Empirical Software Engineering (2023) 28:62
https://doi.org/10.1007/s10664-023-10291-1

Bugs in machine learning-based systems: a faultload benchmark

Mohammad Mehdi Morovati[1] · Amin Nikanjam[1] · Foutse Khomh[1] · Zhen Ming (Jack) Jiang[2]

Accepted: 11 January 2023

Abstract

The rapid escalation of applying Machine Learning (ML) in various domains has led to paying more attention to the quality of ML components. There is then a growth of techniques and tools aiming at improving the quality of ML components and integrating them into the ML-based system safely. Although most of these tools use bugs' lifecycle, there is no standard benchmark of bugs to assess their performance, compare them and discuss their advantages and weaknesses. In this study, we firstly investigate the reproducibility and verifiability of the bugs in ML-based systems and show the most important factors in each one. Then, we explore the challenges of generating a benchmark of bugs in ML-based software systems and provide a bug benchmark namely *defect4ML* that satisfies all criteria of standard benchmark, i.e. relevance, reproducibility, fairness, verifiability, and usability. This faultload benchmark contains 100 bugs reported by ML developers in GitHub and Stack Overflow, using two of the most popular ML frameworks: *TensorFlow* and *Keras*. *defect4ML* also addresses important challenges in Software Reliability Engineering of ML-based software systems, like: 1) fast changes in frameworks, by providing various bugs for different versions of frameworks, 2) code portability, by delivering similar bugs in different ML frameworks, 3) bug reproducibility, by providing fully reproducible bugs with complete information about required dependencies and data, and 4) lack of detailed information on bugs, by presenting links to the bugs' origins. *defect4ML* can be of interest to ML-based systems practitioners and researchers to assess their testing tools and techniques.

Keywords Benchmark · Machine learning-based system · Software bug · Software reliability engineering · Software testing

Communicated by: Andrea Stocco, Onn Shehory, Gunel Jahangirova, Vincenzo Riccio

This article belongs to the Topical Collection: *Special Issue on Software Testing in the Machine Learning Era.*

✉ Mohammad Mehdi Morovati
mehdi.morovati@polymtl.ca

Extended author information available on the last page of the article.

Published online: 05 April 2023

1 Introduction

Recent outstanding successes in applying Machine Learning (ML) and especially Deep Learning (DL) in various domains have encouraged more people to use them in their systems. ML-based systems refer to software systems that contain at least one ML component (software component whose functionality relies on ML). Given the increasing deployment of ML-based systems in safety-critical areas such as autonomous vehicles (Pei et al. 2017; Ma et al. 2018) and healthcare systems (Esteva et al. 2019), we need to provide an acceptable level of reliability in such systems.

Software reliability is broadly considered to be the most important software quality factor among all software quality attributes (Lyu 2007), where such attributes measure the conformance level of the system, component, or process to the identified functional and non-functional requirements (Pressman 2005). Software Reliability Engineering (SRE) is the methodology to ensure failure-free operations of the software in a specified period of time. Substantial portion of SRE techniques has been developed based on studying the lifecycle of bugs (Radjenović et al. 2013; Lyu 2007).

It is generally accepted that standardized benchmarks are the most efficient tools for evaluating and comparing products and methodologies (Kistowski et al. 2015). A benchmark must satisfy some quality criteria to be considered as standard, including relevance, reproducibility, fairness, verifiability, and usability (Kistowski et al. 2015; Vieira et al. 2012). It is also worth mentioning that benchmark construction is a long-term and iterative process that requires the cooperation of the community (Lu et al. 2005). Accordingly, a standard benchmark of bugs is an essential requirement to evaluate, compare, and improve such research on the SRE approaches focusing on the bug's lifecycle.

Benchmark of software bugs that contains a set of real bugs is known as faultload benchmark (Vieira et al. 2012). Several studies on the faultload benchmark for traditional software systems have been done, e.g., *Defects4J* (Just et al. 2014) (a benchmark of bugs in Java open source projects hosted on GitHub), *BugBench* (Lu et al. 2005) (a benchmark of C/C++ programs' bugs), *ManyBugs* (Le Goues et al. 2015) (a benchmark of defects in C programming language), and *Bears* (Madeiral et al. 2019) (a Java bug benchmark for automatic program repair). On the other hand, some faultload benchmarks such as JaConTeBe (Lin et al. 2015) (a benchmark of java concurrency bugs) are designed for specific types of bugs. Accordingly, *defect4ML* also ignores general bugs and considers bugs that are related to the ML components.

Similar to other faultload benchmarks, benchmark of ML-based systems' bugs is the basic necessity for comparing, tuning, and improving testing techniques/tools of ML-based systems. Extracting, reproducing, and isolating real bugs in traditional software still need considerable time and effort. Concerning the higher complexity level of the ML-based systems in comparison with traditional ones (Amershi et al. 2019) and challenges in the engineering of ML-based systems (Galin 2004), providing reproducible bugs in these systems might require more effort. Although preceding studies have developed some benchmarks of bugs in ML-based systems (Kim et al. 2021; Wardat et al. 2021), they totally disregarded the standard benchmark criteria. As an example, *Denchmark* (Kim et al. 2021) does not provide enough information to reproduce and trigger bugs. Meanwhile, several studies on the testing of ML-based systems have used synthetic bugs for assessment (Nikanjam et al. 2021a, b) which may bias their evaluation by hiding potential weaknesses. Some others have also used a limited number of real bugs (Wardat et al. 2021; Schoop et al. 2021) that may not be representative of a thorough evaluation, implying an incorrect measure of the proposed approach's reliability. So, in this research we aim to answer the following research questions:

RQ1. *What are the key factors in reproducibility of reported bugs in ML-based systems?*
RQ2. *What are the important factors in verifiability of ML-based systems' bug-fixes?*
RQ3. *What are the challenges of generating standard faultload benchmark in ML-based systems?*

To answer these research questions, firstly, we investigate 5 public datasets of ML-based systems' bugs and manually check 513 and 498 bugs that they provided from GitHub and SO, respectively. Then, we checked 1264 additional bug-fix commits extracted from ML-based systems repositories. Furthermore, we review 798 Stack Overflow (SO) posts related to *TensorFlow* (Abadi et al. 2016) and *Keras* (Chollet and et al 2018) frameworks. We examine the reproducibility and verifiability of the bugs in ML-based systems, as two of the most demanding criteria of standard benchmark.

We also provide a faultload benchmark of ML-based systems namely *defect4ML*. Figure 1 illustrates a high-level view of the proposed benchmark. The base layer is the benchmark containing the database of bugs. The next layer, *Python* virtual environment, refers to the environment that should be configured to run applications and trigger the bugs. Because each buggy application has different dependencies and requires different sorts of libraries to be triggered, a *Python* virtual environment would be the best solution for running buggy applications in isolation. The top layer represents the potential usage of bugs which is mostly ML testing tools such as *NeuraLint* and *DeepLocalize*. Given the high cost of providing bugs from ML-based systems that satisfy standard benchmark criteria, we set 100 bugs (62 from GitHub and 38 from SO) as our goal for the first release of *defect4ML*. The included bugs are classified into different categories, based on various criteria including Python version, ML framework, violated testing property, and bug type. Different users such as developers, distributors, and researchers of ML-based systems testing tools/techniques can benefit from our proposed benchmark. They can use *defect4ML* to evaluate their proposed approaches for bug detection, or localization and compare them with previous works. Besides, *defect4ML* has potential to be used for automatic bug repairing tools. To this end, users should remove modifications from bug-fix which are not related to the reported bug (similar to methodology used in Jiang et al. 2021). The data generated/analysed during the current study are available in the benchmark repository.[1]

The contributions of this study can be summarized as follows:

- **First standard benchmark of ML-based systems' bugs**: Proposed benchmark satisfies all of the standard benchmark criteria (e.g. relevance, fairness, etc)
- **Large-scale and accurate bug benchmark**: We collected the bugs from GitHub commits and SO posts to provide a large-scale benchmark. Besides, we applied several steps (including manual checking) to filter the bugs and extract the ones satisfying our defined criteria. We have also provided fine-grained classifications based on different criteria (such as ML framework, bug type, etc.) to enable users to filter the bugs and collect a desired subset. Users can also add new bugs to the benchmark and raise a request for removing an existing bug to keep the benchmark up-to-date.
- **Bug reproducibility**: Our analysis revealed that only about 5.3% of all reviewed GitHub bugs and near to 3.34% of reported bugs in SO posts are reproducible. However,

[1] http://defect4aitesting.soccerlab.polymtl.ca/

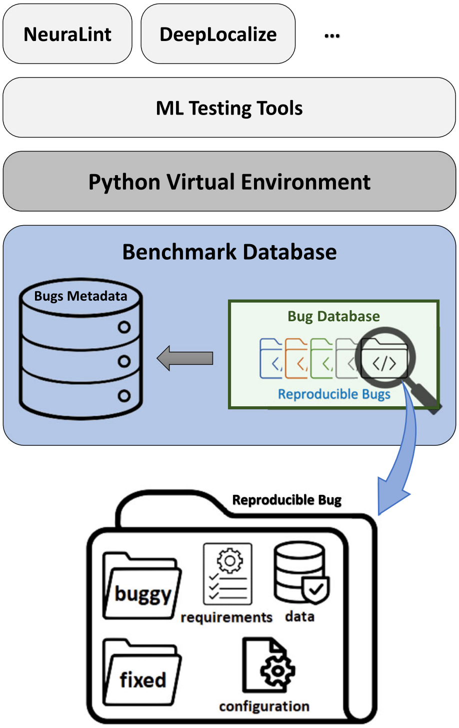


Fig. 1 High-level view of the benchmark

all bugs in *defect4ML* are completely reproducible. We have also provided contextual information for each bug including the needed version of Python, dependencies (necessary libraries and corresponding version), data, and the process of triggering bugs, to allow for reproducibility.

- **Bug-fix verifiability**: Our analysis revealed that only a small portion of the studied ML-based systems' bugs (i.e., 13.3% of collected bugs from GitHub in *defect4ML*) can be verified by the provided test cases in their applications. Moreover, none of the reviewed bugs reported in SO posts has test cases.
- **Detailed information of the bugs**: We provide the URL to the origin of gathered bugs in our proposed benchmark that includes textual information about bugs including: buggy entities such as file name and line of code, bug's root cause, and the fixed version (how the bug got fixed).
- **Diversity**: We have covered 30 different types of bugs based on the taxonomy proposed by Humbatova et al. (2020) (including 95 types of ML-related bugs), to promote diversity of *defect4ML*. Besides, we used GitHub and SO as the primary sources of collecting bugs. To gather bugs from GitHub, we have also explored repositories developed by users with different levels of expertise. In addition, we have presented bugs based on the two most popular ML frameworks, *TensorFlow* and *Keras* (Yalçın 2021; Humbatova et al. 2020).

The rest of the paper is organized as follows. We explain the background of the study in Section 2. The methodology followed to answer the research questions is explained in Section 3. The results of analyzing collected bugs and the proposed benchmark is described in Section 4. Section 5 represents the discussion of our study. Then, the related works are mentioned in Section 6. We discuss threats to the validity of this research in Section 7. Finally, we conclude the paper in Section 8.

2 Background

This section introduces concepts of ML-based systems and SRE in these systems, the general picture of the benchmark, and the criteria it should meet to be considered a standard benchmark.

2.1 ML-Based System

Software systems including at least one ML component are known as ML-based systems. ML components are defined to be software components working based on ML algorithms with the aim of proving intelligent behavior (Martínez-Fernández et al. 2021). An ML component may be only a small part of a much larger system. Figure 2 shows a high-level view of the ML-based systems exposing the role of the ML component in them.

