



Titre: Reliability Assessment Methods for Machine Learning-Based
Title: Systems

Auteur: Mohammadmehdi Morovati
Author:

Date: 2024

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Morovati, M. (2024). Reliability Assessment Methods for Machine Learning-Based
Citation: Systems [Thèse de doctorat, Polytechnique Montréal]. PolyPublie.
<https://publications.polymtl.ca/62551/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/62551/>
PolyPublie URL:

**Directeurs de
recherche:** Foutse Khomh, & Ettore Merlo
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Reliability Assessment Methods for Machine Learning-based Systems

MOHAMMADMEHDI MOROVATI

Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*

Génie informatique

Décembre 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

Reliability Assessment Methods for Machine Learning-based Systems

présentée par **Mohammadmehdi MOROVATI**

en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Daniel ALOISE, président

Foutse KHOMH, membre et directeur de recherche

Ettore MERLO, membre et codirecteur de recherche

Mohammad HAMDAQA, membre

Lei MA, membre externe

RÉSUMÉ

L'apprentissage automatique (Machine Learning, ML) est rapidement devenu une technologie fondamentale, propulsant un large éventail d'applications, des utilitaires quotidiens des smartphones aux systèmes à grande échelle critiques pour la sécurité. À mesure que le ML s'étend dans divers domaines, la demande pour des systèmes fiables basés sur le ML devient plus pressante. Cependant, la nature distincte des systèmes ML, qui diffèrent fondamentalement des systèmes logiciels traditionnels, introduit des défis uniques dans l'évaluation de la fiabilité des systèmes ML. Cette thèse aborde ces défis en se concentrant sur quatre aspects critiques des systèmes basés sur le ML : comprendre la nature des bogues spécifiques au ML, ce qui est essentiel pour développer des techniques d'évaluation de la fiabilité ; identifier les défis de développement dans des domaines spécialisés du ML qui affectent la fiabilité des logiciels basés sur le ML ; mettre en œuvre une référence standard de bogues, une exigence clé pour le processus d'évaluation de la fiabilité ; et faire progresser les techniques de localisation des fautes.

La première étape essentielle dans la conception de techniques d'évaluation efficaces de la fiabilité des systèmes basés sur le ML est de comprendre la nature des bogues ML. À cette fin, nous avons mené une enquête détaillée sur 386 problèmes rapportés dans les 300 meilleurs dépôts GitHub utilisant des frameworks ML populaires tels que Keras, TensorFlow et PyTorch. Notre étude révèle que près de la moitié de ces problèmes concernent spécifiquement les composants ML, les bogues ML présentant des caractéristiques distinctes, telles qu'une complexité accrue et des coûts de correction plus élevés, par rapport aux bogues non-ML. Sur la base de cette analyse fondamentale, nous avons ensuite exploré les défis spécifiques à l'assurance qualité des systèmes basés sur le ML, en nous concentrant particulièrement sur l'apprentissage par renforcement (RL).

Le RL, un sous-domaine du ML, a fait l'objet d'un examen limité dans la littérature malgré son adoption croissante dans l'industrie. Pour explorer ces défis, nous avons analysé 927 posts liés au DRL sur Stack Overflow (SO) et développé une taxonomie des défis courants dans le développement d'applications DRL. Cette taxonomie, validée par une enquête auprès de 65 développeurs DRL, met en évidence que la *compréhension*, l'*utilisation des API* et les *problèmes de conception* sont les défis les plus courants, tandis que le *traitement parallèle* et les *bibliothèques/cadres DRL* sont identifiés comme les plus complexes. Ces résultats soulignent la nécessité critique d'outils et de ressources pour relever ces défis et améliorer la qualité des systèmes basés sur le DRL.

Soutenir le développement de tels outils nécessite une référence standard de bogues pour évaluer les performances de ces techniques et permettre des comparaisons standardisées. Ainsi, la troisième partie de cette thèse examine la reproductibilité et la vérifiabilité des bogues dans les systèmes basés sur le ML, conduisant au développement d’un référentiel appelé *Defect4ML*. *Defect4ML* comprend 100 bogues ML reproductibles issus des dépôts GitHub d’applications basées sur le ML et des posts SO liés au ML, en utilisant les deux frameworks ML les plus courants, Keras et TensorFlow. Ce référentiel répond aux critères standards tels que la pertinence, la reproductibilité, l’équité, la vérifiabilité et l’utilisabilité.

En s’appuyant sur ce référentiel, nous avons développé une nouvelle approche de localisation des fautes à l’échelle du système basée sur les graphes de connaissances (KG) pour traiter les fautes tout au long du pipeline de développement DL, y compris le prétraitement des données, l’entraînement des modèles et le déploiement du système. Notre approche, validée par rapport aux méthodes existantes telles que DeepLocalize et autoTrainer, démontre une performance supérieure en fournissant une liste classée des fautes et de leurs causes probables, offrant une solution complète pour assurer la fiabilité des systèmes basés sur le ML.

En résumé, cette thèse apporte des contributions significatives au domaine des systèmes basés sur le ML en (1) caractérisant la nature unique des bogues dans les composants ML, (2) identifiant et catégorisant les défis de développement dans le DRL, (3) créant le référentiel *Defect4ML* pour les bogues des systèmes basés sur le ML, et (4) en proposant une technique innovante de localisation des fautes qui couvre l’ensemble du pipeline de développement DL. Ensemble, ces contributions font progresser la compréhension et la fiabilité des systèmes basés sur le ML, ouvrant la voie à des applications plus robustes et fiables à l’avenir.

ABSTRACT

Machine Learning (ML) has rapidly become a fundamental technology, driving a diverse array of applications from everyday smartphone utilities to large-scale, safety-critical systems. As ML continues to expand into various domains, the demand for reliable ML-based systems has grown more pressing. However, the distinct nature of ML systems, which differ fundamentally from traditional software systems, introduces unique challenges in the reliability assessment of ML-based systems. This thesis addresses these challenges by focusing on four critical aspects of ML-based systems: understanding the nature of ML-specific bugs which is a necessity for developing reliability assessment techniques, identifying development challenges in specialized ML domains that affect the reliability of ML-based software, implementing a standard benchmark of bugs, a critical requirement for the reliability assessment process, and advancing fault localization techniques. The first essential step in devising effective reliability assessment techniques for ML-based systems is to understand the nature of ML bugs. To this end, we conducted a detailed investigation of 386 reported issues from the top 300 GitHub repositories that use popular ML frameworks such as Keras, TensorFlow, and PyTorch. Our study reveals that nearly half of these reported issues pertain specifically to ML components, with ML bugs exhibiting distinct characteristics, such as increased complexity and higher fixing costs, compared to non-ML bugs. Building on this foundational analysis, we then explored challenges specific to the quality assurance of ML-based systems, with a particular focus on reinforcement learning (RL).

RL, a subfield of ML, has seen limited examination in the literature despite its growing adoption in industry. To investigate these challenges, we analyzed 927 DRL-related posts on Stack Overflow (SO) and developed a taxonomy of common challenges in DRL application development. This taxonomy, validated by a survey of 65 DRL developers, highlights that *Comprehension*, *API usage*, and *Design problems* are the most prevalent challenges, while *Parallel processing* and *DRL libraries/frameworks* are identified as the most complex. These findings underscore the critical need for tools and resources to address these challenges and enhance the quality of DRL-based systems.

Supporting the development of such tools necessitates a standard benchmark of bugs for evaluating the performance of these techniques and enabling standardized comparisons. Thus, the third part of this thesis investigates the reproducibility and verifiability of bugs in ML-based systems, leading to the development of a benchmark called *Defect4ML*. *Defect4ML* comprises 100 reproducible ML bugs sourced from GitHub repositories of ML-based appli-

cations and ML-related SO posts, utilizing the two most common ML frameworks, Keras and TensorFlow. This benchmark meets standard criteria such as relevance, reproducibility, fairness, verifiability, and usability.

Leveraging this benchmark, we developed a novel system-level fault localization approach based on Knowledge Graphs (KG) to address faults across the entire DL development pipeline, including data preprocessing, model training, and system deployment. Our approach, validated against existing methods such as DeepLocalize, and autoTrainer, demonstrates superior performance by providing a ranked list of faults and their probable root causes, offering a comprehensive solution for ensuring the reliability of ML-based systems.

In summary, this thesis makes significant contributions to the field of ML-based systems by (1) characterizing the unique nature of bugs in ML components, (2) identifying and categorizing development challenges in DRL, (3) creating the *Defect4ML* benchmark for ML-based system bugs, and (4) proposing an innovative fault localization approach that spans the entire DL development pipeline. Together, these contributions advance the understanding and reliability of ML-based systems, paving the way for more robust and trustworthy applications in the future.

TABLE OF CONTENTS

RÉSUMÉ	iii
ABSTRACT	v
TABLE OF CONTENTS	vii
LIST OF TABLES	xi
LIST OF FIGURES	xiii
LIST OF SYMBOLS AND ACRONYMS	xv
CHAPTER 1 INTRODUCTION	1
1.1 Research Context	1
1.2 Problem statement	1
1.2.1 Studying the Characteristics of Bugs in ML-based Systems	1
1.2.2 Understanding Development Challenges in ML Applications	2
1.2.3 Developing Effective Fault Localization Techniques for DL-based Systems	2
1.3 Research contribution	3
1.4 Thesis Contribution	4
1.5 Thesis Organization	5
CHAPTER 2 BACKGROUND	6
2.1 Chapter Overview	6
2.2 Software Reliability Engineering (SRE)	6
2.2.1 Software Error, Fault, and Failure	7
2.2.2 Software Testing	7
2.3 Machine Learning (ML) and Deep Learning (DL)	7
2.4 ML-based System	10
2.4.1 ML-based System Development Pipeline	11
2.4.2 Faults in ML-based Systems	12
2.5 Benchmark of Bugs	12
2.6 Software Fault Localization	13
2.6.1 Spectrum-based Fault Localization (SBFL)	14
2.6.2 Mutation-based Fault Localization (MBFL)	14

2.6.3	Fault Localization Using Program Slicing	14
2.6.4	Fault Localization Using Stack Trace Analysis	15
2.6.5	Predicate Switching Fault Localization	15
2.6.6	Information Retrieval-based Fault Localization	15
2.6.7	History-based Fault Localization	16
2.6.8	Learning to Rank Fault Localization	16
2.7	Knowledge Graph (KG)	16
2.8	Chapter Summary	18
CHAPTER 3 LITERATURE REVIEW		19
3.1	Chapter Overview	19
3.2	Bug in ML-based Systems	19
3.3	SO Post Analysis	22
3.4	Challenges of DRL Applications Development	23
3.5	Benchmark of Bugs	26
3.6	Fault Localization in ML-based Systems	27
CHAPTER 4 BUG CHARACTERIZATION IN MACHINE LEARNING-BASED SYS- TEMS		32
4.1	Chapter Overview	32
4.2	Study Design	33
4.3	Methodology	35
4.3.1	Collecting repositories	35
4.3.2	Manual inspection of repositories	36
4.3.3	Manual labeling of bugs	38
4.4	Result	41
4.4.1	RQ1: Distribution of bugs in ML-based Systems	42
4.4.2	RQ2: Complexity of bug fixes in ML-based systems	44
4.4.3	RQ3: Needed resources for fixing bugs in ML-based systems	49
4.5	Discussion	51
4.6	Threats to Validity	54
4.6.1	Construct validity	54
4.6.2	Internal Validity	54
4.6.3	External Validity	55
4.6.4	Conclusion Validity	55
4.7	Chapter Summary	56

CHAPTER 5 COMMON CHALLENGES OF DEEP REINFORCEMENT LEARNING APPLICATIONS DEVELOPMENT	57
5.1 Chapter Overview	57
5.2 Study Design	57
5.3 Methodology	59
5.3.1 Extracting Posts from Stack Overflow (SO)	59
5.3.2 Manual Inspection	60
5.3.3 Taxonomy Construction and Validation	62
5.4 Result	64
5.5 Discussion	87
5.6 Threats to Validity	89
5.6.1 Construction validity	89
5.6.2 Internal validity	89
5.6.3 External validity	90
5.6.4 Reliability validity	90
5.7 Chapter Summary	91
CHAPTER 6 FAULTLOAD BENCHMARK OF BUGS IN MACHINE LEARNING-BASED SYSTEMS	92
6.1 Chapter Overview	92
6.2 Study Design	92
6.2.1 Objectives	92
6.3 Methodology	93
6.3.1 Collecting Bugs from Public Datasets	95
6.3.2 Collecting Bugs from GitHub	96
6.3.3 Extracting Bugs from Stack Overflow	98
6.3.4 Labeling Collected Bugs	100
6.4 Results	101
6.4.1 RQ1. Key factors in reproducibility of ML-related bugs	102
6.4.2 RQ2. The important factors in verifiability of fixing ML-related bugs	102
6.4.3 RQ3. Providing a standard faultload benchmark for ML-based systems	103
6.5 Discussion	111
6.6 Threats to Validity	112
6.6.1 Internal Validity	113
6.6.2 External Validity	114
6.7 Chapter Summary	114

CHAPTER 7	FAULT LOCALIZATION IN MACHINE LEARNING-BASED SOFTWARE: A SYSTEM-LEVEL VIEW	116
7.1	Chapter Overview	116
7.2	Study Design	116
7.3	<i>FL4Deep</i>	119
7.3.1	Dataset Preparation	119
7.3.2	Extracting Required Information from DL codes	120
7.3.3	Creating KG	123
7.3.4	Predicting Missed Relationship in KG	124
7.3.5	Ranking the Identified Root Causes	126
7.4	Results and Analysis	126
7.4.1	Experimental Design	127
7.4.2	RQ1. How does <i>FL4Deep</i> compare against other fault localization approaches?	128
7.4.3	RQ2. To what extent does each component contribute to the overall performance of <i>FL4Deep</i> ?	130
7.4.4	RQ3. What are the characteristics of faults misclassified by <i>FL4Deep</i> ?	132
7.5	Discussion	134
7.5.1	Challenges in Fault Localization of DL-based Systems	134
7.5.2	Most Fitted Fault Localization Techniques for DL-based systems	135
7.6	Threats to validity	136
7.6.1	Construct Validity	136
7.6.2	Internal Validity	136
7.6.3	External Validity	137
7.6.4	Reliability validity	137
7.7	Chapter Summary	137
CHAPTER 8	CONCLUSION	139
8.1	Limitations	141
8.2	Future Research	142
REFERENCES		144

LIST OF TABLES

Table 4.1	Details of the selected ML frameworks.	36
Table 4.2	Detailed information about the number of remaining repositories after each filtering out step.	37
Table 4.3	Detailed information about labeled bugs.	40
Table 4.4	The identified root causes of ML bugs in open-source ML-based software systems.	41
Table 4.5	Detailed result of <i>Mann–Whitney U</i> and <i>Cliff’s delta</i> test.	47
Table 5.1	Detailed information on the manual labeling process.	62
Table 5.2	Survey structure	65
Table 5.3	Detailed information regarding tags and keywords used to extract DRL-related SO posts. Columns show the number (#) and percentage (%) of SO posts categorized in the main categories having mentioned tags/keywords (‘DRL’, ‘DRL LF’, ‘DL’, ‘PP’, and ‘Gen’ refer to ‘DRL issues’, ‘DRL libraries/frameworks’, ‘DL issues’, ‘Parallel processing’, and ‘General programming’, respectively).	76
Table 5.4	Result of the survey of DRL practitioners.	80
Table 6.1	Detailed information about the selected ML frameworks	94
Table 6.2	Detailed information regarding the collected bugs from public datasets	96
Table 6.3	Detailed information about the number of remaining bug-fix commits after each filtering step	99
Table 6.4	Detailed information about the number of remaining posts after each filtering out step.	101
Table 6.5	Detailed information of bugs.	104
Table 6.6	Results of studied tools as case study	113
Table 7.1	Statistical operators used to analyze extracted dynamic information from model training logs	124
Table 7.2	Fault that can be detected by <i>FL4Deep</i>	125
Table 7.3	Sample results of comparing <i>FL4Deep</i> , and other popular fault localization techniques for DL-based systems	129
Table 7.4	Results of comparing the proposed method with previous studies . .	131
Table 7.5	Results of <i>FL4Deep</i> sensitivity analysis by removing Static Information (SI), Dynamic Information (DI), and KG Link Prediction (LP) components	131

Table 7.6	Comparison of False Positive (FP), False Negative (FN), Precision (PR), and Recall (RC) of various approaches in localizing ML-related faults	134
-----------	---	-----

LIST OF FIGURES

Figure 2.1	Agent interacting with its environment [1].	9
Figure 2.2	Sample architecture of an ML-based online transcription system (inspired from [2]).	10
Figure 2.3	Pipeline of ML-based system development process (adopted from [3])	12
Figure 2.4	High-level architecture of ML components.	13
Figure 2.5	Sample KG of an DL-based system	17
Figure 2.6	Link prediction in KG based on the type of inference (a) transductive link prediction, (b) inductive link prediction.	18
Figure 3.1	Taxonomy of bugs in DL programs [4]	21
Figure 4.1	Venn diagram of bug categorization in ML-based systems based on the location they occurred.	33
Figure 4.2	High-level view of the followed methodology.	35
Figure 4.3	Bug in ML-based systems: (a) distribution of issues, (b) root causes of the ML bugs, and (c) symptoms of ML bugs.	43
Figure 4.4	Classification of bugs in ML-based systems (a) based on the taxonomy provided by [4], (b) bugs falling into ‘other’ category.	45
Figure 4.5	Complexity of Fixes: (a) number of bug-fixes commit, (b) number of bug-fixes changed files, (c) number of bug-fixes changed LOC, (d) bug-fixes entropy, (e) number of comments on an issue, (f) number of comments on bug-fix.	46
Figure 4.6	Complexity of Fixes: (a) number of collaborators on an issue, and (b) number of collaborators on bug-fix, (c) number of comments/collaborator on issue, (d) number of comments/collaborator on a fix.	47
Figure 4.7	Number of developers who review PRs that fixed ML and non-ML bugs	52
Figure 5.1	High-level view of the used methodology.	60
Figure 5.2	Taxonomy of common challenges in DRL development. Numbers represented with each category/subcategory indicate the number of SO posts categorized into that category/subcategory.	66
Figure 5.3	Distribution of DRL related SO posts per years (a) high-level categories of the provided taxonomy, (b) subcategories of ‘ <i>DRL issues</i> ’ category.	75
Figure 5.4	Duration to receive an accepted answer (hours).	77
Figure 5.5	SO posts’ required time to receive an accepted answer.	78

Figure 5.6	Programming languages mentioned in the posts belong to various categories of challenges.	84
Figure 5.7	Common libraries/frameworks used for developing DRL applications.	86
Figure 6.1	Methodology of collecting bugs.	94
Figure 6.2	Distribution of bugs' symptoms in defect4ML.	105
Figure 6.3	A screenshot of <i>defect4ML</i> web application.	111
Figure 7.1	High-level view of <i>FL4Deep</i> methodology	119
Figure 7.2	Pipeline of DL-based system development process divided into three parts (adopted from [3])	121
Figure 7.3	Process of creating training dataset and training Random Forest (RF), Decision Tree (DT), and K-nearest Neighbors (KNN) models	122
Figure 7.4	Training a model for KG link prediction	126

LIST OF SYMBOLS AND ACRONYMS

SE	Software Engineering
AI	Artificial Intelligence
NN	Neural Network
ANN	Artificial Neural Network
ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
GNN	Graph Neural Network
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
SO	Stack Overflow
Q&A	Question & Answer
SRE	Software Reliability Engineering
KG	Knowledge Graph
RDF	Resource Description Framework
N3	Notation 3
RF	Random Forest
ST	Decision Tree
KNN	K-Nearest Neighbors
SO	Stack Overflow
API	Application Programming Interface
SBFL	Spectrum-based Fault Localizatio
MBFL	Mutation-based Fault Localization
IR	Information Retrieval
SI	Static Information
DI	Dynamic Information
PR	Pull Request
SQA	Software Quality Assurance
GPU	Graphics Processing Unit
AST	Abstract Syntax Tree
LSTM	Long Short-Term Memory
SUS	Stochastic Universal Sampling

AIAPIC	Absence of Inter API Compatibility
WD	Wrong Documentation
UT	Unaligned Tensor
APIC	API Change
AOTC	Absence of Type Checking
APIM	API Misuse
IMPC	Incorrect Model Parameter or Structure
CWCM	Confusion With Computation Model
ONNX	Open Neural Network Exchange
LOC	Lines of Code
LDA	Latent Dirichlet Allocation
SARSA	State–Action–Reward–State–Action
Q-learning DQN	Deep Q-Network
DDQN	Double Deep Q-Network
PPO	Proximal Policy Optimization
A3C	Asynchronous Advantage Actor-Critic
MDP	Markov Decision Process
RQ	Research Question
VAR	Variance
STD	Standard Deviation
SEM	Standard Error of the Mean
OS	Operating System
EYE	Euler Yet another proof Engine
LP	Link Prediction
FP	False Positive
FN	False Negative
PR	Precision
RC	Recall
LLM	Large Language Model

CHAPTER 1 INTRODUCTION

1.1 Research Context

Machine Learning (ML) has rapidly evolved into a critical technology, driving applications ranging from everyday smartphone functions to large-scale, safety-critical systems. As ML continues to expand across various fields, the demand for reliable ML-based systems has grown significantly. However, the unique characteristics of ML systems, which differ fundamentally from traditional software, pose distinct challenges in ensuring their reliability. Traditionally, software systems are built deductively, where the rules governing system behavior are explicitly defined in the program code. In contrast, ML-based systems derive these rules inductively from training data. This paradigm shift in software development introduces significant challenges in reasoning about the behavior of systems with ML components. The lack of explicit, complete specifications or source code for critical behaviors makes ML systems difficult to test and verify, complicating efforts to ensure reliability.

1.2 Problem statement

Ensuring the reliability of ML-based systems is critical. However, traditional approaches to software reliability do not fully address the unique challenges presented by ML components. This thesis identifies key problems in achieving reliable ML-based systems, focusing on three main areas: the need for a deeper understanding of ML-specific bugs, the development challenges unique to ML applications, and the creation of effective fault localization techniques tailored to ML systems.

1.2.1 Studying the Characteristics of Bugs in ML-based Systems

A key aspect of reliability is the identification and resolution of bugs. While previous studies have provided insights into bugs in traditional software, these findings do not adequately apply to ML-based systems due to differences in underlying paradigms. ML systems involve complex, data-driven behavior, leading to new types of bugs that are often unpredictable and harder to detect. A detailed characterization of these bugs provides developers with a clearer understanding of the most common bug types and their root causes in ML-based systems. This allows developers to address bugs more effectively, thereby improving the reliability of the final product. Additionally, software test engineers working on ML-based systems can create more precise test cases, catching a higher number of bugs and further enhancing system

reliability. Researchers can also develop more effective reliability assessment methods for ML-based systems by gaining comprehensive knowledge of the characteristics of these bugs. Therefore, a deeper understanding of bugs in ML-based systems benefits various stakeholders involved in ensuring the reliability of these systems.

1.2.2 Understanding Development Challenges in ML Applications

The process of developing ML applications, particularly in areas like deep reinforcement learning (DRL), introduces unique challenges that are not encountered in traditional software development. DRL applications, while sharing some commonalities with other ML systems, involve dynamic environments and learning processes that make them more difficult to test and validate. To address these challenges, it is crucial for DRL developers to have a clear understanding of these obstacles and account for them during the development process. Failure to adequately address these challenges may lead to significant problems, such as performance or functional bugs [5], which can compromise the reliability of the system. Therefore, recognizing these challenges is essential for building more robust and reliable ML applications. Additionally, this understanding can guide the development of specialized testing and fault localization techniques that are better suited to the intricacies of ML-based systems.

1.2.3 Developing Effective Fault Localization Techniques for DL-based Systems

Fault localization is a crucial step in ensuring the reliability of any software system. Since DL-based software systems are also susceptible to errors, fault localization plays a vital role in enhancing their reliability. However, traditional fault localization approaches are inadequate for DL-based systems. Fault localization is generally part of the software debugging process, which is a key aspect of software development. Debugging involves identifying and resolving software bugs to improve system reliability. An effective fault localization approach can detect a greater number of bugs, thereby contributing to more reliable software. Given the limitations of traditional fault localization techniques for DL-based systems, there is a clear need for approaches specifically designed to efficiently localize faults in these systems. DL-based systems are composed of several interconnected components—such as data preprocessing, model training, and deployment—each of which can introduce faults. Existing tools often focus primarily on the model itself, overlooking other critical parts of the DL pipeline. To ensure overall reliability, effective fault localization techniques must address the entire lifecycle of DL systems, from data handling to deployment, to accurately detect and resolve faults.

Thesis Statement. This thesis aims to develop effective methods for improving the reliability of ML-based systems by conducting a comprehensive analysis of the unique characteristics of ML-specific bugs and the challenges encountered during ML application development. A specific focus is placed on advancing fault localization techniques to address the unique characteristics of ML components, with the goal of improving overall system reliability and robustness.

1.3 Research contribution

To characterize ML bugs, we conducted an in-depth investigation of 386 reported issues in GitHub repositories. Our findings reveal that ML bugs display unique characteristics. Building on this foundational study, we further examined the specific challenges in the quality assurance of ML-based systems, particularly focusing on Reinforcement Learning (RL). Despite increasing industrial adoption of RL, it has received limited attention in the literature.

To examine DRL applications challenges, we conducted an analysis of 927 DRL-related posts on Stack Overflow (SO) and developed a comprehensive taxonomy of common challenges in DRL application development. The findings of this research emphasize the urgent need for specialized tools and resources to effectively address these challenges and improve the quality of DRL-based systems.

To support the development of such tools, a standardized benchmark of bugs is essential for evaluating the performance of various techniques and facilitating standard comparisons. Therefore, the third part of this thesis contributes a benchmark named *Defect4ML* which includes 100 reproducible ML bugs.

Using this benchmark, we developed a system-level fault localization method that localizes faults throughout the whole DL development pipeline. Our proposed approach, compared to existing methods (like DeepLocalize and AutoTrainer), shows improved performance by generating a ranked list of faults and their likely root causes. This method provides a more comprehensive solution for enhancing the reliability of ML-based systems.

In summary, this thesis makes substantial contributions to the field of ML-based systems by (1) representing the distinct characterizations of bugs in ML components, (2) identifying and categorizing development challenges in DRL, (3) creating a standard benchmark (*Defect4ML*) for ML system bugs, and (4) proposing a novel fault localization technique that covers the whole DL development pipeline. These contributions enhance the understanding and reliability of ML-based systems, paving the way for more robust and trustworthy

ML-based applications in the future.

1.4 Thesis Contribution

The contributions of this thesis are summarized as follows.

- We provide a large-scale study on the reproducibility of bugs in ML-based systems.
- We present the first comprehensive empirical study on the characteristics of bugs in ML-based systems.
- We provide insight into the cost and complexity of fixing ML and non-ML bugs in ML-based systems.
- We indicate and discuss the root causes and symptoms of the ML bugs.
- We provide the first large-scale empirical study of the challenges in the development of DRL applications.
- We propose the first standard benchmark of bugs in ML-based systems.
- We provide the first technique analyzing the whole pipeline of DL-based systems development to localize the bugs.
- We highlight the key challenges in fault localization of DL-based systems.

This thesis led to the publication/submission of the following research papers:

- **Morovati, M.M.**, Nikanjam, A., Tambon, F., Khomh, F. and Jiang, Z.M., 2024. Bug characterization in machine learning-based systems. *Empirical Software Engineering*, 29(1), p.14.
- **Morovati, M.M.**, Tambon, F., Taraghi, M., Nikanjam, A. and Khomh, F., 2024. Common challenges of deep reinforcement learning applications development: an empirical study. *Empirical Software Engineering*, 29(4), p.95.
- **Morovati, M.M.**, Nikanjam, A., Khomh, F. and Jiang, Z.M., 2023. Bugs in machine learning-based systems: a faultload benchmark. *Empirical Software Engineering*, 28(3), p.62.

- **Morovati, M.M.**, Nikanjam, A. and Khomh, F., 2024. Fault Localization in Deep Learning-based software: A System-level Approach, Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM).

The following papers have been written during my PhD. While they all investigate reliability issues in ML-based systems, they did not fit within the specific narrative of the thesis and so were excluded.

- Nikanjam, A., Braiek, H.B., **Morovati, M.M.** and Khomh, F., 2021. Automatic fault detection for deep learning programs using graph transformations. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(1), pp.1-27.
- Nikanjam, A., **Morovati, M.M.**, Khomh, F. and Ben Braiek, H., 2022. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. Automated software engineering, 29(1), p.8.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. **Chapter 2** provides the background information of this thesis. **Chapter 3** reviews related studies and research in this field. In **Chapter 4**, we present our study on bug characterization in ML-based systems, highlighting the differences between ML and non-ML bugs, identifying the most common ML bug types and their root causes, and comparing the complexity of fixing ML versus non-ML bugs. **Chapter 5** presents the results of our empirical study on the challenges developers face when developing DRL applications. Additionally, we introduce a taxonomy of common challenges in DRL application development, validated by 65 DRL developers. **Chapter 6** discusses the main obstacles to creating a standard benchmark of bugs and introduces a benchmark of bugs in ML-based systems that meets all standard benchmark criteria. In **Chapter 7**, we propose a system-level fault localization approach that can be effectively applied to DL-based systems. Finally, **Chapter 8** provides a summary and conclusion of this thesis.

CHAPTER 2 BACKGROUND

2.1 Chapter Overview

This Chapter provides the essential background for this thesis. First, we introduce Software Reliability Engineering (SRE) and clarify the concepts of error, fault, and failure in Software Engineering. Next, we discuss ML-based systems, highlighting how the definition of faults in these systems differs from that in traditional software systems. Next, we explore the significance of benchmarks, particularly faultload benchmarks, in assessing the reliability of software systems. we discuss the role of fault localization within the software engineering process and outline its fundamental techniques.

2.2 Software Reliability Engineering (SRE)

Quality is defined as the extent to which a system meets its requirements. In software engineering, quality refers to the degree of conformance of a system, component, or process to its specified functional and non-functional requirements [6]. Among various quality factors, reliability is recognized as the most critical in any engineering field. In software engineering, reliability is widely acknowledged as the most important quality attribute [7]. Software reliability primarily concerns the risk of software failures, which originate from software errors—parts of the code that are partially or entirely incorrect. In essence, software failure is the system’s inability to perform the required functionality [8].

Software Reliability Engineering (SRE) is a methodology aimed at ensuring the failure-free operation of software systems over a specified period. Various SRE techniques strive to achieve high reliability by monitoring different stages of the fault lifecycle, which can be categorized into four main classes [7]: 1) Fault prevention, which focuses on preventing the occurrence of faults; 2) Fault removal, which involves detecting and eliminating existing faults, often through validation and verification techniques; 3) Fault tolerance, which employs redundancy to ensure that services can continue functioning even when faults occur; 4) Fault/Failure forecasting, which predicts the likelihood of faults and failures in the software system. Most software reliability modeling approaches are centered on this last category.

2.2.1 Software Error, Fault, and Failure

In the software community, a fault is considered the manifestation of a software error that results in incorrect software functionality [9]. A software error is a mistake made by the programmer, which can be either grammatical (an issue in one or more lines of code) or logical (a failure to meet one or more software requirements) [8]. However, not all software errors necessarily become faults. An erroneous line of code becomes a fault and affects software functionality only when it is executed. When a user attempts to execute a faulty part of the software, activating the fault, it can result in a software failure. A software failure refers to the software's inability to perform the required functionality [10].

2.2.2 Software Testing

Software Quality Assurance (SQA) is a systematic approach that encompasses essential tasks to provide the necessary confidence in a software system to meet its technical requirements [8]. Software testing is the primary SQA tool used to verify the expected behavior of a software unit, multiple integrated units, or the entire system before its deployment in the customer's environment [8,9]. The main purpose of software testing is to identify and address defects that negatively impact software quality by utilizing a set of test cases. Each test case consists of specific test data, execution conditions, and expected outcomes to ensure that the developed software aligns with the specified requirements [9]. If the test cases reveal discrepancies between the actual and expected software behavior, the development team proceeds with a debugging process to locate and fix the errors [11].

2.3 Machine Learning (ML) and Deep Learning (DL)

Artificial Intelligence (AI) and especially Machine Learning (ML) have exerted a significant influence on the industrial software systems during the last decades. Generally, AI refers to the methodologies used for building intelligent machines, especially intelligent computer programs with the aim of learning, reasoning, and perception [12]. ML refers to a branch of AI that can learn from the data. ML systems can improve their performance over time based on their experience [13]. With regards to the fact that data plays a key role in ML, we can categorize ML systems into three main categories based on the data used for training [14]: **1)Supervised learning**, the most popular type of ML system, that uses labeled data as the learning target to be trained, **2)Unsupervised learning** which is trained using unlabeled data, and **3)Reinforcement Learning (RL)**, a subcategory of ML in which data is considered as a sequence of observation of the environment, actions that taken, and

rewards. Systems using RL try to learn how to interact with the environment and take the actions to achieve the highest reward. ML can be applied on some standard tasks including [15]: **1)classification** used to assign a category to each item, **2)regression** referring to the task of predicting a real value for each item, **3)ranking** applied to order items based on some criteria, **4)clustering** which is mostly used for very large data to partition items into some homogeneous classes, and **5)dimensionality reduction** that is used to transform the initial items to lower-dimensional representation, without losing the properties of initial representation.

Artificial Neural Networks (ANN) or more prevalent, Neural Networks (NN) is a type of model for ML that receives great attention lately which is associated with the invention of DL techniques [16]. NN is inspired by studying the neurons of the brain. It consists of artificial neurons to discover the relationship amongst the data and tries to mimic the way that the human brain operates. Among all types of NN, Feed Forward Neural Networks are considered the most prominent type. Multilayer perceptron and convolutional neural networks (CNN) are also known as the most popular kinds of Feed Forward Neural Networks [10].

