2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) | 978-1-7281-8710-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/MSR52588.2021.00070

Denchmark: A Bug Benchmark of Deep Learning-related Software

Misoo Kim
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
misoo12@skku.edu

Youngkyoung Kim
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
agnes66@skku.edu

Eunseok Lee
College of Computing
Sungkyunkwan University
Suwon, Republic of Korea
leees@skku.edu

***Abstract*—A growing interest in deep learning (DL) has instigated a concomitant rise in DL-related software (DLSW). Therefore, the importance of DLSW quality has emerged as a vital issue. Simultaneously, researchers have found DLSW more complicated than traditional SW and more difficult to debug owing to the black-box nature of DL. These studies indicate the necessity of automatic debugging techniques for DLSW. Although several validated debugging techniques exist for general SW, no such techniques exist for DLSW. There is no standard bug benchmark to validate these automatic debugging techniques. In this study, we introduce a novel bug benchmark for DLSW, Denchmark, consisting of 4,577 bug reports from 193 popular DLSW projects, collected through a systematic dataset construction process. These DLSW projects are further classified into eight categories: framework, platform, engine, compiler, tool, library, DL-based application, and others. All bug reports in Denchmark contain rich textual information and links with bug-fixing commits, as well as three levels of buggy entities, such as files, methods, and lines. Our dataset aims to provide an invaluable starting point for the automatic debugging techniques of DLSW.**

***Index Terms*—Automatic debugging, Bug report, Bug Benchmark, Deep learning-related software**

I. Introduction

Deep learning (DL) is a useful technique for big data management and augmentation of software (SW) intelligence. DL frameworks provide building blocks for the design, training, and validation of DL models via a high-level programming interface. Moreover, several tools and libraries support the efficient addition of DL functionality to the software. SW developers have developed DL-based SW applications employing these frameworks and tools to provide SW with DL capabilities [1]. Recently, DL-related SW (DLSW), which provides or utilizes DL functionality, significantly increased.

The increasing dependency of current SW on DL has aroused research for DLSW in the SW engineering field. Saleema et al. noted the fundamental difference between AI applications, which exploit the power of DL models, and conventional applications [2]. Gonzalez et al. also showed these differences in communication perspective [3]. Han et al. introduced a high-level workflow for DLSW and examined discussion topics on popular deep learning frameworks from discussion platforms, such as GitHub [4]. Zhang et al. made seven recommendations for DLSW practitioners based on their findings, such as the difficulty of controlling big data during the SW maintenance phase [5]. Conversely, Chen et al. focused on the deployment of DLSW. They proposed the taxonomy and challenges behind DLSW deployment and mentioned that DLSW-specific fault localization techniques for automatic debugging are required [1]. At the same time, various studies have analyzed the code, bug, and bug-fixing pattern of DLSW. Jebnoun et al. examined code smells in DLSW and confirmed that the presence of code smells may increase bug occurrence [6]. Nargiz et al. manually classified the real faults of DLSW and introduced a large taxonomy for faults in DL systems. [7]. Johirul et al. showed that data bugs and logic bugs are the most severe types of bugs in DLSW [8]. In their subsequent research, they demonstrated that the bug and its fixing patterns of the DL models fundamentally differ from those of conventional SW [9]. These diverse studies on DLSW illuminate the unique characteristics of DLSW and its bugs. Their findings indicate that it is imperative to consider distinctive automatic debugging techniques for DLSW to improve its quality.

Software debugging is an essential process for bug resolution. Automatic debugging techniques offer a possible remedy to solve the quality issue of DLSW. While several debugging techniques, such as bug localization and fixing, have been validated in conventional SW, no such techniques exist for DLSW. As previously mentioned, DLSW is more complicated than traditional SW. Unlike general SW, DL models are powered by data and are inherently non-deterministic owing to their hidden feedback loop and randomness [2]. This leads to a strong dependency of DLSW on the data quality. Such characteristics make existing debugging techniques unreliable for DLSW. Therefore, for the development of debugging techniques for DLSW, validation of previous debugging techniques on DLSW must be performed.