To simplify the design, implementation, and integration of ML components in software systems, several ML frameworks such as *TensorFlow* (Abadi et al. 2016), *Keras* (Chollet and et al 2018), and *PyTorch* (Paszke et al. 2019) have been developed. They help developers to create, train, and then deploy various types of ML models. Hence, ML frameworks play a vital role in developing ML-based systems (Zhang et al. 2020). Similar to any other software components, ML components are also error-prone. However, current ML frameworks do not provide any capability to validate and verify the developed ML components.

2.2 Bugs in ML-Based Systems

In general, software bug is known as the inconsistency between the existing and expected software functionality, also called deficiency in satisfying software requirements (Zubrow

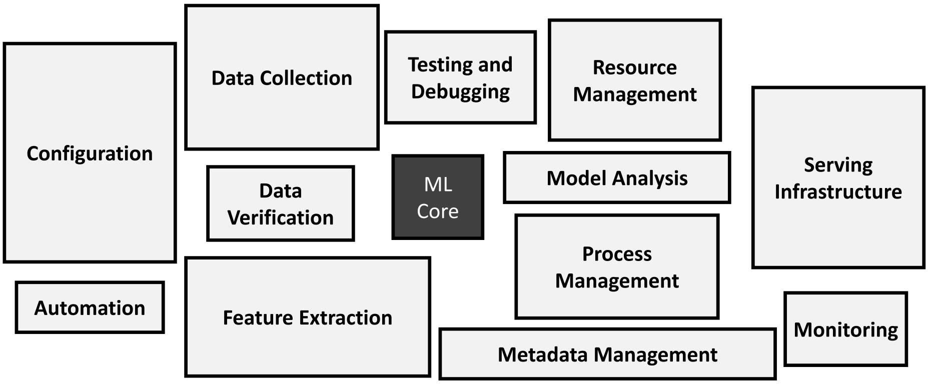


Fig. 2 An high-level view of the ML-based systems (Sculley et al. 2015)

2009). Accordingly, ML bug refers to the deficiencies in ML components, which may lead to discrepancies between existing and the required behavior of ML component (Zhang et al. 2020). An ML bug can occur in the ML framework (Jia et al. 2021b; Rivera-Landos et al. 2021; Tambon et al. 2021), program code (Islam et al. 2019; Zhang et al. 2018b), or the data. So, researchers study the bugs in each area separately (Zhang et al. 2020). In this study, we just investigate the bugs in program code and do not consider bugs in ML framework and data.

There are three different testing levels for ML-based systems: model testing, integration testing, and system testing (Riccio et al. 2020). At the model testing level, we consider the ML component in isolation and ignore other software components. Integration testing level aims to assess the interaction between ML and other components. System testing level studies all software components to evaluate the ML-based system's conformance to the intended requirements. Accordingly, we can consider three different categories of bugs in ML-based systems, each one related to one of testing levels.

- Bugs in ML components: At this level, we consider the ML components in isolation and study the bugs inside them ignoring other components of the system.
- Bugs in any component that affect functionality of ML components: At this level, all identified bugs in the previous level have been taken into account plus the bugs which are out of the ML components but affect the ML component's functionality. In this paper, we call this category as ***ML-related*** bugs.
- Bugs in all software components: At this level, we do not make any difference among ML components and others considering all system bugs as the same.

Based on the definition of faults in IEEE standard glossary of software engineering terminology (2010), software fault is manifestation of a bug in software. In other words, when a software bug causes an incorrect software operation, it becomes a software fault (Galin 2004). Concerning that faults emerge from the discordance between software requirements and the existing behavior, faults can be functional or non-functional. Functional faults refer to the inability of the software to meet the required functionality. Non-functional faults stem from the deficiencies in methodologies to achieve the required functionality, not the functionality. An ML fault is also considered as an inadequacy in the behavior of the ML component (Humbatova et al. 2020; ISO 2019). A software fault results in a failure only when a user tries to use the faulty software component, leading to fault activation (Galin 2004). Generally speaking, failure in software engineering is known as the inability of the system or its components to fulfill required functions (Riccio et al. 2020). Faults in ML-based systems may also lead to bad performance, crash, data corruption, hang, and memory out of band which are considered as failure in these systems (Islam et al. 2019).

2.3 SRE in ML-Based Systems

Because of the essential differences between the paradigm of traditional and ML-based software systems, we are facing several new challenges in SRE of ML-based systems. Various studies have acknowledged the significant challenges in SRE of the ML-based systems. Fast changes in the new versions of ML frameworks is one of the major challenges that Islam et al. (2020) reported. As an example, they exposed that almost 26% of operations have been changed from version 1.10 to 2.0 in *TensorFlow*. Code portability is another crucial challenge in the SRE of ML-based systems (Lenarduzzi et al. 2021). There are multiple ML frameworks (e.g. *TensorFlow*, *Keras*, *PyTorch*, etc.) just for Python programming language. Although they have some similarities, there are major differences among them. Therefore, understanding and porting ML codes from one framework to another can be a nontrivial task. Bug reproducibility is another significant challenge in the SRE of ML-based systems (Zhang et al. 2018b). Wardat et al. (2021) also reported the lack of detailed information regarding the bugs in ML-based systems as a basic challenge in SRE of ML-based systems. We aim to cover these challenges in our proposed benchmark.

SRE techniques mostly use the lifecycle of bugs (Lyu 2007). One of the major SRE approaches using the bug's lifecycle is fault removal that aims to detect the existing faults and remove them. Such techniques use validation and verification approaches to cope with reliability concerns that are known as software testing techniques. Overall, software testing is considered as one of the most complicated tasks of the software development process. It is well-accepted that the complexity of the testing has a direct relation with the complexity level of the system to be tested (Galin 2004). That means, by increasing the complexity of the system, testing becomes more complicated to be able to deal with the system quality flaws. It has also been proved that the complexity level of the ML-based systems stays at a higher place compared to traditional ones (Amershi et al. 2019). Consequently, testing of the ML-based systems is considered as more complicated tasks, in comparison with traditional software systems.

ML testing properties refer to the conditions that are needed to be guaranteed for a trained model during testing. In other words, ML testing properties represent the quality attributes that should be tested and satisfied in ML-based systems (Zhang et al. 2020). Existence of bugs in ML-based systems may result in violation of various ML testing properties, depending on the impact of bugs on the system. In this study, we used the introduced ML testing properties by Zhang et al. (2020) which are briefly reviewed in this subsection. Correctness represents the probability that the ML system works in the right way, as it is intended. Model relevance checks the complexity of the ML model to make sure it is not more complicated than required. In fact, model relevance aims at preventing model overfitting. Overfitting happens when the complexity of the employed ML algorithm is more than required. Robustness is defined as the extent to which the ML system is able to handle invalid inputs and functions correctly. Efficiency refers to the speed at which ML systems operate and perform the defined tasks (e.g., prediction). Fairness ensures that ML systems make decisions without bias. Interpretability is the degree to which humans can understand the reasons behind the decisions that ML systems make. Although testing properties have been categorized into six different classes, they may overlap with each other (Zhang et al. 2020).

Bad performance, crash, data corruption, hang, incorrect functionality, and memory out of bound are symptoms of ML-related bugs which are known as various failure types in ML systems (Islam et al. 2019; Zhang et al. 2018b). Bad/poor performance refers to the situation where the accuracy of the ML component is not as good as expected. Crash is the most common symptom of ML-related bugs in which ML-based software stops running with or without showing an error message. Data corruption means data has been corrupted when it passes the network which leads to wrong output. When the ML software stops responding to the input without prompting an error, it is known as hang. Incorrect functionality refers to the situation that ML software behavior differs from the expected, without any error. Memory out of bound occurs due to the unavailability of required memory for training. It should be also taken into consideration that symptoms of bugs belonging to each ML testing property can be different. For instance, if the correctness testing property of an ML component gets violated, its symptoms may be any of known types such as bad performance, crash, hang, etc.

It should be also taken into consideration that symptoms of bugs belonging to one ML testing property can be different. For instance, when the correctness testing property of the ML component has been dissatisfied, its symptoms may be any of known ML-based systems failure types such as bad performance, crash, hang, etc.

It is not surprising that researchers have used testing techniques from traditional software systems to cope with testing challenges of ML-based systems. However, traditional testing methodologies would not be sufficient and efficient testing approaches for ML-based systems (Marijan et al. 2019). Traditional testing methods require adaptation to the context of ML to be effective for them (Bourque et al. 1999). Moreover, the concept of quality is not well-defined in ML-based systems and its terminology is different from the traditional ones (Lenarduzzi et al. 2021; Borg 2021). It is also worth noting that the intrinsic difference between ML-based and traditional software systems generates new types of bugs which do not exist in the traditional software systems (Riccio et al. 2020). For instance, the behavior of the ML-based systems is heavily dependent on factors such as training dataset, hyperparameters, optimizer, etc. Besides, it is hardly possible for humans to debug the learned behavior which is encoded by weights within the ML model.

Several studies have been carried out to provide tools to test ML-based systems. Wardat et al. (2021) conducted research to localize the bugs in ML-based systems. They explained that because understanding ML models' behavior is challenging, existing debugging methods for ML-based systems do not support localization of the bugs. They provided a dynamic mechanism to analyze the ML components and implemented an alternative "callback" mechanism in *Keras* to collect the detailed information of the ML component during the training phase. Then, their proposed tool analyzes the collected data to discover possible bugs and their root causes. Islam et al. (2020) carried out an empirical study on the challenges that the automated repairing tools should address. They reviewed SO posts and GitHub bug fixes using the five most popular ML frameworks (*Caffe*,[2] *Keras*,[3] *TensorFlow*,[4] *Theano*,[5] and *Torch*[6]) to identify fix patterns. They classified the bug-fix patterns specially used in Deep Neural Networks (DNN) into 15 different categories and provided various solutions to fix bugs belonging to different classes. Schoop et al. (2021) offered a system namely UMLAUT to assist non-expert users in identifying, understanding, and fixing bugs in the DL programs. UMLAUT can be attached to the DL program to check the model structure and its behavior. Then, it suggests the best practices to improve the quality of ML components. Nikanjam et al. (2021a) provided an automatic fault detection tool for DL programs namely *NeuraLint* that validates the DL programs by detecting faults and design inefficiencies in the implemented models. They identified 23 different rules using graph transformations to detect various types of bugs in DL programs.

2.4 Benchmark

Benchmark is known as a standard tool for the competitive evaluation of systems and for making comparisons amongst systems or components, in terms of specific characteristics

[2] https://caffe.berkeleyvision.org/

[3] https://keras.io/

[4] https://www.TensorFlow.org/

[5] https://github.com/Theano/Theano

[6] http://torch.ch/

like performance, security, and efficiency (Vieira et al. 2012). It is widely acknowledged that a standardized benchmark is the most significant requirement to evaluate and compare the methodologies (Kistowski et al. 2015). To generate a standardized benchmark, several criteria should be satisfied in the development process of the benchmark, including (Kistowski et al. 2015; Vieira et al. 2012):

- **Relevance:** asserts that the result of benchmark can be used to measure the performance of the operation in the problem domain. That is to say, how the benchmark behavior relates to the behavior of interest to its consumers. Relevance is mostly considered as the most important factor of any standard benchmark (Kistowski et al. 2015). Without providing relevant information to the benchmark users, it is highly possible that the benchmark will not be in the users' interest, even if it gives perfect services for other criteria. As a general rule of thumb, the benchmark that is well-suited for a particular domain has limited applicability, while the benchmark trying to cover a broader range of domains will be less meaningful for any specific domains (Huppler 2009).
- **Reproducibility:** refers to the benchmark ability to provide the same results while it is run using the same configuration.
- **Fairness:** explains that all competing systems be able to use the benchmark equally. In other words, the benchmark should be usable for all systems, without generating artificial limitations.
- **Verifiability:** ensures that the benchmark results are accurate.
- **Usability:** means that the benchmark should be understandable easily to prevent credibility shortage. Credibility shows the level of confidence that users have in the results (Rodríguez-Pérez et al. 2018). Ease of use is also another important property belonging to this criterion.