A software component is a self-contained part of software or application that is able to work independently. The reason behind dividing software systems into components is to provide smaller parts with less complexity and more manageability [17]. The ML community defines ML components as software components that work based on ML algorithms and aim to provide intelligent behavior. There are fundamental differences between traditional (non-ML) and ML software components. In traditional software components, developers implement a program that directly encodes the logic to fulfill the functional requirements. However, to develop ML components, one needs to implement an ML model, such as a neural network, and then train it using a dataset [14]. In other words, ML algorithms learn a complex function from the given dataset and subsequently apply the learned function to unseen data points. The trained model is then used in the ML components for inference and generating desired results.

Deep Learning (DL), as a branch of ML, includes neural networks with numerous layers that results in large (deep) models. Recent DL outstanding successes in decision-making and human-competitive tasks (e.g. stock trading [18]) have made it a part of several state-of-the-art software systems [19]. DL-based systems are software systems where the DL model is implemented, trained with a large dataset, and integrated into the software systems [20]. Nowadays, there is an increasing demand for providing reliable DL-based systems, because of the increasing adoption of DL in different software areas, and especially safety-critical areas where any single bug may result in catastrophe (e.g., Tesla autopilot accident [21]).

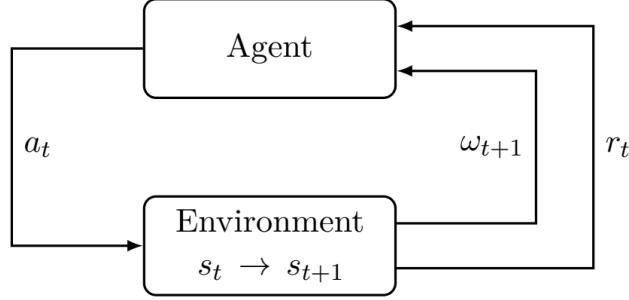


Figure 2.1 Agent interacting with its environment [1].

Reinforcement Learning (RL) is a subfield of ML where an agent interacts with an environment to learn optimal sequences of actions aimed at maximizing cumulative rewards. The goal of the agent is to incrementally refine its behavior or explore new strategies to improve its performance over time. This learning process is driven by trial and error, as the agent continuously interacts with the environment and gathers information from its experiences [1]. In general, RL focuses on the autonomous learning of optimal decision-making over time [22,23]. Formally, the RL problem is modeled as a discrete-time stochastic control process. At each time step t , the agent selects and executes an action a_t from a set of possible actions A . After taking the action, three things happen: 1) the agent receives a reward r_t from a set of rewards R , 2) the state of the environment transitions to a new state $s_t + 1 \in S$, and 3) the agent observes a new observation $\omega_{t+1} \in \Omega$ representing the next observation from a set of observations. Figure 2.1 illustrates the interaction between the agent and the environment. The RL agent’s objective is to learn a policy π from the set of all possible policies Π maximizing the expected cumulative reward. A discount factor $\gamma \in [0, 1]$ is also applied to future rewards to account for their relative importance [1].

In recent years, researchers have successfully combined DL with RL to address complex sequential decision-making problems [24]. This integration is known as Deep Reinforcement Learning (DRL). DRL leverages the power of DL to learn hierarchical representations from data, enabling it to handle large state-action spaces with minimal prior knowledge. For instance, DRL agents have demonstrated the ability to master video games using only raw visual inputs, such as pixel data from thousands of pixels [25]. Unlike traditional RL methods, DRL algorithms can process vast input spaces and identify actions that maximize rewards (e.g., improving a game score), allowing them to exhibit some human-like problem-solving abilities [26,27].

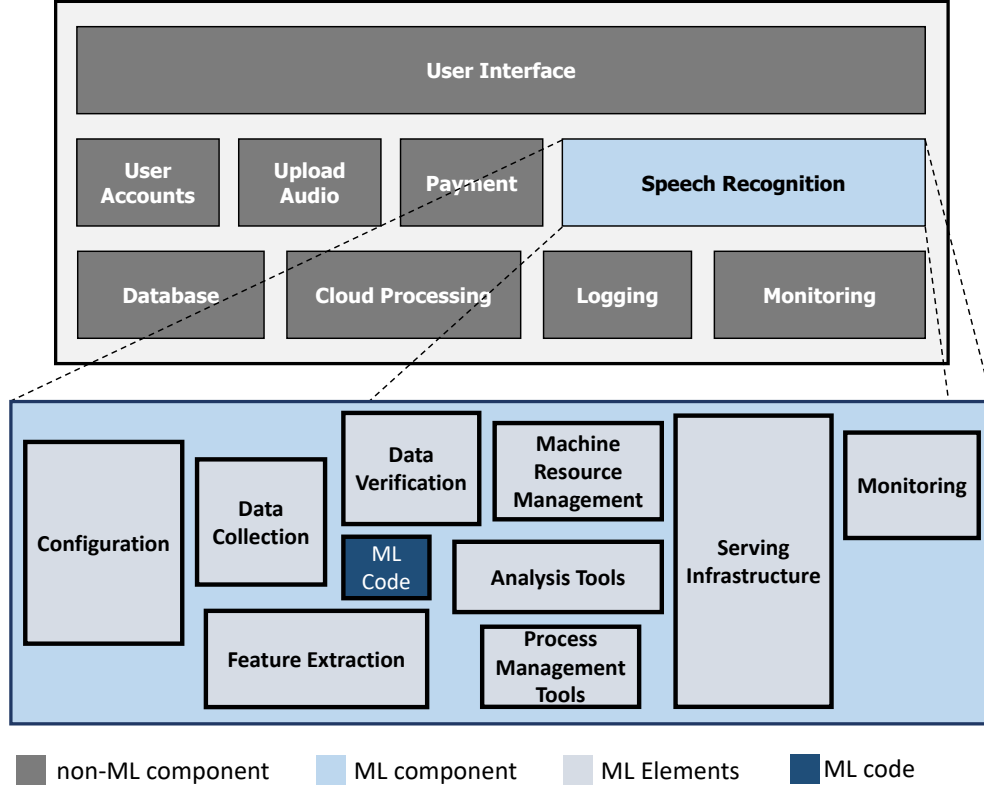


Figure 2.2 Sample architecture of an ML-based online transcription system (inspired from [2]).

2.4 ML-based System

ML-based systems refer to software systems having at least one ML component [28]. Figure 2.2 shows a sample ML-based system; i.e., a transcription service where the ML component provides the key functionality which is converting uploaded voice to text. However, to be able to use the ML model in the production environment, several additional software components, referred to as non-ML components, are necessary. For instance, a user interface is required to enable users to create a user account, upload the audio file(s), make payment of service expenses, and view the resulting transcript generated from the uploaded audio. Additionally, database and processing components are required to manage related tasks for converting uploaded audio to text, saving transcripts, and a monitoring component to ensure that operational system metrics meet the specified requirements. It is worth mentioning that the ML component itself comprises various elements (from configuration to monitoring) and ML code represents only a small part of the entire ML component [2].

Due to the central role of the trained ML model in the functionality of ML components, we are facing new types of bugs in ML components, referring to the malfunctioning of learned functions for unseen data points, and new challenges in characterizing them [29]. Bugs

in ML-based systems can be divided into two main categories: ML bugs that affect the functionality of ML components and non-ML bugs that affect the functionality of other (non-ML) components. It is important to note that ML models serve as an integral element within ML components that requires interaction with other non-ML and ML components.

2.4.1 ML-based System Development Pipeline

A software system is defined as a system comprising one or more software components [9]. Each software component is a distinct entity with an independent structure representing a set of functions [30]. An ML-based system is considered a type of software system that includes at least one ML component [28]. An ML component is also a software component whose functionality depends on machine learning. Fig. 2.3 illustrates the general pipeline of the ML-based system development pipeline, encompassing nine steps [3].

- **Model Requirement:** In this step, system designers assess the feasibility of implementing various system features using ML and decide on the most appropriate models.
- **Data Collection:** This step involves integrating available datasets or creating a dataset tailored to the specific software system.
- **Data Cleaning:** Here, inaccurate and noisy data are filtered out to ensure a high-quality dataset.
- **Data Labeling:** This step involves assigning informative labels to each record in the dataset.
- **Feature Engineering:** This step includes a set of activities focused on selecting the most suitable features for the ML models.
- **Model Training:** The selected ML models are trained using the chosen features, along with the collected and cleaned data.
- **Model Evaluation:** The trained model is evaluated using a testing dataset and pre-defined metrics, such as model accuracy.
- **Model Deployment:** The trained and evaluated models are deployed in the target environments and devices.
- **Model Monitoring:** Finally, the deployed models are continuously monitored in the real environment to detect and address potential system bugs during execution.

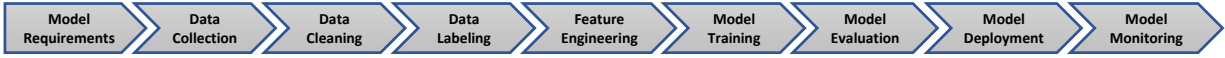


Figure 2.3 Pipeline of ML-based system development process (adopted from [3])

2.4.2 Faults in ML-based Systems

According to the definition of the software bug, an ML bug refers to the differences between the existing and required behavior of an ML component [14]. ML bugs can emerge from three main sources: program level (ML program written to build ML model), production level (i.e., using the ML model for inference), and infrastructure level (ML frameworks) (see Figure 2.4). Bugs at any level may also affect the overall quality of the ML component and accordingly, the quality of the ML-based system. In this study, we focus on the ML bugs at the program level. In addition to the general concept of system failure, ML bugs may also result in a bad performance, crash, data corruption, hang, and memory out of band which are considered ML failures as well [31].

2.5 Benchmark of Bugs

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 like performance, security, and efficiency [32]. It is widely acknowledged that a standardized benchmark is the most significant requirement to evaluate and compare the methodologies [33]. To generate a standardized benchmark, several criteria should be satisfied in the development process of the benchmark, including [32, 33]:

- **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 [33]. 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 [34].
- **Reproducibility:** refers to the benchmark ability to provide the same results while it

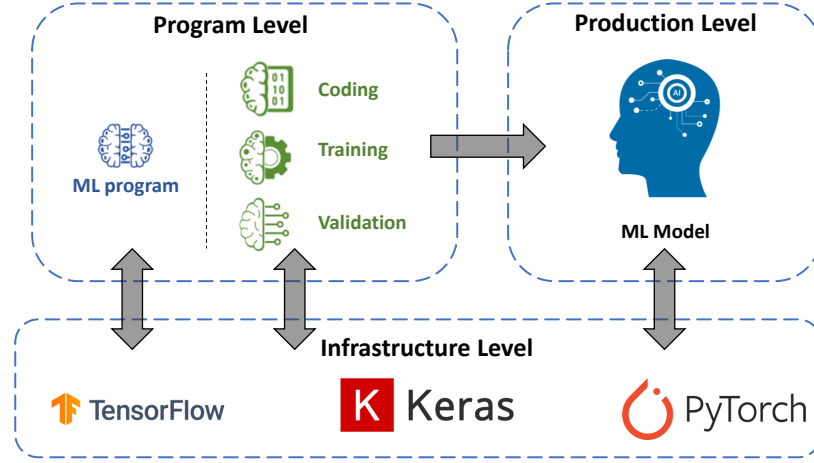


Figure 2.4 High-level architecture of ML components.

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 [35]. 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 [32].

2.6 Software Fault Localization

Generally speaking, fault localization is considered as an operation that receives a faulty program and a set of test cases as input and generates a ranked list of suspicious program elements [36]. Program elements can be taken into account in three levels including 1) statement (line of code), 2) method, and 3) file. Fault localization techniques have been categorized into two main groups, based on the methodology used for the analysis [37, 38].

- **Static:** Methods that rely solely on the source code, without executing the application, extract necessary information directly from the available code and analyze it to identify suspicious elements.
- **Dynamic:** Techniques that require running the program and utilizing runtime information (e.g., stack traces, bug reports, etc.). These techniques try to monitor the system execution during the testing process and collect necessary dynamic data. Dynamic analysis techniques may also extract static information from the available source code to enrich the collected dynamic data for analysis.

Various techniques have been developed for fault localization in software systems, each bringing a unique approach with specific strengths and limitations. The following sections present an overview of these methods.

2.6.1 Spectrum-based Fault Localization (SBFL)

A program spectrum refers to a measurement of a program's runtime behavior (such as code coverage). Spectrum-based fault localization leverages these runtime behavior measurements of test cases to identify and locate faults [39]. In other words, this approach analyzes the dynamic behavior of the program during the execution of a test suite to determine how likely each statement contains a fault [40].

2.6.2 Mutation-based Fault Localization (MBFL)

Mutation analysis is the assessment process of the test suite's effectiveness in detecting various types of faults. Generally, each mutant replaces an operand or expression with another, thereby altering a statement [41]. Mutation testing is a software testing technique that involves introducing faults into the program under test (referred to as mutants) and analyzing the differences in behavior between the mutants and the original program [42]. Mutation-based fault localization identifies suspicious mutants and uses this information to pinpoint the location of faulty statements [43]. Suspiciousness level of a program statement is determined by how frequently it affects both passed and failed tests.

2.6.3 Fault Localization Using Program Slicing

As software systems have grown larger and more complex, software debugging has become increasingly challenging. Program slicing is a debugging technique that narrows the focus by isolating program statements relevant to a specific computation, effectively removing

irrelevant parts and transforming the program into a minimal form [44]. A program slice represents a subset of the program that influences a particular system behavior. In other words, it is the portion of the program that affects the values of specific statements of interest [45]. Program slicing is generally divided into two main types: static and dynamic. Static slicing relies solely on information available statically, without executing the program. In contrast, dynamic slicing considers only the statements executed for a particular input, resulting in a smaller, more focused slice compared to static slicing [46]. In bug localization, program slicing simplifies the program by extracting a minimal subset of the program that directly influences the incorrect behavior, making it easier to investigate. The next step is to identify the specific statements within this slice that are responsible for the faulty behavior [47].

2.6.4 Fault Localization Using Stack Trace Analysis

A stack trace is a sequence of active stack frames generated during program execution, offering crucial information for debugging [36]. Each function call generates a stack frame that remains active until the function returns. Fault localization methodologies based on stack trace gather active traces of passed and failed system execution. Next, the fault localizer tries to identify the active functions at the point of a system crash and localize the fault, by analyzing these stack traces [48]. It is worth mentioning that this technique is mostly used to check the reason for program crashes.

2.6.5 Predicate Switching Fault Localization

A predicate controls the execution of a program by determining its branching paths. Predicate switching involves running a program through various control flows [49]. If altering the outcome of a condition expression causes a test case to pass instead of fail, the predicate is considered critical and may be the root cause of the failure. The predicate-switching fault localization technique uses mutations to modify predicate conditions and evaluates the results of the program's execution.

2.6.6 Information Retrieval-based Fault Localization

Information Retrieval (IR) is a methodology used to extract relevant information from large collections of unstructured data [50], primarily employed for text indexing and searching in documents. In the context of fault localization, IR-based techniques take bug reports as input to generate a ranked list of files likely related to the reported issue. Notably, this approach

does not require program execution information (e.g., test case results) and relies solely on the content of the bug reports [51].

2.6.7 History-based Fault Localization

History-based fault localization uses development history information to localize bugs, operating on the premise that source files with a history of a higher number of bugs are more likely to contain defects in the future. Put simply, history-based methods require a substantial amount of processed historical data [52]. In the history-based fault localization approach, program elements are ranked according to their likelihood of being defective, similar to the process used in bug prediction [53].

2.6.8 Learning to Rank Fault Localization

With the significant increase in computational power over the past decade, DL has emerged as one of the most popular fields in computer science. DL techniques have also been used to enhance fault localization techniques. The Learning to rank Fault Localization techniques train DL models to prioritize files by their likelihood of containing defects, leveraging suspiciousness scores derived from various SBFL formulas [54].

2.7 Knowledge Graph (KG)

A Knowledge Graph (KG) is both a specialized type of Knowledge Base (KB) system and a form of labeled, directed graph that organizes and represents large amounts of structured knowledge. KGs are designed to store entities and their relationships in a way that allows for efficient querying, reasoning, and extraction of meaningful insights [55]. Google started the development of KG in 2012 to develop a structure including important aspects of human knowledge that can be found in data sources [56]. KG can organize knowledge from various information sources effectively to represent knowledge about certain domains [57]. Since KG provides a contextualized understanding of data, it has received more attention in recent years [58]. A key feature of KG in data representation is that they treat the links between facts (the edges in the KG) as equally important as the facts themselves. Figure 2.5 represents a simple KG of a DL-based system.

The Resource Description Framework (RDF) is a standard model for representing information about resources [59], which is the core building block of KG. RDF is designed to enable efficient information processing by applications, going beyond merely presenting data in a human-readable format. By representing relationships between resources, RDF is particularly

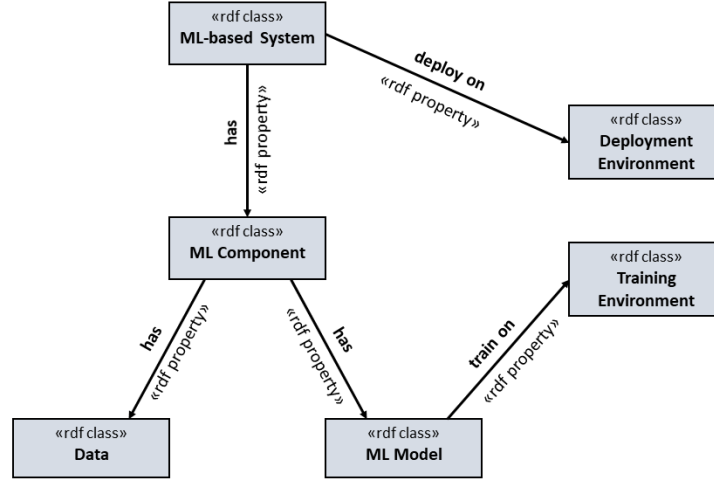


Figure 2.5 Sample KG of an DL-based system

well-suited for building interlinked datasets [60]. In RDF, linked data is represented as triples consisting of a *subject*, *predicate*, and *object*. The *subject* and *object* denote entities, while the *predicate* specifies the relationship between entities.

Notation3 (N3) is a logical programming language considered a superset of RDF [61]. N3 extends RDF’s capabilities by enhancing its representational power and enabling decision-making through data manipulation, information access, and reasoning. It allows developers to make statements about entities (including logical implications) which is known as declarative programming. Consequently, an N3 reasoning engine can infer new information from the declared statements, making it a suitable option for automating decision-making processes or enriching KG [62].

Despite recent advances in KG, even the largest KGs remain incomplete. For instance, FreeBase [63], one of the largest public domain KGs, has over 70% of its person entities lacking information about their place of birth [64,65]. To address this issue, link prediction has been developed to infer missing relationships among entities [66]. Traditionally, link prediction has relied on various heuristic metrics based on the paths between graph nodes, such as Katz [67] and PageRank [68]. However, recent advancements in DL inspired researchers to develop new techniques for link prediction in KGs using Graph Neural Networks (GNNs) [69,70]. At the same time, several datasets have also been created specifically for link prediction tasks in graphs, including FB15K [63] and WN18 [71]. There are generally two main techniques for link prediction in KGs, based on the type of inference: transductive and inductive [72]. Transductive link prediction involves training the DL model on the same graph used for inference [73], meaning that it uses known entities to predict links between them. In contrast,

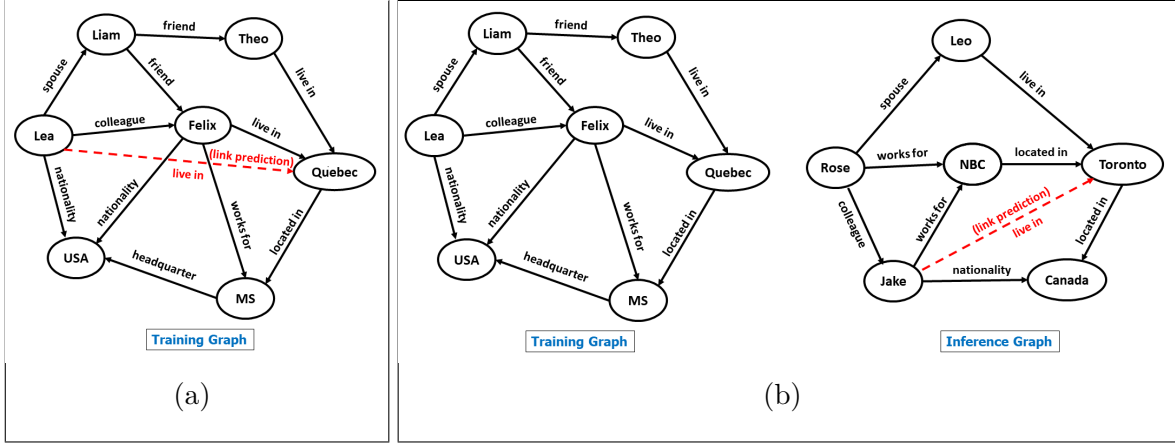


Figure 2.6 Link prediction in KG based on the type of inference (a) transductive link prediction, (b) inductive link prediction.

inductive link prediction involves using a different graph for training than the one used for inference [74], enabling the prediction of possible links between previously unseen entities. Fig.2.6 shows the difference between transductive and inductive link prediction.

2.8 Chapter Summary

This chapter provided the foundational knowledge necessary to comprehend the contributions of this thesis. We began by introducing the software reliability engineering process and the concept of software fault, which is central to this process. Following that, we provided an overview of ML, along with two of its most prominent subfields: DL and RL. In the subsequent part, we explored ML-based systems, detailing the entire development pipeline and explaining how faults manifest in such systems. Additionally, we discussed the characteristics of a standard benchmark in software engineering and emphasized the importance of a bug benchmark in software testing. We then moved on to software fault localization, discussing its objectives and the most common techniques used to pinpoint faults. Lastly, we introduced the concept of KG, explained their key features, and discussed the importance of link prediction within KG for various applications.

CHAPTER 3 LITERATURE REVIEW

3.1 Chapter Overview

In this chapter, we review the previous studies related to this thesis including bug characterization in ML-based systems, challenges in the development of ML-based systems, benchmark of bugs in ML-based systems, and fault localization for ML-based systems.

3.2 Bug in ML-based Systems

Zhang et al. [29] carried out research on the possible bugs in the ML programs using TensorFlow framework. They reported several major reasons for TensorFlow bugs including training data, program code, execution environment, and TensorFlow framework. But they just focused on the bugs in the program code. They also mentioned three main symptoms for the bugs in the TensorFlow programs. 1) error which can be an exception or crash in the application, 2) Low effectiveness which means poor accuracy or any other unexpected output, and 3) Low efficiency that refers to the execution of the program slowly. Furthermore, they represented six main symptoms for the studied bugs including 1) Incorrect model parameters or structure which is related to some problems in learning rate (model parameters) or missing model nodes/layers (model structure). 2) Unaligned tensor referring to the mismatch between the input and expected tensor shape. 3) Confusion with the TensorFlow computation model which is about the problems resulting from the unfamiliarity of the users with the underlying computations assumed by the TensorFlow. 4) TensorFlow API change is another root cause that seems more serious in comparison with traditional software systems, 5) TensorFlow API misuse referring to the problems in using TensorFlow API without satisfying the assumptions, and 6) structure inefficiency that is similar to the incorrect model parameters/structure, while this problem has influence on the performance efficiency of the system. In other words, the structure inefficiency results in the performance inefficiency, while bugs belong to the incorrect model parameters/structure class cause incorrectness in the functionality.

Humbatova et al. [4] developed a taxonomy of DL-related bugs. They provided five major categories of bugs in DL programs including model, tensors and inputs, GPU usage, training, and API. Figure 3.1 represents the diagram of the bug taxonomy they provided. After analyzing the bugs that they collected from StackOverflow and GitHub, they reported that bugs belonging to the training category are the most prevalent bug category in DL programs.

Model and tensors, and input classes also stay at the second and the third most popular bugs, respectively.

Islam et al. [31] conducted an empirical study on the characteristics of bugs developers may face in development of DL software systems. They extracted StackOverflow posts and GitHub bug fix commits from the repositories that use the most popular DL frameworks (cafe, Keras, TensorFlow, Theano, and Torch). They categorized the bugs based on three criteria including bug types, bug root causes, and bug symptoms. They have reported API, coding, data, structural, and non-model structural as the different types of bugs that may occur in the development of DL systems. API bugs refer to the problem in using DL APIs and its main root causes can be API change in various versions of frameworks, lack of inter-API compatibility, and confusing documents. Coding bugs mean the problems result from programming mistakes. Data bugs are related to the problems in data that is used by the DL system, not the DL system. Actually, this kind of bugs are related to the preprocessing of the data, before using it by the DL system. Structural bugs, one of the most prominent classes of DL bugs, emerge from the problems in DL model definitions. Non-model structural bugs are similar to structural bugs, while they arise in DL development stages except for the modeling. They also identified six main root causes for the introduced bugs in DL including 1) lack of inter API compatibility, 2) lack of type checking, 3) API change, 4) API misuse, 5) confusion with computational model, and 6) incorrect model parameter or structure. Finally, they presented six major symptoms for bugs in DL including bad performance, crash, data corruption, hang, incorrect functionality, and memory out of bound. Besides, they reported that most of the DL bugs (more than 66%) result in the system crash.

Kumar et al. [75] categorized the failures in ML-based systems into two main classes, intentional and unintentional failures. Intentional failure mode refers to the failures generated by an adversarial attempt to compromise different aspects of the ML-based system. In other words, this type of failure tries to subvert the system to do its identified function. Intentional failure is also known as the adversarial attack in the ML community. Different adversarial attacks try to carry out attacks on confidentiality, integrity, and availability of ML-based systems. Unintentional failures are considered as the unpredicted failures in ML-based systems. They introduced six different types of unintentional failures in ML-based systems including reward hacking, side effects, distributional shifts, natural adversarial examples, common corruption, and incomplete testing. Reward hacking is completely related to the Reinforcement Learning (RL) systems and occurred because of the mismatch between started reward and the true rewards in RL. Side effects refer to the disruption of the environment by RL system to achieve its goal. Distributional shifts mean that the ML system which is tested in a type of environment can not be adopted to change in other kinds of environments. Natural adver-

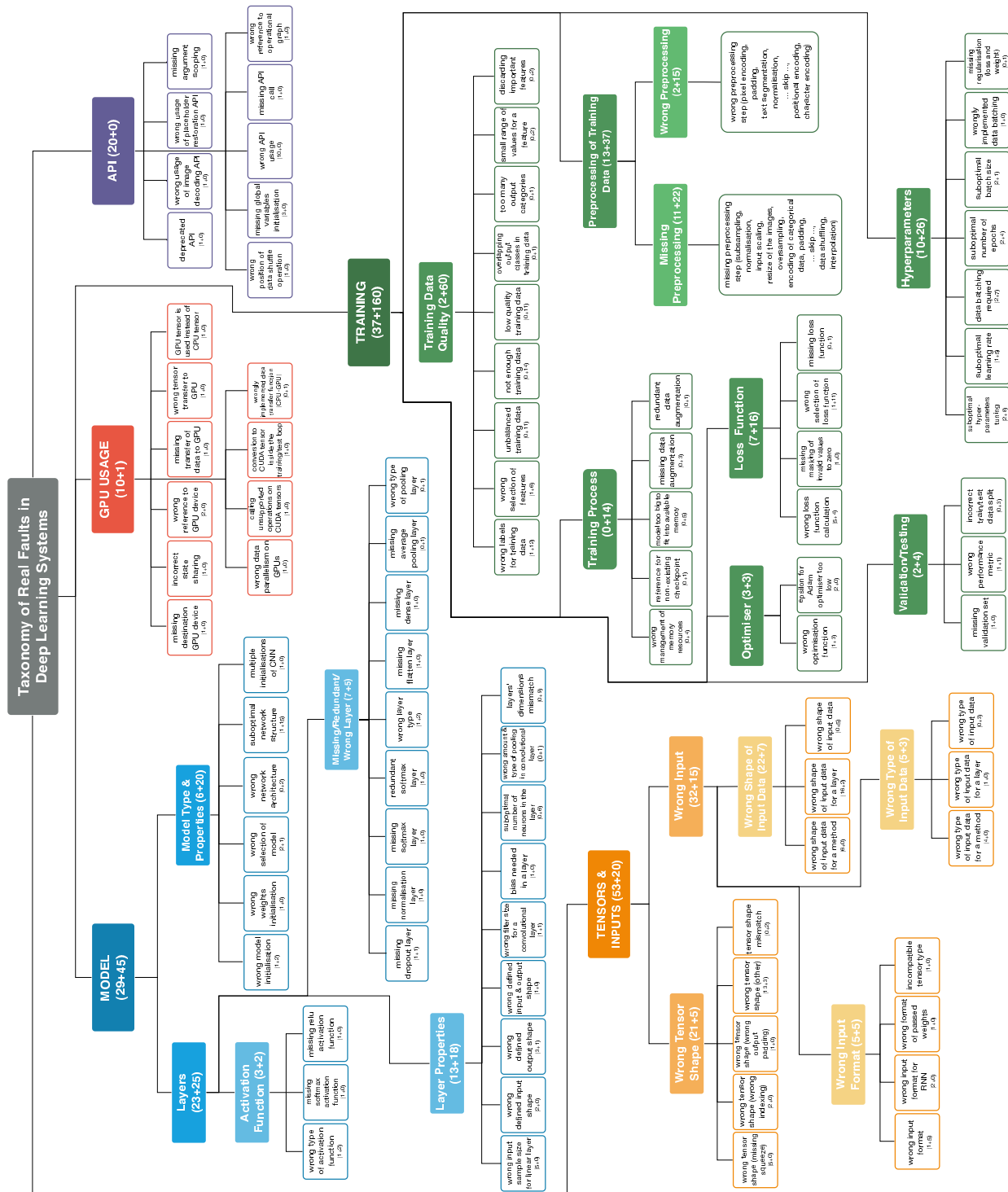


Figure 3.1 Taxonomy of bugs in DL programs [4]

serial examples explain that the functionality of ML-based system might be failed because of lack of perturbation attacks. Another class is common corruption that is related to the inability of the ML-based system to handle some common challenges such as noisy images. Incompleted testing refers to the fact that ML-based systems can not be tested in realistic conditions.

Although there has been significant research on bugs in ML-based systems and their classification, existing taxonomies and bug types have become outdated and require revision. The rapid growth of ML usage across various domains in recent years may have introduced new types of bugs that are not adequately captured by current classifications. Moreover, having a more precise understanding of bugs in ML-based systems is essential for effective detection, repair, and localization, ultimately leading to more reliable ML-based software systems.

3.3 SO Post Analysis

Beyer et al. [76] investigated the automatic classification of SO posts. They manually labeled 1000 posts, and identified 7 categories of questions: 1) API changes, 2) API usage, 3) Conceptual, 4) Discrepancy, 5) Learning, 6) Errors, and 7) Review. Leveraging the labeled dataset, they developed two approaches for the automatic classification of SO posts. In the first approach, they used the labeled dataset to extract some regular expression patterns and used these patterns to predict the category of other posts; achieving a performance of 0.91 for both precision and recall. In the second approach, they trained Random Forest and Support Vector Machine (SVM) classifiers using the labeled dataset. The best results were obtained using the Random Forest classifier, i.e., a precision of 0.88 and a recall of 0.87.

Alshangiti et al. [77] investigated SO posts related to ML development. They used a tag-based snowball sampling approach to extract SO posts related to ML, starting with the ‘machine-learning’ tag. Their results revealed that a higher number of ML-related posts remain without any accepted answer (61%), in comparison with general domain questions (48%). They also reported that ML-related questions need 10 times longer to receive an answer, compared to general domain questions. Next, they compared the ratio of expert users in ML and web development (the most popular domain of programming in SO), showing that the number of ML experts is significantly less than that in web development. Afterward, they reviewed the most challenging ML development phases revealing that data preprocessing and manipulation, and model deployment and environment setup are the two most error-prone phases.

Bangash et al. [78] conducted an empirical study of SO posts related to ML. They used the

‘machine_learning’ tag to extract 28,010 posts published between 2008 and 2018. Next, they used Latent Dirichlet allocation (LDA) [79] to categorize the extracted posts into 44 detailed topics. Then, they showed that *code errors*, *Algorithms*, and *Labeling* are the most discussed topics in ML. They then combined the 44 topics identified using LDA into 4 main groups including frameworks, implementation, sub-domain (RE), and algorithms. They reported that nearly 51% of all ML-related SO posts belong to the implementation group. Afterward, two of the researchers manually examined 230 sampled posts and reported that most of the questions stem from the fact that ML novice developers try to use ML in their software systems. They extracted information about the number of questions with accepted answers and concluded that ML-related questions are harder to answer than general domain SO questions. They also observed that only 65.6% of ML-related SO posts have appropriate tags; which might suggest that many users are not knowledgeable enough to assign the proper ML-related tags to their posts.

Hamidi et al. [80] examined the challenges that developers may face in the development of ML systems, based on their discussion in SO. They studied 43,950 ML-related SO posts submitted between 2008 and 2020. First, they showed that *Python* is the most popular programming language for ML development, and C# and C/C++ are the least popular programming languages for ML development. Then, they reported that model building and model evaluation are the two most challenging steps in ML development while model monitoring is the least questioned phase. They also report that questions regarding model requirements, data collection/processing, and model-building steps receive less accepted answers than others. This may stem from the fact that questions about these steps are more difficult to answer or the lack of active knowledgeable developers on SO to answer questions related to these steps.