Prior to any validation, development or study of automatic debugging techniques for DLSW, a DLSW bug benchmark is required. A bug benchmark generally contains the following data: (1) a bug report that describes the bug; (2) a bug-fixing commit that splits the software status before and after the bug resolution (a fixing commit is also necessary to obtain human

978-1-7281-8710-5/21/$31.00 ©2021 IEEE
DOI 10.1109/MSR52588.2021.00070

bug-fixing patches); and (3) buggy entities such as buggy files, methods, and lines, that need to be found and modified to resolve the bug.

While several bug benchmarks for traditional software exist [10]–[15], no currently released bug benchmarks are devoted to DLSW. In this study, we introduce a well-made bug benchmark for DLSW, named Denchmark. We constructed a large-scale bug benchmark by utilizing bug reports as a starting point. Denchmark consists of 4,577 bug reports from 193 DLSW projects, which were classified into eight categories based on project description. As the first bug benchmark to validate automatic debugging techniques for DLSW, Denchmark's main advantages are as follows.

- **Well-made bug benchmark.** A well-made benchmark should be large-scale and accurate. In order to build a large-scale benchmark, we collected large-scale bug-fixing commits from commit logs and pull requests. Then, we performed five tasks including manual validation to filter out inaccurate commits.
- **First bug benchmark for the DLSW.** To our knowledge, no bug benchmark for DLSW currently exists. We systematically selected DLSW and constructed the bug benchmark, containing bug reports, bug-fixing commits, and buggy entities.
- **Diversity of DLSW.** The 193 DLSW projects were manually classified into eight categories: framework, platform, engine, compiler, tool, library, DL-based applications, and others. Different programming languages and DL frameworks are used for the DLSW.
- **Rich textual information of bug reports.** Bug reports in Denchmark include textual information and specific tagging information "code," which is the main structured text of GitHub issue reports. This text provides a wealth of information for text-based debugging techniques.
- **Fine-grained buggy entities.** Bug location is the primary data for bug localization and fixing. Denchmark provides information about buggy entities in various granularity levels, such as a buggy file, method, and line.

In summary, Denchmark provides rich information that enables the validation and development of automatic debugging techniques for DLSW. This dataset can provide valuable research opportunities to researchers in software engineering fields, with a focus on DLSW.

II. Methodology

A. Select target DLSW projects

To select DLSW, we focused on popular DLSW projects on GitHub. Utilizing the GitHub search API[1], we performed a keyword-based search. Based on the popular DL frameworks [16] and the popular DL models [2], we defined 22 DL-related keywords as follows.

[1]https://developer.github.com/v3/search/

[2]https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

Popular DL frameworks
tensorflow, keras, cntk, caffe, caffe2, torch, pytorch, mxnet, chainer, theano, deeplearning4j, dl4j, paddlepaddle
Common DL keywords
deep-learning, deep-neural-network, deep-reinforcement-learning, deep-q-learning, convolution-neural-network, recurrent-neural network, deep-belief-network, long-short-term-memory-network, deep-boltzmann-machine

After retrieving these projects, we selected only DLSW that met the following three constraints. First, the projects should be popular and active. We chose projects with a Stargazers count exceeding 100, whose latest commit is after 2020. Second, the projects should be implemented using the top-10 programming languages[3], namely JavaScript, Python, Java, Go, C++, Ruby, TypeScript, PHP, C#, and C. Third, the projects should have more than 10 closed bug reports. We considered an issue report with a label containing the keyword "bug" (excluding "not bug") as a bug report to filter out irrelevant issues (e.g., feature requests or tasks).

After the steps above, 275 projects were chosen as our initial candidates. We then manually investigated these candidates, filtering out 10 projects that are redirected and duplicated projects and 25 projects that do not possess DL functionality. For example, the Carla project[4] was selected in the initial set as it contains the keyword "deep-learning". However, Carla does not contain explicit DL functionality even though the sub-project of Carla contains this functionality. Such a project falls outside of our focus. We excluded the 24 other projects for similar reasons.

B. Collect bug reports

From 240 DLSW projects, we automatically downloaded 26,156 closed bug reports. The bug reports contain textual descriptions, comments, and HTML tags.

The HTML tags in the raw bug reports indicate the type of information provided by the text wrapped by the tag [17]. Unlike previous datasets [11], [14], [15], which excluded all HTML tags, our dataset provides a bug report with "CODE" tags among HTML tags to keep track of human tagging and provide code-related information. "CODE" is commonly used to define a piece of computer code and so it contains valuable information for software debugging techniques. We utilize the "<denchmark-code>" tag in our dataset to represent this text.