Faultload benchmark is one of the main categories of standard benchmark that includes a set of faults and tries to provide experience of the real faults occurring in the system. It is also commonly confirmed that faultload benchmark is the most complex one among all benchmark categories, because of the complicated nature of the faults (Vieira et al. 2012).

Concerning the drastic influence of ML in several safety-critical areas during the last few years, the reliability engineering of ML-based systems has become more crucial. A faultload benchmark of ML-based systems can play a vital role in the assessment of methods working on the reliability engineering of the ML-based systems. Although there may exist a great number of benchmarks in each software domain, a few numbers of them satisfied requirements of the standard benchmark (Vieira et al. 2012). Accordingly, there are several public datasets of bugs in ML-based systems provided as either replication package of their study or a faultload benchmark, but they have some considerable problems. For example, bug's dataset provided in Zhang et al. (2018b) does not provide any information about the application dependencies and ignores reproducibility of the collected bugs entirely. Besides, several reported bugs were based on deprecated versions of Python (older than version 3.6) which might be inefficient for assessment of the current ML-based systems testing tools.

Another public bugs dataset that is provided by Islam et al. (2020) has some similar problems. Firstly, they completely disregarded the reproducibility of the bugs. While dealing with the dependency challenge, we came across bugs unrelated to the ML or which occurred in old versions of *Python* (older than 3.6). On the other hand, some mentioned bugs were based on ML frameworks which are discontinued. For instance, this public dataset noted 15 bugs using *Theano* (a *Python* library to define, optimize, and evaluate mathematical expressions) (Al-Rfou et al. 2016). With regards to the fact that *Theano* is a deprecated library and not supported anymore, adding bugs that use *Theano* would not be valuable. Also, it reported 17 bugs based on Torch (Collobert et al. 2002), a scientific computing framework for ML algorithms, which development has been deactivated since 2018 (Organisation 2021).

Similar to the former studies, in the public bugs dataset published by Humbatova et al. (2020), many reported bugs depend on *Python* older than 3.6. We also found several commits mentioned as bug-fix, while they are not a real bug like (2016) or not accessible like (2018). Lack of dependency information is another shortcoming of this public dataset.

It is worth noting that datasets provided in Humbatova et al. (2020), Islam et al. (2020), and Zhang et al. (2018b) are replication packages of those studies and authors do not aim to introduce a faultload benchmark. However, those datasets can be useful for researchers who are studying bugs in ML-based systems.

In the public bugs dataset that Wardat et al. (2021) have delivered, the reproducibility of the bugs received no attention resulting in many non-reproducible bugs. Besides, the coverage of the provided bugs is also relatively limited. That is to say, they cover limited types of bugs in ML-based systems.

To the best of our knowledge, there is no benchmark for ML-based systems' bugs that satisfies the mentioned criteria of the standard benchmark.

3 Methodology

In this section, we describe the methodology that we followed to answer our RQs. To this end, we need to collect and investigate the ML-related bugs. Figure 3 represents the methodology that we used to collect the bugs for answering our RQs.

We used two main sources to gather bugs: (1) public datasets of previous studies on the bugs in ML-based systems, and (2) ML-related bugs reported in GitHub or SO. To gather the bugs from the prior research, we reviewed several articles that were about bug's lifecycle in ML-based systems and provided public bugs datasets (Zhang et al. 2018b; Islam et al. 2020; Wardat et al. 2021; Humbatova et al. 2020). As the second source, we extracted the bugs reported in (a) bug-fix commits of GitHub repositories or (b) SO posts. We focused on two of the most popular ML frameworks, *TensorFlow* and *Keras* (Yalçın 2021; Humbatova et al. 2020), respecting the well-known popularity metrics (Zerouali et al. 2019) (e.g., number of stars and number of forks) of their GitHub repositories. Table 1 represents the detailed information regarding the popularity metrics of the selected ML frameworks (on the date we checked them).

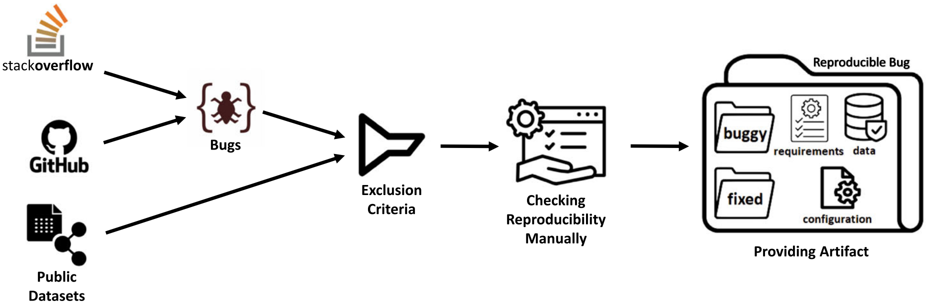


Fig. 3 Methodology of collecting bugs

Table 1 Detailed information about the selected ML frameworks

ML Framework	#stars	#forks	#subscribers
TensorFlow	$160K$	$85.7K$	$8K$
Keras	$52.2K$	$18.8K$	$2K$

We narrowed down our study to the bugs in DL systems, based on the main categories of DL systems faults' taxonomy provided by Humbatova et al. (2020): model, tensors and input, training, GPU usage, and API categories. "Model" refers to the bugs related to the structure of the model (such as (2018)). "Tensor and input" category covers the bugs regarding the problems in the shape and format of data (such as (2018)). "Training" includes the bugs in the model training process (such as (2018)). "GPU usage" category deals with bugs occurred when using GPU devices in DL systems (such as (2018)). "API" refers to the problems related to the API usage in ML frameworks (such as (2018)).

To examine the collected bugs and remove those which could not satisfy the standard benchmark criteria in manual checking step, we used some exclusion criteria:

- **bugs using Python version older than 3.6.** The reason behind this criterion is removing applications using deprecated version of Python or ML frameworks.
- **bugs irrelevant to the identified ML frameworks (*TensorFlow* and *Keras*).** Some of the collected buggy applications use ML frameworks other than our favorable (e.g., *Caffe*). They are collected because their repositories have had "tensorflow" or "keras" keywords in their description. So, we remove them using this criterion.
- **bugs without dependency information.** One of the most significant needed information to reproduce bugs is their dependencies. We applied this criterion to filter out bugs that are not reproducible.
- **bugs without required data or description for achieving it.** Used data while facing the bug is another key requirement for reproducing ML-based systems' bugs. We establish this criterion to delete irreproducible bugs.
- **bugs which are not related to the ML.** With respect to our primary aim to investigate characteristics of ML-related bugs and provide faultload benchmark of ML-based systems, we use this criterion to exclude bugs that their root cause is not ML.
- **bugs with description in any language other than English.** Because we use commit messages as material to inform the users about the detailed information of bug and the bug-fix solution, we deleted the commits that use other languages than English for their commit messages.

3.1 Collecting Bugs from Public Datasets

Zhang et al. (2018b) carried out a study on the bugs in *TensorFlow* programs to identify the root causes and symptoms of various types of bugs. They provided a public dataset of 88 and 87 bugs studied in their paper, extracted from GitHub and SO, respectively. After applying exclusion criteria, we obtained 9 GitHub related bugs to be added to the benchmark. But none of their reported bugs from SO remains after applying exclusion criteria. Islam et al. (2020) provided a public dataset along with their research on the bug fix patterns in DNN programs. DNN program refers to the DL model and training algorithm, where they investigated for bug fix patterns. Their public dataset contains 347 and 320 bugs from GitHub and SO, respectively. By filtering bugs using the exclusion criteria, we obtained 8

Table 2 Detailed information regarding the collected bugs from public datasets

Dataset	Num of collected bugs	
	GitHub	SO
Zhang et al. (2018b)	9	0
Islam et al. (2020)	8	3
Wardat et al. (2021)	4	14
Humbatova et al. (2020)	0	0
Nikanjam et al. (2021a)	8	10
Total	***29***	***27***

Bold italics mean final results or total of some sub-categories

reproducible *GitHub* related bugs and 3 SO related. Wardat et al. (2021) provided a benchmark of bugs in DNN programs using *Keras*. They reported 11 and 29 bugs from GitHub and SO, respectively. After applying our refinement process, we obtained 4 *GitHub* related and 14 SO related bugs to add to the benchmark. Humbatova et al. (2020) studied different types of bugs in DL programs and proposed a taxonomy on the identified bugs. They also published the dataset of their studied bugs containing 60 bugs from GitHub and 109 from SO. Our filters eliminated all their mentioned bugs and we could not achieve any bug from their dataset. Nikanjam et al. (2021a) delivered a public dataset with their automatic bug detection tool including 34 real bugs in DL programs, 26 from SO and 8 from GitHub. After checking their provided bugs using mentioned exclusion criteria, we added 8 of *GitHub* bugs and 10 SO ones to the benchmark.

Table 2 represents the number of bugs that we added to *defect4ML* from public datasets of bugs. It is also worth noting to mention that some SO bugs are repetead in different public datasets. As an example, post (2016) exists in three public datasets (Nikanjam et al. 2021a; Wardat et al. 2021; Islam et al. 2020).

3.2 Collecting Bugs from GitHub

GitHub[7] is considered the most significant resource of open source software repositories in the computer programming community. As of September 2020, GitHub hosts more than 56 million users and about 190 million software repositories (GitHub 2021) including more than 28 million public repositories. GitHub provides API to simplify the data extraction process that allows developers to create their own requests and extract preferred data. We also used GitHub rest API v3 (2021) to gather repositories.

3.2.1 Selection of ML-based Systems Repositories

To collect the ML-based systems' repositories, we used GitHub search API. Firstly, we limited the results to the repositories that use Python programming language which is defined as *"Python"* and *"Jupyter Notebook"* programming languages in GitHub. *Python* is the most popular programming language for ML (Voskoglou 2017; Gupta 2021). On the other hand, *Keras* and *TensorFlow* also provide *Python* APIs. As it is mentioned, we analyzed

[7] https://github.com/

ML-based systems that use *TensorFlow* and/or *Keras*. We extracted the repositories using each of *TensorFlow* and *Keras* separately. To this end, we used “tensorflow” and “keras” keywords to extract the repositories using these ML frameworks. In the next step, we limited the repositories to the ones with at least one push after 2019. This criterion is to decrease the possibility of reviewing repositories using the old versions of ML frameworks or *Python*, which may not be beneficial to add to the benchmark. At the same time, since it does not prevent the inclusion of repositories using old versions of *Python* or ML framework, we also filter those repositories during our manual inspection. Furthermore, repositories that are forked or defined as “disabled” are excluded in the search query.

GitHub search API limits the users to access just the first 1000 results. So, we divided the whole duration of search for push command from Jan 1, 2019 to Aug 30, 2021 (the time we run the queries) into snapshots of 5 days to restrict the number of results to less than 1000. That is to say, we raised 192 search requests for each ML framework to extract repositories. We collected 30,387 and 51,151 repositories that use *Keras* and *TensorFlow*, respectively. In the filtering process, we did not filter the repositories based on the popularity criteria (e.g., number of stars, number of commits, etc.) to keep more diversity in the collected bugs. In other words, we aimed to collect as much as bugs from developers with various expertise levels to avoid generating the benchmark using a biased set of bugs.

