs that crash after raising a signal, the code snippet is placed in a seperate file and executed, while the file contaning the actual test asserts the presence of the signal raised by the code. Upon successful reproduction, the original code crashes raising the proper signal and the assertion passes.

Example The issue no. 120903 in PyTorch has the following test code that checks the oracle:

```
def test_f():
    input_data = torch.randn(3, 4, 5, 6)
    scale = np.array([0.1, 0.2, 0.3, 0.4])
    zero_point = torch.tensor([1, 2, 3, 4], dtype=torch.int32)
    axis = 1
    quant_min = 0
    quant_max = 255
    with pytest.raises(RuntimeError) as e_info:
        output = torch.fake_quantize_per_channel_affine(input_data,
            ↪ torch.from_numpy(scale), zero_point, axis, quant_min,
            ↪ quant_max)
    print(f'{e_info.type.__name__}: {e_info.value}')
```

In this example, the unexpected behavior is a `RuntimeError` thrown by the call to the `fake_quantize_per_channel_affine` API from PyTorch. Since the test oracle should pass when the bug is successfully reproduced, the code catches the exception and prints information about it, and upon catching the exception successfully, the test passes. If the bug reproduction fails i.e. the exception is not thrown or a different exception is thrown, the test will fail.

4 BugsInDLLs

This section presents the interface of BugsInDLLs and walks the reader through a demonstration showing the tool's functionality.

Table 3: BugsInDLLs command-line interface.

command	description
list-tests	List the tests available on this dataset
run-test	Runs one test
run-tests	Runs all the tests
run-tool	Runs a testing tool in a given buggy environment
show-info	Shows information about available tests
stats	Shows statistics about this dataset (e.g., number of tests that require GPU, etc)

4.1 Interface

Table 3 shows the list of commands in the BugsInDLLs's interface along with a short description for these commands. In the following we demonstrate BugsInDLLs.

4.2 Usage

To enable system-wide access of the framework, it is necessary to add the directory `/framework` to the `PATH` environment variable. Run the following commands for that:

```
$> git clone git@github.com:ncsu-swat/bugsindlls.git
$> cd bugsindlls
$> export PATH=$PATH:`pwd`/framework
```

4.2.1 Running one bug instance. Let us use bug 120903 [22] from PyTorch to demonstrate the command `run-test`, which runs a test to reproduce a bug on an environment with needed dependencies installed. Use the following command to reproduce the bug:

```
$> run-test --library-name pytorch --bug-id 120903
```

Execution of this command produces the following output:

```
...
Versions of relevant libraries:
[pip3] numpy==1.26.4
[pip3] torch==2.2.0+cpu
[conda] torch 2.2.0+cpu pypi_0 pypi
====== test session starts ======
platform linux -- Python 3.10.0, pytest-8.2.0, pluggy-1.5.0
...
test_issue_120903.py Pytorch issue no. 120903
Seed: 120903
RuntimeError: !needs_dynamic_casting<func_t>::check(iter) INTERNAL
    ↪ ASSERT FAILED at "../aten/src/ATen/native/cpu/Loops.h":310,
    ↪ please report a bug to PyTorch.
====== 1 passed in 0.96s ======
```

The output shows the dependencies installed in the environment to reproduce the bug and the verdict of the test oracle from pytest. In this case, the execution of the bug-revealing test throws a runtime error. Note that this test passes as it reproduces the intended bug.

4.2.2 Running FreeFuzz [24] on BugsInDLLs. A developer needs to provide three scripts to integrate a fuzzing tool: (1) a script that contains commands to run the tool (e.g. `run_freefuzz_docker.sh`); (2) a preprocessing script to extract error types and buggy APIs for a specific library version (`preprocess.py`), and (3) a post-processing script to match the execution log with the expected errors (`postprocess.py`). The directory `tool-integration` contains templates for these scripts and their instantiations for FreeFuzz [24], a popular API fuzzer for DL libraries. We demonstrate the integration of FreeFuzz [24]. The following script creates a Docker container for FreeFuzz.

```
$> cd tool-integration/FreeFuzz && bash install_freefuzz_docker.sh
```

The script `install_freefuzz_docker.sh` builds the docker container that encapsulates FreeFuzz.

The following script runs FreeFuzz on all bugs associated with version 2.2.1+cu121 of PyTorch. The command `run-tool` takes the library version as an input, along with the name of the docker container (already created), the name of the library, and the user-provided script containing the commands to run the tool (`run_freefuzz_docker.sh`). The script reports how many bugs that are reproducible with this version of the library were successfully reproduced by the given tool. Tool execution proceeds as follows. First, a Docker container associated with the tool is created using `install_freefuzz_docker.sh`. Next, a reproducible bug is tested using the specified library version as input to verify the setup. Following this, the preprocessing script (`preprocess.py`) is executed to extract error types and identify buggy APIs for the given library version. The environment inside the Docker container is then updated to match the specified version, ensuring compatibility before running the fuzzing tool. Once execution is complete, a postprocessing script (`postprocess.py`) is run to analyze the execution log against the ground truth, determining whether the bugs were successfully reproduced.

```
$> run-tool -c freefuzz -l pytorch -v 2.2.1+cu121 \
    --run-script tool-integration/FreeFuzz/run_freefuzz_docker.sh
```