There are several text-based automatic debugging techniques, such as IR-based bug localization [15], [18], bug report-based fixing techniques [19], [20] and bug report-based fault injection [21], which can take advantage of valuable hints from tags. The tags can provide hints such as bug location, patch-related codes, and test-related codes.

C. Identify bug-fixing commits

To extract bug-fixing commits, we linked 26,156 closed bug reports of 240 DLSW with their corresponding commits that resolved the bugs. Here, we present two approaches used to discover bug-relevant commits.

The first approach is to retrieve the commit messages using the bug identifier (ID) and bug-related keywords (error, fix,

[3]https://madnight.github.io/githut/#/pull_requests/2020/3

[4]https://github.com/carla-simulator/carla

bug, crash, and #) as queries [11], [22]. In this way, we were able to find 12,538 commit links of 6,967 bug reports for 228 DLSW.

The second approach involves retrieving the relevant pull requests from the project's merged pull requests and then retrieving the corresponding commits of each pull request ID [14], [23], [24]. In GitHub, the event history of each bug report contains relevant pull requests that include the bug ID. To minimize the number of irrelevant pull requests, we only extracted pull requests with an ID greater than that of the bug ID. This extraction strategy guarantees that we only collect pull requests registered after the bug report is reported. Accordingly, we collected 9,455 relevant pull requests from 8,511 bug reports of 204 DLSW. We then retrieved the commit messages by utilizing pull request IDs and bug-related keywords as queries, replicating the methodology of the first approach. We found 8,654 commits linked with 6,511 pull requests and then linked 5,886 bug reports with these commits from 172 DLSW projects.

Finally, we merged the bug-fixing commits extracted from these two separate approaches. In the end, we identified 26,604 relevant commits for 10,283 bug reports from 229 DLSW projects.

D. Filter out bug-fixing commits

From the 10,283 bug reports with relevant commit links, we performed five tasks to discard some commits and bug reports to minimize incorrect links between bugs and commits.

The first task is to exclude bug reports linked to two or more commits. Multiple commits linked to a single bug report can include not only bug-fixing commits but also refactoring commits that modify the already fixed code [25]. The automatic linking process cannot precisely weed out refactoring commits from multiple commits. For this reason, we performed the first filtering task.

The second task involves excluding the commits with no modified files. Some commits are performed to add new files or delete source files. Current bug localization and fixing techniques can localize and fix the actual buggy codes [12], [18], [19], [26], [27]. However, if all committed files are newly added or deleted, it is difficult to find or change the codes in the buggy-state software. To avoid this situation, we excluded all commits in which there were no modified files in the committed files.

Furthermore, we excluded any commit performed before the bug report opening date and after its closing date. These past commits are referenced for bug resolution but clearly did not fix the bug directly.

We excluded several bugs that shared the same commit. When the commit mentioned more than two bug IDs, the bugs related to this commit were removed from candidates because automatically identifying the buggy entity for each bug report from one commit can be imprecise [12], [28].

Through these tasks, 5,863 bug reports were removed leaving 4,577 bug reports with bug-fixing links from 193 DLSW projects. To guarantee the correctness of the final dataset, finally, we manually validated that no errors existed in the ground-truth data. To do so, we randomly sampled 450 bug reports (≈10%) and manually investigated them. We confirmed that there were no wrong bug-fixing commits linked to bug reports. Every bug report in our investigation scope was linked with actual commits which were modified to fix the bug.

E. Extract buggy entities

Buggy entities are important data for the validation of bug localization and fixing. We extracted the buggy entities by utilizing PyDriller [29], a Python-based framework that can effectively extract information from the Git Repository. We extracted the files in each of the 4,577 commits, regardless of the added, deleted, and modified files. We also extracted buggy methods and lines when the files were modified.

III. Denchmark

Table I summarizes the Denchmark with overall statistics for 193 DLSW projects and detailed statistics for exemplary 39 DLSW projects. Exemplary projects indicate the projects with more than 30 bug reports.