Generally speaking, extensive research has been conducted on analyzing SO posts across various domains, leading to well-established approaches for extracting useful information from this data. However, while some studies have focused on investigating SO posts related to ML and its applications, there is still significant potential for uncovering valuable insights through further analysis of ML-related posts. For instance, analyzing posts about DRL could reveal key challenges that DRL developers encounter when building their systems, an area that has not been thoroughly explored yet. Besides, discussions which are recently mentioned in SO posts may reveal new knowledge on the ML-related applications.

3.4 Challenges of DRL Applications Development

Zhang et al. [81] proposed strategies to help DL and DRL developers detect and resolve quality issues in their applications. Nikanjam et al. [82] proposed a methodology for auto-

matically detecting faults in DL applications, using graph transformations. Schoop et al. [83] also proposed a strategy for detecting common issues during the development of DL models. These studies primarily targeted issues occurring in the training program of DL models. The issues considered in these aforementioned study fall within the ‘*DL issues*’ category of our provided taxonomy, which constitutes only a small portion of DRL issues.

Nikanjam et al. [84] also investigated challenges categorized as *DRL issues* in our proposed taxonomy. They examined questions/discussions about four popular DRL frameworks (including *gym* [85], *Tensorforce* [86], *Dopamine* [87], and *Keras-rl* [88] on GitHub and SO) and extracted 329 SO posts about DRL. They categorized these posts into six groups: basic concepts, without acknowledgment, implementation issues, answered by the owner, relative questions, and others. They reported that ‘without acknowledgment’ and ‘implementation’ questions are the most common DRL-related questions in SO, accounting for 32% and 27%, respectively. They also showed that in 2% of their studied SO posts, the answer has been posted by the questioner. They report that DRL-related SO posts take an average and median time of 2.07 days and 13 hours, respectively, before receiving an accepted answer. This period is longer than the time taken by DL-related SO posts to receive an accepted answer; which is 5 hours on average. This finding implies that DRL-related questions might be more difficult to answer than DL-related questions. Nikanjam et al. [84] also proposed a taxonomy of faults in DRL models. with 11 different issue types. Our study differs from what Nikanjam et al. [84] carried out in several aspects. Firstly, they concentrated on four specific libraries/frameworks designed for DRL development, ignoring other SO posts containing source codes using other Python-based libraries/frameworks or not mentioning any script. For instance, a number of SO posts inquire about DRL concepts (e.g. #52838439) without mentioning any scripts, a scope not covered by Nikanjam et al. [84]. Moreover, their research was just on the DRL model, whereas our study delves into the challenges developers may face throughout the development of entire DRL applications, without any limitation to a specific section of DRL applications. Last but not least, Nikanjam et al. [84] examined SO posts reporting program faults during the development of DRL applications. It is imperative to distinguish between challenges and program faults and note that challenges do not necessarily equate to program faults. A fault denotes a defect or error leading to a discrepancy between the expected and achieved results or observed behavior [28]. Software development challenges, in contrast, encompass any difficulty or complexity encountered in completing a development task. These challenges may arise from various factors, including technical complexities, resource constraints, or lack of expertise. Nikanjam et al. [84] also investigated challenges categorized as ‘*DRL issues*’ in our proposed taxonomy. They examined questions/discussions about four popular DRL frameworks (including *gym* [85],

Tensorforce [86], *Dopamine* [87], and *Keras-rl* [88] on GitHub and SO) and extracted 329 SO posts about DRL. Since Nikanjam et al. [84] focused on software faults, they excluded 305 SO posts from their dataset, generating their taxonomy based on 24 such posts that included faults explicitly. Notably, all SO posts in their dataset are included in our study. Furthermore, our research incorporates SO posts that do not pertain to program faults but instead pose queries related to the comprehension of DRL concepts. For example, the comprehension subcategory includes 253 SO posts, none of which reports program faults. They categorized these posts into six groups: basic concepts, without acknowledgment, implementation issues, answered by the owner, relative questions, and others. They reported that ‘without acknowledgment’ and ‘implementation’ questions are the most common DRL-related questions in SO, accounting for 32% and 27%, respectively. They also showed that in 2% of their studied SO posts, the answer has been posted by the questioner. They report that DRL-related SO posts take an average and median time of 2.07 days and 13 hours, respectively, before receiving an accepted answer. This time period is longer than the time taken by DL-related SO posts to receive an accepted answer; which is 5 hours on average. This finding implies that DRL-related questions might be more difficult to answer than DL-related questions. Nikanjam et al. [84] also proposed a taxonomy of faults in DRL models. The most significant difference between their study and ours is their focus on the DRL model only and the fact that they only collected data about faults mentioned in SO; disregarding all the other types of questions.

Yahmed et al. [89] conducted a study on the challenges of deploying DRL systems, based on the questions that developers ask on SO. In the first step, they extracted 357 SO posts related to the deployment of DRL systems. Next, they categorized collected SO posts into 4 categories with respect to their deployment platform, including ‘server/cloud’, ‘mobile/embedded system’, ‘browser’, and ‘game engine’. Their results showed that the number of SO posts regarding deployment has grown over the 7 studied years. They manually examined the extracted SO posts and identified 31 challenges related to DRL deployment. They grouped these challenges into 11 categories (and proposed a taxonomy): general questions, deployment infrastructure, data preprocessing, RL environment, communication, agent load/save, performance, environment rendering, agent export, request handling, and continuous learning. The proposed taxonomy has been evaluated via a survey with practitioners. Their results show that DRL developers struggle the most with deployment infrastructures and RL environments. Their results also show that communication-related challenges (procedure, connection loss, configuration of remote setting, and model convergence) are the most difficult challenges to address, in terms of the time to an accepted answer. The main difference between the work of Yahmed et al. and our study is the focus of the study. Yahmed et al.

examined DRL deployment challenges while we examined challenges faced by DRL developers during the application development phase, i.e., prior to deployment. In other words, they ignored all DRL development steps before deployment and focused only on the deployment phase of the DRL applications, which occurs after complete implementation of them. At the opposite extreme, we investigated the SO posts regarding the whole pipeline of DRL applications development. Besides, comparing the number of studied SO posts in these two studies (357 vs 927 SO posts) demonstrates that Yahmed et al filtered out the whole dataset of SO posts on DRL development to achieve posts specifically talking about the deployment step of the DRL application.

While several studies have been conducted to identify challenges in the development of ML-based applications, to the best of our knowledge, no research specifically addresses the challenges faced by developers during the implementation of DRL applications. Although some researchers have explored challenges related to specific aspects of DRL development, such as deployment challenges [89], there is a noticeable lack of comprehensive studies that examine the main challenges across the entire pipeline of DRL application development.

3.5 Benchmark of Bugs

Benchmark of software bugs that contains a set of real bugs is known as faultload benchmark [32]. Several studies on the faultload benchmark for traditional software systems have been done, e.g., *Defects4J* [90] (a benchmark of bugs in Java open source projects hosted on GitHub), *BugBench* [91] (a benchmark of C/C++ programs' bugs), *ManyBugs* [92] (a benchmark of defects in C programming language), and *Bears* [93] (a Java bug benchmark for automatic program repair). On the other hand, some faultload benchmarks such as *JaConTeBe* [94] (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.

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 [32]. 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 [29] 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. [95] 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) [96]. 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 [97], a scientific computing framework for ML algorithms, which development has been deactivated since 2018 [98]. Similar to the former studies, in the public bugs dataset published by Humbatova et al. [4], 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 [99] or not accessible [100]. Lack of dependency information is another shortcoming of this public dataset. It is worth noting that datasets provided in [4, 29, 95] 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. [101] 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.

A bug benchmark is a fundamental tool for validating reliability assessment methods. A benchmark specifically for ML-related bugs is also crucial to validate approaches which are developed to assess the reliability of ML-based systems. Given the significant differences between bugs in traditional software and those in ML systems, it is essential to provide a benchmark that specifically addresses ML-related bugs. Although some research claims to have developed benchmarks for ML-related bugs, these efforts often overlook standard benchmark criteria. As a result, there is still a need for an ML-related bug benchmark that fully meets the standards of a reliable and comprehensive benchmark.

3.6 Fault Localization in ML-based Systems

DEBAR is a static analysis tool that detects numerical bugs at the architecture level of DL programs [102]. *DEBAR* uses two main categories of abstracting methods to detect numerical faults [103] including 1) tensor and 2) numerical values. Regarding tensor abstracting

methods, *DEBAR* uses array expansion, array smashing, and tensor partitioning abstracting methods. Furthermore, *DEBAR* uses interval and affine relation analysis abstraction techniques as numerical abstraction methods. Numerical faults refer to the issues represented as ‘NaN’, ‘INF’, or crashes during the training phase. It also checks the program source codes for the most common unsafe operations such as *Exp*, *Log*, etc. Results of evaluating *DEBAR* show that it outperforms other existing static numerical fault detection techniques, in terms of accuracy (93.0%) without decreasing the performance.

DeepLocalize introduces a white box-based technique to localize faults in DL programs using two basic steps [101]. The first step involves generating intermediate code from the source code of the DL program. This intermediate code is created to verify whether the DL statements are identifiable. In the subsequent step, *DeepLocalize* employs a white-box method to dynamically analyze the traces generated during model training. *DeepLocalize* detects the faulty layers or hyperparameters leading to bugs in DL program [101]. It is worth noting that *DeepLocalize* requires both the source code of the DL software system and the model training logs. That is, it needs an executable DL program without any compile error to be able to analyze it and find possible bugs. To evaluate *DeepLocalize*, 40 buggy samples in total were collected from GitHub and SO, showing that *DeepLocalize* detects bugs and their root cause in 34 and 21 out of 40, respectively.

UMLAUT, the Usable Machine LeArning Utility, is a tool that helps DL developers identify, understand, and debug DL programs [83]. *UMLAUT* integrates with the DL program to collect model training information and heuristically assess the model’s structure and behavior. After analyzing the gathered data, Umlaut identifies potential issues, representing them as error messages and offering best practice solutions in the form of code snippets. *UMLAUT* provides a Keras callback function that enables the capture of model training details. Based on the severity of the identified issue, Umlaut classifies the messages into three levels: warning, error, and critical. To recommend best practices for addressing these issues, *UMLAUT* draws from various sources, including lecture notes [104], books [24], and expert blogs [105]. *UMLAUT* claims to detect issues across multiple stages of the DL pipeline, such as data preparation, where it identifies problems like input data exceeding typical limits, NaN values in loss or input data, shape mismatches in image inputs, and unexpected validation accuracy. It also detects model architecture issues, such as missing activation functions, the absence of a softmax layer, and the use of multiple activation functions in the final layer. Furthermore, Umlaut flags parameter tuning issues, including suboptimal learning rates, potential overfitting, and excessively high dropout rates. By providing these insights, Umlaut helps optimize the DL model development process.

NeuraLint, a model-based fault detection approach for ML programs provide a technique to detect faults in the DL programs using meta modeling and graph transformation [82]. To analyze a DL program, *NeuraLint* translates it into a model, based on the provided meta-model for DL programs. Next, it uses a model-based verification to check 23 rules. The rules have been inferred from 1) research papers studying bugs in DL programs, 2) public datasets of faulty DL programs, and 3) official tutorials of DL frameworks. *Neuralint* was evaluated using 34 faulty ML/DL programs extracted from GitHub and SO. The results show a recall of 70.5% and a precision of 100% when detecting bugs and design issues.

Braiek et al. [106] introduced a property-based debugging approach called *TheDeepChecker*, aimed at identifying 17 deep learning (DL) bugs previously documented by Humbatova et al. [4]. To achieve this, they extracted key features for each DL component (such as initial random parameters and output activation functions) and developed principles based on these features to detect DL bugs. To debug a DL program, *TheDeepChecker* begins by collecting dynamic information during model training, such as hidden activations, predictions, and losses. It then applies various statistical calculations to reduce the dimensionality of the extracted data. In the following step, *TheDeepChecker* flags layers where more than half of the neurons have died, as determined by a threshold in the outputs. Next, it evaluates the DL program against critical values and erroneous behaviors using pessimistic boundaries. Finally, an approximative component assesses the results from the previous steps against an anticipated set of possible states and behaviors to identify bugs within DL programs. To evaluate *TheDeepChecker*, the authors tested it on real-world DL buggy samples sourced from SO and GitHub. The comparison between *TheDeepChecker* and *Amazon SMD* showed that *TheDeepChecker* outperformed *Amazon SMD*, achieving 75% accuracy in bug identification compared to *Amazon SMD*'s 60%.

DRLinter, a model-based fault detection technique for deep reinforcement learning (DRL), is a tool to discover bugs in DRL programs [84]. Reinforcement Learning (RL) is a subcategory of ML, where the main goal is to achieve maximum reward. DRL refers to the integration of the DL methodology into RL approaches to improve sequential decision-making [24]. In the first step, *DRLinter* transforms the DRL program into a graph. Next, it uses a generic meta-model for DRL programs to check 11 defined rules and validate them against possible faults. *DRLinter* has been evaluated using 15 synthetic DRL programs, where errors were artificially injected into the source code, and 6 real-world DRL programs. The results show that *DRLinter* successfully detects faults in all synthetic examples. In the real-world samples, it achieved a recall of 75% and a precision of 100% in identifying faults.

AutoTrainer is an automated system for detecting and repairing training issues in deep

neural networks. It identifies five distinct types of training faults in Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) [81], helping to optimize the training process. CNN is known as a kind of Neural Network (NN) using convolution in at least one of the network layers. CNN is mostly used for analyzing a grid of values (e.g., images) [24]. RNN is also considered a subcategory of NN used for analyzing sequential data. It is important to take into account that connections in RNN can create a cycle where the output of some nodes may be the subsequent input of the same nodes [107]. *AutoTrainer* collects and analyzes model training logs to detect possible training faults. As *AutoTrainer* focuses on the model training phase and its related faults, it can detect 1) vanishing gradient, 2) exploding gradient, 3) dying ReLU, 4) oscillating loss, and 5) slow convergence problems. *AutoTrainer* also provides a built-in solution to fix any of the identified faults, ensuring prompt correction when issues are detected. Evaluation results show that *AutoTrainer* identified 316 faults in 262 reviewed DL programs and successfully fixed 309 of them. Additionally, the average accuracy of the fixed programs improved by up to 1.5 times.

DeepFD is a learning-based fault diagnosis and localization technique for DL programs [108]. *DeepFD* collects and analyzes data from model training to identify potential faults within DL programs. It extracts 20 distinct features, such as loss and accuracy at each epoch, to detect faults in DL programs. It employs multi-label versions of three classifiers—K-Nearest Neighbors (KNN), decision tree, and random forest—to diagnose these faults. Subsequently, the source code of the DL program is converted into an Abstract Syntax Tree (AST) to localize the faults identified in the previous step. Finally, *DeepFD* reports several suspicious lines of code that are likely to be the root causes of the detected faults.

DeepDiagnosis is a tool that recognizes faults, reports symptoms of faults, localizes the found faults, and provides suggestions for fixing them [109]. Similar to the previous approaches, *DeepDiagnosis* also collects information from model training and analyzes it to detect possible faults. To this end, it provides a callback function that should be passed to the model training method. Next, it analyzes the gathered data to detect 8 different fault symptoms; including saturated activation, exploding tensor, accuracy not increasing, dead node, loss not decreasing, unchanged weight, exploding gradient, and vanishing gradient. In the next step, it uses a decision tree to localize the root cause of the identified faults and formulate recommendations for their correction. Several changes to the model structure can be recommended by the tool to fix the identified faults; change loss/activation function, change optimizer, change weight/bias initialization, change learning rate, change number of training layers, change batch size, and change size of training data. *DeepDiagnosis* was evaluated on a total of 444 ML programs, sourced from GitHub, Stack Overflow, or generated by *AutoTrainer* [81]. The comparison of *DeepDiagnosis* with *UMLAUT* [83], *DeepLocalize* [101], and

AutoTrainer [81] approaches, based on 56 buggy models from GitHub and SO, demonstrated that *DeepDiagnosis* outperformed all the other approaches. However, its accuracy was lower than all mentioned approaches when applied to the samples generated by *AutoTrainer*.

Wardat et al. [110] introduced an approach called *Deep4Deep* to automatically debug DNN programs and localize faults by mapping extracted model features to specific model problems. They implemented a Long Short-Term Memory (LSTM) model to learn the relationship between symptoms of a faulty model and their root causes. The LSTM model captures patterns of model issues using features extracted from the DNN model. *Deep4Deep* leverages both dynamic and static information from DNN models. Dynamic information includes model parameters (e.g., weights, metrics) observed during training, such as loss, activation functions, data range, and the vanishing gradient. Static information is derived directly from the DNN model’s source code, independent of execution. Additionally, Wardat et al. compared *Deep4Deep* with existing methods, including UMLAUT [83], DeepLocalize [101], Autotrainer [81], and DeepDiagnose [109]. Their results showed that *Deep4Deep* outperforms these methods in terms of fault detection and localization accuracy, time efficiency, and the clarity of information provided to users regarding faults and their root causes.

Given the fundamental differences between the paradigms of traditional software and ML-based software, it is essential to develop tools and approaches that can efficiently localize faults in ML-based systems. Although several studies have been conducted on fault localization in ML-based systems, no existing approach addresses all stages of the ML development pipeline. Additionally, the increasing usage of ML across various domains in recent years may introduce new types of bugs in ML-based systems that are not adequately handled by current fault localization tools and approaches.

CHAPTER 4 BUG CHARACTERIZATION IN MACHINE LEARNING-BASED SYSTEMS

4.1 Chapter Overview

The rapid growth of applying ML in different domains, especially in safety-critical areas, increases the need for reliable ML components, i.e., a software component operating based on ML. Since corrective maintenance, i.e. identifying and resolving systems bugs, is a key task in the software development process to deliver reliable software components, it is necessary to investigate the usage of ML components, from the software maintenance perspective. Understanding the bugs' characteristics and maintenance challenges in ML-based systems can help developers of these systems to identify where to focus maintenance and testing efforts, by giving insights into the most error-prone components, most common bugs, etc. In this chapter, we investigate the characteristics of bugs in ML-based software systems and the difference between ML and non-ML bugs from the maintenance viewpoint. We extracted 447,948 *GitHub* repositories that used one of the three most popular ML frameworks, i.e., *TensorFlow*, *Keras*, and *PyTorch*. After multiple filtering steps, we select the top 300 repositories with the highest number of closed issues. We manually investigate the extracted repositories to exclude non-ML-based systems. Our investigation involved a manual inspection of 386 sampled reported issues in the identified ML-based systems to indicate whether they affect ML components or not. Our analysis shows that nearly half of the real issues reported in ML-based systems are ML bugs, indicating that ML components are more error-prone than non-ML components. Next, we thoroughly examined 109 identified ML bugs to identify their root causes, and symptoms, and calculate their required fixing time. The results also revealed that ML bugs have significantly different characteristics compared to non-ML bugs, in terms of the complexity of bug-fixing (number of commits, changed files, and changed lines of code). Based on our results, fixing ML bugs is more costly and ML components are more error-prone, compared to non-ML bugs and non-ML components respectively. Hence, paying significant attention to the reliability of the ML components is crucial in ML-based systems. These results deepen the understanding of ML bugs and we hope that our findings help shed light on opportunities for designing effective tools for testing and debugging ML-based systems.

4.2 Study Design

Software error is an incorrect part of a source code, which can be a grammatical, logical, or any other type of mistake that a software developer may make [8]. An ML error is any mistake that occurred in any part of the code of an ML component (e.g. data collection, feature extraction, and ML training code) [14]. A Software system containing erroneous code can produce an abnormal behavior, i.e., a fault (software bug). A software bug is a problem causing a disparity between defined software requirements and developed functionality [9]. Accordingly, an ML bug refers to the differences between the existing and required behavior of an ML component [14].

Based on the location of the bugs (in the code) and their impact, bugs in ML-based systems can be classified into ML and non-ML bugs. We define ML bugs as issues that affect the functionality of ML components. In contrast, a non-ML bug is an issue that deteriorates the functionality of non-ML components. Similar to research conducted by Riccio et al. [10], we categorize bugs in ML-based systems into three main classes based on the location of the code where the bugs occur (Figure 4.1):

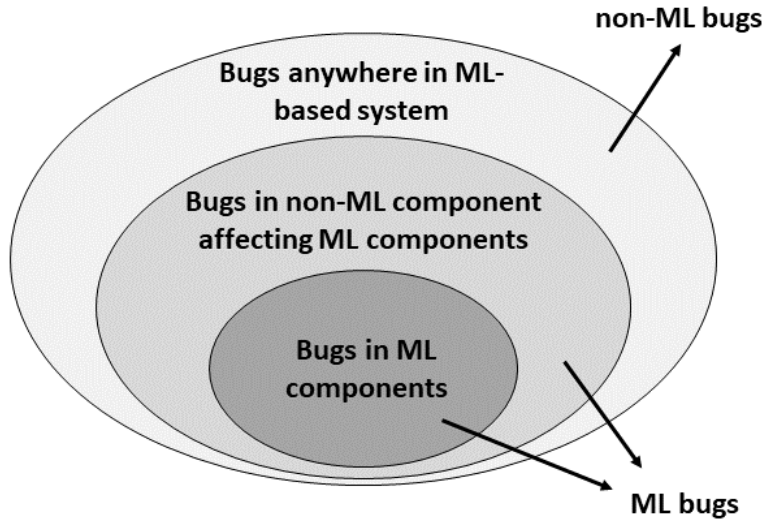


Figure 4.1 Venn diagram of bug categorization in ML-based systems based on the location they occurred.

- ***Bugs in ML components:*** In this class, we investigate only bugs inside ML components in isolation. As an example, issue [111] is related to a problem in training logs, hence it is related to an issue inside the ML component.

- ***Bugs in non-ML components, but affecting ML components:*** This class considers bugs inside any component affecting ML components’ functionality. This category includes all bugs classified into the previous category (Bugs in ML components) as well. For example, issue [112] explains a problem in the output server channel causing a bug in interactive learning. Although this issue is not inside the ML component, it results in failure in the inference of the ML component.
- ***Bugs anywhere in the system:*** This class contains any bugs in all the system’s components, related to ML components or not. An issue of the Rasa project (#4142) [113] is an example of a non-ML bug referring to a problem in the presentation of the output.

Software maintenance is the last part of the software development process known for its high required effort and cost [114]. Corrective maintenance, a type of software maintenance focusing on detecting and fixing bugs, is considered the most important task for delivering reliable software systems [9]. Given that ML-based systems are often more complex than traditional software systems [3], the nature and level of difficulty of bugs occurring in these systems are likely to be very different compared to traditional systems. Since software bugs are at the core of corrective maintenance activities, a good understanding of their characteristics is critical. This study aims to provide a comprehensive characterization of ML and non-ML bugs occurring in ML-based systems from the software maintenance point of view.

Although a number of studies have investigated bugs in ML programs [115, 116], differences between bug types in ML programs [4], and their symptoms and root causes [29, 31], none of them provides detailed information on the characteristics of ML bugs from the software maintenance viewpoint, in comparison with non-ML bugs. This study fills this gap in the literature by providing insights about the characteristics of ML and non-ML bugs in ML-based systems. We also examine the impact of employing ML on the maintenance of software systems. Our study is focused on bugs occurring in ML systems built using the following three most popular ML frameworks: TensorFlow, Keras, and PyTorch. We answer the following Research Questions (RQs):

RQ1. *What are the distributions of the ML and non-ML bugs in ML-based software systems?*

RQ2. *What are the differences between the complexity level of fixing ML and non-ML bugs in ML-based systems?*

RQ3. *What is the difference between needed resources (time-to-fix and developer expertise) for fixing ML and non-ML bugs?*

To answer these questions, we analyzed 386 closed issues gathered from 40 open-source ML-based software systems using *TensorFlow* [117], *Keras* [118], and *PyTorch* [119].

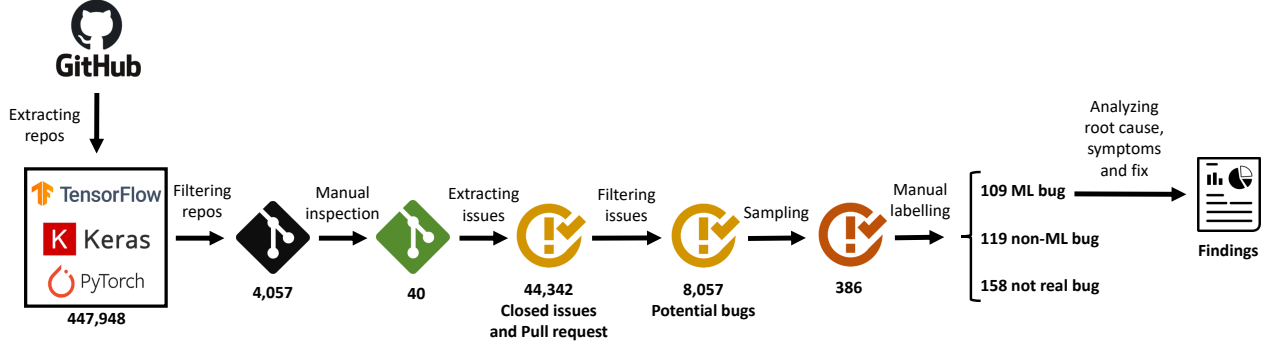


Figure 4.2 High-level view of the followed methodology.

4.3 Methodology

In this section, we describe our methodology to investigate bugs in ML-based software systems answering our RQs. The general process consists of three steps: (1) Collecting repositories, (2) Manual inspection of repositories, and (3) Manual labeling of bugs. Figure 4.2 illustrates the methodology that we followed in our study.

4.3.1 Collecting repositories

To identify relevant ML issues, this study focuses on projects developed using the three most popular ML frameworks: *TensorFlow*, *Keras*, and *PyTorch*. We select them based on their popularity metrics in *GitHub* as shown in Table 4.1, e.g., the number of stars and forks.

With respect to the fact that computer society also considers *GitHub* as the most important resource of open source software systems [120], we use *GitHub* as the main source to extract the needed data for our analysis. Since *Python* is the most popular programming language for ML [121,122], and *TensorFlow*, *PyTorch*, and *Keras* provide *Python* APIs, we also narrow down the search to the repositories developed using *Python* by adding ‘`language:python`’ parameter to the search queries. In *Python*, when a developer wants to use a library, she adds the library to the script using the ‘`import`’ command before its usage. For example, developers add ‘`import keras`’ to the script to use *Keras* in their application. So, we run search queries with ‘`import <ML framework>`’ (*TensorFlow*, *Keras*, and *Pytorch*) to fetch all repositories using these frameworks. Since *Keras* may be imported using ‘`import tensorflow.keras`’, we also search for this command as well. *GitHub* limits the users to access only the first 1000 results of the search API. To overcome this issue, we manage to perform different search queries by adding a parameter to fetch files with the size inside the indicated range (i.e., `size:<min>..<max>`). We define the range for each query in such a

Table 4.1 Details of the selected ML frameworks.

ML Framework	#stars	#forks	#subscribers
TensorFlow	174k	88.3k	3.4k
Keras	58.3k	19.3k	2k
PyTorch	66.7k	18.3k	1.7k
Caffe	33.3k	19k	269
Jax	23.1k	2.2k	504
MXNet	20.4k	6.9k	875
CNTK	17.4k	4.4k	201
sonnet	9.6k	1.4k	51

way that achieves less than 1000 results. Because our query returns an empty list for the repositories using the files greater than 500 *MB*, we divide the whole range between 1 *KB* and 500 *MB* into ranges of 10 *KB*. So, we raise 150,000 search API calls in total, 50,000 for each framework to fetch repositories.

Next, we conduct four filtering steps based on repositories’ metadata to remove personal, inactive, and unpopular repositories, and ones with trivial history (Table 4.2). For mining repositories’ metadata such as the number of open/closed issues, and the number of commits, we use *GitHub* GraphQL API [123] that outperforms API V3 for extracting such information. In other words, we replace several *GitHub* API V3 calls with one GraphQL API call. We use the following exclusion criteria that were employed successfully in previous studies [4,124,125]:

- **unpopular repositories:** repositories with less than 10 stars or 10 forks.
- **personal repositories:** repositories with 1 collaborator.
- **inactive repositories:** repositories without any activity during last year.
- **repositories with trivial history:** repositories with less than 100 commits.

We are left with 4,057 repositories after this step.

4.3.2 Manual inspection of repositories

After collecting repositories and filtering out irrelevant ones, we proceed with manual inspection of them. This is necessary to identify true ML-based systems. In the first step, we selected 30 repositories randomly (10 for each framework) with the highest number of closed issues and Pull Requests (PRs) similar to the methodology followed by Humbatova et al. [4]. We then manually check all 30 repositories to ensure that they are ML-based systems and have at least one ML component. We establish the following exclusion criteria to investigate collected repositories and remove non-ML-based repositories:

Table 4.2 Detailed information about the number of remaining repositories after each filtering out step.

	ML frameworks		
	<i>TensorFlow</i>	<i>Keras</i>	<i>PyTorch</i>
All extracted	144415	98462	205071
After removing unpopular	10480	4108	12556
After removing inactive	4393	2989	7171
After removing repos with trivial history	1633	738	2189
After removing personal	1445	661	1950
Repos with ML keywords	156	488	250
After manual checking top 100	12	15	13
Total extracted repos			447948
Total remained repos			4057
Identified ML-based repos			40

- Repositories that use a language other than English to describe their application, such as [126],
- Tutorial and training repositories which are a set of sample codes or implemented examples of books, such as [127],
- Repositories using ML framework, but not for implementing the functionality of components based on ML algorithms (Figure 2.4), such as [128] that uses *Keras* for data preprocessing,
- Repositories using ML frameworks for testing their functionality or providing examples of their application usage, such as [129],
- Repositories that implement a wrapper for the ML frameworks to extend their functionality, such as [130].

To proceed, the first three authors went through 30 randomly selected repositories and checked them separately, to categorize them into ML-based and non-ML-based, using an open coding procedure [131]. After a meeting to discuss the results, they concluded that only 3 out of the 30 manually checked repositories are ML-based systems. Hence, we added another filtering criterion to decrease the number of false positives. We investigate repositories' scripts for keywords related to defining, training, and evaluating ML models and exclude repositories without any of such keywords. For instance, in *Keras*, '*Sequential*' and '*Model*' APIs are used to define the model, and '*fit*', '*compile*', '*evaluate*', '*predict*', '*train_on_batch*', '*test_on_batch*', '*predict_on_batch*', and '*run_eagerly*' APIs for training and evaluating models [132]. We followed the same approach for *TensorFlow* and *PyTorch*, and reported

details in our replication package [133]. By running this filtering step, we retained 894 repositories.

We then selected a sample of 100 repositories for each framework, in total 300. The top 100 repositories with the highest number of closed issues and PRs were selected for each ML framework, similar to the methodology followed by Humbačová et al. [4] to be checked manually. To label repositories into ML-based or non-ML-based, the first three authors classified the first 15 repositories of each ML framework (45 in total) independently. To assess inter-rater agreement among them, we used Fleiss’ kappa [134] like in similar works [135, 136], and obtained an inter-rater agreement of about 33%. Next, the three authors met to discuss the conflicts and resolve them. Afterward, we repeated the labeling process for 45 repositories and the agreement rate reached 89.9%, which is acceptable to keep going through the rest of the repositories (Fleiss’ kappa agreement greater than 81% is interpreted as almost perfect agreement [137]). So, we proceeded to label the rest of the repositories, and the three authors labeled all 300 repositories and achieved 87.7% agreement. In the end, 40 repositories were identified as ML-based. The list of repositories is available in our replication package [133]. The list encompasses projects such as AirSim¹ a simulator for drones and cars, FaceSwap² a deep learning tool to swap faces on videos or DeepSpeech³ a speech-to-text engine. Besides, with respect to the fact that we aim to compare characteristics of bugs in ML-based vs non-ML-based systems in this study, we do not make any distinctions between various types of ML algorithms (CNN, Transformer, RNN, etc.).

4.3.3 Manual labeling of bugs

Since our goal in this study is to characterize bugs in ML-based systems, following the existing work [115, 138], we mine the 40 identified ML-based repositories and extract closed issues and merged Pull Requests (PR) showing a bug-fix. We choose such PRs because 1) bugs mentioned in such PRs have been already accepted and then got fixed and 2) these PRs usually have more comprehensive information about the fixed bugs (e.g., code changes, links to related issues, and discussions among developers), which facilitates understanding the bugs. To identify PRs with the purpose of bug-fixing from repositories, following the existing study [115, 139], we collect PRs whose tags/titles include at least one bug-relevant keywords (i.e., fix, defect, error, bug, issue, mistake, incorrect, fault, and flaw). Similarly for closed issues, as users raise *GitHub* issues for several purposes (e.g. asking questions, enhancement, feature request, and reporting bugs) and assign some tags to represent their goal, closed

¹<https://github.com/microsoft/AirSim>

²<https://github.com/deepfakes/faceswap>

³<https://github.com/mozilla/DeepSpeech>

issues with at least one of the mentioned keywords in tags or titles were extracted.

We initially extracted a total of **44,342** closed issues and merged PRs. Out of them, we achieved **8,057** with the bug-relevant keywords. Following previous works [20,140], to ensure a 95% confidence level and a 5% confidence interval, we randomly sample **367** issues/PRs. With respect to the different number of raised issues/PRs in each ML-based repository of our dataset, we use the Stochastic Universal Sampling (SUS) [141] method to obtain a fair set of sampled artifacts and prevent any bias against the repositories with the small number of issues/PRs. In other words, we extract bugs from each repository, based on the ratio of each repository's issues/PRs to the total number of extracted ones. We round calculated values to the closest greater integer number, and that left us with **386** issues/PRs at the end.