3.2.2 Selection of Bug-fix Commits

To extract bug-fix commits from the collected repositories, we searched commits’ messages for a list of bug-related keywords (*bug, fail, crash, fix, resolve, failure, broken, break, error, hang, problem, overflow, issue, stop, etc.*), which are used successfully in the literature (Abidi et al. 2019a, b, 2021). We also used PyDriller (Spadini et al. 2018), a python library to mine the GitHub repositories, to collect bug-fix commits. In this step, we collected 157,190 bug-fix commits from repositories using *TensorFlow* and 98,562 from the ones using *Keras*. To exclude bug-fix commits which are irrelevant to ML, we performed another filtering step based on the approach used successfully in Humbatova et al. (2020). We searched a list of keywords that are related to the various bug types in ML components (e.g., *optimize, loss, layer*, etc.) in the commits’ messages and exclude ones with none of those keywords. Thus, we reached 38,463 and 26,326 bug-fix commits for *TensorFlow* and *Keras*, respectively.

Afterwards, we used sampling with 95% for confidence level and 5% for confidence interval that gives us 380 bug-fix commit for repositories using *TensorFlow* and 379 for *Keras*. In the next step, we manually checked all of the gathered bug-fix commits, applied the exclusion rules, and removed the inappropriate ones. To achieve exclusion criteria, the first author reviewed 200 bug-fix commits and shared the results with the second and third authors (all with more than 2 years of experience in engineering ML-based systems). After three meetings, we achieved an agreement on the following exclusion criteria:

- Bug-fix commits with messages written in languages other than English.
- Bug-fix commits that do not demonstrate the problem clearly.
- Bug-fix commits which used ML frameworks other than *TensorFlow* or *Keras*.

Besides, based on the changed LOC and manipulated APIs by the commit, we consider the bug-fix commits that can be categorized as one of the most recent taxonomy of DL bugs (Humbatova et al. 2020). To identify ML-related bug-fix commits, the first two authors separately checked the 100 randomly selected bug-fix commits, 50 from repositories

Table 3 Detailed information about the number of remaining bug-fix commits after each filtering step

	ML frameworks	
	TensorFlow	*Keras*
All extracted ML-based repos	51151	30387
Bug-fix commits	157190	98562
ML-related bug-fix commits	38463	26326
Sampled bug-fix commits	380	379
Added bugs to the benchmark	**17**	**29**

Bold italics mean final results or total of some sub-categories

developed using TensorFlow and 50 by Keras, attaining 54.7% agreement using Cohen's Kappa (McDonald et al. 2019). So, we had two meetings to recognize the main reasons for disagreements and resolve them. Next, we again checked the 100 reviewed bug-fix commits achieving 89.6% agreement based on Cohen's kappa which is considered as almost perfect agreement (McHugh 2012). Afterward, the first author reviewed the rest of randomly selected bug-fix commits. Then, the second author checked those bug-fix commits that gained a level of agreement of 86.4%. Concerning most of the reviewed repositories do not provide complete information of their dependencies, we tried to find the match version of the used ML framework with the date of bug-fix commit. For instance, it is obvious that commits which are done before March 5, 2018 could not use *Keras* higher than version 2.1.4, because the release date of *Keras* 2.1.5 is March 6, 2018 (Keras 2016). Afterward, we attempted to find out the Python version and complete list of required libraries and their version, matching the version of used ML framework. Finally, we achieved 38 bug-fix commits which meet benchmark requirements. Then, we continued checked manually 505 other bug-fix commits from repositories using *TensorFlow* and/or *Keras* randomly to attain our goal. Table 3 represents the number of bugs that we added to the *defect4ML* from GitHub.

With respect to the need for both buggy and fixed versions of the application for each identified bug in our proposed benchmark, we have provided a snapshot of the application's repository exactly before fixing the bug. To this end, we use `git log -p <fileName>` command and extract the commit prior to the bug-fix (`fileName` refers to the buggy file that bug-fix commit will change). Using that commit, we can gather the version of repository exactly before fixing reported bug.

3.3 Extracting Bugs from Stack Overflow

Stack Overflow[8] (SO) is taken into account as the largest Q&A platform for software developers, with over 21 million questions and 14 million registered users on March 2021 StackOverflow (2021). To collect intended posts, we used Stack Exchange Data Explore[9] platform, where one can gather information regarding SO posts using SQL queries. Like for data extraction from GitHub, we used some criteria for filtering SO posts and collecting relevant ones. In the first step, we collected posts that have "tensorflow" or "keras" as post tags. So, we gathered 50,001 and 37,887 question regarding *TensorFlow* and *Keras*, respectively. Besides, 21,908 posts have both "tensorflow" and "keras" tags at the same time. We considered all of them as posts related to *Keras*. Then, we filtered out

[8] https://stackoverflow.com/

[9] https://data.stackexchange.com/stackoverflow/query/new

Table 4 Detailed information about the number of remaining posts after each filtering out step

	ML frameworks	
	TensorFlow	*Keras*
All extracted posts	50001	37887
Posts with accepted answer	18821	14590
Sampled posts	376	374
Added bugs to the benchmark	3	7

the posts without an accepted answer where one can not make sure of fixing the issue, in the SO posts without accepted answer. So, we reached 18,812 posts regarding *TensorFlow* and 14,590 about *Keras*. Afterward, we used sampling with 95% and 5% for confidence level and confidence interval, respectively. Thus, we attained 376 posts related to *TensorFlow* and 374 for *Keras*. It is worth noting that instead of selecting the posts randomly, we selected the posts with the highest scores. Next, we manually checked all of the collected bugs' root cause to keep ones related to the ML and remove irrelevant ones. Similar to the GitHub manual checking step, we used some exclusion criteria to filter out the irrelevant SO posts. To obtain exclusion criteria, The first author analyzed 100 SO posts, 50 related to the *TensorFlow* and 50 regarding *Keras* with the highest score and discussed the results with the second author. After two meetings, we reached the following exclusion criteria:

- posts which mentioned conceptual questions about ML/DL components (such as (2018)).
- posts related to the users questions on developing ML/DL components, not resolving a bug (such as (2017)).
- posts that their root causes were not ML and they were just the typical programming mistakes (such as (2018)).
- posts that do not include the required script to reproduce the bug.
- posts without mentioning the employed dataset or a clear description about it (such as (2017)).

In the next step, the first author checked the 50 posts with the highest score for each ML framework (a total of 100) to identify relevant ones. In the next step, the second author reviewed them which obtained 89.3% agreement using Cohen's kappa. Then, the first author labeled the rest of the SO posts and the second author checked them, gaining 87.4% agreement.

From the availability of the required data viewpoint, we can categorize the SO posts into three main categories. First, the posts that mention popular datasets such as MNIST (LeCun et al. 1998) or CIFAR-10 (Krizhevsky et al. 2009) (such as (2019)). We utilized *Keras* datasets[10] to reproduce bugs belonging to this group. Second, posts that provide the link to achieve the needed data to reproduce the bugs (such as (2018)). Third, posts that did not give any description about the required data (such as (2018)). To address the required data problem of the posts that belong to this group, we tried to reproduce the bug using popular datasets. In case of inability to reproduce the reported bug with the same root cause or symptom, we excluded the post.

[10] https://keras.io/api/datasets/

By manual investigating 376 sampled bugs related to *TensorFlow* and 374 regarding *Keras* (750 in total), we achieved 10 reproducible bugs (3 *TensorFlow* and 5 *Keras*) to add to the *defect4ML*. Table 4 depicts the number of bugs that we added to *defect4ML* from SO posts. We continued manual checking for 48 more randomly selected SO posts regarding *Keras* that concluded 2 bugs to be added to *defect4ML*.

3.4 Labeling Collected Bugs

To categorize the collected bugs, we used three kinds of labels, firstly based on violated testing property (Zhang et al. 2020), secondly bug type, and finally according to symptoms of bugs (Islam et al. 2019). Bug type refers to the class of ML bug taxonomy (Humbatova et al. 2020) to which the reported bug belongs. For the first step, the first two authors held a meeting to discuss ML testing properties and achieve common understanding on each ML testing property. Then, the first two authors labeled the first 25% bugs reaching 85.7% agreement based on Cohen's kappa (McDonald et al. 2019) which is interpreted as almost perfect agreement (McHugh 2012) and allows us to continue labeling the rest of bugs. To assign ML testing property to each bug, we studied commit message or SO post message to understand the property to which bug-fix aims. For instance, a GitHub commit (2018) which is trying to improve the performance of the ML model is categorized as efficiency property. As another example, GitHub (2018) that resolves the problem in the model structure is labeled as correctness. For labeling the rest of bugs, we labeled them in three parts (25% of bugs in each part) and held a meeting after each part to identify the major reasons of disagreements and resolve them. In the case of disagreements between two raters, they discussed disagreements with the third author which result in labeling all bugs consistently, which is used in previous studies successfully (Shen et al. 2021). Finally, we achieved 88.6% agreement using Cohen's kappa after labeling all bugs.

For the second labeling step, the first two authors labeled the 25% bugs separately achieving a 58% agreement based on Cohen's kappa (McDonald et al. 2019), which is known as a moderate agreement (McHugh 2012). Hence, to improve their understanding of bug types, they had a meeting to have a clear knowledge about each label, identify the main reasons of disagreements, and build a common understanding of bug types. Afterward, raters labeled the first 25 bugs again resulting in 87% agreement, implying an almost perfect agreement between them. The rater continues labeling the rest of bugs with this approach in three parts (every 25% of the bugs in each part). After labeling bugs in each part, we had a meeting to recognize the main reasons behind the disagreements and resolve them. Finally, we achieved 88.7% agreement by labeling all bugs, using Cohen's kappa. For the remaining disagreements, we used methodology mentioned for labeling bugs based on the violated properties.

To label the bugs according to their symptoms, firstly we had a meeting to achieve an obvious understanding about symptom of ML-related bugs. Next, the first two authors labeled 25% of the bugs gaining 88.5% agreement that is interpreted as almost perfect agreement (McHugh 2012). Then, raters labeled the rest of bugs in three parts (similar to two prior steps) reaching 91.4% agreement using Cohen's kappa (McDonald et al. 2019). To achieve consistent labels for all bugs, raters discussed the disagreements with the third author and resolved them (same as previous steps).

4 Results

Generally, we manually checked 513 and 498 reported ML-related bugs extracted fron GitHub and SO, and provided in previous articles. Also, we collected 64,789 bug fix commits from ML-based systems repositories using *TensorFlow* and/or *Keras*. We selected 1264 out of them randomly and manually checked them as well for satisfaction of standard benchmark criteria. Moreover, we manually inspected 798 SO posts related to *TensorFlow* and/or *Keras*. In the following, we present our answers to each of our formulated research questions.

4.1 RQ1. Key Factors in Reproducibility of ML-Related Bugs

Bug reproducibility plays a key role in this study, because it is taken into account as a substantial challenge in SRE of ML-based systems on the one hand (Zhang et al. 2018b) and in the benchmark generating on the other hand (Kistowski et al. 2015). To make sure of reproducibility of the collected bugs, we did several manual checking steps. Firstly, we checked the buggy application for needed dependencies information to run it without dependency issues. In the next step, we looked into needed data for running the buggy application and triggering the reported bug. After addressing dependency and data requirements of each bug, we faced some new challenges while trying to run the buggy applications. Regarding the extracted ML-related bugs from GitHub, we have found several bugs with compilation errors while they are running (such as (2017)). Besides, some of the bugs were not triggered using the mentioned condition in bug-fix commit (such as (2018)). We coped with the reproducibility problem of 62 out of 1777 (513 from public datasets and 1264 from GitHub) reviewed ML-related bugs extracted from GitHub.

About the reported bugs in SO, almost all of the scripts mentioned in the posts are just code snippets, not a complete code of the ML component, as it is popular in SO. Therefore, we had to complete them by the default configuration and then check to ensure that the considered bug would occur. In some cases, we failed to use the bug after spending considerable time. Moreover, most of the SO posts do not include the required dependency and data information to reproduce the reported bugs. We could reproduce 38 out of 1296 (498 from public datasets and 798 from SO) SO posts.