A. Project statistics

As shown in Table I, Denchmark includes 193 DLSW projects collected from GitHub. The first column contains the class of each project. We classified DLSW by the SW type they described in their project description to avoid misunderstanding the intentions of each SW developer. There were 36 frameworks (F) in total and 10 frameworks had more than 30 bug reports: general frameworks mentioned in Section II-A and domain-specific frameworks. The number of platforms (P) and engines (E) among the target DLSW projects were 12 and five, respectively. There were three compilers, 44 toolkits (T), 63 libraries (L), and 17 DL-based SW applications (A). Remaining 13 DLSW were the other types such as development tools, standard, and DL practice code repository.

The third column depicts the number of programming languages utilized, among the top-10 programming languages described in Section II-A. The average number of employed languages was 2.2. The fourth column indicates the number of utilized DL frameworks among the popular DL frameworks mentioned in Section II-A. We extracted their respective numbers based on "import" and "from" and the name of each DL framework [7]. The average number of frameworks used was 2.7. These statistics suggest that DLSW researchers should understand multi-language and multi-framework environments to develop automatic debugging techniques.

B. Bug report statistics

The fifth column in Table I summarizes the numbers of all bug reports and the number of bug reports with the CODE tag. Ranging from 1 to 370 for each project, the total number of bug reports was 4,577. The total number of bug reports with CODE was 1,969, while their ratio among all bug reports was approximately 43%.

The sixth column displays the average number of comments in the bug reports, which was 3.5. A sufficient number of

Table I: Denchmark Statistics.

Class* (#Project)	Project	#PL	#FR	#Bugs All (Code)	#Cmt◇	Day◇	#BuggyEntity◇ File / Mth / Line
F (36)	BentoML	3	3	49 (25)	3.0	9.2	4 / 6.3 / 42
	Deeplearning4J	8	6	51 (27)	4.7	80.7	11.7 / 18.3 / 106.8
	Haystack	1	1	36 (15)	4.1	10.8	2.7 / 4.2 / 36.6
	Horovod	2	4	48 (30)	4.6	13.3	2.9 / 5.5 / 55.6
	mxnet	4	6	252 (168)	4.8	79.5	4 / 6.6 / 55.9
	ncnn	3	4	46 (9)	2.3	18.5	3 / 3.8 / 59.9
	PaddlePaddle	4	4	60 (21)	2.4	8.3	3.2 / 5.1 / 40.4
	ray	6	3	370 (239)	3.7	31.5	3.8 / 7.3 / 60.9
	Tensorflow	8	5	306 (173)	7.1	64.4	2.6 / 4.1 / 40.7
	Tensorflow.NET	3	1	36 (2)	1.4	4.4	5.5 / 8.4 / 41.9
	All**	**2.7◇**	**3.0◇**	**1,466 (805)**	**3.3**	**31.2**	**3.1 / 4.9 / 35.8**
P (12)	Kubeflow	4	4	81 (29)	6.0	39.4	2.8 / 2.9 / 41.5
	OpenPAI	6	5	45 (8)	1.8	48.0	2.9 / 2.2 / 33.8
	All**	**3.3◇**	**3.4◇**	**275 (94)**	**3.3**	**30.8**	**11.8 / 11.8 / 46.8**
E (5)	ONNX Runtime	7	6	57 (23)	5.2	28.1	3.3 / 6.9 / 72.6
	All**	**4.6◇**	**2.8◇**	**87 (38)**	**6.8**	**15.4**	**6.2 / 13.9 / 102.8**
C (3)	SQLFlow	4	3	48 (18)	1.2	17.0	2.3 / 3 / 32.8
	All**	**5.0◇**	**3.3◇**	**60 (28)**	**3.8**	**14.3**	**2.3 / 4 / 45.9**
T (44)	Fairseq	2	2	35 (17)	1.7	18.9	2.7 / 3.8 / 21.3
	garage	1	3	69 (11)	2.4	29.5	6.8 / 11 / 81.7
	GluonTS	1	3	41 (29)	3.0	17.8	3.3 / 3.4 / 31.3
	MONAI	3	3	49 (11)	2.1	3.8	4.1 / 7.5 / 39.9
	NNI	3	3	49 (17)	3.0	27.8	2.6 / 2.8 / 25.2
	OpenNMT-tf	1	2	33 (12)	3.0	5.8	2.1 / 2.2 / 12.6
	TensorPack	1	3	48 (18)	2.0	0.6	2.2 / 2.3 / 10
	TorchIO	1	1	37 (3)	2.4	4.5	3.1 / 4.6 / 23.7
	Turi Create	4	6	195 (89)	2.6	73.4	6.8 / 71.4 / 340.9
	All**	**1.7◇**	**2.8◇**	**895 (351)**	**3.6**	**20.7**	**3.5 / 5.3 / 69.6**
L (63)	AllenNLP	2	2	31 (9)	3.4	24.2	3.8 / 3.8 / 33.5
	DGL	3	5	35 (14)	3.4	15.4	2.5 / 4.6 / 28.7
	Ignite	1	3	40 (21)	2.4	12.2	4.1 / 11 / 56.2
	oneDNN	3	8	42 (22)	3.8	15.5	4 / 21 / 49.2
	Open3D	4	3	45 (21)	4.3	55.0	5.5 / 7.2 / 61.3
	OpenCV	5	3	157 (85)	3.3	171.8	2.3 / 4.8 / 49.8
	PYRO	2	2	56 (19)	2.8	18.2	2.9 / 5.1 / 34.1
	PyTorch Lightning	1	3	240 (133)	5.4	17.7	3.7 / 7.3 / 45.8
	Syft	1	2	34 (17)	3.0	34.0	3.8 / 4.3 / 56.9
	Tensorflow Addons	2	3	60 (31)	3.4	16.4	2.9 / 3.6 / 28.1
	Tensorflow.js	7	3	55 (25)	6.7	24.3	4.6 / 4.8 / 90
	TorchBearer	1	2	44 (1)	0.6	6.1	4 / 9 / 83.7
	All**	**1.9◇**	**2.5◇**	**1,169 (538)**	**3.5**	**38.7**	**5.9 / 10.9 / 52**
A (17)	CVAT	3	2	48 (12)	3.2	30.7	4 / 4.2 / 48.3
	OctoBot	1	0	34 (2)	0.3	23.3	3 / 4.5 / 24.4
	PhotoPrism	2	1	49 (13)	6.5	28.0	3.6 / 5 / 36.1
	All**	**1.9◇**	**1.2◇**	**228 (61)**	**3.2**	**41.0**	**2.8 / 3.7 / 133.4**
Others○ (13)	DeepForge	2	2	312 (30)	0.2	3.8	1.7 / 5.4 / 47.9
	All**	**1.8◇**	**3.3◇**	**397 (54)**	**2.6**	**60.0**	**2.3 / 2.6 / 65.3**
Total**	**193**	**2.2◇**	**2.7◇**	**4,577 (1,969)**	**3.5**	**33.4**	**4.6 / 7.3 / 62**