In the next step, we inspect the extracted issues/PRs manually to identify their type as *ML* or *non-ML*, their root causes and symptoms following an open coding procedure [131]. To this end, we use a two steps manual labeling: 1) to categorize the issues/PRs into ML and non-ML and, 2) to classify the ML ones based on their root causes and symptoms. To categorize the issues/PRs into ML and non-ML, each of the first three authors went independently through the first 40 issues/PRs (almost 10% of all sampled issues) as a pilot analysis, and we achieved a 44.8% inter-rater agreement Fleiss' kappa. To identify the main reasons for disagreements and resolve them, two meetings were held and a clear criterion was agreed upon for each class. Generally, we label issues/PRs affecting the quality of ML components as ML bugs and the rest as non-ML bugs. During the manual labeling, several collected closed issues turned out not to be real issues. As an example, some issues are generated automatically by the repository's bot [142] or closed by it, because nobody replied to the issue [143]. So, we added an additional group as '*not real bug*' referring to the issues which can be classified as neither ML nor non-ML, satisfying one of the following rules:

- issues generated automatically by a bot.
- issues which are end-user questions (users do not know how to handle the problems) or user's mistakes.
- issues that are closed without fixing (not having enough information from the issue report or having a rejected PR).
- issues that are not reproducible.
- issues that are unclear or without explanation.

Table 4.3 Detailed information about labeled bugs.

Framework	Issue	PR	ML bug	non-ML bug	Not real bug
TensorFlow	88	32	18	41	61
Keras	76	11	8	46	33
PyTorch	118	61	83	32	64
Total	282	104	109	119	158

We also filter out the PRs that are found to be irrelevant to bug fixing (labeled as *no real issue*). Moreover, some PRs fixed multiple bugs, and we, therefore, labeled each of them as an individual bug where applicable, similar to the existing work [115, 139].

Then we labeled those 40 issues/PRs again and achieved an 89.6% agreement (interpreted as almost perfect agreement [137]), which is reasonable to keep labeling the rest of the issues/PRs. To label the remaining issues/PRs, we label every 100 bugs, have a meeting to discuss the results, explore the main reasons for disagreements, and resolve them for the next parts. Eventually, we achieve **89.7%** agreement, resulting in **109** ML bugs (28.2%), **119** non-ML bugs (30.8%) and **158** not real bugs (40.9%). Table 4.3 illustrates the detailed information regarding manual labeling of issues as ML bugs or non-ML bugs.

Next, we carry out the second part of bug labeling to indicate their **root cause** and **symptoms**. Since many of our selected repositories used DL models, we employ the recent classification of root causes and symptoms of bugs in DL systems introduced in [31] which extended a former study [29]. For root causes, we have: *absence of inter API compatibility* (AIAPIC), *absence of type checking* (AOTC), *API change* (APIC), *API misuse* (APIM), *confusion with computation model* (CWCM), *incorrect model parameter or structure* (IMPS), *structure inefficiency* (SI), *unaligned tensor* (UT), and *wrong documentation* (WD). Table 4.4 represents the detailed description of root causes of ML bugs in ML-based systems. The types of symptoms are as follows: *bad performance*, *crash*, *data corruption*, *hang*, *incorrect functionality*, and *memory out of bound*. We mainly rely on the definitions presented in their original sources [29, 31]. To label the bugs, we first selected 20 ML bugs out of the 109 ML bugs and labeled them as a pilot step, achieving 30% and 84.9% agreements based on Fleiss’ kappa for root causes and symptoms, respectively. We again meet two times to identify the major reasons for disagreements and reach a consensus on the labels’ concepts and definitions. Then, labeling the first 20 ML bugs again resulted in an 86.5% inter-rater agreement for root causes and 89.7% for symptoms. For the rest of the bugs, we inspected them in three rounds. In each round, we label around 1/3 of the bugs, then meet to discuss the disagreements and resolve them. In the end, we achieved **88.4%** and **97.2%** agreement for root causes and symptoms of ML bugs.

Table 4.4 The identified root causes of ML bugs in open-source ML-based software systems.

Bug root cause		Description
Absence of Inter API Compatibility (AIAPIC)		Inconsistency of two different types of libraries. For example, issue [144] emerges from a problem in using ‘ <i>pytest</i> ’ in the application developed using PyTorch.
Absence of Type Checking (AOTC)		Data type mismatch in calling API methods. As an example, issue [145] stems from the absence of type checking of “ <i>trainer.logger</i> ” in different places.
API Change (APIC)		Releasing new versions of ML frameworks that their APIs are not compatible with their previous versions. Issue [146] is considered as an example of APIC that is raised because of changing APIs of PyTorch Lightning in version 1.2, compared to older versions.
API Misuse (APIM)		Trying to use an API developed by an ML framework, without a sound understanding about it.
Confusion with Computation Model (CWCM)		Confusion about the function of an API of the ML framework leading to the misuse of the assumed computation model by the ML framework.
Incorrect Model Parameter or Structure (IMPS)		Constructing the ML model such as creating an ML model with incorrect structure or using inappropriate parameters. For instance, issue [147] raises because of ignoring parameters’ default value of the early stopping method.
Structure Inefficiency (SI)		Problems in the modeling step of ML component/software. Although this category is similar to IMPS, their symptoms are different. In fact, IMPS leads to system crashes, however SI results in bad performance of the system. Issue [148] is an example that its root cause is using incorrect training parameters resulting in bad performance of the system.
Unaligned Tensor (UT)		Trying to build a computation graph in an ML process without providing input data satisfying input specification of the ML framework API. For instance, issue [149] stems from the shape mismatch of the intermediate tensors.
Wrong Documentation (WD)		Incorrect information presented in the formal documentation of the library. As an example, issue [150] is related to the unclear explanation in the document of the LightningAI library for using early stopping API.

4.4 Result

In this section, we provide and discuss empirical results for each RQ. The materials and all collected data used for this study are publicly available in our replication package [133].

4.4.1 RQ1: Distribution of bugs in ML-based Systems

Our goal in RQ1 is to evaluate the distribution of ML and non-ML bugs in ML-based software systems. Results are presented in Figure 4.3. Manual labeling of a selected sample (with a 95% confidence interval) of potential bugs reveals that 40.9% of them are not real bugs. This finding suggests that almost half of the issues/PRs reported in the collected repositories are not real bugs. Some of these issues are identified as users' questions/mistakes while others lack sufficient description to be considered as a bug (e.g., not reproducible or have unclear description/not enough information). This finding aligns with a previous research conducted by Anvik et al. [151] which found that only 58% of the issues raised for Eclipse and 44% of the issues raised for Firefox have been either fixed or considered for fixing. On the other hand, Long et al. [152] carried out research on the performance and accuracy bugs within ML frameworks (e.g. TensorFlow, Keras, PyTorch, etc) and reported only 3% of the raised issues as bugs without sufficient information or non-reproducible. This discrepancy can be explained by the fact that our study encompasses a broader range of bug types in ML-based systems, beyond just accuracy and performance issues, which seem to be more easily identifiable (and likely fixed) than other types of bugs. Furthermore, we focus on the bugs within ML-based systems rather than inside the ML framework itself. Moreover, it is worth noting that localizing and identifying the root causes of bugs inside ML-based systems are likely more challenging than bugs inside ML frameworks, mainly due to the utilization of various libraries in the implementation of ML-based systems. Among the remaining bugs, our results demonstrate that 47.8% of the real bugs in ML-based systems are dealing with ML components (Figure 4.3a). Although ML components of software take just about 10% of development time [153], it represents roughly half of the real issues encountered in a typical ML-based system. These findings emphasize the critical need for automatic testing tools for ML components and ML-based software systems.

Finding 1. A large number of issues in ML-based systems either turn out to be users' questions/mistakes or did not provide enough information to decide whether it is a bug or to help fix it. Moreover, although almost 10% of development time is consumed for ML components, nearly half of the "real" bugs in ML-based systems are related to ML components.

As mentioned in the methodology, to classify the root cause and symptom of our bugs, we employed the categories introduced by Islam et al. [31]. Figure 4.3b presents the indicated root causes. The most important observation is that 72.5% of the identified ML bugs could not be classified in any of the 9 root causes defined in [31], so we classified them

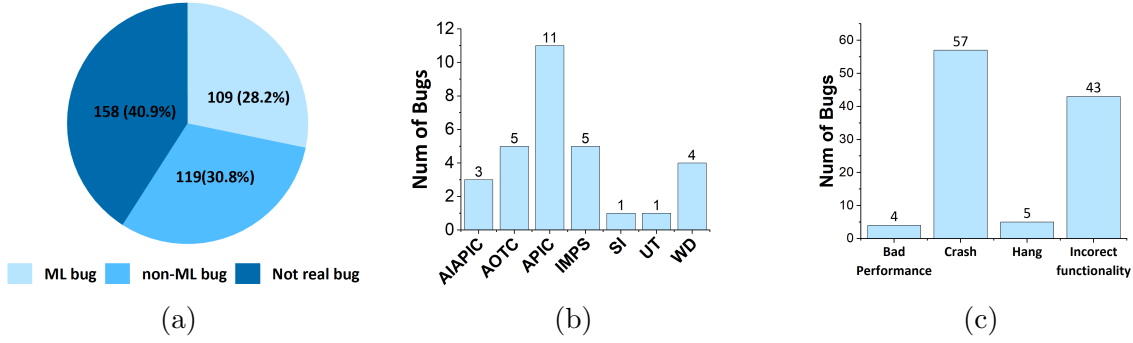


Figure 4.3 Bug in ML-based systems: (a) distribution of issues, (b) root causes of the ML bugs, and (c) symptoms of ML bugs.

as *Other* (we do not report this category in Figure 4.3b). The reason is that the defined category focused on root causes related to APIs, models (structure and parameter), and documentation. Therefore, for further investigation of root causes, we went through the bugs categorized as *Other* using the 11 labels provided by [154] for root causes of bugs in DL libraries. We followed a similar methodology to what is done initially for root causes. We found the following results: Dimension mismatch (1.3%), Processing (15.2%), Algorithm (15.2%), Corner case (2.5%), Logic error (36.7%), Configuration error (19.0%), Concurrency (10.1%) while we found no occurrence (0%) for Type confusion, Inconsistency, Referenced types error, and Memory. We refer the reader to [154] for a full description of the categories. The root cause of the majority of bugs is identified as Logic error, as it covers bugs that occur in the logic of the code like incorrect program flow or wrong order of actions. This also includes bugs that lead to wrong calculations of gradients. Since ML code is highly dependent on configurations (like hyperparameters), the next largest portion of root causes is Configuration error which represents bugs caused by wrong configurations. However, both classifications overlooked problems raised during training and using trained models. It looks like that, by increasing the development and usage of ML-based software systems, we need a new category of bug’s root causes for such systems.

Categorizing bugs based on their symptoms, as reported in Figure 4.3c, shows that crash with 52.2% is the most prominent symptom which is in accordance with previous studies [31]. However, the ratio of ‘incorrect functionality’ rises to 39.4% compared to 12% in the previous report [31]. The reason can be an increase in the application of ML components in software systems from 2019 (the previous study [31]) to 2022.

Then, we categorized ML bugs based on the taxonomy of faults in DL systems introduced by Humbatova et al. [4]. Their taxonomy consists of 5 top-level categories that we used to label ML bugs: “Tensors&Inputs”, “Model”, “Training”, “API”, and “GPU Usage”. The results

are reported in Figure 4.4a. Overall, the “Training” category is the most frequent type of bug with 52.7% which is similar to the original study [4] where authors reported 52.5% of issues related to the training. It also echoes [152] where accuracy/performance bugs are mainly concerned with the training phase (although it concerns bugs inside DL frameworks). The next prevalent type of bug is Tensors&Inputs which deals with problems related to the wrong shape and type/format of the data. Similar to root causes, here, we encountered 9 bugs that did not fit into any of the 5 categories. So, we have added the ‘other’ category to cover them. These bugs include issues happening during the loading/saving of a trained model, deployment of the trained model, model inference, checkpoints, and monitoring of a trained model. Figure 4.4b shows the distribution of bugs in the “other” category. The “loading/saving of a trained model” category refers to the issues which may occur when developers try to save a trained model to be able to use it in another program or load a saved model which is trained in the previous steps (e.g., issue [155]). Model deployment issues are related to the possible problems in deploying a trained ML model on a specific platform. For instance, issue [156] tries to fix an architectural problem of the model to be able to deploy on ONNX. Problems categorized as inference issues refer to the errors occurring in the prediction step of the ML component. Issue [157] is an example of an inference issue explaining a problem in the implementation of the prediction step of the trained model. Checkpoint issues are explained as the problems that developers faced while trying to save the ML model during its training process to be able to continue the training process from checkpoints, in case of any crash/stop in the model training process. Issue [158] is considered an example of the model checkpoint issue. Monitoring issues are related to the problems of monitoring a deployed model.

Finding 2. Because of the increasing usage of ML in different domains and the increasing maturity of ML-based software systems over time, we need to revise existing ML bug classifications and add new possible categories to be able to classify root causes/symptoms of ML bugs. Moreover, most of the issues of ML components seem to originate from the training phase.

4.4.2 RQ2: Complexity of bug fixes in ML-based systems

To compare the complexity of bug fixes for different types of bugs in ML-based systems, we used metrics that are utilized successfully by previous studies [159, 160] including the number of commits to fix the bug, number of changed files, number of changed Lines of Code (LOC), entropy of changes, number of comments on the issue, number of comments on the

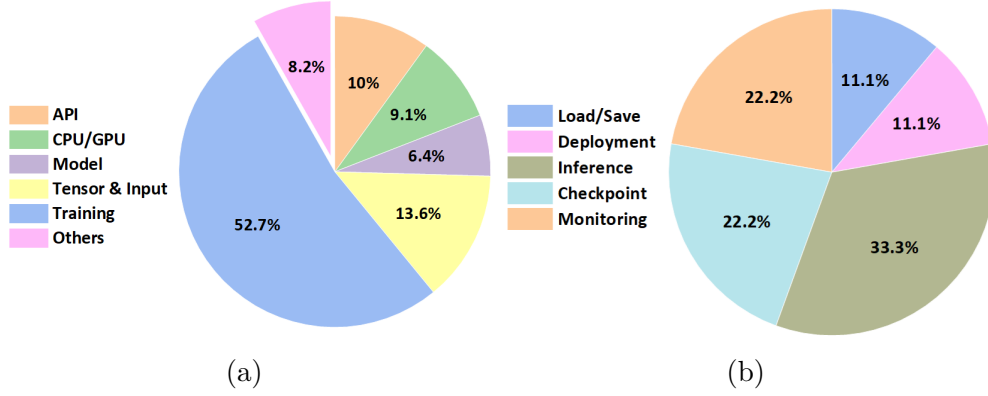


Figure 4.4 Classification of bugs in ML-based systems (a) based on the taxonomy provided by [4], (b) bugs falling into ‘other’ category.

bug-fix related to the issue, number of users who discussed the issue, and number of users who discussed and collaborated on bug-fix. Figure 4.5 and Figure 4.6 report the results.

To ensure a statistically significant difference between metrics for ML and non-ML bugs, we also run statistical tests. As we make no assumptions on the distribution of data and since the variance of data is different, *Mann–Whitney U* test is the recommended test [161]. We used the two-sided test of the *Mann–Whitney U* test which tests the null hypothesis that the probability of a given metric of ML bugs being greater than non-ML bugs is equal to the probability of a given metric of non-ML bugs being greater than ML bugs. Finally, as reporting simply p-value might be misleading [162], we also computed effect size to assess the magnitude of the differences [163]. Moreover, “Cliff delta” test is a recommended test for calculating effect size for data with non-normal distribution [164]. For both tests, we used their official implementation in the R statistical environment (with “effsize” library for Cliff’s delta). Results are presented in Table 4.5.

As we can see in Table 4.5, regarding *p*-value, using the widely accepted 0.05 significance threshold, the results show that differences in the number of commits, number of changed LOC, entropy of changes, number of collaboration on fix, and number of comment/collaborator on fixes are significant while the number of changed files, number of collaborators on issues, number of comments on both issues and fixes, and number of comment/collaborator on issues are not significant. To interpret the effect size, we use the traditional scale associated with Cliff’s delta [165] that is $|d| = 0.147$ is small, $|d| = 0.33$ is medium and $|d| = 0.474$ is large. All *statistical* significant tests previously described are also *practically* significant with a *small* effect size, except *number of commits* which results in a *medium* effect size. Moreover, the number of commits, changed LOC, number of collaborators on a fix and entropy features

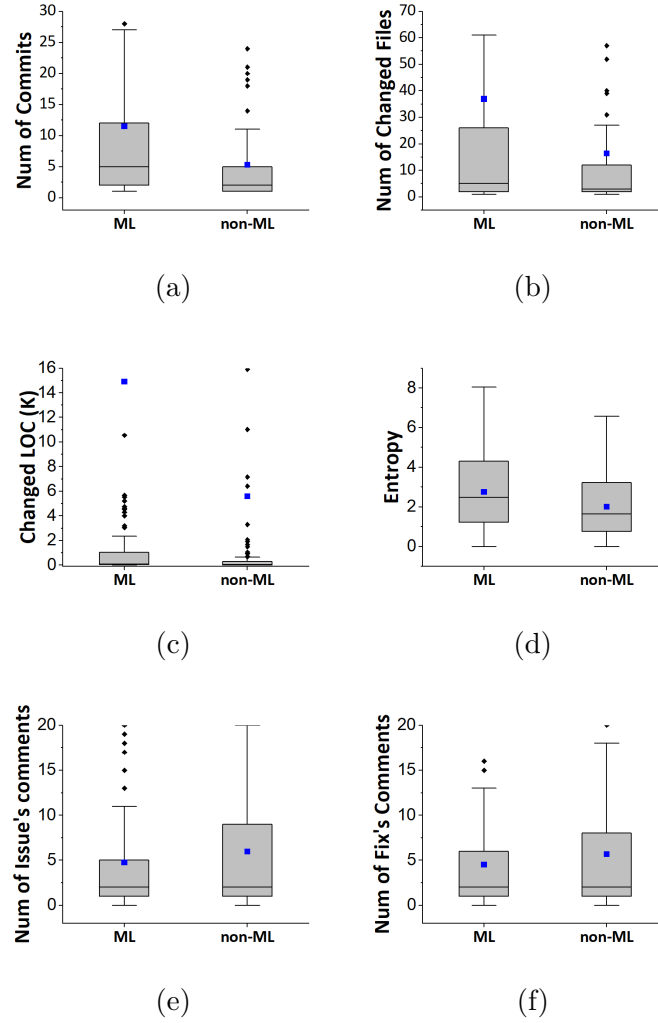


Figure 4.5 Complexity of Fixes: (a) number of bug-fixes commit, (b) number of bug-fixes changed files, (c) number of bug-fixes changed LOC, (d) bug-fixes entropy, (e) number of comments on an issue, (f) number of comments on bug-fix.

have a positive effect size which indicates that this feature is more prevalent in ML bugs than non-ML bugs. Besides, the negative effect size of the number of comments/collaborators signifies that this feature tends to be higher in non-ML bugs compared to ML ones. According to obtained results, ML bugs tend to require more commits as well as more changed LOC compared to non-ML bugs. They also require more collaborators to be fixed as well. Yet, the presence of numerous outliers in commits for non-ML bugs (Figure 4.5a) shows that some non-ML bugs can prove harder to address. Regarding the number of changed LOC, there seem to be multiple outliers in both cases (Figure 4.5c), which can imply that this feature is independent of the nature of the bug, and is basically bug-dependent, although there is a significant difference.

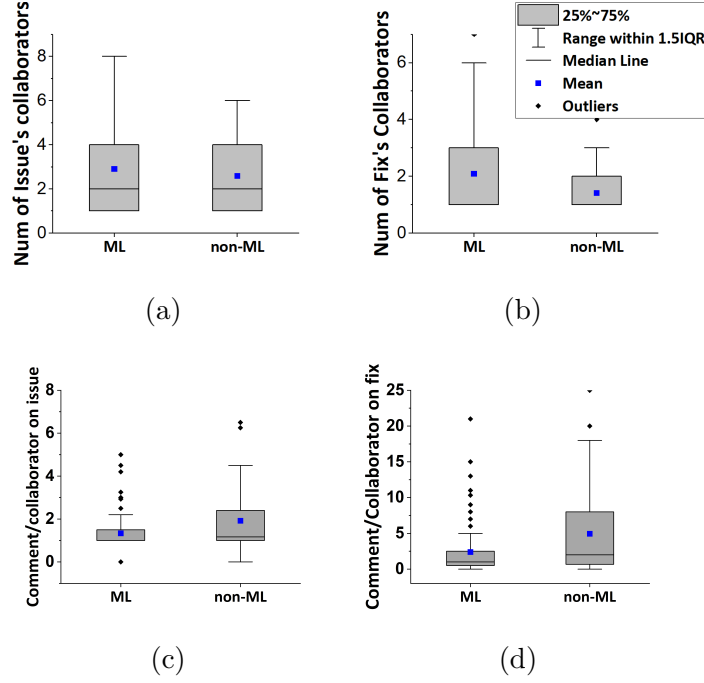


Figure 4.6 Complexity of Fixes: (a) number of collaborators on an issue, and (b) number of collaborators on bug-fix, (c) number of comments/collaborator on issue, (d) number of comments/collaborator on a fix.

Table 4.5 Detailed result of *Mann-Whitney U* and *Cliff's delta* test.

Metric	<i>p-value</i>	<i>effect size (d)</i>
# commits	$< 1e-3$	0.337^M
# changed files	0.080	0.133 ^N
# changed LOC	0.033	0.164^S
# comments on issue	0.347	-0.072 ^N
# comments on fix	0.441	-0.058 ^N
# collaborators on issue	0.904	-0.009 ^N
# collaborator on fix	$< 1e-3$	0.272^S
# comment/collaborator on issue	0.086	-0.131 ^N
# comment/collaborator on fix	0.014	-0.192^S
entropy	0.008	0.204^S

^M medium effect

^S small effect

^N negligible effect

To give a picture of the number of comments on issues and fixes, we provide the average number of comments per collaborator (Figures 4.6c and 4.6d). Results reveal that collabora-

tors put more comments on average on non-ML issues, compared to ML ones. As Table 4.5 represents, the difference between the number of comments/collaborators on issues in ML and non-ML bugs is not significant, but it is significant for fixes with small effect size ($|d| = -0.192$). One possible reason behind this discrepancy may be the fact that developers who put comments on ML bugs collaborate passively. In other words, collaborators just put a few comments on fixes and then stop collaborating on fixes actively, which may be because of their limited knowledge of ML. For instance, there are 8 different collaborators on a PR (#4309) of *lightning-AI* project which is the fix of an issue (#3660). However, according to activities, only 2-3 of them are collaborating on fixing the issue.

For the number of issue’s collaborators (Figure 4.6a), there is no statistical difference in our case which signifies that while ML bugs require more developers to be fixed (Figure 4.6b), there is a similar number of collaborators discussing on the issue. As such, ML bugs tend not to require significantly more collaborators (experts) to be identified and diagnosed in ML-based systems. Regarding the number of comments on different issues (Figure 4.5e and Figure 4.5f), while not being statistically significant, the distribution of data reveals that the number of comments in non-ML bugs is on average higher than ML ones. The reason behind it can be that a higher number of developers are more skilled in general programming than in ML. However, there seems to be a high number of outliers, which means that while on average ML bugs receive fewer comments than non-ML ones, some bugs require extensive discussions, potentially harder to pinpoint the root cause of the problem for instance. Finally, ML bugs tend to significantly have higher entropy than non-ML bugs (Figure 4.5d). This means, in general, ML bugs happen in some specific files whereas non-ML bugs tend to be more scattered across the project’s files.

To further investigate why ML bugs need more change iterations (i.e., more commits), we performed a thematic analysis of the bug-fixing PRs using Latent Dirichlet Allocation (LDA) from the Gensim library [166]. We removed any kind of text known as PR template (repetitive text in all PRs) which could bias the model in our 109 ML bugs’ PRs. We then applied LDA for topic modeling based on Mallet [167] to obtain the relevant number of topics. The algorithm returned the best coherence score (in parenthesis) for a number of topics of 7 (0.6326), 26 (0.6297), and 20 (0.6218). We chose the case of the 7 topics, as it yielded the best score and a higher number of topics might mean the model started grouping issues based on the application rather than the proper cause of the issue (for instance, in the 20 topics proposed, one of them is based on the keywords “state”, “action”, “intent” or “slot” which are keywords heavily used in one of the applications we collected). Reviewers then analyzed the keywords of the 7 topics and the issues labeled with the given topic to come up with topic labels. The final topics are: Test related (15.8%), Requirements and

Dependencies (19.3%), Training Accelerator and Model checkpoint (18.4%), Scripting Bugs (14.9%), Model Bugs (16.7%), Training Bugs (7.9%), Documentation (7.0%). The topics that are the most represented are commits linked to Requirements and Dependencies and Training Accelerator and Model checkpoint with over **18%** each, while the least represented topics are the Documentation and Training bugs with around **7%**. Interestingly, the largest part of the issues for ML bugs do not relate directly to ML (Requirements/Dependencies), nonetheless, such bugs are expected in ML systems as they often rely on multiple libraries and frameworks to work.

Finding 3. In ML-based systems, ML bugs have significantly different characteristics from non-ML ones: they need more commits, a higher number of LOC, and more collaborators involved. Analysis of the ML-related commits shows bugs based on Requirements/Dependencies are the most widespread while Documentation is the least represented.

4.4.3 RQ3: Needed resources for fixing bugs in ML-based systems

To measure the number of resources spent for fixing bugs in ML-based systems, we consider two metrics: time-to-fix and expertise level of developers who fix the bugs (which were used in previous studies as well [159, 168]). To indicate the time-to-fix, we used the time range between the issue opening and closing date [169]. For the expertise level of developers, we consider the number of commits done in all other repositories by the developer who fixed the bug, exactly before fixing each issue of our dataset. Therefore, in our list, some users appeared multiple times with different numbers of commits depending on the date of the bug-fix. We kept only the highest expertise level observed (that is, the latest update of the number of commits). We added all users gathered that way whether they only were present on ML, non-ML, or both types of bugs in our dataset. Similar to RQ2 (Section 4.4.2), we use *Mann-Whitney U* and *Cliff's delta* test.

Our results reveal that fixing time is not statistically significant ($p\text{-value} = 0.423$, $d = 0.061$ which is negligible), that is fixing ML bugs requires no more time compared to non-ML ones. This does not necessarily mean that ML bugs are as "easy to fix" as non-ML ones since other metrics we calculated such as the number of commits or the number of collaborators working on a fix were significantly higher for ML bugs. As we focused on repositories containing ML components, this non-significant difference might mean that developers are experts in solving both types of issues, so all issues take almost the same amount of time. This observation can be further emphasized when comparing the expertise level of users.

In all cases, the tests are not statistically significant which means that the number of prior commits of developers working on ML or non-ML issues is similar. This reflects on the observation made previously: as developers working in those repositories are used to both ML and non-ML issues, they are likely to be experts at the same level, even if in our dataset we flagged them as working on only one type of bug. Yet, some users worked on both types of issues. To assess the impact they have, we removed them from ML/non-ML issues groups and retested our data again, which resulted in still no significance even if the metric decreased. This means that users having some expertise in both types of bugs, in our dataset, do not change the outcome, implying that other users working on a single type of issue take as much time as the other group. Similarly, removing outliers (that is, users that have an overly high number of commits) does not affect the results.

Finding 3 seems to suggest that ML bugs are more complex in terms of bug fixing than non-ML bugs. Nonetheless, the fact that we find no evidence of statistically different expertise levels or time-to-fix between ML and non-ML bugs in this study would seem to contradict Finding 3. As we do not have access to the *effective time* spent on the fix, it can be the case that ML bugs were given priority and the actual time spent on fixing the non-ML issues is lower than the time-to-fix we have access to (date of fix and date of beginning). While there might be some threats to the validity of considered criteria used to measure needed resources for fixing ML and non-ML bugs (expertise level of developers and time-to-fix), no other information was available to compute such a metric. We examined the correlation between entropy and time-to-fix using the *Spearman* correlation method from *pandas* Python library. Results reveal that time-to-fix has a moderate positive correlation (based on correlation formal definition [170]) with entropy (0.42). With respect to the fact that entropy is considered a measure to show the complexity of changes [160], it can imply that time-to-fix is a good candidate to show the needed effort for fixing issues.

Because of the negative impact of bug reopening on software quality [171], we examined the reopening rate of ML and non-ML bugs from the studied system. We aim to understand if developers experience higher bug reopening rates when dealing with ML bugs than they do for non-ML bugs. Results show that the reopening rate of ML issues (5.45%) is a bit higher than that of non-ML bugs (4.2%). We assessed the statistical significance of this result using the *Chi-squared* test. We leveraged the *scipy* Python library to run this test and obtained a p -value of 0.894. Meaning that there is no significant difference between the reopening rate of ML and non-ML bugs. Besides, the rate of issue reopening in ML-based systems with 4.8% is a bit less than the one reported for traditional software systems (between 6% and 10%) by Zimmermann et al. [172].

Considering the major impact of code reviews on the software quality [173], we examined the rate of developers’ invitations to review PRs fixing ML and non-ML bugs. We found that in 49.05% of ML bugs, developers were invited to review PRs compared to only 23.36% for non-ML-bugs. This finding suggests that higher attention is given to the quality of bug-fix changes when dealing with ML bugs. We explain this result by the fact that fixing ML bugs requires ML knowledge which may not be mastered by all of the developers on the team. We also investigated the number of PRs’ reviewers in ML and non-ML issues since it has a positive correlation with the time-to-fix [174]. As is shown in Figure 4.7, the number of developers who reviewed bug-fixing PRs for ML bugs is higher than the number of developers who reviewed bug-fixing PRs for non-ML bugs. Furthermore, results of *Mann–Whitney U* and *Cliff’s delta* tests (similar to 4.4.2) indicate that the difference between the number of reviewers involved in fixing ML and non-ML issues is significant ($p\text{-value} < 1e\text{-}6$, $d = 0.378$ which is interpreted as medium effect size).

Moreover, we checked the number of bug-fix PRs merged without review in our collected bugs. We found that 37.8% of merging bug-fix PRs for non-ML bugs have been accomplished without conducting any review compared to 13.7% for ML bugs. This lower number of non-reviewed bug-fixing PRs may suggest that development teams are more cautious when dealing with PRs related to ML bugs (reviewing 87.3% of them), which may also explain the lower rate of bug reopening observed for ML bugs.

Finding 4. There is no statistically significant difference between ML and non-ML on expertise level and time-to-fix when fixing bugs in ML-based systems. However, since the time-to-fix an issue does not capture all the effort that went into fixing the issue because it doesn’t necessarily account for preparatory work done before starting editing the code, future work should investigate the cost of fixing ML bugs in more detail.

4.5 Discussion

We found that a large number of issues in ML-based systems either are users’ questions/mistakes or do not come with enough information to decide whether it is a bug or to help fix it. Moreover, although almost 10% of development time is spent on ML components, nearly half of the “real” bugs in ML-based systems are related to ML components. Given the increasing usage and maturity of ML-based software systems over time, we believe that the software engineering community needs to revise existing ML bug classifications and add new possible categories to be able to classify root causes/symptoms of ML bugs. Our results revealed

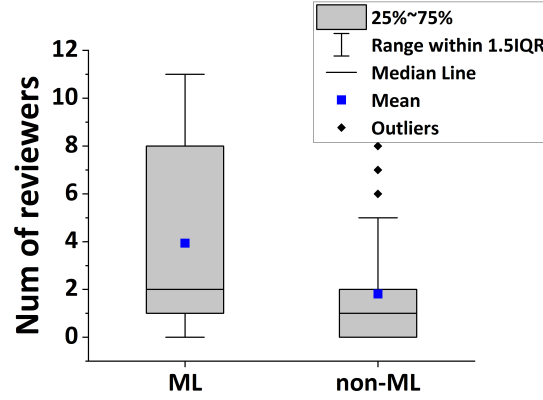


Figure 4.7 Number of developers who review PRs that fixed ML and non-ML bugs

that ML bugs have significantly different characteristics from non-ML bugs, in ML-based systems: they need more number of commits, a higher number of LOC, and more collaborators involved, i.e. they seem to need more revision on average before the bug is fixed. The analysis of ML-related commits shows that bugs related to Requirements/Dependencies are the most widespread while Documentation bugs are the least represented. We did not find any statistically significant difference between ML and non-ML on the level of expertise of developers involved in their correction and the time taken to fix them in ML-based systems. However, the time-to-fix of an issue does not capture all the effort that went into fixing the issue because it does not necessarily account for preparatory work done before starting to edit the code.

Our work can help in planning and forecasting quality assurance activities for ML-based systems. Based on our findings, the maintenance team can allocate appropriate resources for detection and correction of bugs. Our findings show where and how ML-based system developers should invest their efforts to have the most efficient software maintenance (lowering the cost of maintenance). Similar to previous research works (e.g., compiler bugs in DL systems [115]), we shed light on the root causes and symptoms of bugs in open-source ML-based systems. These findings can be used to facilitate the development of (automatic) debugging tools for ML-based software systems [175]. Furthermore, our findings indicate the necessity of adopting novel techniques for the prediction and detection of ML bugs, given their distinctive characteristics and distribution of ML bugs in comparison to non-ML bugs, as shown in this study.

It should be taken into consideration that applying ML in software systems results in generating new types of bugs which do not exist in traditional software systems. On the other hand,

although developing ML components take just about 10% of the ML-based systems development time [153], it represents roughly half of the real issues that a typical ML-based system will face. That is, ML components are more error-prone causing more serious suffering in comparison with non-ML ones. From the software maintenance viewpoint, it is obvious that developing ML components places a heavier financial burden on the software development process, with respect to their higher proneness to bugs. As a software maintenance task, this observation emphasizes the need for automatic testing tools for ML components and ML-based software systems. A number of studies on the automatic bug detection [81, 82], bug localization [101], debugging [109], and bug repair [95] in ML-based systems that have been carried out during the last few years showed that there is an increasing need for automatic testing tools for ML-based systems.