Finding 1: The most influential factors in reproducing ML-related bugs are required dependencies (including used libraries and their exact version, ML framework version, and Python version) and data to run the buggy application. Besides, only near 3.48% and 2.93% of all manually inspected ML-related bugs extracted from GitHub and SO can be reproduced.

4.2 RQ2. The Important Factors in Verifiability of Fixing ML-related Bugs

Software verification refers to the process of checking software to ensure that it achieves defined goals without any bug (2017). To verify the fixing of reported bugs, we need to ensure that the mentioned problem will be resolved after applying the fix. Software systems use software testing methods to verify their goals and objectives (Lyu 2007). We investigated all reproducible ML-related bugs for the test cases provided by buggy applications that check the verification of the software component consisting of reported bugs. We found 10 out of 75 buggy applications extracted from GitHub that can be verified using provided test cases by the applications. In case the buggy application does not give any test cases, we have to trust the bug-fix commit message to verify the bug fixes, like almost all of the previous articles studying ML-related bugs and their fixes (Humbatova et al. 2020; Zhang et al. 2018b; Islam et al. 2020).

Verifiability of fixing reported bugs in SO posts is a more severe challenge. Most of the scripts mentioned in SO posts are code snippets and do not have additional information like test cases. So, we could not verify any of the reproducible ML-related bugs extracted from SO using provided test cases by developers who ask the question. Instead, we used the accepted answer flag to verify bug fixes.

Finding 2: Lack of test cases and immaturity of providing test cases for ML-related bugs are the most influential factors for verifying fix of ML-related bugs. Moreover, just near 13.3% of ML-related bugs gathered from GitHub can be verified using provided test cases by their applications. However, none of the reproducible ML-related bugs from SO could be verified by test cases implemented by the user who created the post.

4.3 RQ3. Providing a Standard Faultload Benchmark for ML-Based Systems

Overall, we collected 100 bugs for *defect4ML*, 62 from GitHub, and 38 from SO. Table 5 shows the detailed information of collected bugs. We will explain the challenges we faced while generating the *defect4ML* and its merits in comparison with others in the upcoming subsections. Figure 4 also depicts the the distribution of bugs, based on their symptoms.

4.3.1 Satisfied Criteria of Standard Benchmark

We consider all criteria of standard faultload benchmark as the challenges of generating standard bug benchmark in ML-based systems. We will describe the methodologies we used to satisfy each criterion in the upcoming subsections.

Relevance To meet the relevance criterion, we just include ML-related bugs in *defect4ML*. Because there are well-suited benchmarks for traditional programming bugs (non ML-related bugs) such as Widyasari et al. (2020), Madeiral et al. (2019), Le Goues et al. (2015), Lu et al. (2005), Just et al. (2014), and Lin et al. (2015), we exclude any bugs that are not related to ML. For example, commit (2019) which is related to the fix in *README* file has been excluded from *defect4ML.*

Reproducibility To fulfill reproducibility, we have provided accurate information required for reproducing bugs. The information consists of complete list of dependencies (all needed libraries and their exact version including ML framework), Python version compatible with ML framework and other libraries that application uses, used dataset while facing the reported bug, and the instruction to activate the bug.

Table 5 Detailed information of bugs

Source	Framework	Bug category	Bug type	#bugs	Total
GitHub	TensorFlow	API	Deprecated API	2	**6**
			Missing variable initialization	1	
			Wrong API usage	1	
		Model	Missing softmax layer	1	**4**
			Wrong network architecture	1	
			Wrong type of activation function	1	
			Wrong weights initialisation	1	
		Tensors & inputs	Tensor shape mismatch	1	**2**
			Wrong shape of input data	1	
		Training	Redundant data augmentation	1	**8**
			Suboptimal learning rate	2	
			Wrong loss function calculation	4	
			Wrong selection of loss function	1	
	Keras	API	Deprecated API	3	**8**
			Missing API call	2	
			Missing argument scoping	1	
			Wrong API usage	2	
		Model	Missing dense layer	1	**14**
			Suboptimal network structure	4	
			Wrong filter size for convolutional layer	1	
			Wrong layer type	2	
			Wrong network architecture	3	
			Wrong type of activation function	3	
		Tensors & inputs	Wrong tensor shape	1	**1**
		Training	Missing preprocessing step	1	**19**
			Suboptimal batch size	4	
			Suboptimal number of epochs	4	
			Wrong loss function calculation	1	
			Wrong optimisation function	4	
			Wrong selection of loss function	5	
SO	TensorFlow	API	Deprecated API	2	**2**
		Tensors & inputs	Wrong input format	1	**1**
	Keras	API	Wrong API usage	4	**5**
			Deprecated API	1	
		Model	Suboptimal network structure	1	**11**
			Missing Flatten layer	2	
			Wrong type of activation function	8	
		Training	Suboptimal learning rate	3	**9**
			Wrong loss function	2	
			Missing preprocessing	4	
		Tensors & inputs	Tensor shape mismatch	2	**10**
			Wrong type of input data	1	
			Wrong shape of input data	6	
			Wrong tensor shape	1	
Total					*100*

Bold italics mean final results or total of some sub-categories

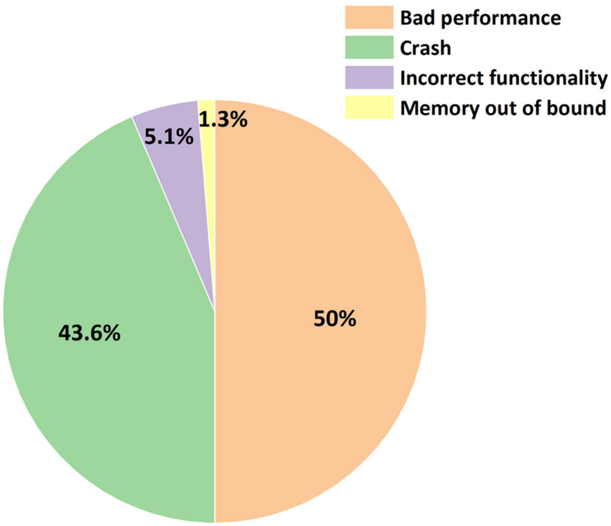


Fig. 4 Distribution of bugs' symptoms in defect4ML

Fairness We have applied three main measures to meet the fairness and prevent generating artificial limitations for *defect4ML*. Firstly, we have equipped the benchmark with two of the most popular ML frameworks (*TensorFlow* and *Keras* (Yalçın 2021; Humbatova et al. 2020)). Secondly, we have used the taxonomy of bugs in DL programs introduced in Humbatova et al. (2020) and tried to label the benchmark bugs based on the leaves of that taxonomy. We have covered 30 types of bugs mentioned in this taxonomy. So, the benchmark can satisfy various users' requirements who focus on specific types of bugs. We have also provided different types of bugs using various versions of ML frameworks and Python.

Verifiability We have implicitly satisfied this criterion because all collected bugs are real and discussed in GitHub projects or SO posts. The benchmark also provides a link to the bug's origin (GitHub bug-fix commit or SO post) for all bugs. Bug-fix commit messages and SO posts represent detailed information about the occurrence of the bug and the solution to fix it. The benchmark delivers two versions of the application where the bug has occurred, i.e., the *buggy* and *fixed* (or clean) versions. The *Buggy* version indicates the version of the application before fixing the bug. The *Fixed* version refers to the application after fixing the identified bug. Regarding the verifiability of fixing bugs collected for *defect4ML*, 10 out of 62 reproducible ML-related bugs gathered from GitHub are verifiable by the provided test cases and the rest based on the bug-fix commit massage. We also used accepted answer flag for SO posts that prove the verifiability of the solution provided to resolve the reported bug.

Usability We aim to generate an understandable benchmark by delivering information with each bug, such as violated testing property, bug's type, and a link to the bug's origin.

Users can achieve detailed information regarding the bug's root cause, symptoms, and possible fixing methods using the mentioned link. We have also provided several categorizations (bug's origin, Python version, ML framework, violated testing property, and bug's type) to prepare a set of bugs that fit users' aims. Users may use *defect4ML* for different goals, such as assessing the ability of testing tools that focus on bug detection, repair, and localization.

4.3.2 Addressed SRE Challenges in ML-based Systems

Several new challenges in the SRE of ML-based systems make it more complicated compared to the traditional software systems (Islam et al. 2020; Wardat et al. 2021). Neglecting these challenges may deteriorate the satisfaction level of relevance, reproducibility, verifiability, and usability of ML-based systems. We rise to these challenges in *defect4ML* to ensure the relevance of our proposed benchmark. In the following subsections we elaborate on the methods used to handle each challenge.

Fast Changes in ML Frameworks To handle the challenge of fast changes in the ML framework, users need to have the exact information regarding the version of the used ML framework in the application containing the bug. We have presented this information for current bugs in *defect4ML*.[11] Also, *defect4ML* has been equipped with bugs that appeared in different versions of each ML framework.

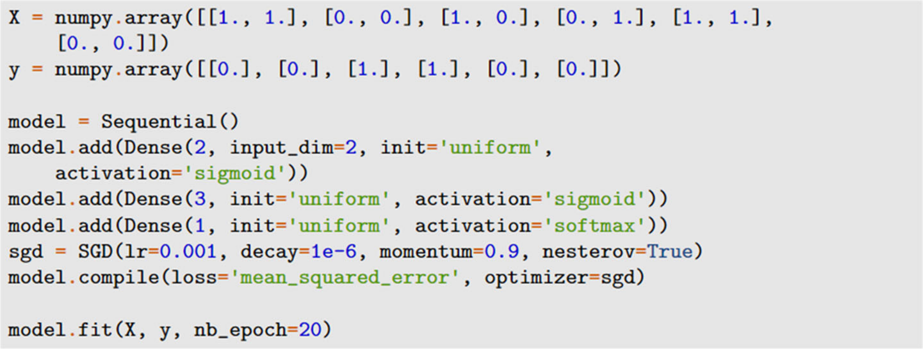
Code Portability To tackle the code portability challenge, defect4ML has provided different bugs that occurred in the applications developed using the two most popular ML frameworks: *TensorFlow* and *Keras* (Yalçın 2021; Humbatova et al. 2020). Therefore, users have access to a list of bugs in their preferred ML framework, without requiring porting bugs from one ML framework to another one. Users can also enhance the benchmark by adding bugs from ML frameworks other than those studied in this paper or recreating bugs in new ML frameworks.

Bug Reproducibility Bug reproducibility is known as one of the most critical challenges in all SRE areas. But it is a more severe challenge in ML-based systems, because of direct effect on the other challenges of SRE in ML-based systems such as fast changes of ML frameworks and code portability. This difficulty may result from 1) a high amount of operational changes in versions of the ML frameworks (Islam et al. 2020), and 2) different dependencies specified for every single version of ML frameworks (Zhang et al. 2018b). To meet this challenge, we have delivered a complete list of dependencies (and corresponding versions) needed to run the application.[11] We have also presented specific configuration of each bug including Python version, and process to trigger the bug.[12] Moreover, because of the substantial effect of data in ML components operations (Zhang et al. 2020; Felderer and Ramler 2021), we have delivered the required training/testing datasets to reproduce the bugs.

Lack of Detailed Information about the Bugs When an ML component faces a bug, the compiler mostly gives no detailed information regarding the root cause of the bug or the exact bug location. For example, the compiler may not recognize the ML model structural bugs, or provides just an error message to inform the developer about the existence of a problem in the ML model structure. But it does not give any clue (such as bug location) to the developer for debugging the ML model. Figure 5 shows a sample script trying to implement a classifier for XOR problem using *Keras* (bug #84 of *defect4ML*) (2016). Although loss remains constant during training (because of wrong activation function in the last dense layer), compiler does not give any information about the problem to the user. To cope with

[11] In *requirements.txt* file, that consists of detailed information of dependencies per bug.