PL: used programming language, **FR**: used frameworks, **Cmt**: comment in each bug report, **Day:** resolution days.
*: F: Framework, P: Platform, E: Engine, C: Compiler, T: Toolkit or Toolset, L: Library, A: Application
○: Others: DLSW development tool, standard format and practice codes, ◇: Average of overall projects
**: Statistics of all projects in the category, not just exemplary projects

comments can help DLSW researchers understand bugs. The seventh column shows the average time of bug resolution, which was 33.4, which is larger than one month; i.e., if researchers develop automatic debugging techniques, developers can save an average of 33 days of both time and cost.

C. Buggy entity statistics

The eighth column presents the average numbers of buggy entities (based on entity level) as 4.6, 7.3, and 62. In case of OpenPAI on Platform class, the average value of modified methods is less than that of modified files. This value potentially means that the bugs could be fixed by 1) changing outside the methods (e.g., global variables) rather than changing the methods or by 2) changing the files written by non-programming languages (e.g., YAML files [30]).

IV. Research Opportunity

The foremost opportunity based on Denchmark is the study of bugs and debugging techniques for DLSW. Our dataset represents a well-curated bug benchmark for each project as we have systematically filtered out irrelevant data by integrating several methods from existing studies. It may reduce the cost of suitable DLSW selection because it offers DLSW researchers a representative class for each project and an experimental dataset construction.