Based on our dataset of ML bugs, one can develop an automatic approach to predict the type of bugs in ML-based systems, as the most recent taxonomy [4] does not cover deployment issues. By increasing development and usage of ML-based software systems in various new areas, developers may face new categories of bugs which raises demand for revising existing bug’s taxonomy and adding new discovered bugs to them. In other words, a novel taxonomy of bugs that covers issues related to all stages of development, deployment and maintenance of ML-based software systems is necessary. Zhang et al. [29] is considered as one of the first studies that provided a list of root causes and symptoms of various bugs in ML-based systems. Shen et al. [115] also studied bugs related to ML-based systems from ML compiler point of view. In fact, they studied bugs in ML compilers (such as TVM⁴, Glow⁵, and nGraph⁶), not in the ML-based systems. Tomban et al. [176] also generated a list of bugs which may occur inside ML frameworks. Therefore, it is obvious that understanding about bugs in ML-based systems is still progressing. Accordingly, we would be required to update ML bug’s root causes, based on the newly added categories of ML bugs.

We believe there might be some specific metrics (compared to the studied metrics) to measure the complexity of ML components. There exist some studies reviewing complexity of the ML components from different viewpoints. As an example, Hu et al. [177] studied the complexity of DL models as the complexity of the problem that the DL model can express. They explained that model complexity can be measured based on various factors including model framework (feedforward neural network, convolutional neural network, etc), model size (number of parameters, number of hidden layers, etc), model optimization process (objective function’s form, learning algorithm, hyperparameters, etc), and the complexity of the

⁴<https://tvm.apache.org/>

⁵<https://ai.meta.com/tools/glow/>

⁶<https://www.intel.ca/content/www/ca/en/artificial-intelligence/ngraph.html>

data used for model training (data distribution, data dimensionality, information volume, etc). Zhang et al. [14] also represents model relevance as an ML testing property that checks the complexity of the model and tries to prevent model from overfitting. When the complexity of the model is higher than the problem, the model overfits to the training data and can not generalize. On the other hand, Yao et al. [178] carried out an empirical study to show the relationship between model complexity and model performance. They introduced complexity as the ability to control data preprocessing, feature selection, classifier selection and parameter tuning. Their results showed that models with higher complexity (more dimension to be controlled) gain better performance and classifier selection plays the most effective role in achieving higher performance. Therefore, ML engineers should be careful about the model complexity and the type of classifiers that they use.

With respect to the fact that about 70% of software development costs belong to software maintenance [114], automatic bug detection, and debugging tools improve the quality and speed of software development leading to a decrease in the software maintenance cost and accordingly software development, significantly [179].

4.6 Threats to Validity

We now discuss threats to the validity of our study.

4.6.1 Construct validity

Limitations of our approach could come from: 1) keywords used for extracting repositories, and 2) choice of exclusion criteria for repositories and issues. The rest of the approach (bug filtering, issues classification of symptoms/root cause, statistical sampling...) is based on existing works. For keywords, we used ‘import <ML framework>’ and ‘language: python’ which is sufficiently general to catch a lot of repositories and is more likely to generate False Positive (repositories that we would end up discarding) rather than False Negative. Exclusion criteria were chosen after some preliminary discussions among authors to weed out irrelevant repositories. Raters further agreed to flag issues/repositories they were not sure about, to allow for careful discussion in order to avoid incorrect labeling.

4.6.2 Internal Validity

The first source of internal validity is the manual checking of repositories and issues. To alleviate this threat, the first three authors (one Ph.D. and two Ph.D. candidates who are practitioners of ML-based systems) selected 300 repositories, and 386 issues, and then labeled

them. After several meetings, we reached an agreement on the labeling process and the rules to differentiate between bug types. Another internal threat to validity is using sampling for gathering issues from the extracted repositories. To mitigate the sampling bias, we did the sampling with 95% and 5% of confidence levels and confidence intervals, respectively, which are the most common sampling in the Software Engineering community. The next threat is the fixes that we consider for each issue. In general, it is supposed that each PR works on one specific issue. But in some cases, developers use one PR to fix more than one issue. So, in a few cases, we could not separate the changes that are exactly related to the studied issues which may impact our results.

4.6.3 External Validity

The main threat to external validity is the selected ML frameworks. We studied ML systems built using *TensorFlow*, *Keras*, and *PyTorch*, because these frameworks are the most popular ML frameworks in GitHub, with the highest number of stars/forks among all ML frameworks. However, by focusing on these three frameworks that are mostly used for DL applications, it is possible that the findings obtained in this study are DL-specific and do not generalize to other ML-based systems that do not make use of DL. Future works should consider expanding our study to cover a more diverse set of ML-based systems, including those using “classic” ML algorithms. Additionally, investigating other ML frameworks, such as *JAX*, which has recently gained significant attention, could be considered as future work for this study. Indeed, the use of these frameworks may introduce new types of ML bugs. Another threat to external validity stems from our focus on *Python* code. Since *Python* is the most used language in ML development [121,122], we believe that our findings are relevant for the majority of ML-based systems. Nevertheless, future works should consider expanding our study to include ML-based systems developed using other programming languages. In the sampling process, we focused on repositories with the highest number of stars, forks, commits, and issues to study the most mature repositories and mitigate the generalization threat. Another threat to the external validity of this study would be the generalization of the results, because of using GitHub as the main source for data collection. We selected GitHub because it is the most important platform for hosting open-source projects.

4.6.4 Conclusion Validity

Conclusion limitations can be potentially wrongly classified, missing issues/repository type, and the replicability of the study. We manually inspected 4,057 repositories out of over 400,000 repositories scraped from GitHub. Similarly, for issues, we extracted 44,342 issues

from the ML-based repositories and sampled 386 using a well-known statistical procedure to ensure the credibility of our results. In both cases, we believe that the sample data is big enough to be representative, and not mislead us in our conclusions. Similarly, the labeling of both repositories and issues was done independently by three raters and then discussed to mitigate potential errors. At last, we provided a replication package [133] to allow for the reproducibility of our results, and also for other researchers to build on our study.

4.7 Chapter Summary

In this study, we aimed to characterize different types of bugs in ML-based systems, from the software maintenance perspective. Therefore, we manually checked 386 selected issues raised in ML-based systems developed using the three most popular ML frameworks (*TensorFlow*, *Keras*, and *PyTorch*). Firstly, we showed that ML components are more error-prone in comparison with non-ML components. Furthermore, the results showed that the previous classifications on ML-related bug types, their root cause, and symptoms are becoming stale and should be revised. We also observed that fixing ML bugs are mostly more complicated than non-ML bugs. Moreover, the required resources (time-to-fix and developer expertise level) for fixing ML bugs are almost similar to non-ML ones. Besides, we presented that development teams fix ML issues more cautiously compared to non-ML ones, with respect to the percentage of bug-fix PRs that are merged without review.

CHAPTER 5 COMMON CHALLENGES OF DEEP REINFORCEMENT LEARNING APPLICATIONS DEVELOPMENT

5.1 Chapter Overview

Deep Reinforcement Learning (DRL) is a subdomain of ML used to produce intelligent agents. Despite recent developments in DRL technology, the main challenges that developers face in the development of DRL applications are still unknown. Since software development challenges considerably affect the quality of the software product, in this chapter, we conduct a large-scale empirical study of **927** DRL-related posts extracted from Stack Overflow, the most popular Q&A platform in the software community. Through the process of labeling and categorizing extracted posts, we created a taxonomy of common challenges encountered in the development of DRL applications, along with their corresponding popularity levels. This taxonomy has been validated through a survey involving 65 DRL developers. Results show that at least 45% of developers experienced 18 of the 21 challenges identified in the taxonomy. The most frequent source of difficulty during the development of DRL applications are *Comprehension*, *API usage*, and *Design problems*, while *Parallel processing*, and *DRL libraries/frameworks* are classified as the most difficult challenges to address, with respect to the time required to receive an accepted answer.

5.2 Study Design

Reinforcement Learning (RL) has begun making its mark across a range of industrial sectors, from autonomous vehicles [180] and traffic engineering [181] to healthcare systems [182]. Recently we have been also witnessing an increasing adoption of RL to solve different software engineering tasks, from automatic code improvement [183], to test case prioritization [184], and program debloating [185]. Reinforcement Learning differs significantly from other sub-categories of Machine Learning (ML) such as supervised and unsupervised learning, as it includes an agent that interacts with an environment to learn how to perform a sequence of actions leading to the best cumulative final rewards [84]. In other words, in RL, an agent learns to act in a way that modifies its behavior gradually to achieve the best final result; which makes traditional software quality assurance techniques inadequate for RL. Deep Reinforcement Learning (DRL) is an integration of Deep Learning (DL) and RL, also known as Deep RL, to address challenges, such as high-dimension input data [186]. Combining DL and RL enables DRL to discover compact low-dimensional representations of high-dimensional

data automatically [186].

Although there exist studies on testing and debugging RL programs [187, 188], the main challenges and obstacles that developers face while developing RL applications are still unclear [189]. Moreover, because of basic differences between the paradigm of traditional software applications and ML applications [28, 95], it is expected that developers of ML applications face different types of challenges in the implementation process of such applications. Thus, DRL developers may face different challenges from other types of software systems (including traditional software systems as well as other subcategories of ML applications) [190–192]. As an example, Listing 5.1 shows a SO post (70562317) related to a DRL application, representing a challenge in implementing the method to choose an optimal action which is specific to DRL development and differs from ML and DL development challenges.

Though there exist some studies regarding challenges in the development of DL [193, 194], ML applications [195, 196], to the best of our knowledge there is no study on the challenges that developers face when developing DRL applications. The study by Yahmed et al. [89] is the most closely related work to this research. It examines the challenges that developers face during the deployment process of DRL systems but does not consider the challenges occurring in the early development phases prior to deployment. In this study, we examine the following research questions:

RQ1. *What are the common challenges of DRL application development?*

RQ2. *How are the identified challenges perceived by DRL practitioners?*

RQ3. *Are DRL application development challenges language- and/or framework-specific?*

To answer these research questions, we manually examined and categorized 927 Stack Overflow (SO) posts that are related to DRL development. We report our results as a taxonomy of challenges in DRL application development. Besides, we conducted a survey of DRL developers/practitioners to validate our findings. Moreover, we investigated the dependency of

```

1 def act(self, some_input, state):
2     mu, var, state_value = self.model(some_input, state)
3     # mu contains info required for gradient
4     mu = mu.data.cpu().numpy()
5     # mu is detached and now has forgot all the operations performed in "
    self.action_head"
6     sigma = torch.sqrt(var).data.cpu().numpy()
7     action = np.random.normal(mu, sigma)
8     action = np.clip(action, 0, 1)
9     action = torch.from_numpy(action/1000)
10    return action, state_value

```

Listing 5.1 SO post (70562317) showing a challenge in the development of the RL action.

the identified challenges on programming languages and libraries/frameworks used for DRL development.

5.3 Methodology

This section describes the methodology we follow in this study. This methodology is illustrated in Fig.5.1.

5.3.1 Extracting Posts from Stack Overflow (SO)

We rely on Stack Overflow (SO) as the main source of information in this study; similar to several previous studies which utilized data exclusively obtained from SO for analyses [77, 80, 193]. SO is known as the largest technical question and answer (Q&A) website creating a public knowledge base in various areas [197], with 23.4 million questions and 19.6 million users as of December 2022 [198]. In the software development community, SO provides a platform for developers to exchange about coding issues; improving their coding knowledge. To extract SO posts related to DRL, we use Stack Exchange Data Explorer¹ which provides access to up-to-date SO data. Overall, we use a list of DRL-related tags and keywords to collect DRL-related SO posts. Listing.5.2 presents an example of a query used to collect SO posts containing ‘*deep-learning*’ and ‘*reinforcement-learning*’ tags at the same time.

```
1 SELECT * FROM Posts
2 WHERE Tags LIKE '%<deep-learning>%'
3 AND Tags LIKE '%<reinforcement-learning>%'
```

Listing 5.2 Sample query to extract SO posts that have ‘*deep-learning*’ and ‘*reinforcement-learning*’ tags simultaneously.

To gather the list of DRL-related tags and keywords, we follow previous study [77] using a snowballing approach in which we start with posts that have ‘*deep-learning*’ and ‘*reinforcement-learning*’ tags, simultaneously. In the next step, we collect all tags assigned to the SO posts gathered in the previous step. Then, we include DRL-related tags to our list of tags (e.g., ‘*dqn*’). We continue this process and expand the list of DRL-related tags until we are unable to add any new tags to our list. Besides, we create a list of DRL-related keywords based on the list of collected DRL-related tags. Firstly, we include all DRL-related tags (such as ‘reinforcement learning’) to the list of DRL-related keywords. Moreover, we add expanded forms of DRL-related tags which are acronyms. As an example, we add ‘*Deep Q-Learning*’ which is the expanded form of ‘*dqn*’. The complete list of tags and keywords used to extract

¹<https://data.stackexchange.com/stackoverflow/query>

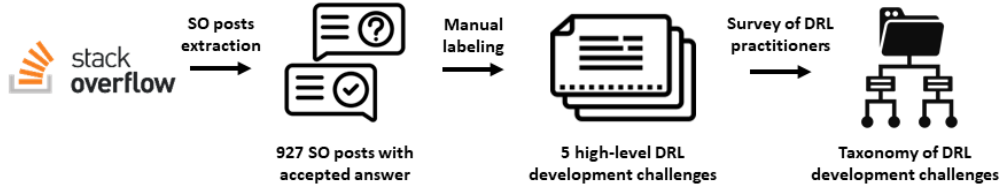


Figure 5.1 High-level view of the used methodology.

SO posts are available in our replication package [199]. In summary, we collected SO posts that meet at least one of the following criteria:

- Posts having one of the identified tags (e.g., ‘*drl*’, ‘*dqn*’, etc.)
- Posts with a combination of identified tags (e.g., combination of ‘*deep-learning*’ and ‘*reinforcement-learning*’)
- Posts with a combination of identified tags and keywords in their title or body (e.g., ‘*reinforcement-learning*’ tag and ‘*deep*’ in the post title)
- Posts including identified keywords in their title or body (e.g., ‘*drl*’)

After extracting all posts and removing duplicates, we obtained 3,083 posts. Then, we filtered out the posts without an accepted answer which leaves us with 927 posts. We chose to remove posts without an accepted answer, similar to previous studies [84,200] because the correctness of any of such responses would not be inferable, which could potentially bias our results. We also collected information about the time taken by each post to receive an accepted answer and used it as an indicator of the level of difficulty of the question, similar to the approach employed in previous works [201,202].

5.3.2 Manual Inspection

During this step, a team of four raters (three Ph.D. candidates and a senior research staff who all are practitioners of DRL development) is responsible for labeling the collected SO posts. Following a methodology similar to prior works [4,31], we split the collected SO posts into 10 parts, each of which is inspected in a dedicated labeling round. All the discussions and referenced source codes in each post are thoroughly reviewed. The raters use an open coding method [203] to label SO posts and categorize them. Each SO post is reviewed by two raters. We use the “*Google Sheets*” platform [204] to save all extracted labels in an online environment. That is, all raters put generated labels in the shared document, but they do not have access to the labels that other raters assigned to each post. After labeling

SO posts in each round, the raters meet to discuss disagreements and resolve conflicts. In the case they fail to resolve a disagreement, a third rater reviews the SO post and makes a decision about its label, acting as a tie-breaker. Besides, the raters review the generated labels in each meeting to ensure their comprehensiveness and granularity (combining similar labels generated by different raters or dividing a label into separate ones). The justification for dividing SO posts into 10 labeling rounds, with approximately 10% of the total collected posts in each round, emerges from the necessity for raters’ discussion in iterative meetings. These discussions are essential for resolving labeling conflicts and reaching an agreement, reviewing generated labels, and creating finer or coarser-grained labels. In case of changing existing labels, raters re-review the previously labeled posts to ensure that assigned labels are in synchronization with newly generated labels. We made this decision to allow for continuous improvement of the labeling process. This way raters have the opportunity to resolve their conflicts at the end of each round, similar to the technique used in previous studies [4, 31]. Besides, in the case that any of the raters suggests generating a new label, all raters meet to discuss and reach an agreement on that new label. After completing labeling all 927 posts, all raters meet to finalize the generated labels, categorize them, and create the taxonomy. Then, the first two authors review all of the labeled posts again to ensure that their assigned labels are in sync with the final generated taxonomy. Regarding the posts in which the questioner asks more than one question belonging to multiple categories of challenges, we repeat that record in our dataset and assign a different label to each record. For instance, post #45382763 has been identified to belong to two categories, comprehension and design problems. Although we could not report inter-rater agreement level due to the lack of prior defined categories, we calculate inter-rater agreement between the pair of raters who investigate each SO post after finalizing the labels using Cohen’s Kappa [205] and obtained an 86% agreement level. Table 5.1 presents detailed information on the labeling procedure.

During the manual inspection of SO posts, we filtered out 57 posts that were not related to DRL development. Generally, some questioners may add DRL-related tags to their posts by mistake or because of unfamiliarity with DRL and its real capabilities to solve their considered problems (such as #60958362). We also filtered out posts that are too general and which could not be considered as reporting about a challenge in DRL development (e.g., #3972812).

It is worth mentioning that 70% of the DRL-related questions in SO still remain without any accepted answer. This is consistent with previous findings by Alshangiti et al. [77] that 61% of ML-related SO posts remain without any accepted answer. Multiple factors could explain this finding. In some cases, the person who asks the question responds to the question after a while, but she does not assign the accepted answer badge to the post (e.g., #45364837). Some

Table 5.1 Detailed information on the manual labeling process.

Round	Analyzed SO posts	Conflicts	Relevant to DRL	New categories
1	100	11	95	15
2	100	7	93	1
3	72	10	67	–
4	100	16	96	4
5	100	14	90	1
6	100	9	96	–
7	100	5	95	–
8	100	7	93	1
9	100	5	94	–
10	55	3	51	–
Total	927	87	870	22

users also ask basic questions irrelevant to DRL but assign DRL-related tags to them. These questions receive negative scores and remain without any accepted answer (e.g., #50544568). We also observe posts where the person asking the question forgot to assign the accepted answer badge to an answer, based on upvotes to the response, comments of the person who asked the question, or other people with the same problem under the response (e.g., #63250935). About the posts with accepted answers, it should be mentioned that 16% of them have been answered by the user who published the question. This usually happens when a user asks a very specific question that remains unanswered for a long time, and then the same user finds the response elsewhere and adds it to his original post (e.g., #2723999).

5.3.3 Taxonomy Construction and Validation

Similar to previous studies [4,206], we use a bottom-up methodology to create the taxonomy. In fact, after completing each labeling round, we categorize all generated labels belonging to a similar theme into a group. Next, we build up parent nodes in a way that makes sure that categories and their subcategories adhere to a ‘*is a*’ relationship. Concerning that the raters may provide new labels during each labeling round, we need to update the taxonomy which means adding a new category, a new subcategory, or combining two categories/subcategories. After any update on the taxonomy which leads to a new version of it, all the authors have a debate on the newly generated version of the taxonomy in a group meeting. By completing the final labeling round and integrating all updates on the taxonomy, all study’s authors carry out a careful inspection of the produced taxonomy (including all categories and subcategories) over a meeting and finalize it.

Interviews and surveys stand as the two popular methods to validate the results of quali-

tative studies [207, 208]. Considering the advantages of conducting surveys, including cost-effectiveness, generalizability, reliability, and versatility [209, 210], we assessed the comprehensiveness and representativeness of the obtained taxonomy using a survey with DRL developers/practitioners who are not involved in the construction of the taxonomy. Nevertheless, it is noteworthy that several preceding studies have presented their findings without undergoing any validation process [31, 193].

While we build our taxonomy based on SO posts, we use Github to identify potential respondents to our survey. We collect a list of survey participants from collaborators of GitHub repositories related to DRL. Specifically, we extract GitHub repositories mentioning ‘*deep reinforcement learning*’ in their description using GitHub’s search API V3; a rest API that receives a query and returns a list of repositories that satisfy conditions stated in the query. In other words, we use ‘*deep reinforcement learning*’ as the keyword to generate the search query of GitHub search API V3². Given that GitHub search API limits access to only the first 1,000 results, we follow the methodology used in [28] and run several different queries to achieve less than 1,000 repositories for each query. That is, we divide the duration of the search for repository creation date between Jan 1, 2010, and Jan 31, 2023 (the date of running queries) into snapshots of 1 month. Thus, we execute 157 GitHub search requests to collect 7,244 repositories. Subsequently, we filter out forked and disabled repositories. The complete list of repositories and a sample search query to extract DRL-related repositories are accessible via the replication package [199]. In the next step, we check the repositories’ contributors and collect the contributors mentioning their email addresses, obtaining 2531 unique email addresses of developers.

We use Qualtrics [211], an online survey tool for designing and conducting surveys, to create survey forms. Table 5.2 presents the structure of our survey questionnaires. The survey starts with general questions regarding the participant’s current role and experience in DRL development (Section 1 of Table 5.2). Subsequently, we delve into specific questions regarding each DRL development challenge which is pointed out in the finalized taxonomy (Section 2 of Table 5.2). Concerning the potential complexity and difficulty of comprehending the whole taxonomy as a single figure within the survey, we present challenges (subcategories) in groups based on their respective main categories. Besides, to ensure clarity, a detailed description is given with each challenge offering participants a thorough understanding of the asked challenges. For each identified challenge, we ask three questions including 1) a ‘*yes/no*’ question identifying whether the answerer has faced the identified challenge, 2) the severity of the challenge, and 3) the amount of effort required to address the challenge. If a participant

²<https://docs.github.com/en/rest>

answers ‘yes’ to the first question, s/he will have the next two Likert-scale questions which are related to the severity of challenges and the required effort to address them. We also provide a free-text question in the final part of the survey asking the participants about any challenges in developing DRL applications not listed in our provided taxonomy (Section 3 of Table 5.2). These free-text questions allow us to collect possible challenges that we may miss in the taxonomy. The full survey questionnaire is available in our replication package [199].

We conduct a comprehensive analysis of top DRL-related GitHub repositories to ensure the completeness of the generated taxonomy. Besides, investigating the challenges faced by developers engaged in the development of real-world DRL applications may further enhance the generalizability of the generated taxonomy. Our methodology for selecting DRL-related repositories aligns with established approaches documented in similar prior studies [4,28,212]. Initially, we extracted 7,244 repositories to identify contributors of DRL-related repositories. Then, we select the top 100 repositories based on the highest number of stars. Subsequently, we conduct searches for ‘challenge’, ‘difficult’, and ‘complex’ keywords within repositories’ documentation, commit messages, and closed issues to identify potential challenges encountered by developers during the development of these repositories. Our search yielded 746 occurrences of the aforementioned keywords within the top 100 DRL-related repositories. Using a confidence level of 95% and a confidence interval of 5%, we conduct sampling, resulting in the selection of 254 instances. In the next step, two raters independently examine 254 randomly selected commits and issues to find out whether they pertained to real DRL-related development challenges. The raters carefully review commit messages, issue discussions, and investigate any changes made in relation to the specified keywords.

5.4 Result

This section presents and discusses the results of our study. All the materials used in this study, including the collected data, are publicly available online in our replication package at [199].

RQ1: What are the common challenges of DRL application development?

The final taxonomy of challenges of DRL application development is arranged into a tree structure with five high-level categories in which leaves (subcategories) refer to the challenges. Fig.5.2 shows the taxonomy of common challenges in the development of DRL applications. The number in parentheses presented next to each category/subcategory is the absolute number of identified SO posts that are categorized into that category (the absolute frequency

Table 5.2 Survey structure

Sec.1 General questions				
<i>Email address (optional)</i>				
<i>Years of experience on DRL</i>				
...				
Sec.2 DRL development challenges review				
DRL Challenges	Facing this challenge	Challenge severity	Needed effort to address	
<i>Comprehension</i>	☐ Yes	☐ Minor	☐ Major	☐ Low
<i>Reward</i>	☐ Yes	☐ Minor	☐ Major	☐ Medium
...	☐ No	☐ Critical	☐ High	☐ High
Sec.3 Free-text question				
<i>Any challenge that we have missed</i>				
<i>Any suggestion or comment</i>				

of each challenge). To give a better understanding of identified challenges, a brief description

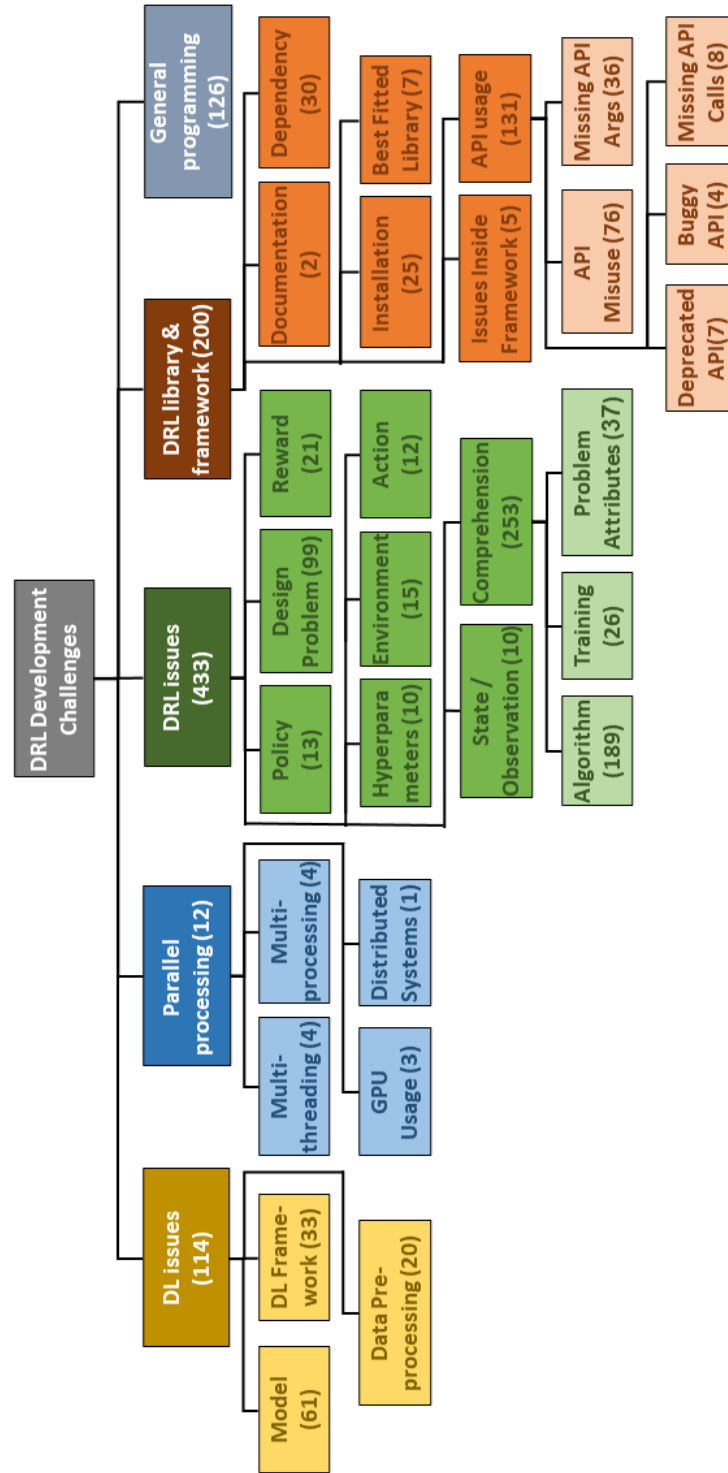


Figure 5.2 Taxonomy of common challenges in DRL development. Numbers represented with each category/subcategory indicate the number of SO posts categorized into that category/-subcategory.

of each category/subcategory is provided in the following.

DRL Issues. This category focuses on the challenges that developers may face while developing the DRL part of their applications. Challenges that belong to this category are specific to DRL application development. That is, compared to challenges classified under other categories that can be shared among other kinds of ML-related applications (e.g. supervised learning), challenges of the *DRL issues* category are only faced during DRL application development.

- a. *Design Problem:* Instances where the user asks advice for designing a solution and implementing a DRL application for their specific problem or scenario. For example, developers asked for recommendations to implement different parts of DRL applications for Curve Fever or mini-golf games. Another such challenge in this category is related to designing the properties of each object in a tank game. Although we tried to make finer-grained subcategories under this subcategory, we found out that it is impractical to split this subcategory further, as the majority of this subcategory’s posts primarily comprised users seeking guidance on high-level conceptual queries about defining their problem within a DRL context.
- b. *Comprehension:* Challenges about the meaning or details of theoretical concepts in DRL, i.e., misunderstanding about basic formulas of different DRL algorithms. For instance, a developer mentioned “*I’m trying to make a learning football game from scratch using Deep Q-learning algorithm (without convolutional network though). I just couldn’t figure out what does Φ stand for in this algorithm.*”, or one which is related to the difference between SARSA and Q-learning algorithms in terms of collecting the next policy value.
 - b.1. *Training:* This subcategory comprises inquiries concerning the theoretical concepts of the DL model and its significance in DRL applications. The DL model plays an essential role in the training and decision-making process of the DRL agents. Indeed, DL models in DRL applications serve to represent policy of the agent, estimate the value of state-action, and learn mappings from states to values.
 - b.2. *Problem attributes:* SO posts containing theoretical queries about attributes of DRL problems (such as reward, action, state, etc.) are categorized in this subcategory. SO posts in this category are related to the conceptual aspects of reward, action, and state, not their implementation details. For example, questions on how to formulate states, actions, and the reward signal for a particular DRL problem, or why a particular definition of the states is not suitable for a problem.

- b.3. *Algorithm*: Questions related to fundamental concepts of various DRL algorithms, such as actor-critic, Q-learning, SARSA, etc., are categorized as algorithm subcategories. The main duty of DRL algorithms is updating parameters of the policy and value function according to the observed states and achieved rewards. Our investigation into these algorithms reveals that Q-learning (28%) and DQN (15.3%) are the most common DRL-related algorithms posing challenges for developers. Conversely, Actor-Critic (3.1%), Proximal Policy Optimization (2.6%), and Trust Region Policy Optimization (0.5%) are the least queried DRL-related algorithms.
- c. *Policy's Loss*: This category refers to challenges about DRL learning policy's loss. As an example, questions regarding implementing a customized loss function or any problems in loss calculation methods are categorized in this group.
- d. *Reward*: Challenges in the implementation of reward, e.g., not using the negative reward to penalize each added time step are categorized in this subcategory. An example of this category is a post asking *"I am implementing the basic RL algorithm to play the game Flappy Bird. I want to be able to process the screen and recognize whether a point has been scored or the bird has died. Processing the screen returns a stacked numpy array. The reward function then needs to assign a reward to the provided array, but I have no idea how to go about this"*.
- e. *Action*: Questions/Challenges related to the action(s). e.g., possible actions in a specific game or implementing a *'chooseAction'* method for a PacMan bot.
- f. *State/Observation*: Questions/Problems regarding the state(s) or the agent observation(s), e.g., handling large state spaces. An example of such instance is a user asking *"I implemented a 3x3 OX game by q-learning (it works perfectly in AI v.s AI and AI v.s Human), but I can't go one step further to 4x4 OX game since it will eat up all my PC memory and crash... Since I need to calculate each Q value (for each state, each action), I need such a large number of array, is it expected? any way to avoid it?"* and the accepted answer offered suggestions on reducing their state space size by considering symmetries and other tricks.
- g. *Environment*: Questions/Problems pertinent to the environment, e.g., designing a custom environment. For example a user asked *"I'm very new to Ray RLlib and have an issue with using a custom simulator my team made. We're trying to integrate a custom Python-based simulator into Ray RLlib to do a single-agent DQN training. However, I'm uncertain about how to integrate the simulator into RLlib as an environment"*.

- h. *Hyperparameters*: Questions/Challenges related to the hyperparameters of the RL algorithm, e.g., setting the discount factor too high. A good example of this group is demonstrated in a post where the user shared the code for the learning algorithm and reported that the loss keeps increasing and the model is not learning. The accepted answer states that *“The main problem I think is the discount factor, gamma. You are setting it to 1.0, which means that you are giving the same weight to the future rewards as the current one”*.

DRL Libraries/Frameworks. This category refers to the challenges that developers face when they are trying to use DRL-specific libraries/frameworks (e.g., KerasRL [88], RLlib [213], Tensorforce [214], etc.). Challenges that software developers face when using libraries/frameworks have been extensively studied for traditional software systems development [215,216] and also for DL applications development [217]. However, despite the large number of SO posts related to the usage of DRL libraries/frameworks (i.e., 200 SO posts), these challenges are yet to be examined for DRL application development.

- a. *Installation*: Questions/problems regarding installing/uninstalling DRL-related libraries/frameworks or missing libraries. Issues categorized in this subcategory can often stem from an incompatibility between DRL-related frameworks/libraries and other libraries. For example, a user described her issue as *“when I try to install gym[box2d] I get the following error: I tried: pip install gym[box2d]. on anaconda prompt I installed swig and gym[box2d] but I code in python3.9 env and it still not working (my text editor is pycharm) gym is already installed”*.
- b. *Dependency*: This subcategory includes questions/challenges about the mismatch between versions of installed libraries/frameworks and problems in installed versions of libraries, e.g., when the version of the installed *OpenAI Gym* is not compatible with *Python*. An instance of this subcategory is a user reporting getting an error while installing *OpenAIGym* and the answer pointed out that *“the error means that the package has dependency requirements that conflict with one another”*.
- c. *API usage*: This subcategory includes questions about the usage of arguments, attributes, methods, etc. of an API. It also includes questions about the default values, implemented method, existence of attributes, or methods in an API. An example from this group of issues is a user reporting not knowing how to get the weights of the network using the correct API methods: *“I’m using RLlib to train a reinforcement learning policy (PPO algorithm). I want to see the weights in the neural network underlying the*

policy. After digging through RLlib’s PPO object, I found the TensorFlow Graph object. I thought that I would find the weights of the neural network there. But I can’t find them”. This subcategory is subdivided into five subcategories to delineate more detailed challenges.