[12] In *conf.ini* file, that contains the required configuration per bug.



```
X = numpy.array([[1., 1.], [0., 0.], [1., 0.], [0., 1.], [1., 1.],
    [0., 0.]])
y = numpy.array([[0.], [0.], [1.], [1.], [0.], [0.]])

model = Sequential()
model.add(Dense(2, input_dim=2, init='uniform',
    activation='sigmoid'))
model.add(Dense(3, init='uniform', activation='sigmoid'))
model.add(Dense(1, init='uniform', activation='softmax'))
sgd = SGD(lr=0.001, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)

model.fit(X, y, nb_epoch=20)
```

Fig. 5 Sample implementation of XOR classification problem using *Keras* (2016)

this challenge, we use GitHub bug-fix commit messages and SO posts description that give detailed bugs specifications and the possible solutions to fix them. Bug-fix commit messages may also include the link to the raised issue in the issue tracker, which has in-depth information about the bug.

4.3.3 Artifacts

Each bug consists of several components to meet the prerequisites of the benchmark. These components include:

- ***Buggy and fixed versions of the application:*** each bug has two different versions of the application containing that bug. *Buggy* version is the application including the bug and is generally used to evaluate ML-based systems testing tools' ability to detect the bugs. *Fixed* version is the same application just after fixing the bug and is mainly used to assess the repairing tools. Repairing tools try to detect the bugs and provide another version of the application where the identified bug has been fixed based on the best practices.
- ***Dependencies (Required libraries):*** each bug comes with a complete list of dependencies required to run the buggy version (e.g., ML framework and its version, and needed libraries and their versions).
- ***Data:*** All bugs are equipped with the needed data to trigger the reported bug.
- ***Configuration:*** that is used to produce bug categorization on the benchmark. Moreover, the process of running the application to trigger the bug is mentioned in the configuration.
- ***Test case:*** Each bug has its own test case which can execute the buggy application and trigger the bug, without requiring any user's manipulation. It has been exposed that providing test cases to discover bugs in ML-based systems is more effective than using assertion inside the application (Jia et al. 2021a). Besides, the test case enables users to observe the effect of the bug and compare the behavior of the system in buggy and fixed versions. Because the result of an ML application could be different for each run, even with the same hyperparameters and dataset, providing assertion on exact values can be ineffective for ML application, as shown by Nejadgholi and Yang (2019). Hence, we use a range for assertion of provided test cases in defect4ML. Besides, according to the previous studies showing that accuracy of fixed and buggy version will be close to each

other, in case the buggy version does not lead to crash/hang (Jia et al. 2022), we use the average result of running buggy and fixed versions 10 times each to address challenges of determining threshold in the provided assertions.

- ***Detailed description:*** each bug has a detailed description including its root cause, symptom, and an explanation that represents the situation triggering the bug to ease understanding of bugs.

Users who want to use *defect4ML* bugs should be informed that we provide test case assertions on that part of results mentioned as symptoms of bug in GitHub issues/SO posts. For example, in bug #92, the user obviously asked for a solution to resolve the problem of low accuracy. Thus, we provide assertions on the accuracy of the buggy and fixed versions. As reported in Wardat et al. (2022), diagnosing an ML bug may require running the code, collecting information on training and validation phases, and monitoring various values. We are aware that relying on model accuracy to diagnose a bug in ML applications may not always be precise (Pham et al. 2021), because of the non-deterministic nature of ML applications leading to different results/accuracy in various executions with the same hyperparameters and dataset. To this end, in cases where a bug is detected based on its impact on the model accuracy, we run each version of an ML application multiple times (mostly 10 times) in the test cases and use the averaged accuracy achieved in multiple executions.

We have classified bugs based on different criteria in *defect4ML*. The first criterion is testing properties of ML-based systems (Zhang et al. 2020) which includes correctness, model relevance, robustness, security, efficiency, fairness, interpretability, and privacy. This criterion refers to the conditions that should be guaranteed for the trained model.

Correctness refers to the ability of a trained model to predict unseen future data correctly (Zhang et al. 2020). That is, when an ML model is not designed and/or trained optimally, it can manifest low accuracy during test or deployment. Thus, to meet correctness which requires model accuracy improvement, developers should revise the designed ML model based on recommendations of ML experts. As an example, in the code shown in Fig. 5, because of using the wrong activation function (softmax) in the last dense layer, model accuracy stays near 66%. In fact, the softmax activation function is helpful, where the number of target classes is more than 2. Thus, removing this activation function resolves the problem and increases the model accuracy. All of the currently presented bugs in *defect4ML* fall into the correctness category.

Model relevance checks for a proper match between the design of the model and training data (Kirk 2014). In other words, model relevance asserts that a designed ML model should not be more complicated than what is required. Model overfitting usually leads to low model relevance (Zhang et al. 2019). Providing a neural network with unnecessary hidden layers may cause model overfitting, and then low model relevance. For example, when a model is more complex and has more learning capacity than needed, it may fit noises of training data resulting in contaminating model generalization (Hawkins 2004). Thus, decreasing model learning capacity (adding dropout layer, weight decay, etc) would improve model generalization and accordingly, model relevance.

Robustness is defined as the degree to which an ML system can handle any perturbation on ML components (e.g. data) (IEEE 1990; Zhang et al. 2020). A trained ML model should be able to handle small perturbation on data, like adversarial samples. As an example, studies showed that existence of adversarial examples in safety-critical systems (such as autonomous vehicles) may lead to significant improvement in the robustness of the system (Tian et al. 2018; Zhang et al. 2018a).

Security refers to the resiliency of a ML system against harmful or dangerous actions by illegal access or manipulation of ML components (Xue et al. 2020). Systems with low accuracy may deal with data poisoning if perturbed data is employed as training data. A security attack may mislead a trained model or lead a model to be trained badly by manipulating training data (Liu et al. 2020). For example, accessing "stop sign" training data of an autonomous vehicle and manipulating it to decrease detection performance of the model may lead to a catastrophe (Zhu et al. 2019).

Efficiency measures consumed computational resources for training or inferring processes in the ML system (Zhang et al. 2020). Overall, an ML system suffering from suboptimal model structure may need more training time compared to the optimal model structure. For example, training an ML model aiming at decreasing loss may be faced with stopping loss decrement after some training loops (Brownlee 2020). Afterwards, continuing training may be considered a waste of resources. Hence, ML developers use early stopping to monitor evaluation metrics and cease training, whenever training no longer improves evaluation metrics (Rice et al. 2020).

Fairness aims at preventing ML decisions to suffer from ethical issues (e.g. human rights, discrimination, etc.) (Chouldechova and Roth 2018). In general, human beings have a bias in labeling or collecting data (Barocas and Selbst 2016). Fairness ensures that ML decisions are in the right way and free of bias. Unfair models may produce discrimination, where they do not work for some subpopulations. An example of an unfair model can be a medical image processing model that works inaccurate, except for white males.

Interpretability refers to the degree that reasons behind decisions made by ML models can be understandable by human beings (Lipton 2018). As an example, an ML model with high interpretability used for medical treatment decisions may be trusted more by medical experts. Last but not least, privacy aims at protecting private information that can be used as training data (Dwork 2008; Zhang et al. 2020). For example, data used for an ML model that plays a role of assistant in medical treatment decisions should preserve privacy of patients information.

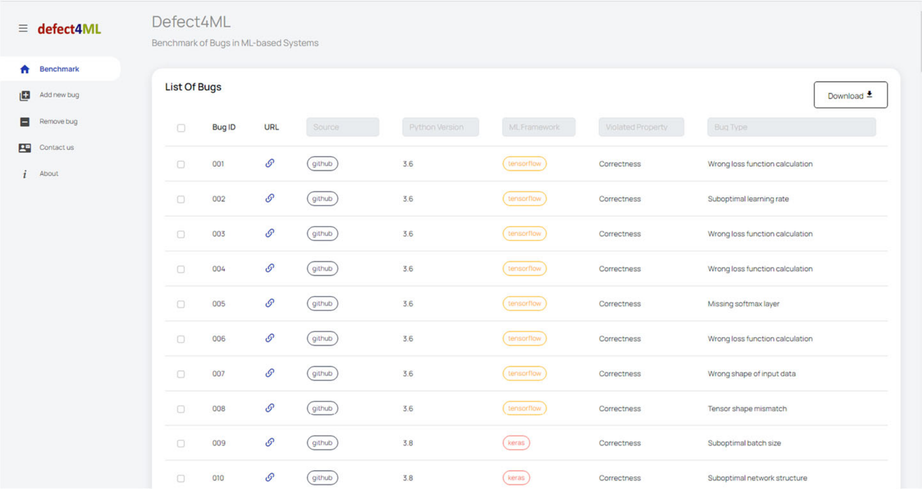
Another filtering criterion is the ML framework used to implement the buggy ML component. To indicate the types of bugs, we used the taxonomy of DL bugs proposed by Humbatova et al. (2020).

4.3.4 Provided API

In order to deliver an easy-to-use benchmark, we provide a web application endpoint for *defect4ML* (accessible via http://defect4aitesting.soccerlab.polymtl.ca). Figure 6 represents a screenshot of the *defect4ML* web application. Users can browse the bugs, and filter them based on different criteria to attain a list of bugs that is suitable for their goals. Since *defect4ML* is in the process of expansion, we have provided the possibility for the users to add new bugs to the benchmark or raise a request for removing the existing ones. Users can submit new bugs by providing an artifact of the bug. Detailed explanation of adding new bugs has been presented in benchmark web application page.

5 Discussion

This section presents an example application of *defect4ML* as a case study: comparing two ML-based systems testing tools: *NeuraLint* and *DeepLocalize*.



Defect4ML

Benchmark of Bugs in ML-based Systems

List Of Bugs

□	Bug ID	URL	Source	Python Version	ML Framework	Violated Property	Bug Type
□	001	🔗	github	3.6	tensorflow	Correctness	Wrong loss function calculation
□	002	🔗	github	3.6	tensorflow	Correctness	Suboptimal learning rate
□	003	🔗	github	3.6	tensorflow	Correctness	Wrong loss function calculation
□	004	🔗	github	3.6	tensorflow	Correctness	Wrong loss function calculation
□	005	🔗	github	3.6	tensorflow	Correctness	Missing softmax layer
□	006	🔗	github	3.6	tensorflow	Correctness	Wrong loss function calculation
□	007	🔗	github	3.6	tensorflow	Correctness	Wrong shape of input data
□	008	🔗	github	3.6	tensorflow	Correctness	Tensor shape mismatch
□	009	🔗	github	3.8	keras	Correctness	Suboptimal batch size
□	010	🔗	github	3.8	keras	Correctness	Suboptimal network structure

Fig. 6 A screenshot of *defect4ML* web application

5.1 Benchmark Applications

According to the characteristics of the *defect4ML*, it can be beneficial to studies on bugs in ML-based systems. Developers of ML-based systems testing tools can use *defect4ML* to show the advantages of their proposed tools and techniques compared to the existing ones. The researchers can also use our proposed benchmark to evaluate existing ML-based systems testing tools and clarify the most critical challenges that should receive attention from new studies. We provide a case study of using *defect4ML* to evaluate ML-based systems testing tools. So, the primary goals of this case study are the assessment of ML-based systems testing tools and comparing them.

In this case study, we compare two testing tools for ML-based systems. We selected two up-to-date testing tools published recently: *NeuraLint* (Nikanjam et al. 2021a) and *DeepLocalize* (Wardat et al. 2021). *NeuraLint* is a model-based automatic fault detection tools for DL programs. The authors proposed a meta-model for DL programs that contains their basic properties. To detect the bugs, *NeuraLint* first extracts the graph of the DL program from its code. In the next step, it identifies the bugs using graph transformations that represent detection rules. It is worth noting that *NeuraLint* is based on static analysis of DL programs meaning that it does not need to run the DL program to identify the bugs.