The output looks like the following:

```
Using bug-id 121725 for library pytorch version 2.2.1+cu121
...
RuntimeError: Please look up dimensions by name, got: name = None.
====== 1 passed in 0.63s ======
Updating environment in the container of the testing tool
...
Running the testing tool on the environment of the bug
APIs under test:
torch.logsumexp
torch.autograd.grad
Testing on ['torch']
torch.multinomial
...
No violation of precision-oracle in the compare-bug category
No violation of precision-oracle in the potential-bug category
No violation of cuda-oracle in the compare-bug category
No violation of cuda-oracle in the potential-bug category
No violation of crash-oracle in the compare-bug category
No violation of crash-oracle in the potential-bug category
-> torch.logsumexp did not face any failures
-> torch.autograd.grad did not face any failures
Reproduced 0 out of 2 bugs
```

The execution log shows that the tool did not reproduce any of the two bugs that were expected to be reproduced. Among the two buggy APIs, FreeFuzz supports `torch.logsumexp` but the bug could not be reproduced, whereas FreeFuzz does not support `torch.autograd.grad`. The outputs from the tool are saved in the container so that the execution log can be inspected further manually. From this result, the developer can debug and refine their technique to catch bugs that it missed, as well as add support for APIs that are currently unsupported.

4.3 Contributions

BugsInDLLs allows users to contribute to the dataset by adding new bug instances. This opens up the tool to the community to help in expanding the dataset and improving the quality of the dataset. The following steps show how to contribute a new bug instance:

(1) Identify issues in the issue tracker of the library of interest;
(2) Create an issue in the repository of BugsInDLLs with the template "Reproduce Bug" (available in the repository);
(3) Create a branch linked to the issue;
(4) Add a self-contained bug reproduction script in a sub-directory named with the GitHub issue identifier under the directory of the library;
(5) Prepare the execution environment (Docker containers [12] for CUDA-dependent bugs, `Conda Enviroment` [4] for others);
(6) Follow the steps in 3.3 to create the bug reproduction script;
(7) Update the spreadsheets of the library with bug details;
(8) Create a pull request to the main branch.

5 Related Work

Several bug datasets have been proposed in literature for general software. Some popular examples includes software-artifact infrastructure repository (SIR) [5] which was one of the bug datasets containing 81 artificial faults in projects across multiple languages, Defects4J [9] which includes 357 real bugs across five large real-world Java projects, BugsInPy [25] containing 493 real bugs from 17 real-world Python programs, and BugsJS [21] containing 453 bugs across 10 Javascript projects. Google's FuzzBench [14] provides an evaluation platform for comparing general purpose fuzzers. Magma [8] is another fuzzing benchmark built by *front-porting* real bugs in latest version of projects. In contrast to these general purpose benchmarks, BugsInDLLs includes reproducible bugs for Deep Learning libraries like PyTorch and TensorFlow, and allows systematic benchmarking for DLL fuzzers.

6 Conclusion and Future Work

Deep Learning Libraries provide the basic blocks for creating AI-enabled applications. Several testing techniques have been proposed in the literature to find bugs in these libraries. Unfortunately, the lack of a dataset of reproducible bugs in those libraries poses an important barrier to the proper evaluation of these techniques. BugsInDLLs fills this gap. It includes a total of 112 bugs across three major DL libraries and provides an infrastructure to evaluate fuzzers. To facilitate evaluation of testing techniques even further we plan to continue expanding this dataset and to explore automated approaches to "frontport" bugs into single version of the library as in the Magma benchmark [8].

Acknowledgements

This work was supported and funded by the National Science Foundation grant numbers CCF-2349961.

References

[1] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[2] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[3] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.

[4] Anaconda Software Distribution. 2025. Conda environment. https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

[5] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10 (2005), 405–435.

[6] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1418–1430.

[7] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 486–498.

[8] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[9] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[10] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[11] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 657–669.

[12] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[13] Meshy. 2024. Python Wheels. Online. https://pythonwheels.com Accessed: 2025-01-11.

[14] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.

[15] M M Abid Naziri, Aman Kumar Singh, Feiran Qin, Benjamin Wu, Saikat Dutta, and Marcelo d'Amorim. 2024. BugsInDLLs. Online. https://github.com/ncsu-swat/bugsindlls Accessed: 2025-03-21.

[16] M M Abid Naziri, Aman Kumar Singh, Feiran Qin, Benjamin Wu, Saikat Dutta, and Marcelo d'Amorim. 2025. BugsInDLLs Demonstration. Online. https://www.youtube.com/watch?v=NslNWrULT1c Accessed: 2025-03-21.

[17] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[18] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[19] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. Acetest: Automated constraint extraction for testing deep learning operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 690–702.

[20] Marius Smytzek, Martin Eberlein, Batuhan Serce, Lars Grunske, and Andreas Zeller. 2024. Tests4Py: A Benchmark for System Testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 557–561.

[21] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. 2020. Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 308–319.

[22] vwrewsge. 2024. INTERNAL ASSERT FAILED. Online. https://github.com/pytorch/pytorch/issues/120903 Accessed: 2025-01-11.

[23] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. 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*. 788–799.

[24] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[25] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[26] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 176–188.

A Walkthrough

Section 4.2 demonstrates usage of BugsInDLLs. The reader can also follow the steps in the README.md file on our GitHub repository.