- c.1. *API misuse*: This subcategory covers SO posts that mention misunderstanding of API usage. In other words, API misuse occurs when DRL developers try to utilize an API in a manner that is not aligned with its intended purpose.
- c.2. *Missing API call*: Questions related to the absence of necessary API calls within a code snippet are classified in this subcategory.
- c.3. *Missing API args*: When SO posts discuss challenges that DRL developers face due to the absence of one or more essential arguments in an API call, we classify them under this subcategory.
- c.4. *Buggy API*: This subcategory includes SO posts that inquire about calling APIs resulting from the bugs within the implementation of APIs. It is worth mentioning that this challenge is distinct from issues related to the implementation of DRL applications themselves; rather, they pertain to the implementation of the API.
- c.5. *Deprecated API*: In this subcategory, we cover questions about calling a deprecated API which has been altered or removed from the library/framework.
- d. *Documentation (using newly added features)*: This subcategory of issues occurs when a developer wants to use a feature of a DRL library/framework, but there is no documentation for it. For example, a user who could not find the required documentation for the *Neural Network Approximator* in *ReinforcementLearning.jl*: “*I have decided to use a Neural Network Approximator. But the docs do not discuss much about it, nor are there any examples where a neural network approximator is used. I am stuck on how to figure out how to use such an approximator*”.
- e. *Best fitted library for a special task (library suitability)*: Instances where the user asks about the best DRL libraries/frameworks for customizing agents, based on the requirements of the problem. An instance of this group was observed in a post where a user had a customized state space and was looking for a library that supports it: “*I’ve had some luck training an agent using keras-rl, specifically the DQNAgent, however, keras-rl is under-supported and very poorly documented. Any recommendations for RL packages that can handle this type of observation space? It doesn’t appear that openai baselines, nor stable-baselines can handle it at present*”.

- f. *Problems inside DRL frameworks*: Including issues that are encountered because of internal faults, i.e., bugs in the DRL frameworks. For example, a user kept getting a *numpy* error when calling the *model.learn()* function and it was found to be an official bug in the used library *Stable Baselines3*.

DRL Libraries/Frameworks. This category refers to the challenges that developers face when they are trying to use DRL-specific libraries/frameworks (e.g., KerasRL [88], RLlib [213], Tensorforce [214], etc.). Challenges that software developers face when using libraries/frameworks have been extensively studied for traditional software systems development [215,216] and also for DL applications development [217]. However, despite the large number of SO posts related to the usage of DRL libraries/frameworks (i.e., 200 SO posts), these challenges are yet to be examined for DRL application development.

- a. *Installation*: Questions/problems regarding installing/uninstalling DRL-related libraries/frameworks or missing libraries. Issues categorized in this subcategory can often stem from an incompatibility between DRL-related frameworks/libraries and other libraries. For example, a user described her issue as “*when I try to install gym[box2d] I get the following error: I tried: pip install gym[box2d]. on anaconda prompt I installed swig and gym[box2d] but I code in python3.9 env and it still not working (my text editor is pycharm) gym is already installed*”.
- b. *Dependency*: This subcategory includes questions/challenges about the mismatch between versions of installed libraries/frameworks and problems in installed versions of libraries, e.g., when the version of the installed *OpenAI Gym* is not compatible with *Python*. An instance of this subcategory is a user reporting getting an error while installing *OpenAIGym* and the answer pointed out that “*the error means that the package has dependency requirements that conflict with one another*”.
- c. *API usage*: This subcategory includes questions about the usage of arguments, attributes, methods, etc. of an API. It also includes questions about the default values, implemented method, existence of attributes, or methods in an API. An example from this group of issues is a user reporting not knowing how to get the weights of the network using the correct API methods: “*I’m using RLlib to train a reinforcement learning policy (PPO algorithm). I want to see the weights in the neural network underlying the policy. After digging through RLlib’s PPO object, I found the TensorFlow Graph object. I thought that I would find the weights of the neural network there. But I can’t find them*”. This subcategory is subdivided into five subcategories to delineate more detailed challenges.

- c.1. *API misuse*: This subcategory covers SO posts that mention misunderstanding of API usage. In other words, API misuse occurs when DRL developers try to utilize an API in a manner that is not aligned with its intended purpose.
- c.2. *Missing API call*: Questions related to the absence of necessary API calls within a code snippet are classified in this subcategory.
- c.3. *Missing API args*: When SO posts discuss challenges that DRL developers face due to the absence of one or more essential arguments in an API call, we classify them under this subcategory.
- c.4. *Buggy API*: This subcategory includes SO posts that inquire about calling APIs resulting from the bugs within the implementation of APIs. It is worth mentioning that this challenge is distinct from issues related to the implementation of DRL applications themselves; rather, they pertain to the implementation of the API.
- c.5. *Deprecated API*: In this subcategory, we cover questions about calling a deprecated API which has been altered or removed from the library/framework.
- d. *Documentation (using newly added features)*: This subcategory of issues occurs when a developer wants to use a feature of a DRL library/framework, but there is no documentation for it. For example, a user who could not find the required documentation for the *Neural Network Approximator* in *ReinforcementLearning.jl*: “*I have decided to use a Neural Network Approximator. But the docs do not discuss much about it, nor are there any examples where a neural network approximator is used. I am stuck on how to figure out how to use such an approximator*”.
- e. *Best fitted library for a special task (library suitability)*: Instances where the user asks about the best DRL libraries/frameworks for customizing agents, based on the requirements of the problem. An instance of this group was observed in a post where a user had a customized state space and was looking for a library that supports it: “*I’ve had some luck training an agent using keras-rl, specifically the DQNAgent, however, keras-rl is under-supported and very poorly documented. Any recommendations for RL packages that can handle this type of observation space? It doesn’t appear that openai baselines, nor stable-baselines can handle it at present*”.
- f. *Problems inside DRL frameworks*: Including issues that are encountered because of internal faults, i.e., bugs in the DRL frameworks. For example, a user kept getting a *numpy* error when calling the *model.learn()* function and it was found to be an official bug in the used library *Stable Baselines3*.

DL Issues. This category represents the challenges that arise specifically from the DL part of DRL applications. As the challenges belonging to this category are shared by both DL and DRL applications, we use the high-level categories of the taxonomy provided by Humbatova et al. [4] for DL applications.

- a. *Model:* Questions regarding the DL model including model layer, activation function, load/save model, etc. For instance, a user who was implementing a DRL model asked for advice on back propagation: *“I am struggling with the implementation of the back propagation. Since the rewards are so big, the error values are huge, which creates huge weights. After a few training rounds, the weights to the hidden layer are so big, my nodes in the hidden layer are only creating the values -1 or 1”*.
- b. *Data Preprocessing:* Questions about preparing data to be fed into the DL model, e.g., the shape of the input matrix. As an example, there was a user who asked *“I am learning how to use Gym environments to train deep learning models built with TFLearn. At the moment my array of observations has the following shape: (210, 160, 3). Any recommendations on what is the best way to reshape this array so it can be used in a TensorFlow classification model?”*
- c. *DL framework:* Questions about the usage of DL frameworks (e.g., Keras, TensorFlow, PyTorch, etc.) in development of DL part in DRL applications. For instance, a user who wanted to use *Huber loss* in their model which was written using *Keras*, but this loss function is not readily available in *Keras*, and the accepted answer implements the *Huber loss* function.

Parallel Processing & Multi-threading. This category focuses on the challenges associated with running DRL applications as parallel or distributed applications. While running different types of DL applications in a parallel style is a common practice (e.g., using GPU or multi-core CPU), it is important to note that the architecture of DRL applications differs from other types of DL applications. As a result, running a DRL application as a parallel or distributed application introduces new challenges that may not occur in DL application development.

- a. *GPU usage:* Questions/Problems regarding utilizing GPU for running DRL applications. For example, a user was facing performance issues when training a DQN on GPU: *“I am try to train a DQN model with the following code. The GPU (cuda) usage is always lower than 25 percent. I know the tensorflow backend is consulting the GPU resources, but the usage is low. Is there any way I can improve the utilization*

of the GPU (When I train a CNN network, the GPU (cude) utilization is around 70 percent)?”

- b. *Distributed processing*: Questions/Challenges about running DRL applications as a distributed software. For example when a user wanted to implement the *Asynchronous Advantage Actor Critic (A3C)* model for reinforcement learning in the local machine, she posed a question about the possibility of implementing it in a distributed manner: “*Would it be easier/faster/better to implement this using the distributed TensorFlow API? In the documentation and talks, they always make explicit mention of using it in multi-device environments. I don’t know if it’s an overkill to use it in a local async algorithm*”.
- c. *Multi-threading*: Questions/Problems about running DRL applications as a multi-threaded software. An example of this category was observed in a post where the user asked about the possibility of reducing the DRL application’s training time by running it on multiple threads concurrently: “*My friend and I are training a DDQN for learning 2D soccer. I trained the model about 40.000 episodes but it tooks 6 days. Is there a way for training this model concurrently? For example, I have 4 core and 4 thread and each thread trains the model 10.000 times concurrently. Therefore, time to training 40.000 episodes are reduced 6 days to 1,5 days like parallelism of for loop*”.
- d. *Multi-processing*: This subcategory refers to challenges stemming from running DRL applications on multiple processing units (e.g., multiple CPUs). As an example, a user asked a question about using Ray³ in a multi-processing style.

General Programming Issues. This category contains programming and coding mistakes occurring when developing DRL applications. The challenges in this category could not be classified in any of the other categories described above. For example, a user had a question about how to slice a 3D *numpy* array in an RL finite MDP application (#67089715).

Overall, the majority of the analyzed SO posts have been assigned to categories *Comprehension*, *Design problem*, *Model*, and *API Usage*. Aside from *Design problem* related questions, which are often quite specific (i.e., related to particular implementations of DRL), the majority of questions asked by DRL developers concern issues that apply to DRL applications in general.

³<https://www.ray.io/>

Finding 1: The taxonomy of challenges in DRL development is structured into five main categories including *DRL issues*, *DL issues*, *DRL libraries/frameworks*, *parallel processing & Multi-threading*, and *general programming issues*, where *DRL issues* (48.9%) and *DRL libraries/frameworks* (22.6%) categories are the most popular. Among the challenges, *comprehension* (28.6%), *API usage* (14.8%), and *design problem* (11.2%) are the most prevalent DRL development challenges.

Table 5.3 shows the number and percentage of the SO posts within our dataset having different tags and/or keywords that we used in subsection 5.3.1. Fig.5.3 presents the distribution of SO posts related to DRL application development for a period of 13 years, from 2009 to 2022. From Fig.5.3a, we observe a substantial surge of inquiries about DRL development in 2016; reaching a peak in 2019 and 2020. Additionally, as can be seen on Fig.5.3b, *comprehension* and *design problem* questions dominated posts about DRL application development challenges. It is also noticeable that *API usage*, the second most common DRL application development challenge, was at its peak in 2018.

Fig.5.4 depicts the distribution of time taken by SO posts from different categories to receive an accepted answer. This duration is an indicator of the difficulty level of the questions mentioned in the SO posts in the development of both traditional [201, 202] and ML software [20, 77]. *Parallel Processing* is the category with the highest average time taken before receiving an accepted answer. This can be explained by the fact that using multi-processing or distributed processing in DRL is not necessarily widespread and also requires particular

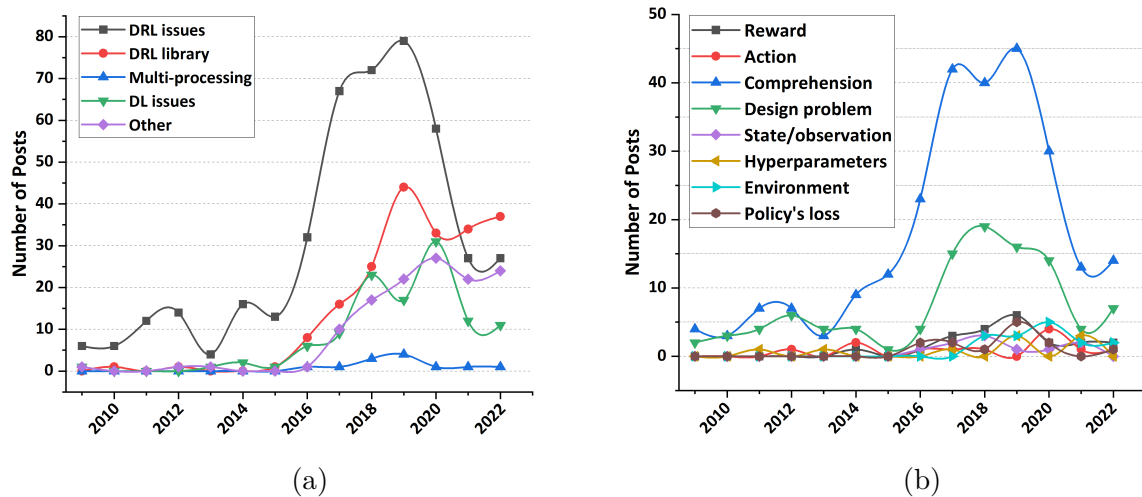


Figure 5.3 Distribution of DRL related SO posts per years (a) high-level categories of the provided taxonomy, (b) subcategories of ‘DRL issues’ category.

Table 5.3 Detailed information regarding tags and keywords used to extract DRL-related SO posts. Columns show the number (#) and percentage (%) of SO posts categorized in the main categories having mentioned tags/keywords (‘DRL’, ‘DRL LF’, ‘DL’, ‘PP’, and ‘Gen’ refer to ‘DRL issues’, ‘DRL libraries/frameworks’, ‘DL issues’, ‘Parallel processing’, and ‘General programming’, respectively).

	Tag/ Keyword	Posts (#)	DRL (%)	DRL LF(%)	DL (%)	PP (%)	Gen (%)
Tags	reinforcement-learning	710	96	58	86.8	83.3	54.8
	openai-gym	208	10	59	10.5	25	25.4
	tensorflow	159	11.3	14	47.4	16.7	20.6
	q-learning	137	25.2	2	13.2	-	7.1
	deep-learning	127	13.4	6.5	32.5	-	15.1
	keras	78	5.1	6	29.8	-	7.9
	pytorch	65	6.5	3	18.4	-	7.9
	dqn	39	4.4	2	7	8.3	5.6
	stable-baselines	24	-	5.5	1.8	-	5.6
	keras-rl	19	-	3.5	5.3	-	2.4
	tensorflow-agents	12	0.5	2.5	2.6	-	1.6
	rllib	9	0.5	4	-	-	-
	tensorboard	8	-	2	-	-	2.4
	starcraftgym	5	-	2	-	-	0.8
keywords	reinforcement learning	313	44.8	24	36	41.7	19.8
	deep	136	16.4	10.5	23.7	16.7	11.9
	ppo	117	11.5	18	11.4	8.3	13.5
	deep reinforcement learning	37	4.6	2.5	5.3	8.3	4
	a3c	17	1.6	0.5	3.5	25	1.6
	deep q-learning	16	2.5	1	0.9	-	1.6
	actor critic	11	1	-	4.4	8.3	0.8
	ddqn	9	-	1	1.8	8.3	-
	drl	5	0.7	0.5	-	-	0.8
	dql	5	0.7	0.5	0.9	-	-
	deep rl	2	0.5	-	-	-	-
	trpo	2	0.5	-	-	-	-
	hierarchical	1	0.2	-	-	-	-
	trust region	1	0.2	-	-	-	-
	policy optimization	1	0.2	-	-	-	-

knowledge and expertise. The remaining categories need a nearly similar average time frame to receive an accepted answer, with the *general programming issues* category having the shortest average duration. We attribute the shortest average time of the *general programming issues* category to the fact that this category contains generic challenges that do not require expertise in DL or DRL.

Although we note a high proportion of outliers in Fig.5.4, we have a relatively low median,

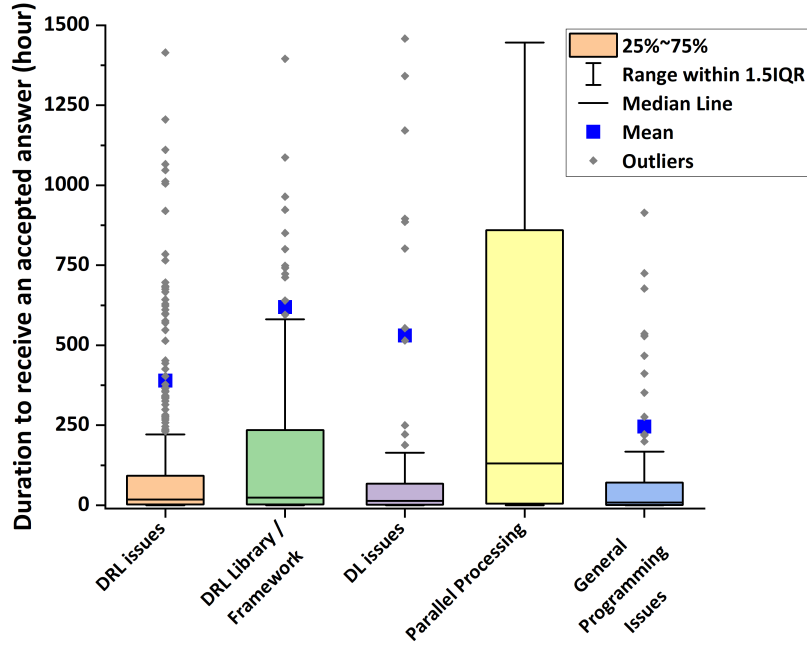
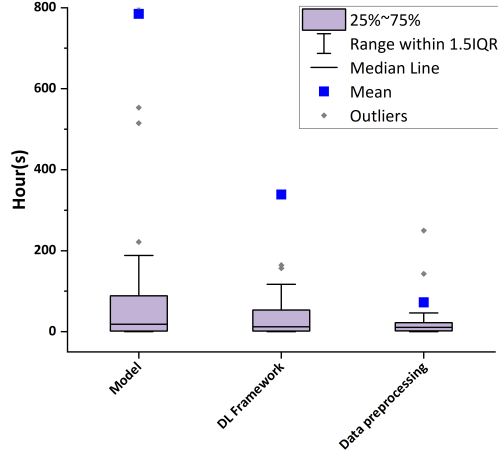


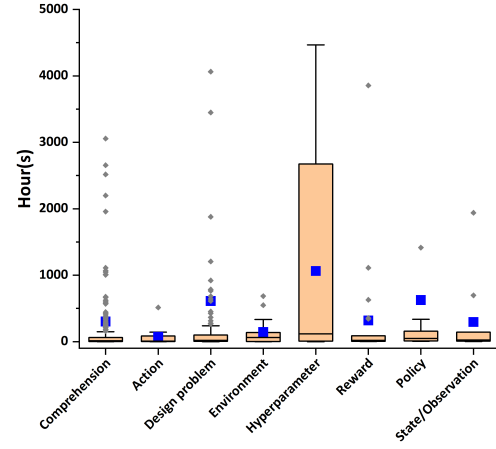
Figure 5.4 Duration to receive an accepted answer (hours).

first, and third quartiles, for all categories compared to the average. Meaning that, although the majority of posts received an accepted answer in a relatively short period of time (in general less than 10 days), a sizable number of posts required a longer time (more than 20 days). Therefore, even within categories, there is quite some discrepancies inside the subcategories (challenges) themselves. As can be seen on Fig.5.5a, SO posts categorized as *Model* have a higher average waiting time to receive an accepted answer than the other subcategories of the *DL Issues* category. Among subcategories under *DRL Issues*, *hyperparameters* includes SO posts with the highest average required time for receiving an accepted answer (Fig.5.5b). About *DRL libraries/frameworks*, SO posts belonging to *API usage*, *Dependency* and *Installation* subcategories require the longest average time before receiving an accepted answer (Fig.5.5c). Regarding subcategories within *parallel Processing & multi-threading*, it is notable that the average duration required to receive an accepted answer for SO posts classified under *multi-processing* subcategory exceeds one year. It should be also taken into account that the small number of SO posts in this subcategory might bias the results, with respect to the fact that small data may not represent the distribution of classes in the population adequately [218].

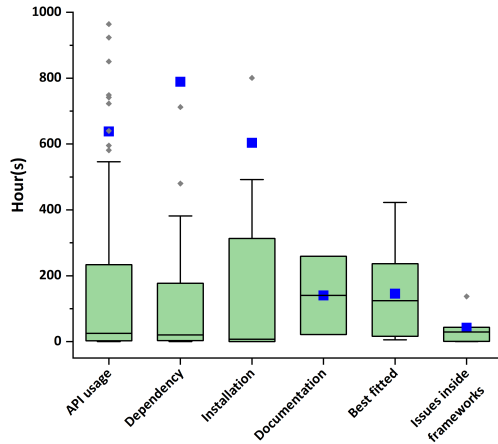
Although our methodology to measure the difficulty level of addressing challenges aligns with prior studies analyzing SO posts [77,215], it is worth noting that some studies have utilized the number of posts within each category as a metric to illustrate the difficulty associated



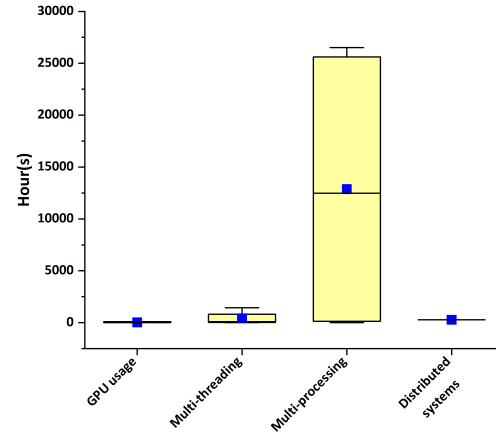
(a) DL issues



(b) DRL issues



(c) DRL libraries/frameworks



(d) Parallel processing & multi-threading

Figure 5.5 SO posts' required time to receive an accepted answer.

with addressing the related challenges [78,80]. In the case of considering the number of posts as an indicator of the difficulty level for tackling challenges (the number mentioned along with each category in Fig. 5.2), the results would closely mirror the difficulty levels observed for the challenges categorized as *DL issues* and *parallel processing* categories depicted in Fig.5.5. For instance, *design problem* is the second most difficult challenge in the *DRL issues* category when we use 'number of posts' or 'duration to receive an accepted answer' to measure the difficulty level. Conversely, the difficulty levels for challenges falling under *DRL issues* and *DRL libraries/frameworks* differ by using these two metrics. For example, as illustrated in Fig.5.5b, *hyperparameters* emerges as the most challenging subcategory within the *DRL issues* category, even though *comprehension* has the highest number of posts. This

discrepancy may be attributed to the nature of inquiring about *hyperparameters*, which may necessitate various implementations and a longer time to respond compared to other challenges. Furthermore, addressing SO posts categorized as *comprehension* mainly seeks users' background knowledge, which does not necessarily require practical implementation or application execution.

Finding 2: The proportion of SO posts containing questions related to DRL application development increased significantly during the period (2016 - 2021), reaching a peak between 2019 and 2020. In terms of average time before the reception of an accepted answer, *Parallel processing* is the category that required the longest period of time, in comparison to the other categories. Overall, we observe a high variance in the time to answer of the posts from the different categories of challenges identified in our taxonomy, suggesting that their impact on the development process of DRL applications is not equal.

RQ2: How are the identified challenges perceived by DRL practitioners?

We cross-check the taxonomy generated based on SO posts using a validation survey. **65** ML practitioners participated in our survey to assess our identified challenges, including 55% researchers (Master's and Ph.D. students, research assistants, and professors), 27% ML/SE engineers, 11% developers, and 7% data scientists. Among our respondents, 86% have at least 1 year of DRL development experience and 46% have more than 3 years of experience. Table 5.4 summarizes the responses of participants for each DRL development challenge contained in the taxonomy. For each challenge, we provide the percentage of developers who reported having experienced that challenge (based on the answers to the 'yes' or 'no' question). Besides, we asked the participants about the severity of each challenge and the effort required to address them. Results show that all challenges presented in our taxonomy were encountered by the survey respondents. Moreover, no additional challenges were proposed by the survey participants through the open-text questions. This indicates that our provided taxonomy is representative of challenges faced by developers during the DRL application development. It is worth noting that the survey questions incorporate the challenges depicted in the third level of the taxonomy (Fig.5.2), ensuring the survey's conciseness.

According to the survey results, the majority of our respondents have been confronted with challenges classified as *DRL issues* (68.9% average over all subcategories in this category). This observation is aligned with the proportion of SO posts categorized as *DRL issues* (see

Table 5.4 Result of the survey of DRL practitioners.

Challenges		Response		Severity			Required effort		
		Yes	No	Minor	Major	Critical	Low	Medium	High
DRL Issues	Comprehension	48%	52%	69%	23%	8%	60%	23%	17%
	Reward	86%	14%	32%	37%	31%	28%	28%	44%
	Action	49%	51%	74%	11%	15%	66%	20%	14%
	Environment	83%	17%	32%	34%	34%	31%	29%	40%
	Hyperparameters	80%	20%	43%	32%	25%	37%	26%	37%
	Design problem	75%	25%	38%	37%	25%	34%	31%	35%
	Policy	62%	38%	54%	37%	9%	54%	32%	14%
	State/Observation	68%	32%	52%	32%	16%	49%	37%	14%
DRL libraries/frameworks	Installation	43%	57%	85%	12%	3%	78%	15%	7%
	Dependency	54%	46%	78%	14%	8%	77%	18%	5%
	API usage	38%	62%	86%	9%	5%	78%	14%	8%
	Documentation	51%	49%	69%	23%	8%	71%	20%	9%
	Bugs inside framework	48%	52%	75%	14%	11%	70%	15%	15%
	Best fitted library	51%	49%	70%	25%	5%	65%	25%	10%
Parallel processing	GPU usage	54%	46%	72%	20%	8%	69%	20%	11%
	Multi-threading	45%	55%	68%	24%	8%	65%	18%	17%
	Multi-processing	49%	51%	57%	31%	12%	51%	23%	26%
	Distributed systems	43%	57%	58%	28%	14%	55%	14%	31%
DL Issues	Model	48%	52%	71%	23%	6%	66%	26%	8%
	Data preprocessing	48%	52%	57%	25%	18%	63%	25%	12%
	DL framework	46%	54%	66%	25%	9%	64%	27%	9%

Fig.5.2). Conversely, challenges belong to *Parallel Processing & Multi-threading* category have been experienced the least with only 45.25% of respondents (which is the lowest proportion among all categories) reporting having faced challenges leveraging parallel processing. This finding is reflected by the results of our quantitative analysis of SO posts, which show that only 1.5% of posts contained questions related to the *Parallel Processing & Multi-threading* category. It should be also taken into consideration that previous research showed that there is a growing trend toward the number of studies on DRL [219, 220]. Among the challenges identified in our taxonomy, developers who participated in our survey specify *reward* (86%), *environment* (83%), *hyperparameters* (80%), and *design problem* (75%) as the most common challenges in DRL development. Although only 14.8% of SO posts contained questions about *API usage*, 38% of survey respondents identified it as a challenging issue in DRL development. Given that *reward*, *environment*, *hyperparameters*, and *design problem* are fundamental components of an RL application [221], it is expected that survey

participants reported them as the most encountered challenges. For instance, defining the environment is known as a crucial step in an RL application development process that affects the convergence of an agent’s behavior significantly [222].

This however contrasts with the number of SO posts identified as *DRL environment* (i.e., 1.6% of SO posts). We also note that *comprehension*, the most frequent challenge in terms of the number of SO posts in our taxonomy (28.6% of SO posts), has been reported by 52% of survey participants as a non-challenging issue. The explanation for this variance lies in the experience level of the survey respondents. Indeed, it is indicated that 84% of the survey participants have at least 1 year of experience in DRL development. In other words, experienced practitioners are less likely to seek help for understanding the fundamental DRL concepts because they have already mastered these basics. Moreover, it can be interpreted as the fact that the most challenging steps in the development of DRL applications for DRL practitioners are related to providing an optimized solution for various DRL-related problems, not just addressing a DRL-related problem. Therefore, to fulfill the specific requirements of various DRL developers with different experience levels, it is important to acknowledge that DRL developers have unique needs at different stages of the DRL development journey. Moreover, it should be taken into consideration that the survey was conducted in 2023; nearly a decade after DRL started to become mainstream [25, 223].

To enhance the completeness of our provided taxonomy, we scrutinize 254 sampled commits and issues extracted from real-world DRL-related repositories (Subsection 5.3.3). Upon thorough examination of the sampled commits, we did not identify any instances of challenges being mentioned in relation to the DRL. Furthermore, an analysis of sampled closed issues revealed a consistent pattern wherein users primarily seek support on the utilization of DRL applications offered by the repositories.

According to our 13-year analysis of the distribution of DRL-related SO posts, there has been a drop in the number of SO posts categorized as *DRL issues* after its peak in 2019 (Fig.5.3a and Fig.5.3b). This phenomenon may be attributed to various factors. Initially, it suggests a potential progression in the mastery of fundamental DRL concepts among developers over these years leading to a reduction in challenges encountered and thereby a decrease in the number of DRL-related posts on SO. This can stem from the fact that the growth of DRL popularity in the community results in increased accessibility of DRL tutorial resources including books, tutorials, videos, and papers. These resources aid DRL developers in enhancing their understanding of foundational DRL concepts. Besides, these resources mostly address various DRL problems, including repositories of practical DRL examples that facilitate comprehension of DRL concepts. Moreover, as time has passed, the

accumulation of SO posts regarding DRL development has delivered a rich source of DRL development challenges. As a result, many DRL developers can potentially find answers to their questions among the existing SO posts. It should be also taken into consideration that previous research showed that there is a growing trend toward the number of studies on DRL [194,223], so the drop in DRL-related SO posts does not imply a drop in the popularity of research on this topic.

Finding 3: Survey respondents encountered all the challenges included in our taxonomy. They identified *reward*, *design problem*, *environment*, and *hyperparameters* as the most common challenges in the development of DRL applications. However, this observation highlights a notable contrast with the findings from the analysis of SO posts, wherein *comprehension* with 28.6% of SO posts is the most frequent challenge. The variability observed can be attributed to the fact that DRL practitioners mostly face challenges when they are trying to provide solutions for different DRL problems.

We also ask survey respondents about the severity and needed effort to address the challenges identified in the provided taxonomy (Table 5.4). In general, the majority (exceeding 57%) of the survey respondents indicated that the most frequent challenges from their viewpoint (i.e., *Reward*, *Environment*, *Design problem* and *Hyperparameters*) are major or critical. Moreover, at least 63% of the survey participants considered the level of effort required to address these challenges, to be "Medium" or "High". The majority (more than 52%) of the survey participants consider the other identified challenges to be of *Minor* severity level, and to require a *Low* level of effort. In general, the participants consider that *Installation* and *API usage* challenges require a low level of effort, which might signal that DRL libraries/frameworks have good documentation and usability in general [224].

We compared the time-to-answer of the posts (from different challenges categories) with the effort reported by our survey participants for the different challenges categories and made the following observations:

- *Hyperparameters*, and *design problems* are the subcategories of *DRL issues* that took longer time before receiving an accepted answer. Survey respondents also reported them as severe and requiring a high effort from ML developers.
- The average time required to receive an accepted answer for *State/observation* and *comprehension* SO questions are comparable (even though some SO posts within *comprehension* subcategory took a bit longer to receive an accepted answer than posts belonging to *State/observation*). The survey participants also assessed these two sub-

categories of challenges (i.e., *comprehension* and *State/observation*) as easy to resolve in general.

- *API usage* and *Dependency* challenges are the groups of challenges that took the longest time before their questions received an answer. This result is in contrast with the survey participants' estimation of the effort required to fix them.

The severity and required effort reported by the survey participants for each challenge are strongly correlated (using kendall's tau [225]) in a positive direction [226] and statistically significant ($P - value < 0.05$). Hence, more severe challenges necessitate more effort from developers.

Finding 4: Survey respondents highlighted *Reward*, *Environment*, *Hyperparameters*, and *Design problem* as the most severe and intricate-to-address challenges in DRL development.

RQ3: Are DRL application development challenges language- and/or framework-specific?