DeepLocalize is another testing tool that can analyze DL programs, detect the bugs, and localize them automatically. It provides a customized callback function for *Keras* that collects DNN detailed information during the training process. In other words, *DeepLocalize* analyses the DNN training traces to identify the possible bugs and their root causes (e.g., faulty layers or hyperparameters). Unlike *NeuraLint* that analyzes the DL programs statically, *DeepLocalize* uses dynamic analysis. That is, DL programs should be executable without any compilation error to be analyzable by *DeepLocalize*.

Concerning the fact that *NeuraLint* can analyze the DL programs that have been written in one file, we had a limited number of bugs to use. On the other hand, due to the existence of compile-time errors in the buggy version of some bugs, they are not usable for *DeepLocalize*.

Table 6 Results of studied tools as case study

Bug ID	Source	Framework	Violated property	Bug type	NeuraLint result	DeepLocalize result
25	GitHub	Keras	Correctness	Wrong network architecture	No identified error	batch 4 layer 9 : error in forward
26	GitHub	Keras	Correctness	Wrong type of activation function	No identified error	batch 4 layer 11 : error in forward
44	GitHub	Keras	Efficiency	Suboptimal network structure	No identified error	batch 4 layer 12 : error in forward
80	SO	Keras	Correctness	Mising flatten layer	Lack of pooling, missing flatten	×
84	SO	Keras	Correctness	Wrong type of activation function	Wrong activation function, wrong units' shape	batch 0 layer 2 : error in delta weights
86	SO	Keras	Correctness	Wrong type of activation function	Wrong activation function, wrong layers' structure	batch 0 layer 0 : error in forward
88	SO	Keras	Correctness	Wrong type of activation function	Wrong activation function	×
89	SO	Keras	Correctness	Wrong type of activation function	Wrong activation function	batch 0 layer 0 : error in forward
92	SO	Keras	Correctness	Wrong type of activation function	Wrong activation function, wrong window size for spatial filtering	batch 0 layer 9 : error in delta weights
95	SO	Keras	Correctness	Wrong API usage	No identified error	model does not learn
111	SO	Keras	Correctness	Missing preprocessing	No identified error	Error in delta weights
112	SO	Keras	Correctness	Missing flatten layer	Missing flatten	×

We selected 20 bugs randomly from *defect4ML*, 10 from SO bugs and 10 from GitHub based ones. With respect to the limitations of the *NeuraLint* and *DeepLocalize* tools, we could just use 12 out of 20 which are usable for at least one of the tools. Table 6 demonstrates the result of evaluating *NeuraLint* and *DeepLocalize*. The cells filled with × refer to the samples that have compile-time errors and could not be used by *DeepLocalize*.

Table 6 demonstrates the result of evaluating *NeuraLint* and *DeepLocalize*. The cells filled with × refer to the samples that have compile-time errors and could not be used by *DeepLocalize*. Based on the gathered results, *NeuraLint* was able to identify bugs in 9 out of 10 samples. Besides, it has detected design issues in some examples, in addition to the reported bugs. Design issues are poor design and/or configuration decisions that can have a negative impact on the performance and then quality of a DL-based software system (Nikanjam and Khomh 2021; Nikanjam et al. 2021a). For instance, bug #91 is related to the wrong activation function of the DL model, while *NeuraLint* has also identified that window size for spatial filtering does not define properly. Conversely, *DeepLocalize* could localize the bug correctly in 1 out of 4 samples.

6 Related Work

The closest work to our proposed benchmark was carried out by Kim et al. (2021) providing a benchmark of bugs in ML-based systems, which is called *Denchmark*. They extracted 4577 bugs reported in the issue tracker of 193 GitHub repositories. Although their benchmark was the first bug benchmark focused on ML-based systems, their study has several shortcomings. Firstly, they have considered repositories with various programming languages such as Java, C, C++, Python, etc., without considering any categorization on them. So, developers might have to inspect and then categorize the bugs based on their favorite programming languages, which can be time-consuming. The second drawback is ignorance of the big difference between bugs related to the ML (ML-related bugs) and other ones. *Denchmark* has reported all bugs without any differentiation. Moreover, this study has taken no notice of bug reproducibility, which is one of the main challenges in the SRE of ML-based systems (Zhang et al. 2018b). Last but not least, this benchmark has neglected standard benchmark criteria, which may result in benchmark effectiveness detriment.

Wardat et al. (2021) also provided a benchmark of 40 bugs to validate their proposed tool. They offered 11 bugs from GitHub and 29 bugs from SO and introduced them as a benchmark of bugs in ML-based systems. The first concern about their proposed benchmark is ignorance of the standard benchmark criteria (e.g., relevance, reproducibility, fairness) and SRE challenges in ML-based systems. Besides, no information has been provided about the execution process of applications extracted from GitHub to trigger the bug.

7 Threats to Validity

We now discuss the threats to validity of our study.

7.1 Internal Validity

The primary source of internal threat to the validity of the results provided in this study can be the categorization of bugs. To diminish this threat, we used the predefined taxonomy of bugs in DL programs discussed in Humbatova et al. (2020). Another internal threat to our proposed benchmark can be the manual inspection of the bugs and making decisions about their inclusion or exclusion. The first author (a PhD student and practitioner of ML development) reviewed 100 bug-fix commits and discussed the result with the second and third authors (two PhDs with research background in engineering ML-based systems) to mitigate this threat. After three meetings, we reached an agreement on the including and excluding rules of filtering the bug-fix commits. To ensure that the benchmark consists of real bugs, we used just the SO posts with an accepted answer and bug-fix commits that clearly explain the bug and its symptoms. Verifiability of the bug fixes may be the last internal threat to the validity of this research. To counteract this threat, we used different methods for ML-related bugs extracted from GitHub and SO.

Regarding the bugs extracted from GitHub, if the buggy applications do not provide appropriate tests to verify the bug fix, we used bug fix commit messages that clearly mention the changes to fix the reported bug. To verify bugs gathered from SO posts, we only used the posts with an accepted answer that is considered as evidence of fixing the bug correctly.

7.2 External Validity

The most crucial threat to the external validity of this study is its limitation to the *TensorFlow* (Abadi et al. 2016) and *Keras* (Chollet and et al 2018) frameworks. Firstly, we have selected them because they are two of the most popular ML frameworks (Yalçın 2021; Humbatova et al. 2020). Second, we have provided the feature to add new bugs using any ML framework to the benchmark and request to remove the existing bugs from it by any user. Furthermore, to achieve the highest diversity of ML-related bugs, we did not use any popularity-based filter on the GitHub repositories. We collected the bugs from the repositories developed by users with various expertise levels. Another thread to the external validity of this study can be the selection of *Python* as programming language of ML-based systems development. The main reason behind the selection of *Python* is the fact that it is the most used programming language for developing ML components (Voskoglou 2017; Gupta 2021). As a result, a larger variety of ML-related bugs can be found in ML-based systems based on the *Python* programming language and defect4ML will be also usable for a higher number of users.

8 Conclusion and Future Works

The growing tendency to apply ML-based systems in safety-critical areas increases the demand for reliable ML-based systems. A benchmark of bugs in ML-based systems, a faultload benchmark, is a key requirement for assessing the effectiveness of studies on ML-based systems' reliability (like testing) which are based on bugs' lifecycle. In this study, we reviewed 1777 ML-related bugs from GitHub and 1296 from SO which are related to the ML-based systems using two of the most common ML frameworks, *TensorFlow* and *Keras*, and represented that only near 3.48% of GitHub bugs and 2.93% of reported bugs in SO are reproducible. Besides, we showed that almost 13.3% of fixing of all reproducible bugs extracted from GitHub can be verified by their provided test cases. However, none of the SO posts has test cases for verifying bug fixes. We have also proposed *defect4ML*, a faultload benchmark of ML-based systems consisting of 100 bugs extracted from the software systems using *TensorFlow* and/or *Keras* (62 from Github, and 38 from SO). All of the standardized benchmark criteria have been satisfied by our proposed benchmark.

defect4ML also addresses the main SRE challenges in ML-based systems by providing bugs in various ML frameworks with different versions, comprehensive information regarding dependencies and required data to trigger the bug, detailed information about the type of bugs, and link to the origin of the bug. Concerning the ongoing nature of creating benchmarks, we plan to add more bugs to cover all types of bugs in ML-based systems. Moreover, we are going to improve *defect4ML* to be usable for automatic bug repair toolsi. Besides, we will add bugs based on other ML frameworks (such as *PyTorch*) to improve the coverage of the *defect4ML*.

Acknowledgements This work was supported by: Fonds de Recherche du Québec (FRQ), the Canadian Institute for Advanced Research (CIFAR) as well as the DEEL project CRDPJ 537462-18 funded by the National Science and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ), together with its industrial partners Thales Canada inc, Bell Textron Canada Limited, CAE inc and Bombardier inc.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). Savannah, USENIX, pp 265–283

Abidi M, Grichi M, Khomh F, Guéhéneuc YG (2019a) Code smells for multi-language systems. In: Proceedings of the 24th European conference on pattern languages of programs, pp 1–13

Abidi M, Khomh F, Guéhéneuc YG (2019b) Anti-patterns for multi-language systems. In: Proceedings of the 24th European conference on pattern languages of programs, pp 1–14

Abidi M, Rahman MS, Openja M, Khomh F (2021) Are multi-language design smells fault-prone? An empirical study. ACM Trans Softw Eng Methodol (TOSEM) 30(3):1–56

Al-Rfou R, Alain G, Almahairi A, Angermueller C, Bahdanau D, Ballas N, Bastien F, Bayer J, Belikov A, Belopolsky A et al (2016) Theano: a python framework for fast computation of mathematical expressions. arXiv e-prints pp arXiv–1605

Amershi S, Begel A, Bird C, DeLine R, Gall H, Kamar E, Nagappan N, Nushi B, Zimmermann T (2019) Software engineering for machine learning: a case study. In: 2019 IEEE/ACM 41st international conference on software engineering: Software engineering in practice (ICSE-SEIP). IEEE, pp 291–300

Barocas S, Selbst AD (2016) Big data's disparate impact. Calif Law Rev 104(3):671–732. http://www.jstor.org/stable/24758720. Accessed 11 Jan 2022

Borg M (2021) The aiq meta-testbed: pragmatically bridging academic ai testing and industrial q needs. In: International conference on software quality. Springer, pp 66–77

Bourque P, Dupuis R, Abran A, Moore JW, Tripp L (1999) The guide to the software engineering body of knowledge. IEEE Softw 16(6):35–44

Brownlee J (2020) Use early stopping to halt the training of neural networks at the right time. https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/. Accessed: 2022-12-29

Chollet F et al (2018) Keras: the python deep learning library. Astrophysics Source Code Library, pp ascl–1806

Chouldechova A, Roth A (2018) The frontiers of fairness in machine learning. arXiv:1810.08810

Collobert R, Bengio S, Mariéthoz J (2002) Torch: a modular machine learning software library. Tech. rep. Idiap

Developer guideline documentation G (2021) Github rest api. https://developer.github.com/v3/. Accessed: 2021-7-27

Dwork C (2008) Differential privacy: a survey of results. In: International conference on theory and applications of models of computation. Springer, pp 1–19

Esteva A, Robicquet A, Ramsundar B, Kuleshov V, DePristo M, Chou K, Cui C, Corrado G, Thrun S, Dean J (2019) A guide to deep learning in healthcare. Nat Med 25(1):24–29

Felderer M, Ramler R (2021) Quality assurance for ai-based systems: overview and challenges (introduction to interactive session). In: International conference on software quality. Springer, pp 33–42

Galin D (2004) Software quality assurance: from theory to implementation. Pearson Education, England