The main applications of our dataset relate to the facilitation of research on locating and fixing DLSW bugs, which are the primary automatic techniques for debugging. Denchmark allows researchers to investigate topics relevant to DLSW bugs and the methods of fixing them, providing a method to determine optimal techniques. Below, we highlight potential research areas that could be leveraged through Denchmark.

- **Text-based debugging techniques for DLSW:** Our dataset provides textual bug reports with both <CODE> and comments, providing valuable textual information.
- **Fine-grained bug localization for DLSW:** Bug reports are linked to the buggy entity on three levels, which provide fine-grained ground-truth data.
- **Automatic bug-fixing for DLSW:** Bug reports are linked to bug-fixing commits so that it can provide human patches. The buggy lines provide a fixed location.

V. Conclusion and Future Works

Considering the ever-increasing interest and demand for DLSW, an automatic debugging technique should be developed to ensure the quality of DLSW. In this study, we created a large-scale bug benchmark for DLSW called Denchmark, constructed via the aforementioned process, containing features introduced in this study. All datasets and implementations are released on https://github.com/RosePasta/Denchmark_BRs.

However, some improvements can still be made for the Denchmark. Currently, there exist some bugs without test files in the fixing commit, impeding reproduction. However, in our dataset, some bug reports were linked with changed test files (approximately 35% in 193 DLSW). These reports are reproducible with their test files. It can be assumed that the failed test file represents the circumstance of bug and buggy behavior. We plan to develop a testing framework for DLSW such as Defects4J and BugsInPy [10], [13]. Based on the testing framework for DLSW, we will validate existing debugging techniques, especially bug localization and fixing.

Furthermore, the bug-fixing patterns in DLSW can provide important insights for researchers with respect to automatic debugging techniques. Thus, based on fine-grained buggy entities on Denchmark, we will collect these patterns associated with DLSW by utilizing several studies [31], [32] and the fault taxonomy for DLSW derived in [7].

Acknowledgment

This work was supported by the National Research Foundation of Korea grant funded by the Korea government (MSIT) (2017M3C4A7068179, 2018R1D1A1B07050073, 2019R1A2C2006411)

REFERENCES

[1] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 750–762, 2020.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[3] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 431–442, 2020.

[4] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. What do programmers discuss about deep learning frameworks. *EMPIRICAL SOFTWARE ENGINEERING*, 2020.

[5] Xufan Zhang, Yilin Yang, Yang Feng, and Zhenyu Chen. Software engineering practice in the development of deep learning applications. *arXiv preprint arXiv:1910.03156*, 2019.

[6] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. The scent of deep learning code: An empirical study. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 420–430, 2020.

[7] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.

[8] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019.

[9] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. Repairing deep neural networks: Fix patterns and challenges. *arXiv preprint arXiv:2005.00972*, 2020.

[10] René Just, Darioush Jalali, and Michael D Ernst. 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*, pages 437–440, 2014.

[11] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 61–72, 2018.

[12] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[13] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 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*, pages 1556–1560, 2020.

[14] Sandeep Muvva, A Eashaan Rao, and Sridhar Chimalakonda. Bugl–a cross-language dataset for bug localization. *arXiv preprint arXiv:2004.08846*, 2020.

[15] Shayan A Akbar and Avinash C Kak. A large-scale comparative evaluation of ir-based tools for bug localization. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 21–31, 2020.

[16] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.

[17] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 474–477. IEEE, 2015.

[18] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.

[19] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 314–325, 2019.

[20] Manish Motwani and Yuriy Brun. Automatically repairing programs using both tests and bug reports. *arXiv preprint arXiv:2011.08340*, 2020.

[21] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Ibir: Bug report driven fault injection. *arXiv preprint arXiv:2012.06506*, 2020.

[22] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[23] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[24] Yue Yu, Zhixing Li, Gang Yin, Tao Wang, and Huaimin Wang. A dataset of duplicate pull-requests in github. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 22–25, 2018.

[25] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, 2014.

[26] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[27] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–10. IEEE, 2019.

[28] Xin Ye, Razvan Bunescu, and Chang Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering*, 42(4):379–402, 2015.

[29] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[31] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020.

[32] Wei Lin, Zhifei Chen, Wanwangying Ma, Lin Chen, Lei Xu, and Baowen Xu. An empirical study on the characteristics of python fine-grained source code change types. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*, pages 188–199. IEEE, 2016.