We extract information about the programming languages used to develop DRL applications from the collected SO posts. It should be mentioned that the posts have been collected without any distinction on the programming language and frameworks used. Fig.5.6 presents the proportion of posts using *Python* programming language for the different identified categories of challenges. As can be seen, *Python* is by far the dominant programming language for all categories of challenges. However, the proportion of posts mentioning other programming languages and containing *DRL issues* is non-negligible (i.e., 20.2%). This high ratio is mostly attributable to *Java* (5.2% of all posts), *C++* (4.7% of all posts), and *R* (4.7% of all posts) programming languages. It is also noteworthy that investigating the relationship between used programming languages and the challenges (subcategories) within each category reveals that *Python* stands out as the predominant programming language across all DRL development challenges. Based on these results, we conclude that there is no relationship between DRL development challenges and used programming languages. This finding is in accordance with prior research [4, 28] which reported that *Python* is the most popular programming language for ML-enabled applications. These results about programming languages used in DRL applications development are also supported by our validation survey where all participants mentioned *Python* as the programming language they use for developing DRL applications. Besides, 20% of participants reported *C/C++*, and 12% mentioned

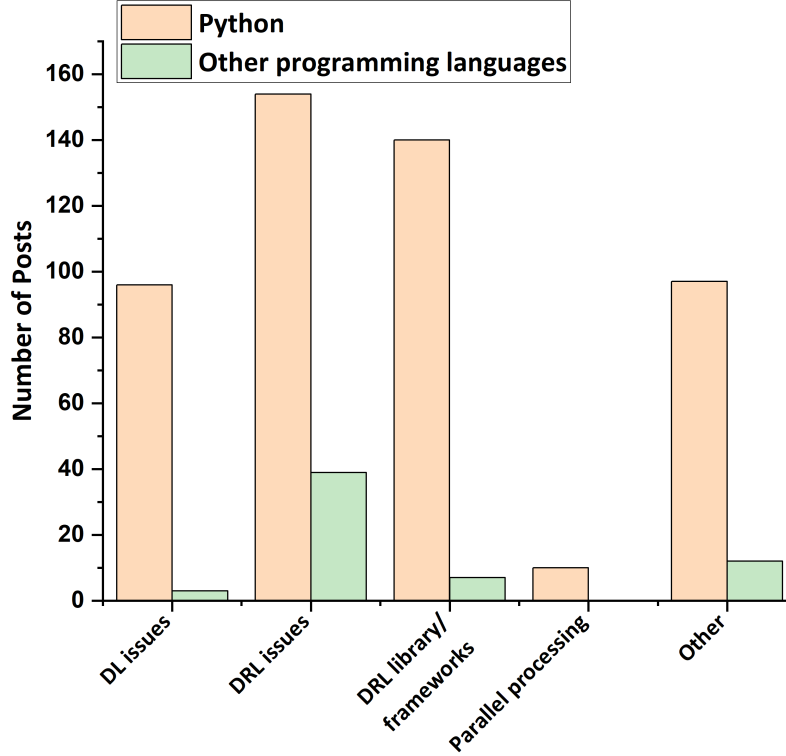


Figure 5.6 Programming languages mentioned in the posts belong to various categories of challenges.

other programming languages in addition to *Python* (e.g., *C#* 4%, and *Java* 3%).

We also examine the mentioned frameworks and libraries in posts related to different challenge categories. Fig.5.7a shows the number of times that each library/framework has been mentioned in posts belonging to various subcategories of the *DRL issues* category. With an exception for the *environment* and *state/observation* subcategories, *Keras*, *Tensorflow*, and *PyTorch* are the most used libraries/frameworks by SO users, and they are the most popular libraries/frameworks for developing ML and DL [212]. Considering that there are several libraries/frameworks specifically designed to ease DRL application development (e.g. *KerasRL*, *RLlib*, etc), Fig.5.7a exposes that SO users usually prefer to use popular ML libraries/frameworks which can be leveraged to implement DRL applications as a subdomain of ML. On the other hand, *gym* [85] is the most popular library/framework in the *environment* subcategory which is reasonable as it is the most popular library for implementing various RL environments and provides a standard benchmark containing a large number of well-known RL environments [227]. This observation can be related to the inherent nature of SO posts, which may not necessarily provide details about the libraries/frameworks employed by the user posting questions. It is noteworthy that 46% of 885 examined SO posts lack any

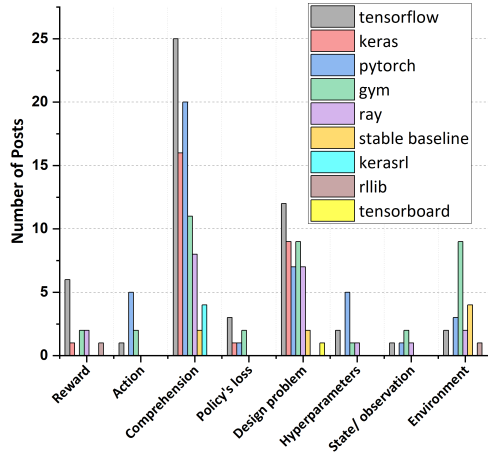
reference to the used DRL-related libraries/frameworks. As an example, post #56312962 serves as an illustrative case where no information has been mentioned regarding the utilized libraries-frameworks to implement the DRL application.

Fig.5.7b presents the libraries/frameworks mentioned in the posts classified as *DL issues*. Results show that *TensorFlow*, *Keras*, and *PyTorch* are the most popular libraries/frameworks in the *DL framework* and *model* subcategories. Given that the challenges within the *DL issues* category pertain to the DL parts of DRL applications, it is not surprising to see *TensorFlow*, *Keras*, and *PyTorch* are frequently mentioned since they are the most used libraries in the development of DL-enabled applications [4, 28]. Moreover, the generality of *Ray* in the *Data preprocessing* subcategory, compared to other libraries/frameworks can be attributed to the fact that *Ray* encompasses not only DRL-related libraries (e.g., *RLlib* [213]) but also various other libraries for a wide range of ML-related tasks at the same time, including scalable datasets, model training, and hyperparameter tuning [228] which may ease developing DRL applications.

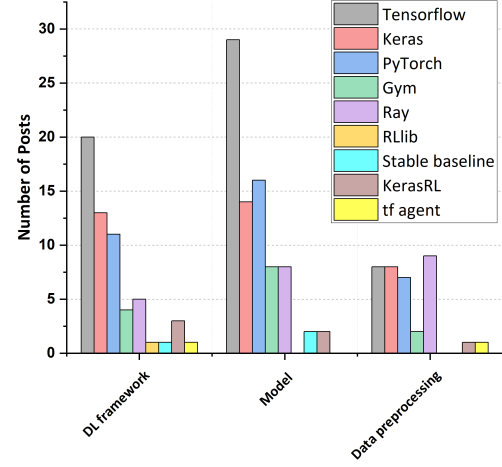
About the most frequently referenced libraries/frameworks in the *DRL libraries/frameworks* category, as illustrated in Fig.5.7c, *gym* received the largest number of questions, especially on the topics of *API usage*, *installation*, and *dependency* challenges. Although *gym* is the most popular library for implementing RL/DRL environments [227], comparing Fig.5.7a and 5.7b may indicate that *gym* has less matured documentation and tutorials, in comparison to *Keras*, *Tensorflow*, and *PyTorch*. It is also worth mentioning that *Keras*, *Tensorflow*, and *PyTorch* are general-purpose ML-related libraries/frameworks which are more developed, compared to *gym* which is implemented specifically for RL development [229].

Regarding parallelization and multi-threading, as can be seen in Fig.5.7d, the majority of issues are reported against *TensorFlow*, particularly regarding *GPU usage* and *distributed processing*. It can be related to the fact that *Tensorflow* is considered the most popular ML framework [230]. On the other hand, most of the *multi-processing* challenges relate to the *ray* framework, which could be attributed to the fact that *ray* supports multiprocessing and RL at the same time [231].

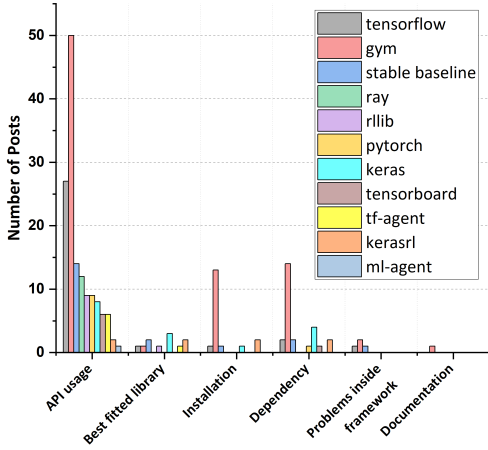
The respondents of our survey corroborated these findings; with 86% of them reporting *PyTorch* as their preferred framework, followed by *TensorFlow* (50%), and *Keras* (39%). Some participants also mentioned *KerasRL* (6%) and *JAX* (6%). We note that *PyTorch* was cited more than *Tensorflow* and *Keras* by participants of our validation survey compared to SO posts. This can be explained by the fact that SO posts do not necessarily include information about the frameworks used by users asking questions. It is also worth mentioning that we collected SO posts over a period of 13 years (from 2009 to 2022), while *PyTorch* was



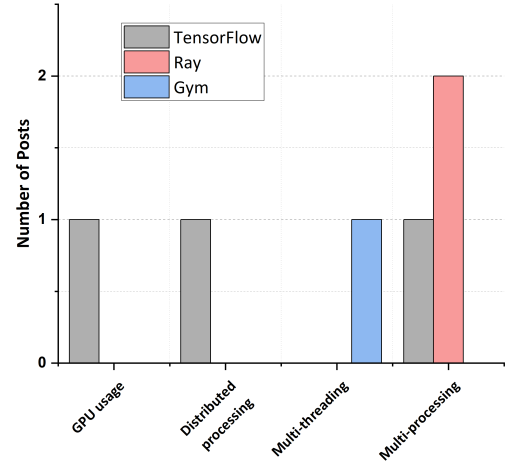
(a) DRL issues



(b) DL issues



(c) DRL libraries/frameworks



(d) Parallel processing & multi-threading

Figure 5.7 Common libraries/frameworks used for developing DRL applications.

introduced only in 2016 (in comparison to *Tensorflow* and *Keras* released in 2015).

Finding 5: *Python* is by far the dominant programming language used for DRL development. While *TensorFlow*, *Keras*, and *PyTorch* are frequently mentioned in posts reporting challenges faced by DRL developers, *gym* is the most challenging libraries to install (see *installation*, and *dependency* challenges) and use (see *API usage* challenges). Our results also show that *ray* is the most challenging library when dealing with *multi-processing* and *data preprocessing* operations.

5.5 Discussion

Based on the findings of our study, in this section, we discuss the state of DRL application development and highlight some research avenues for researchers and practitioners.

Through this study, we gained a thorough understanding of frequently asked questions regarding DRL development to enable the community to explore potential approaches for mitigating these challenges, minimizing errors, and enhancing the reliability of DRL applications. Based on our provided taxonomy, one can see that some challenges faced in the development of DRL applications are common to all types of DL applications. For example, managing dependencies when using DL libraries/frameworks is a prevalent challenge in DL applications. However, dependency management can be more complex in DRL, because of the need for synchronization among a larger number of libraries in the development of DRL applications (e.g., aligning the Python version with the DRL libraries/frameworks and the library that manages RL environment). Similarly to what was suggested by Huang et al. [232] to tackle dependency management challenges in DL applications, DRL researchers can provide a dependency knowledge graph for DRL libraries/frameworks to mitigate this challenge.

Regarding the provided taxonomy, it is worth mentioning that all of the categories and sub-categories of challenges directly relate to the DRL application development. However, some of the challenges may be observed in other ML-related applications. For instance, all of the challenges belonging to *DRL libraries/frameworks* (e.g. *API usage*, *Installation*, *Dependency*, etc.) have been faced by all developers who use ML/DL libraries/frameworks. But it should be also taken into consideration that all of the investigated SO posts in this study have been achieved after a comprehensive filtering process making sure all of the extracted SO posts are about challenges in DRL application development. On the other hand, *Reward*, *Environment*, *Action*, *State/Observation*, and *Policy* challenges are Specific to DRL applications. It should be also taken into consideration that what makes this taxonomy valuable in the DRL community is the fact that their frequency, importance, and severity would be different in DRL application development compared to other ML-related applications. For example, 14.8% of challenges in DRL application development are related to *API usage*, whereas it is only 5.3% in DL-related applications [4].

Finding 2 revealed that 27.3% of DRL development challenges categorized as *comprehension* are related to the lack of sound understanding of basic DRL concepts. In other words, 58.4% of posts belonging to *DRL issues* category (DRL-specific category) are about *comprehension* challenge. This finding highlights the need for documentation and tutorials to help DRL de-

velopers who are not experts in DRL, in the development of DRL applications. A roadmap for the development of DRL applications would also help developers navigate through the implementation of DRL applications with fewer misunderstandings of DRL concepts. The need for such material is emphasized by a post⁴ asking questions about the difference between RL and DRL. By providing a roadmap that systematically expands DRL developers' understanding, developers will be supported in overcoming the most common challenge in DRL application development. An illustration of such guidance is the work conducted by Garg et al. [233] on creating a roadmap for DL development. The need for good documentation and guidance is also emphasized by the survey participants who mentioned that *'although there are a number of tutorials to start working on DRL, a few issues are shared between many of them'*. The participants also noted that many of the DRL tutorial documents cover only a specific domain of DRL. Participants also lamented the poor usability of DRL-related tutorials, claiming that they often contain a lot of unnecessary materials.

Leveraging our Findings 1 and 2, researchers can develop debugging tools to help developers identify the issues early on during DRL application development. Debugging tools can significantly reduce DRL development and maintenance costs. For instance, considering the limited documentation available for most of the DRL libraries/frameworks in comparison with DL libraries for example (e.g., *TensorFlow*), a helpful approach would be proposing techniques and tools to assist DRL developers when using different DRL APIs. This could help mitigate DRL API issues. An example of such techniques focusing on the challenges of software API usage, is the work by Xie et al. [234] which proposes an approach to automatically extract the API parameter constraints of DL libraries/frameworks.

Finding 2 of this study regarding challenges associated with installation and dependency management of DRL libraries/frameworks is aligned with previous studies on dependency management in software development [235] in general, as well as in DL applications [236]. Considering the complex nature of DRL application development (due to the communication of several libraries), challenges regarding libraries/frameworks installation and their dependency management become more intricate, compared to other types of DL applications. This highlights the need for tools (e.g., package manager) to support dependency management. As an example, researchers can provide a tool (such as Maven⁵ for Java) that automates the identification and installation of DRL libraries/frameworks that are best suited for a specific Operating System (OS) and a specific version of *Python*. Additionally, such tools can assist DRL developers in synchronizing the installed DRL-related libraries/frameworks when they need to update some of them.

⁴<https://stackoverflow.com/questions/37973108>

⁵<https://maven.apache.org/>

Our results in Finding 2 also stress the need for supporting tools and documentation, for parallel and distributed DRL application developments. Questions related to this topic took a long time before receiving an accepted answer on Stack Overflow. DRL experts could consider developing pre-configured packages to support parallel/distributed DRL application developments. As an example of the same task in the ML-related development, Openja et al. [230] examined ML application deployment practices on Docker and reported that a significant number of ML developers use Docker to manage dependencies, environment, and the execution of ML applications.

5.6 Threats to Validity

We now discuss threats to the validity of our study.

5.6.1 Construction validity

Our methodology and labeling process can be a potential validity threat. We have thoroughly described our process and the tags used to collect the posts. As no previous taxonomy on this subject exists, we used an open coding approach with multiple rounds and cross-checking to ensure continuous improvement and consistency of the labeling. We further validate our results via a survey with 65 DRL practitioners.

5.6.2 Internal validity

As users do not necessarily provide suitable tags for their questions, our search might have missed some DRL challenges. For example, post *37973108* is related to DRL, but it does not have any specific tag mentioning DRL. Nonetheless, tag usage was necessary for a consistent methodology and we believe that the number of posts gathered and analyzed (927) is sizable enough to provide a good representation of the challenges faced by DRL developers. Moreover, we used a snowballing approach to expand our basic set of DRL-related tags used to extract DRL-related SO posts, similar to previous studies [237]. In addition to extracting posts based on the DRL-related tags, we used a set of keywords to extract DRL-related SO posts without any DRL-related tags to address this issue, as other researchers followed a similar methodology [238]. Another source of threat to the validity of this study arises from the potential overlap between users posing questions in SO posts and the developers associated with the DRL-related GitHub repositories that are used for our survey. To address this concern, we provided a detailed description for each category and subcategory of the taxonomy within the survey, intentionally avoiding any specific information references to any SO

post as an example of challenges. Despite these precautions, the prospect of overlap between these two groups of DRL developers remains a possibility. Although utilizing the duration to receive an accepted answer has been employed in several studies [77,215] as a metric to gauge the difficulty level of SO posts, it may present a potential threat to the internal validity of this research. Reproducibility of issues in ML-enabled systems poses significant challenges and sometimes needs more time compared to traditional software systems [239]. In other words, addressing certain questions in ML-enabled systems (such as those related to parallel processing) requires specific configurations or environments, so the question may be quite simple, but it requires a long time to receive an accepted answer. To address this concern, we also considered the number of posts, which has been used as an indicator to measure the difficulty level of SO posts in a few studies [78,80]. However, as discussed in Section 5.5, this metric yielded similar results to the methodology we employed (i.e., the time to receive an accepted answer).

5.6.3 External validity

While there might exist other DRL challenges that practitioners are facing, we conducted our study using SO which is the largest technical Q&A platform in the software development community. Moreover, all the challenges identified in our provided taxonomy have been validated by our survey participants. Also, respondents of the survey did not report any other challenge that is not included in our provided taxonomy; which is a good result, with respect to completeness. Another external factor that could pose a threat to the validity of our results is the possibility that users who raised SO posts may mostly fall into the category of less experienced users. To mitigate this concern, we examined the top 100 DRL-related GitHub repositories to extract the challenges that GitHub developers mention in their development process. However, it appears that experienced developers generally refrain from detailing their challenges in GitHub commit messages or issues. It is noteworthy that our provided taxonomy may represent high-level categories of challenges, covering almost all aspects of DRL application development. Therefore, refining the taxonomy to offer more detailed categorization could be explored as a potential avenue for future research in this study.

5.6.4 Reliability validity

We described our methodology in detail and provided a replication package [199] to allow others to replicate our results and expand our study.

5.7 Chapter Summary

In this study, we conducted a large-scale empirical study of 927 DRL-related posts extracted from SO. We examined all posts manually to identify the challenges that developers face when developing DRL applications. We found that *Python* is by far the most popular programming language and *TensorFlow*, *Keras*, *PyTorch*, and *OpenAI Gym* are the most frequently used libraries/frameworks for developing DRL applications. We categorized DRL development challenges into five groups including *DRL issues*, *parallel processing & multi-threading*, *DRL libraries/frameworks*, *DL issues*, and *general programming*. An analysis of the received response by the investigated SO posts shows that DRL comprehension, DRL libraries/frameworks API usage, and designing a problem using DRL algorithms are the most challenging parts of DRL applications' development. Furthermore, parallel processing/multi-threading and DRL libraries/frameworks challenges required a longer time to receive an accepted answer. We proposed a taxonomy of challenges and validated it using a survey of 65 DRL developers. The developers confirmed the frequency, severity, and required effort to address identified challenges.

CHAPTER 6 FAULTLOAD BENCHMARK OF BUGS IN MACHINE LEARNING-BASED SYSTEMS

6.1 Chapter Overview

The rapid expansion of ML-based applications across various domains has intensified the focus on ensuring the reliability and quality of ML components. In response, researchers have recently begun developing specialized techniques and tools aimed at improving the reliability of ML components and integrating them safely into broader ML-based systems. Although many of these tools rely on the lifecycle of bugs, there is currently no standardized benchmark to assess their performance, compare their effectiveness, or evaluate their strengths and weaknesses. In this chapter, we first 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.

6.2 Study Design

6.2.1 Objectives

It is generally accepted that standardized benchmarks are the most efficient tools for evaluating and comparing products and methodologies [33]. A benchmark must satisfy some quality criteria to be considered as standard, including relevance, reproducibility, fairness, verifiability, and usability [32,33]. It is also worth mentioning that benchmark construction is a long-term and iterative process that requires the cooperation of the community [91]. 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.

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 [3] and challenges in the engineering of ML-based systems [8], providing reproducible bugs in these systems might require more effort. Although preceding studies have developed some benchmarks of bugs in ML-based systems [101, 240], they totally disregarded the standard benchmark criteria. As an example, *Denchmark* [240] 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 [82, 138] which may bias their evaluation by hiding potential weaknesses. Some others have also used a limited number of real bugs [83, 101] 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?*

6.3 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 6.1 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 [4, 29, 95, 101]. 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* [4, 241], respecting the well-known popularity metrics [242] (e.g., number of stars and number of forks) of their GitHub repositories. Table 6.1 represents the detailed information regarding the popularity metrics of the selected ML frameworks (on the date we checked them).

We narrowed down our study to the bugs in DL systems, based on the main categories of DL

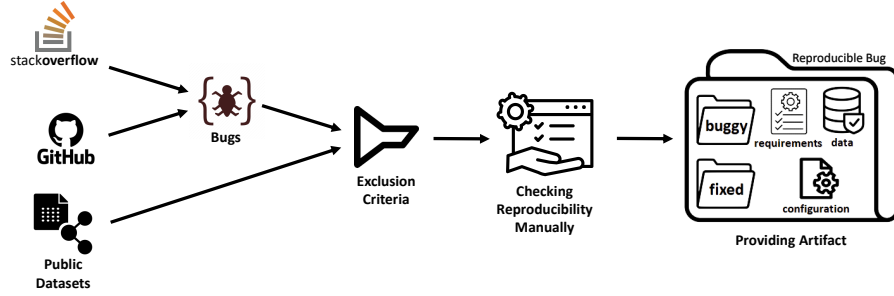


Figure 6.1 Methodology of collecting bugs.

systems faults’ taxonomy provided by [4]: model, tensors and input, training, GPU usage, and API categories. “Model” refers to the bugs related to the structure of the model (such as [243]). “Tensor and input” category covers the bugs regarding the problems in the shape and format of data (such as [244]). “Training” includes the bugs in the model training process (such as [245]). “GPU usage” category deals with bugs occurred when using GPU devices in DL systems (such as [246]). “API” refers to the problems related to the API usage in ML frameworks (such as [247]).

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 infor-

Table 6.1 Detailed information about the selected ML frameworks

ML Framework	#stars	#forks	#subscribers
TensorFlow	174k	88.3k	3.4k
Keras	58.3k	19.3k	2k
PyTorch	66.7k	18.3k	1.7k
Caffe	33.3k	19k	269
Jax	23.1k	2.2k	504
MXNet	20.4k	6.9k	875
CNTK	17.4k	4.4k	201
sonnet	9.6k	1.4k	51

mation 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.

6.3.1 Collecting Bugs from Public Datasets

Zhang et al. [29] 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. [95] 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 reproducible *GitHub* related bugs and 3 SO related. Wardat et al. [101] 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. [4] 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. [82] 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 6.2 Detailed information regarding the collected bugs from public datasets

Dataset	Num of Collected bugs	
	GitHub	SO
Zhang et al. [29]	9	0
Islam et al. [95]	8	3
Wardat et al. [101]	4	14
Humbatova et al. [4]	0	0
Nikanjam et al. [82]	8	10
Total	29	27

Table 6.2 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 repeated in different public datasets. As an example, post [248] exists in three public datasets [82, 95, 101].

6.3.2 Collecting Bugs from GitHub

GitHub¹ 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 [249] 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 [250] to gather repositories.

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 [121, 122]. On the other hand, *Keras* and *TensorFlow* also provide *Python* APIs. As it is mentioned, we analyzed 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

¹<https://github.com/>

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.

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 [251–253]. We also used PyDriller [254], 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 [4]. 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 [4]. To identify ML-related bug-fix commits, the first two authors separately checked the 100 randomly selected bug-fix commits, 50 from repositories developed using TensorFlow and 50 by Keras, attaining 54.7% agreement using Cohen’s Kappa [205]. 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 [255]. 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 [256]. 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 6.3 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.

6.3.3 Extracting Bugs from Stack Overflow

Stack Overflow² (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 [257]. To collect intended posts, we used Stack Exchange Data Explore³ platform, where one can gather information regarding SO posts using SQL queries. Like for data extraction from

²<https://stackoverflow.com/>

³<https://data.stackexchange.com/stackoverflow/query/new>

Table 6.3 Detailed information about the number of remaining bug-fix commits after each filtering step

	ML frameworks	
	<i>TensorFlow</i>	<i>Keras</i>
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

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 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 [258]).
- posts related to the users questions on developing ML/DL components, not resolving a bug (such as [259]).
- posts that their root causes were not ML and they were just the typical programming mistakes (such as [260]).
- 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 [261]).

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 [262] or CIFAR-10 [263] (such as [264]). We utilized *Keras* datasets ⁴ to reproduce bugs belonging to this group. Second, posts that provide the link to achieve the needed data to reproduce the bugs (such as [265]). Third, posts that did not give any description about the required data (such as [244]). 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.

By manually 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 6.4 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*.

6.3.4 Labeling Collected Bugs

To categorize the collected bugs, we used three kinds of labels, firstly based on violated testing property [14], secondly bug type, and finally according to symptoms of bugs [31]. Bug type refers to the class of ML bug taxonomy [4] 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 [205] which is interpreted as almost perfect agreement [255] 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, [245] which is trying to improve the performance of the ML model is categorized as efficiency property. As another example, [243] 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

⁴<https://keras.io/api/datasets/>

Table 6.4 Detailed information about the number of remaining posts after each filtering out step.

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

the third author which result in labeling all bugs consistently, which is used in previous studies successfully [115]. 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 [205], which is known as a moderate agreement [255]. 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 the 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 the 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 [255]. Then, raters labeled the rest of the bugs in three parts (similar to two prior steps) reaching 91.4% agreement using Cohen’s kappa [205]. To achieve consistent labels for all bugs, raters discussed the disagreements with the third author and resolved them (same as previous steps).

6.4 Results

Generally, we manually checked 513 and 498 reported ML-related bugs extracted from 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.

6.4.1 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 [29] and in the benchmark generating on the other hand [33]. 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 [266]). Besides, some of the bugs were not triggered using the mentioned condition in bug-fix commit (such as [267]). 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.

6.4.2 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 [268]. 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 [7]. 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 [4, 29, 95].

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.

6.4.3 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 6.5 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 6.2 also depicts the the distribution of bugs, based on their symptoms.

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 [90–94, 269], we exclude any bugs that are not related to ML. For example, commit [270] 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

Table 6.5 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
	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

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 re-

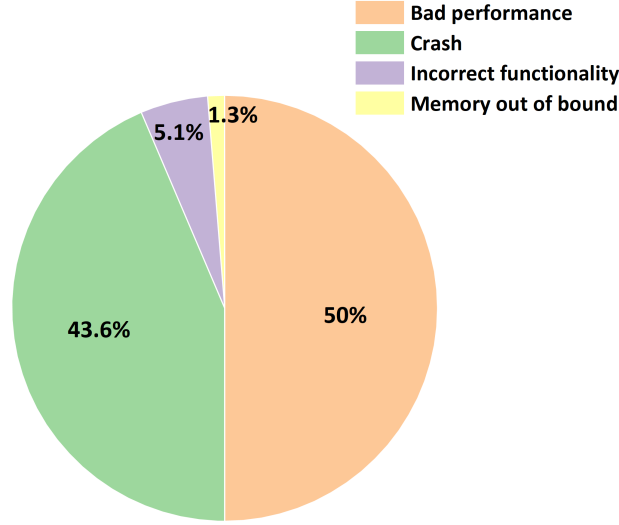


Figure 6.2 Distribution of bugs' symptoms in defect4ML.

ported bug, and the instruction to activate the bug.

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* [4, 241]). Secondly, we have used the taxonomy of bugs in DL programs introduced in [4] 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 message. 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.

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 [95, 101]. 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*⁵. 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* [4, 241]. 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 study 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 [95], and 2) different dependencies specified for

⁵In *requirements.txt* file, that consists of detailed information of dependencies per bug.

every single version of ML frameworks [29]. To meet this challenge, we have delivered a complete list of dependencies (and corresponding versions) needed to run the application⁵. We have also presented specific configuration of each bug including Python version, and process to trigger the bug⁶. Moreover, because of the substantial effect of data in ML components operations [14, 271], 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. Listing 6.1 shows a sample script trying to implement a classifier for *XOR* problem using *Keras* (bug #84 of *defect4ML*) [248]. 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 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.

Artifacts

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

```

1 X = numpy.array([[1., 1.], [0., 0.], [1., 0.], [0., 1.], [1., 1.], [0.,
    0.]])
2 y = numpy.array([[0.], [0.], [1.], [1.], [0.], [0.]])
3 model = Sequential()
4 model.add(Dense(2, input_dim=2, init='uniform', activation='sigmoid'))
5 model.add(Dense(3, init='uniform', activation='sigmoid'))
6 model.add(Dense(1, init='uniform', activation='softmax'))
7 sgd = SGD(lr=0.001, decay=1e-6, momentum=0.9, nesterov=True)
8 model.compile(loss='mean_squared_error', optimizer=sgd)
9
10 model.fit(X, y, nb_epoch=20)

```

Listing 6.1 Sample implementation of XOR classification problem using *Keras* [248].

⁶In *conf.ini* file, that contains the required configuration per bug.

- ***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 [272]. 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 et al. [273]. 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 [274], 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 [109], 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 [275], 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 [14] 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 [14]. 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 Figure 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 [276]. 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 [277]. 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 [278]. 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) [14, 279]. 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 [280,281].

Security refers to the resiliency of an ML system against harmful or dangerous actions by illegal access or manipulation of ML components [282]. 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 [283]. 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 [284].

Efficiency measures consumed computational resources for training or inferring processes in the ML system [14]. 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 [285]. 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 [286].

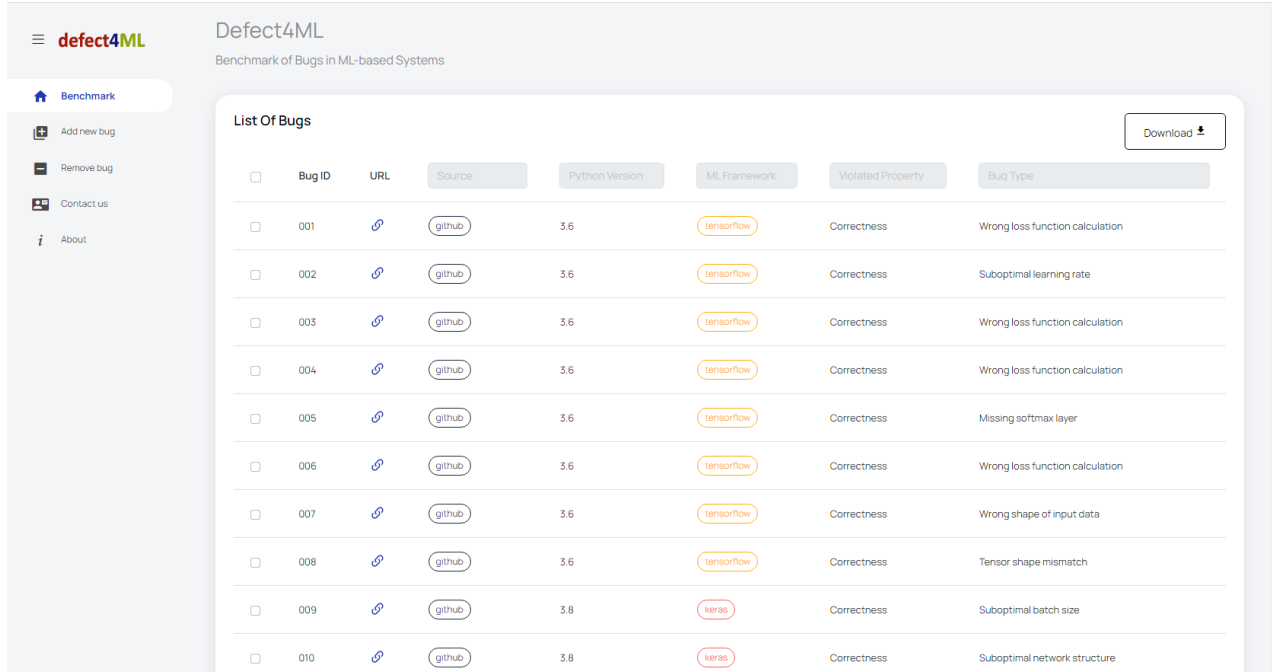
Fairness aims at preventing ML decisions to suffer from ethical issues (e.g. human rights, discrimination, etc.) [287]. In general, human beings have a bias in labeling or collecting data [288]. Fairness ensures that ML decisions are in the right way and free of bias. Unfair models may produce discrimination, 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 [289]. 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 [14,290]. 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 Humbatava et al. [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 6.3 represents a screenshot of the *defect4ML* web application. Users can browse the bugs, and



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

Figure 6.3 A screenshot of *defect4ML* web application.

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.

6.5 Discussion

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* [82] and *DeepLocalize* [101]. *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*.

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 $\times$ refer to the samples that have compile-time errors and could not be used by *DeepLocalize*.

Table 6.6 demonstrates the result of evaluating *NeuraLint* and *DeepLocalize*. The cells filled with $\times$ 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 [82, 291]. 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.

6.6 Threats to Validity

We now discuss the threats to the validity of our study.

Table 6.6 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 pre-processing	no identified error	Error in delta weights
112	SO	Keras	Correctness	Missing flatten layer	missing flatten	×

6.6.1 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 [4]. 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.

6.6.2 External Validity

The most crucial threat to the external validity of this study is its limitation to the *TensorFlow* [117] and *Keras* [118] frameworks. Firstly, we have selected them because they are two of the most popular ML frameworks [4, 241]. 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. Additionally, since we have provided detailed information on the methodology for extracting buggy scripts from various sources and supplying the necessary artifacts, extending the benchmark to include ML bugs from other ML frameworks would be straightforward. This would involve investigating bugs from ML applications using the targeted frameworks and providing the necessary information to meet standard benchmark criteria. 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 [121, 122]. 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.

6.7 Chapter Summary

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 nearly 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 a link to the origin of the bug.