GitHub (2021) Github official website. https://github.com/about. Accessed: 2021-7-27

Gupta S (2021) What is the best language for machine learning? https://www.springboard.com/blog/data-science/best-language-for-machine-learning. Accessed: 2021-10-06

Hawkins DM (2004) The problem of overfitting. J Chem Inf Comput 44(1):1–12

https://github.com/dpressel/baseline/commit/4dad463 (2016). Accessed: 2021-11-01

https://stackoverflow.com/questions/34311586 (2016). Accessed: 2021-11-01

https://stackoverflow.com/questions/38080035 (2017). Accessed: 2021-11-01

https://stackoverflow.com/questions/42264649 (2017). Accessed: 2021-11-01

https://github.com/suchaoxiao/keras-frcnn_modify/commit/2f51f68 (2017). Accessed: 2021-11-01

https://github.com/albu/albumentations/commit/fec1f3b (2018). Accessed: 2021-11-01

https://github.com/vmelan/cifar-experiment/commit/561c82e (2018). Accessed: 2022-06-01

https://stackoverflow.com/questions/53119432 (2018). Accessed: 2021-11-01

https://github.com/acflorea/keras-playground/commit/d44c90c (2018). Accessed: 2022-06-01

https://github.com/keras-team/keras-tuner/commit/3758611 (2018). Accessed: 2022-06-01

https://github.com/hunkim/DeepLearningZeroToAll/commit/9f8fb94 (2018). Accessed: 2022-06-01

https://stackoverflow.com/questions/44924690 (2018). Accessed: 2021-11-01

https://stackoverflow.com/questions/58636087 (2018). Accessed: 2021-11-01

https://stackoverflow.com/questions/50079585 (2018). Accessed: 2021-11-01

https://github.com/PhilippeNguyen/kinopt/commit/fdee16f (2018). Accessed: 2021-11-01

https://stackoverflow.com/questions/56103207 (2019). Accessed: 2021-11-01

https://github.com/vaclavcadek/keras2pmml/commit/4795ec6 (2019). Accessed: 2021-11-01

Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1110–1121

Huppler K (2009) The art of building a good benchmark. In: Technology conference on performance evaluation and benchmarking. Springer, pp 18–30

IEEE standard for system, software, and hardware verification and validation (2017). IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017), pp 1–260. https://doi.org/10.1109/IEEESTD.2017.8055462

IEEE standard glossary of software engineering terminology (1990). IEEE Std 610.12-1990, pp 1–84. https://doi.org/10.1109/IEEESTD.1990.101064

ISO/IEC/IEEE international standard—systems and software engineering—vocabulary (2010). ISO/IEC/IEEE 24765:2010(E), pp 1–418. https://doi.org/10.1109/IEEESTD.2010.5733835

Islam MJ, Nguyen G, Pan R, Rajan H (2019) 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, pp 510–520

Islam MJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: fix patterns and challenges. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE). IEEE, pp 1135–1146

Jia L, Zhong H, Huang L (2021a) The unit test quality of deep learning libraries: a mutation analysis. In: 2021 IEEE International conference on software maintenance and evolution (ICSME). IEEE, pp 47–57

Jia L, Zhong H, Wang X, Huang L, Lu X (2021b) The symptoms, causes, and repairs of bugs inside a deep learning library. J Syst Softw 177:110935

Jia L, Zhong H, Wang X, Huang L, Li Z (2022) How do injected bugs affect deep learning? In: 2022 IEEE International conference on software analysis, evolution and reengineering (SANER). IEEE, pp 793–804

Jiang Y, Liu H, Niu N, Zhang L, Hu Y (2021) Extracting concise bug-fixing patches from human-written patches in version control systems. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). IEEE, pp 686–698

Just R, Jalali D, Ernst MD (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, pp 437–440

Keras (2016) Keras 2.1.5. https://github.com/keras-team/keras/releases/tag/2.1.5. Accessed: 2021-11-01

Kim M, Kim Y, Lee E (2021) Denchmark: a bug benchmark of deep learning-related software. In: 2021 IEEE/ACM 18th international conference on mining software repositories (MSR). IEEE, pp 540–544

Kirk M (2014) Thoughtful machine learning: a test-driven approach. O'Reilly Media, Inc.

Kistowski JV, Arnold JA, Huppler K, Lange KD, Henning JL, Cao P (2015) How to build a benchmark. In: Proceedings of the 6th ACM/SPEC international conference on performance engineering, pp 333–336

Krizhevsky A, Hinton G et al (2009) Learning multiple layers of features from tiny images

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The manybugs and introclass benchmarks for automated repair of c programs. IEEE Trans Softw Eng 41(12):1236–1256

LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324

Lenarduzzi V, Lomio F, Moreschini S, Taibi D, Tamburri DA (2021) Software quality for ai: where we are now? In: International conference on software quality. Springer, pp 43–53

Lin Z, Marinov D, Zhong H, Chen Y, Zhao J (2015) Jacontebe: a benchmark suite of real-world java concurrency bugs (t). In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 178–189

Lipton ZC (2018) The mythos of model interpretability: in machine learning, the concept of interpretability is both important and slippery. Queue 16(3):31–57

Liu X, Xie L, Wang Y, Zou J, Xiong J, Ying Z, Vasilakos AV (2020) Privacy and security issues in deep learning: a survey. IEEE Access 9:4566–4593

Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y (2005) Bugbench: benchmarks for evaluating bug detection tools. In: Workshop on the evaluation of software defect detection tools, vol 5. Chicago

Lyu MR (2007) Software reliability engineering: a roadmap. In: Future of software engineering (FOSE'07). IEEE, Minneapolis, pp 153–170

Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y et al (2018) Deepgauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. Association for Computing Machinery (ACM), New York, pp 120–131

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: an extensible java bug benchmark for automatic program repair studies. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 468–478

Marijan D, Gotlieb A, Ahuja MK (2019) Challenges of testing machine learning based systems. In: 2019 IEEE International conference on artificial intelligence testing (AITest). IEEE, pp 101–102

Martínez-Fernández S, Bogner J, Franch X, Oriol M, Siebert J, Trendowicz A, Vollmer AM, Wagner S (2021) Software engineering for ai-based systems: a survey. arXiv:2105.01984

McDonald N, Schoenebeck S, Forte A (2019) Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. Proc ACM on Human-Comput Interact 3(CSCW):1–23

McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia Medica 22(3):276–282

Nejadgholi M, Yang J (2019) A study of oracle approximations in testing deep learning libraries. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 785–796

Nikanjam A, Khomh F (2021) Design smells in deep learning programs: an empirical study. In: 2021 IEEE International conference on software maintenance and evolution (ICSME), pp 332–342

Nikanjam A, Braiek HB, Morovati MM, Khomh F (2021a) Automatic fault detection for deep learning programs using graph transformations. ACM Trans Softw Eng Methodol 31(1). https://doi.org/10.1145/3470006

Nikanjam A, Morovati MM, Khomh F, Braiek HB (2021b) Faults in deep reinforcement learning programs: a taxonomy and a detection approach. arXiv:2101.00135

Organisation T (2021) Torch official github repository. https://github.com/torch/torch7. Accessed: 2021-9-1

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) Pytorch: an imperative style, high-performance deep learning library. arXiv:1912.01703

Pei K, Cao Y, Yang J, Jana S (2017) Deepxplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th symposium on operating systems principles. Association for Computing Machinery (ACM), New York, pp 1–18

Pham HV, Qian S, Wang J, Lutellier T, Rosenthal J, Tan L, Yu Y, Nagappan N (2021) Problems and opportunities in training deep learning software systems: an analysis of variance. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, ASE '20. Association for Computing Machinery, New York, pp 771–783. https://doi.org/10.1145/3324884.3416545

Pressman RS (2005) Software engineering: a practitioner's approach. Palgrave Macmillan

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: a systematic literature review. Inf Softw Technol 55(8):1397–1418

Riccio V, Jahangirova G, Stocco A, Humbatova N, Weiss M, Tonella P (2020) Testing machine learning based systems: a systematic mapping. Empir Softw Eng 25(6):5193–5254

Rice L, Wong E, Kolter Z (2020) Overfitting in adversarially robust deep learning. In: International conference on machine learning. PMLR, pp 8093–8104

Rivera-Landos E, Khomh F, Nikanjam A (2021) The challenge of reproducible ml: an empirical study on the impact of bugs

Road vehicles—safety of the intended functionality. Standard (2019). https://www.iso.org/standard/70939.html. Accessed 11 Jan 2022

Rodríguez-Pérez G, Robles G, González-Barahona JM (2018) Reproducibility and credibility in empirical software engineering: a case study based on a systematic literature review of the use of the szz algorithm. Inf Softw Technol 99:164–176

Schoop E, Huang F, Hartmann B (2021) Umlaut: debugging deep learning programs using program structure and model behavior. In: Proceedings of the 2021 CHI conference on human factors in computing systems, pp 1–16

Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M, Crespo JF, Dennison D (2015) Hidden technical debt in machine learning systems. Adv Neural Inf Process Syst 28:2503–2511

Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 968–980

Spadini D, Aniche M, Bacchelli A (2018) 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—ESEC/FSE 2018. ACM Press, New York, pp 908–911. https://doi.org/10.1145/3236024.3264598

StackOverflow: Stack overflow annual developer survey. https://insights.stackoverflow.com/survey/2021 (2021). Accessed: 2022-04-01

Tambon F, Nikanjam A, An L, Khomh F, Antoniol G (2021) Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow

Tian Y, Pei K, Jana S, Ray B (2018) Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th international conference on software engineering, pp 303–314

Vieira M, Madeira H, Sachs K, Kounev S (2012) Resilience benchmarking. In: Resilience assessment and evaluation of computing systems. Springer, pp 283–301

Voskoglou C (2017) What is the best programming language for machine learning. https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7. Accessed: 2021-10-06

Wardat M, Le W, Rajan H (2021) Deeplocalize: fault localization for deep neural networks. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). IEEE, pp 251–262

Wardat M, Cruz BD, Le W, Rajan H (2022) Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In: Proceedings of the 44th international conference on software engineering, pp 561–572

Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y 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, pp 1556–1560

Xue M, Yuan C, Wu H, Zhang Y, Liu W (2020) Machine learning security: threats, countermeasures, and evaluations. IEEE Access 8:74720–74742

Yalçın OG (2021) Top 5 deep learning frameworks to watch in 2021 and why tensorflow. https://towardsdatascience.com/top-5-deep-learning-frameworks-to-watch-in-2021-and-why-tensorflow-98d8d6667351. Accessed: 2022-12-29

Zerouali A, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the diversity of software package popularity metrics: an empirical study of npm. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 589–593

Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S (2018a) 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). IEEE, pp 132–142

Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018b) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 129–140

Zhang J, Barr ET, Guedj B, Harman M, Shawe-Taylor J (2019) Perturbed model validation: a new framework to validate model relevance

Zhang JM, Harman M, Ma L, Liu Y (2020) Machine learning testing: survey, landscapes and horizons. IEEE Trans Softw Eng

Zhu C, Huang WR, Li H, Taylor G, Studer C, Goldstein T (2019) Transferable clean-label poisoning attacks on deep neural nets. In: International conference on machine learning. PMLR, pp 7614–7623

Zubrow D (2009) IEEE Standard classification for software anomalies. IEEE Computer Society

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Mohammad Mehdi Morovati[1] · **Amin Nikanjam**[1] · **Foutse Khomh**[1] · **Zhen Ming (Jack) Jiang**[2]

Amin Nikanjam
amin.nikanjam@polymtl.ca

Foutse Khomh
foutse.khomh@polymtl.ca

Zhen Ming (Jack) Jiang
zmjiang@cse.yorku.ca

[1] SWAT Lab., Polytechnique Montréal, Montréal, Canada

[2] York University, Toronto, Canada