CHAPTER 7 FAULT LOCALIZATION IN MACHINE LEARNING-BASED SOFTWARE: A SYSTEM-LEVEL VIEW

7.1 Chapter Overview

Over the past decade, Deep Learning (DL) has become an integral part of our daily lives, with its widespread adoption in various fields, particularly in safety-critical domains. This surge in DL usage has heightened the need for developing reliable DL software systems, making Software Reliability Engineering (SRE) techniques essential. Given that fault localization is a critical task in SRE, researchers have proposed several fault localization techniques for DL-based software systems, primarily focusing on faults within the DL model. While the DL model is central to DL components, there are numerous other elements that significantly impact the performance of DL components. As a result, fault localization methods that concentrate solely on the DL model overlook a large portion of the system. To address this, we introduce *FL4Deep*, a system-level fault localization technique based on a Knowledge Graph (KG). For the first time, *FL4Deep* considers the entire DL development pipeline to effectively localize faults across the DL-based systems. *FL4Deep* first extracts the necessary static and dynamic information from DL software systems, then generates a KG by analyzing the collected information. Finally, it provides a ranked list of potential faults by inferring relationships from the KG. In an evaluation using 100 faulty DL scripts, *FL4Deep* outperformed four previous approaches in terms of accuracy for three out of six DL-related faults, including issues related to data (84%), mismatched libraries between training and deployment (100%), and loss function (69%). Additionally, *FL4Deep* demonstrated superior precision and recall in fault localization for five categories of faults including three mentioned fault types in terms of accuracy, plus issues related to the insufficient training iteration with 0.89 and 0.62 and activation function with 0.89 and 0.92 for precision and recall, respectively. Sensitivity analysis of *FL4Deep* components also indicates that static information has the most significant impact on the performance of *FL4Deep*.

7.2 Study Design

With the increasing dependence of current software systems on DL components, it is crucial to ensure the reliability of these components. DL-based systems, similar to traditional software systems, are prone to a variety of software faults [4]. One of the most critical tasks in ensuring the reliability of software systems (either DL-based or traditional) is debugging, which focuses

on detecting and fixing faults [292]. Even when faults are known to exist due to the system’s faulty behavior, the process of finding the location of faults’ root causes remains a significant challenge [51]. Fault localization, which plays a key role in debugging, involves identifying the specific location of faults’ root causes within the system and fixing them [293].

In traditional software systems, when a mismatch occurs between the expected and actual outputs, it indicates the presence of a fault in the software system [294]. Similarly, faults in DL software refer to discrepancies between the program’s current behavior and the expected outcome [14]. However, unlike traditional software where the logic is represented through control flow coded by developers, the output in DL-based software is determined by a trained model [101]. As a result, DL-based systems introduce new challenges that cause more complex debugging than traditional software [95]. Consequently, fault detection and localization in DL-based systems are more difficult than in traditional systems. For instance, when a classifier produces an incorrect classification, it does not necessarily indicate a fault in the DL-based systems. Although researchers have increasingly focused on developing testing and debugging techniques for DL-based systems, fault localization has received comparatively less attention [109]. This is expected due to the distinct challenges inherent in fault localization within the context of DL-based systems. Compared to traditional software, the root causes of faults in DL-based software are more varied, located in three main components: 1) the DL program code, 2) the DL framework, and 3) the data used to train the DL models [31, 176]. In this study, we focus specifically on faults within the DL software system code, excluding issues related to DL frameworks and data.

The fundamental differences between DL-based and traditional software paradigms introduce novel types of faults unique to DL-based systems. As a result, debugging techniques developed for traditional software cannot be efficiently applied to DL-based systems [19, 212]. For example, Listing 7.1 shows a script from a SO post discussing a DL-based system encountering ‘bad performance’ (#42081257). The developer who submitted the post indicated that she has been unable to detect and localize the issue, specifically faults within the *loss function* and *metrics* parameters (Lines 15 and 16 of Listing 7.1) These faults are mostly identified through manual code review which is time-consuming and needs DL expertise [82]. Since fault localization is one of the most time-consuming aspects of debugging, it significantly impacts the overall effort required for software debugging [51]. That is, inaccurate fault localization can mislead the debugging process, resulting in wasted time and effort for developers. Therefore, providing fault localization techniques for DL-based systems that consider essential features of these systems can significantly assist DL-based systems’ developers. As illustrated in Fig. 2.2, the DL component consists of various elements. While the DL model is the central element, it represents only a small part of the whole DL components [2]. There-

```

1 model.add(embedding_layer)
2 model.add(Dropout(0.25))
3 # convolution layers
4 model.add(Conv1D(nb_filter=32, filter_length=4,
5                 border_mode='valid', activation='relu'))
6 model.add(MaxPooling1D(pool_length=2))
7 # dense layers
8 model.add(Flatten())
9 model.add(Dense(256))
10 model.add(Dropout(0.25))
11 model.add(Activation('relu'))
12 # output layer
13 model.add(Dense(len(class_id_index)))
14 model.add(Activation('softmax'))
15 model.compile(loss='binary_crossentropy',
16              optimizer='adam', metrics=['accuracy'])

```

Listing 7.1 SO post (#42081257) showing a challenge in finding the root cause of achieving bad performance.

fore, faults within the DL component can originate from any of these elements, not just the DL model itself [20]. Therefore, efficient fault localization for DL-based systems should target not only the DL model itself but also the entire training pipeline and its associated components.

To address this gap, we propose a system-level fault localization technique namely *FL4Deep*, which builds upon Information Retrieval (IR)-based and history-based fault localization approaches. *FL4Deep* targets DL faults across all components of the DL pipeline, including data, the DL model and its training, and components dealing with the DL-based system deployment process. To identify and localize faults, *FL4Deep* extracts both static and dynamic information from DL-based systems and constructs a Knowledge Graph (KG) representing the system’s features such as dataset features used to train DL models, model hyperparameters, used environment to train models, etc. Additionally, it generates an ordered list of potential faults and their root causes, inferred through a set of rules designed based on the commonly known faults in DL-based systems. Evaluation results show that *FL4Deep* outperforms other fault localization approaches (including *DeepFD* [108], *AutoTrainer* [81], *DeepLocalize* [101], and *UMLAUT* [83]) in 83% of the 100 buggy samples used for their comparison, in terms of precision and recall. To summarize, this research makes the following contributions:

- We present the first technique that analyzes the entire pipeline of deep learning-based system development to effectively localize faults.
- We highlight the key challenges in fault localization of DL-based systems.

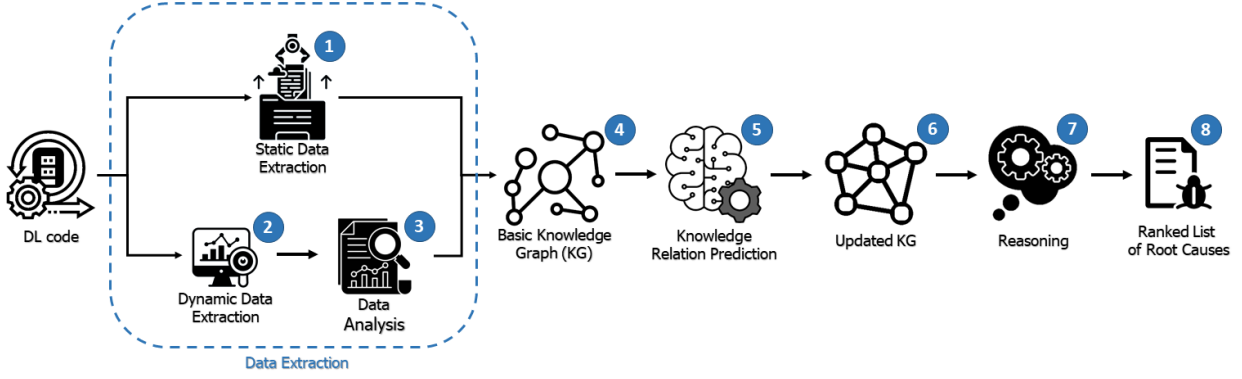


Figure 7.1 High-level view of *FL4Deep* methodology

- We present fault localization techniques originally developed for traditional software systems, which can be adapted to best suit the unique characteristics of deep learning models.
- We provide a dataset of real-world buggy DL codes extracted from SO posts and GitHub repositories.
- We release the source code of *FL4Deep* alongside our datasets to facilitate its use by other researchers [295].

7.3 *FL4Deep*

This section outlines the pipeline of our proposed fault localization technique, *FL4Deep*. Fig. 7.1 provides a high-level view of the methodology we followed in *FL4Deep*. Algorithm 1 represents the pseudocode of the *FL4Deep* algorithm.

7.3.1 Dataset Preparation

In this study, we prepare two datasets of buggy DL code, referred to as the training and validation datasets. The training dataset consists of 75 real-world buggy DL programs, extracted from SO and GitHub repositories. We used this dataset to train the ML models employed in two components of *FL4Deep* represented as (3) and (5) in Fig. 7.1. Of 75 samples, 17 buggy codes were sourced from the research conducted by Cao et al. [108], and 30 were extracted from the *Defect4ML* dataset [28]. Besides, 12 additional buggy samples were reported by both sources. For the 16 remaining buggy samples, we utilized the Stack Exchange Data Ex-

Algorithm 1 *FL4Deep* Algorithm

Input: Dataset, DL code, training environment, deployment environment

Output: Ranked list of fault's root causes

```

 $SI \leftarrow \text{staticInfoExt}(\text{Dataset}, \text{DLcode}, \text{trainingEnvironment})$ 
    // Extracting static information

 $DI \leftarrow \text{dynamicInfoExt}(\text{modelTrainingLogs})$ 
    // Extracting dynamic information

 $KG \leftarrow \text{KgGenerator}(SI, DI)$ 
    // Creating a KG based on the collected information

 $RCs \leftarrow \text{reasoningEngine}(KG)$ 
    // Localizing faults using a reasoning engine

 $\text{RankedRCs} \leftarrow \text{ranking}(RCs)$ 
    // Ranking the identified faults

```

plorer¹, a portal providing an up-to-date database of SO posts. We extracted posts tagged with both *machine-learning*/*deep-learning* and *keras*/*tensorflow*. The first two authors randomly sampled a set of posts for manual review, focusing on those with accepted answers; that included DL code snippets in the questions; and were reproducible. Each selected post was then labeled based on the faults identified in the accepted answers on SO or discussed in the corresponding GitHub Pull Requests (PRs).

The validation dataset includes 100 buggy DL scripts, used to compare the performance of *FL4Deep* with previous fault localization techniques for DL-based software systems. To create this dataset, we utilized 20 SO posts gathered from prior studies [101, 108]. Additionally, we applied various mutation operators specifically designed for DL software systems (such as modifying the loss function and changing the activation function [19, 20]) to these 20 samples to generate new buggy samples. For labeling the validation dataset samples, we used the reported faults from SO posts for those derived from SO. For samples generated using mutation operators, we assigned labels based on the faults introduced by the corresponding mutation operators. All samples in both the training and validation datasets are available in the replication package accompanying this study [295].

7.3.2 Extracting Required Information from DL codes

To extract information from DL software systems from the software under test, we divide the entire DL-based system pipeline into three modules: 1) data preprocessing, 2) model

¹<https://data.stackexchange.com/stackoverflow/query>

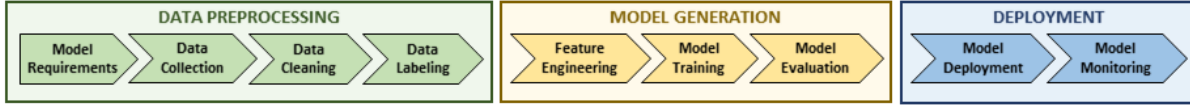


Figure 7.2 Pipeline of DL-based system development process divided into three parts (adopted from [3])

generation, and 3) system deployment. Fig. 7.2 illustrates how the various stages of the DL-based system development pipeline are categorized into these three modules. Algorithm 2 presents the pseudocode of *FL4Deep* information extraction.

FL4Deep utilizes both static and dynamic information (presented as ① and ② in Fig.7.1, respectively) of DL-based systems to identify and localize faults. Static information refers to information within the DL software system which is extracted from the source code before running the DL software system (such as the size of the dataset, DL model structure and its layers, activation functions, optimizer, etc) [296]. To gather necessary static information from DL-based systems, *FL4Deep* takes various DL system components (including the dataset, DL model, training environment, and deployment environment) as input and collects the required information.

Dynamic information is collected during the training of DL-based software which can vary depending on runtime conditions [296]. In *FL4Deep*, we gather model training logs as dynamic information (presented as ② in Fig.7.1). *FL4Deep* extracts neuron weights, accuracy, loss, validation accuracy, and validation loss after each epoch of model training as the dynamic information. Similar to previous studies on fault localization in DL-based systems using dynamic information [108], *FL4Deep* processes the extracted dynamic information using eight statistical operators, as shown in Table 7.1. These operators help capture diagnostic features from the gathered data. For instance, the standard error of the mean indicates how much the sample mean would vary if the study were repeated with new samples from the same population [297].

In the next step, we employ three widely used ML models, i.e., Random Forest (RF) [298], Decision Tree [299], and K-Nearest Neighbors (KNN) [300], to analyze dynamic information and predict potential faults (presented as ③ in Fig.7.1). These models are trained using the training dataset presented in subsection 7.3.1. The results of applying statistical operations to the dynamic information extracted from the 75 training samples are used as input for model training, with the corresponding fault types serving as the output labels. That is, each model is designed as a classifier aiming to predict potential faults based on the extracted

Algorithm 2 *FL4Deep's data extraction*

Input: Dataset, DL code, training environment, deployment environment

Output: static and dynamic info regarding DL code

```

 $S1 \leftarrow \text{Analysis}(\text{dataset})$ 
    // Extracting info regarding dataset

 $S2 \leftarrow \text{Analysis}(\text{trainingEnvironment})$ 
    // Extracting info regarding training environment

 $S3 \leftarrow \text{modelHyperparameters}$ 
    // Extracting info regarding model hyperparameters
repeat
    if end-of-training-epoch then
         $DI.append(\text{modelTrainingLogs})$ 
        // Extracting dynamic info after each training epoch
until end-model-training

 $S4 \leftarrow \text{Analysis}(\text{deploymentEnvironment})$ 
    // Extracting info regarding deployment environment
 $SI \leftarrow \text{combine}(S1, S2, S3, S4)$ 

```

data. Fig. 7.3 represents the process to train these three models. The DL faults that can be identified through dynamic analysis include issues related to the loss function, activation function, optimizer, insufficient iteration, and inappropriate learning rate. These faults are considered the most common issue regarding the model training phase of DL-based software systems [108]. Since a DL software system may encounter multiple faults simultaneously, we use the multi-class versions of these models [301–303]. After obtaining the results from all three models, we apply a majority voting process to aggregate the final prediction of

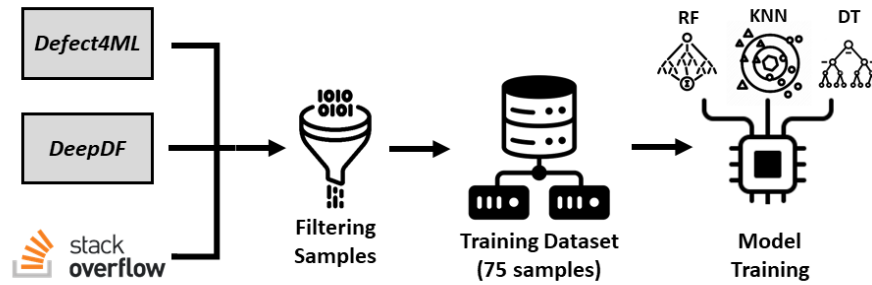


Figure 7.3 Process of creating training dataset and training Random Forest (RF), Decision Tree (DT), and K-nearest Neighbors (KNN) models

potential faults. Algorithm 3 shows the pseudocode of dynamic information analysis. Listing 7.2 shows a code snippet from the SO post #51181393. After extracting dynamic information and applying statistical operations, the result is used as input for all three models. In this case, both the RF and KNN models flagged an issue with the learning rate, while the decision tree model reported no issue. Therefore, through majority voting, the final output identifies a learning rate issue.

Algorithm 3 Prediction of faults based on the dynamic information

Input: Extracted dynamic information

Output: Dynamic Information analysis

```

data ← Process(dynamicInfo)
    // Processing dynamic info using statistical operators

Models ← [RF, DT, KNN]
    // models including Random Forest, Decision Tree, and KNN
faults ← {}
for model in Models do
    F ← predict(model, data)
        // predicting faults using processed dynamic info
    faults.insert(F)

predictedFaults ← majorityVoting(faults)
    // concluding the faults using majority voting

```

7.3.3 Creating KG

In this step, *FL4Deep* constructs a KG to detect faults and localize their root causes. Algorithm 4 represents the pseudocode for constructing the KG and utilizing it to localize faults within the system. The list of the faults identifiable by *FL4Deep*, along with a brief description of each fault is provided in Table 7.2. These faults have been collected from previous studies on various components of DL software systems, such as data [304, 305], model train-

```

1 data = data = linspace(1,2,100).reshape(-1,1)
2 y = data*5
3 model = Sequential()
4 model.add(Dense(1, activation = 'linear', input_dim = 1))
5 model.compile(optimizer = 'rmsprop', loss = 'mean_squared_error', metrics
    = ['accuracy'])
6 model.fit(data,y,epochs =200,batch_size = 32)

```

Listing 7.2 An example of predicting the fault (too small learning rate) by analyzing dynamic information extracted during model training.

Table 7.1 Statistical operators used to analyze extracted dynamic information from model training logs

Operator	Description
min	The minimum value in a feature trace
max	The maximum value in a feature trace
median	The median value of a feature trace
mean	The mean value of a feature trace
var	The variance of a feature trace
std	The standard deviation of a feature trace
skew	The skewness of a feature trace
sem	The standard error of the mean of a feature trace

ing [82, 108, 306, 307], and deployment [20, 28], as well as Q&A forums (e.g., SO posts). These faults are presented in the KG as rules. To generate the KG, *FL4Deep* firstly incorporates the collected static information and the predicted faults obtained from dynamic information analysis as basic facts ((4) in Fig.7.1). Next, KG rules are applied on the basic facts to infer fault-related facts and establish connections between all KG’s facts, including basic and fault-related facts. Fault-related facts refer to the faults identified by *FL4Deep*. Besides, a relationship between a fault-related fact and a basic fact referring to a part of the DL-based system indicates the location of the fault’s root cause. For example, if the KG generator detects that no activation function has been assigned to the model layers, it creates a fault-related fact indicating a ‘missing activation function’ issue in the model structure and links it to the fact representing the DL model.

To derive conclusions from the KG and identify potential faults and their root causes in the DL-based system under test, a reasoning engine is required ((7) in Fig.7.1). The primary goal of the reasoning engine is to simulate rational reasoning, allowing machines to perform complex tasks such as problem-solving, decision-making, and more [308, 309]. For instance, a reasoning engine can be used to determine whether a patient has a specific disease by analyzing symptoms and historical health information.

7.3.4 Predicting Missed Relationship in KG

Link prediction is a fundamental task for KGs used to complete relationships among nodes [310]. *FL4Deep* leverages inductive link prediction to identify and add possible missing relationships within the generated KG (presented as (5) in Fig.7.1). For this purpose, *FL4Deep* employs NodePiece [74], one of the latest and most efficient algorithms for link prediction in KGs. NodePiece is an anchor-based method that learns a connected multi-relational graph

Table 7.2 Fault that can be detected by *FL4Deep*

<i>Fault</i>	<i>Description</i>
suboptimal train/test ratio	split of the dataset into train and test is not optimal [304, 305]
missing preprocessing	data preprocessing which should be done is missing
<i>Python</i> version mismatch	<i>Python</i> versions used in the training and deployed environments are not matched [28]
system architecture mismatch	CPU architecture of the systems used in the training and deployed environments are not matched [20]
OS mismatch	Operating System (OS) used in the training and deployed environments are not matched [20]
libraries mismatch	version of installed libraries and frameworks mismatch in the training and deployed environments [28]
redundant activations	multiple and redundant connected activations can restrict the final activation from utilizing its full output range [306]
biases initialization	it is preferred to initialize biases to zero [307]
units initialization	the weight initialization should not be constant, as it is vital to break the symmetry between neurons [307]
non-linear activation	the activation function for learning layers, such as convolutional and fully-connected layers should be non-linear [306]
loss linkage	the loss function should be properly defined and linked to the final layer’s activation [82]
probability conversion	the final layer should include an activation function to convert the logits into probabilities for classification tasks [82]
suboptimal optimizer	the optimizer should be properly defined and integrated into the computational graph [82]
insufficient iteration	number of epochs is inadequate to reach the best model accuracy [108]
suboptimal learning rate	learning rate is insufficient for achieving good accuracy [108]
loss & activation functions mismatch	loss and activation functions should be matched based on the model structure and data
valid intermediate layer	the intermediate output of the layers should not be “None” or any similar values
wrong activation function	activation function should be defined based on the structure of the data
missing activation function	all models need activation functions in their layers to be able to learn patterns

by generating a combinatorial number of sequences based on the various types of relationships present in the KG. To train the model using NodePiece, we utilize the training dataset,

detailed in Subsection 7.3.1. Fig. 7.4 represents an overall view of the approach we use to train NodePiece.

7.3.5 Ranking the Identified Root Causes

Since *FL4Deep* may detect multiple faults during the localization process, we rank the identified faults to help DL developers prioritize the most probable root causes (presented as ⑧ in Fig.7.1). To achieve this, we base the ranking on the relative frequency of various faults reported in previous studies [4, 20]. That is, according to Humbačová et al. [4] that found faults related to the loss function (accounting for 11.7% of training faults) are more common than those caused by the model optimizer (which account for 3% of training faults), *FL4Deep* orders loss function faults as more likely than optimizer-related faults.

7.4 Results and Analysis

This section presents and discusses the findings of this study, alongside a detailed comparison of *FL4Deep* performance with other state-of-the-art fault localization approaches for DL-based systems. To this end, we aim to answer the following Research Questions (RQs):

RQ1. [Evaluation] *How does FL4Deep compare against other fault localization approaches for DL-based systems?*

RQ2. [Sensitivity Analysis] *To what extent does each component contribute to the overall performance of FL4Deep?*

RQ3. [Error Analysis] *What are the characteristics of faults misclassified by FL4Deep in DL-based systems?*

All materials utilized to answer these RQs, such as the collected datasets and the source code of *FL4Deep*, are accessible in our replication package [295].

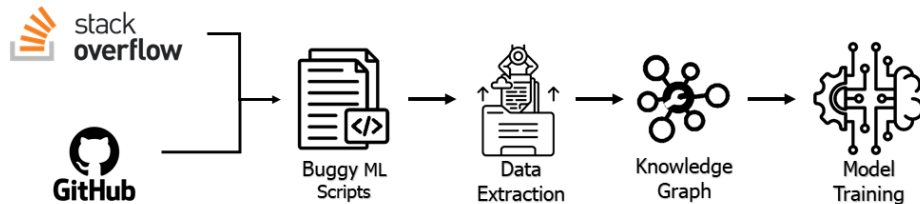


Figure 7.4 Training a model for KG link prediction

Algorithm 4 Localization of faults using generated KG

Input: Extracted data (static and dynamic)

Output: Ranked list of fault's root causes

```

data  $\leftarrow$  predictedFaults + statitcInfo

KG  $\leftarrow$  KgGenerator(data)
    // Generating a KG based on the extracted and analyzed info

updatedKG  $\leftarrow$  KgLinkPrediction(KG)
    // Predicting missed KG's relationship using NodePiece

RCs  $\leftarrow$  reasoningEngine(updatedKG)
    // Localizing faults by reasoning on the created KG

RankedRCs  $\leftarrow$  rank(RCs)
    //Ranking identified faults based on their frequency
  
```

7.4.1 Experimental Design

Given the widespread popularity of *Keras* and *TensorFlow*, as shown by the metrics in Table 6.1, our focus in this study is on DL software systems developed using these two leading frameworks. To extract the static and dynamic information required by *FL4Deep*, we implement three main components to gather data related to the dataset, model training and its environment, and system deployment. Listing 7.3 provides a sample of how *FL4Deep* can be used within the code of a DL-based application. As demonstrated, integrating *FL4Deep*'s APIs into DL-based applications is straightforward and does not require specialized expertise in DL or its development. To collect data-related information, *FL4Deep* extracts various dataset details such as the number of features, the number of rows in the training set, the number of rows in the test set, etc. (illustrated as ① in Listing 7.3). For static information (such as model structure, layers, activation functions, loss function, optimizer, etc.) and dynamic information related to the DL model and its training environment, we implement a *Keras* callback function (illustrated as ② in Listing 7.3). The dynamic information collected includes neuron weights, accuracy, loss, validation accuracy, and validation loss, which are recorded at the end of each training epoch.

To generate KG from the extracted information, we use RDFLib [311], a Python library for the Resource Description Framework (RDF). RDF provides a foundation for decision-making, moving beyond a system of locally trusted facts. Moreover, we utilize Notation3 (N3) [61], a logic language known as a superset of RDF to implement KG. N3 generally enhances RDF by extending its representational capabilities and enabling decision-making through operations on data, information access, and reasoning. Besides, to infer the possible faults and their root causes from the generated KG, we use the 'Euler Yet another proof Engine' (EYE) [312] as

```

1 (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
2 data_analysis(train=x_train, target=y_train, test=x_test) ①
3 model = keras.Sequential([
4     keras.Input(shape=input_shape),
5     ...
6     layers.Dense(num_classes, activation="softmax")
7 ])
8 model.compile(loss="categorical_crossentropy",
9               optimizer="adam", metrics=["accuracy"])
10 model.fit(x_train, y_train, batch_size=128, epochs=10, validation_split
11           =0.1,
12           callbacks=[
13               fl4ml(batch = 128, epochs = 10 ,data=[x_train, y_train, x_test,
14               y_test]) ②
15           ])

```

Listing 7.3 An example usage of *FL4Deep* within a DL code

the reasoning engine of *FL4Deep*. EYE reasoning engine supports RDF and implements N3. EYE is notably more expressive and significantly outperforms other reasoning engines [313]. For example, EYE can solve the Deep Taxonomy Benchmark problem with 100,000 triples in just 4.8 seconds, compared to the CWM reasoner, which takes 9 days and faces out-of-memory issues when using Jena [314,315]. Moreover, to implement NodePiece which is used by *FL4Deep* to train a model for predicting possible missed relationships in the created KG, we have used PyKeen [316] *Python* library.

7.4.2 RQ1. How does *FL4Deep* compare against other fault localization approaches?

To assess the effectiveness of our proposed approach, we use our validation dataset that includes 100 buggy DL samples to compare our approach with previous approaches, including *DeepFD* [108], *DeepLocalize* [101], *AutoTrainer* [81], and *UMLAUT* [83]. To run these approaches, we use their related replication packages that are publicly available. Since our evaluation is based on 100 buggy DL samples, a sample set of gathered results using various approaches is presented in Table 7.3. Besides, Table 7.4 presents the number of faults identified by each approach.

Regarding data-related issues in training DL models, only *UMLAUT* and *FL4Deep* successfully identify the associated faults. While *UMLAUT* successfully localizes 42% of data-related faults in DL software systems, *FL4Deep* outperforms it by accurately identifying and localizing 84% of these faults. For faults caused by mismatches between installed libraries in the training and deployment environments, *FL4Deep* detects 100% of such issues, owing to

Table 7.3 Sample results of comparing *FL4Deep*, and other popular fault localization techniques for DL-based systems

Ref #	Fault localization techniques				
	<i>UMLAUT</i>	<i>DeepFD</i>	<i>AutoTrainer</i>	<i>DeepLocalize</i>	<i>FL4Deep</i>
34311586	(1) Critical: Missing Softmax layer before loss (2) Warning: Last model layer has nonlinear activation	1:[lr] (Lines:27)	–	Batch 0 layer 2: Error in Weights, terminating training	1. suboptimal learning rate 2. wrong activation function
37624102	(1) Critical: Missing Softmax layer before loss (2) Critical: Missing activation functions (3) Warning: Last model layer has nonlinear activation (4) Error: Image data may have incorrect shape (5) Warning: Learning Rate is high (6) Warning: Check validation accuracy	1:[lr] (Lines:66) 2:[Act] (Lines:54, 56, 61, 64)	unstable	Batch 0 layer 9: Error in Output Gradient, terminating training	1. suboptimal learning rate 2. wrong activation function
41600519	(1) Error: Input data exceeds typical limits (2) Critical: Missing Softmax layer before loss (3) Warning: Last model layer has nonlinear activation	1:[loss] (Lines:32)	unstable	Batch 0 layer 6: Error in forward terminating training	1. loss linkage
47352366	(1) Critical: Missing Softmax layer before loss (2) Warning: Last model layer has nonlinear activation	1: [opt] (Lines:40)	explode	Layer-12 Error in delta weights Stop at epoch 1, batch 24	1. loss & activation function mismatch 2. suboptimal optimizer
48385830	(1) Critical: Missing Softmax layer before loss (2) Warning: Possible overfitting	1:[act] (Lines:-) 2:[loss] (Lines:57)	explode	Layer-1 Error in forward Stop at epoch 1, batch 2	1. missing activation function 2. loss linkage
50079585	(1) Critical: Missing Softmax layer before loss (2) Critical: Missing activation functions (3) Warning: Last model layer has nonlinear activation	1:[lr] (Lines:44) 2:[epoch] (Lines:15)	unstable	–	1. suboptimal learning rate 2. insufficient iteration
55328966	(1) Error: Input data exceeds typical limits (2) Warning: Possible overfitting (3) Warning: Check validation accuracy (4) Critical: Missing Softmax layer before loss (5) Critical: Missing activation functions (6) Warning: Last model layer has nonlinear activation	1:[opt] (Lines:49)	explode	–	1. loss & activation functions mismatch 2. suboptimal optimizer
59282996	(1) Error: Input data exceeds typical limits (2) Warning: Check validation accuracy (3) Critical: Missing Softmax layer before loss	1:[epoch] (Lines:309)	unstable	–	1. suboptimal optimizer 2. wrong activation function

specific rules implemented for localizing deployment faults. In contrast, none of the other approaches from previous studies are capable of identifying these issues. Faults related to loss and activation functions are among the most frequent in DL software systems [4], and all studied approaches can detect them to varying degrees of accuracy. For loss function-related faults, *FL4Deep* and *DeepLocalize* achieve the best performance, detecting 69% and 63% of these issues, respectively. In terms of activation function faults, *UMLAUT* leads with 100% accuracy, followed by *FL4Deep* with 92%, and *AutoTrainer* with 38%. Another significant source of faults in DL-based systems is the optimization function. Regarding optimization-related faults, *DeepFD* with 50%, *FL4Deep* with 30%, and *DeepLocalize* with 10% have the best accuracy. Lastly, faults caused by an insufficient number of training iterations are only detected by *DeepFD* and *FL4Deep*, with 60% and 53% detection rates, respectively.

Although *FL4Deep* does not achieve the best performance for all fault types, it demonstrates the most consistent overall performance among all approaches. Specifically, *FL4Deep* strikes a balance in identifying and localizing a wide range of DL faults. For instance, while *UMLAUT*

outperforms *FL4Deep* in identifying faults related to activation functions, its performance for other fault types is relatively low compared to other approaches. Similarly, *DeepFD*, which excels in localizing faults related to insufficient iterations and optimization functions, performs poorly for other types of faults. In summary, while *FL4Deep* leads in three out of the six examined DL fault types, it ranks as the second-best approach for the remaining fault types. In contrast, other methods tend to perform well for only one fault type and show significantly weaker performance across the rest.

Finding 1. Based on the accuracy of identifying and localizing DL-related faults, *FL4Deep* outperforms other techniques for faults in data (84%), loss function (69%), and mismatch between installed libraries on the training and deployment environment (100%).

7.4.3 RQ2. To what extent does each component contribute to the overall performance of *FL4Deep*?

This section highlights the contribution of each component of *FL4Deep* to the overall approach through an ablation study. Using the same 20 samples we used to compare approaches, we assess the sensitivity of *FL4Deep* to its components [317]. The core idea of an ablation study is to systematically remove specific components during each execution and compare the results against a baseline. In this context, we establish the baseline by running *FL4Deep* with all components. Since *FL4Deep* relies on static information, dynamic information, and KG link prediction as its key components, we designed the ablation study to evaluate the impact of removing each component in separate execution scenarios. Table 7.5 presents the results of executing *FL4Deep* under different conditions, showing the effect of removing various components on overall performance.

As the results demonstrate, removing static information has the most significant impact on the performance of *FL4Deep*, leading to a 67% performance reduction. This substantial decrease is due to the fact that static information is crucial for identifying a wider range of fault types compared to other components. For instance, data-related faults can be detected through the use of extracted static information. Besides, the number of faults that their identification and localization highly depend on static information is higher than ones relying on dynamic information. Dynamic information and KG link prediction follow with 16% and 10% impacts, respectively. The relatively lower influence of the KG link prediction component may be attributed to the limited size of the training dataset, as dataset size has a direct effect on model performance [318]. To address this, we plan to enrich our training dataset as a

Table 7.4 Results of comparing the proposed method with previous studies

Issue Type	# samples	<i>DeepFD</i>	<i>DeepLocalize</i>	<i>AutoTrainer</i>	<i>UMLAUT</i>	<i>FL4Deep</i>
Data	19	0	0	0	8	16
Mismatch framework/libraries	20	0	0	0	0	20
Loss function	16	5	10	5	0	11
Insufficient iteration	15	9	0	0	0	8
Optimization function	10	5	1	0	0	3
Activation function	26	2	2	10	26	24

future work, which is expected to improve the efficiency of the KG link prediction component in *FL4Deep*.

For statistical analysis of the results, we employed Fisher’s exact test [319]. This choice is justified by our aim to evaluate the statistical significance between pairs of values (e.g., baseline and SI) given our sample size of approximately 100. [161]. We leverage the SciPy Python library [320] to conduct the test. The results show a significant difference in *FL4Deep*’s performance when the static information component is removed ($p\text{-value} = 0.04$). However, for the scenarios involving dynamic information and KG link prediction removal, no significant differences are found, with $p\text{-values}$ of 0.06 and 0.10, respectively. It is worth noting that $p\text{-values}$ between 0.05 and 0.10 are interpreted as marginal significance, meaning that the evidence against the null hypothesis is weak but not entirely negligible [321].

Finding 2. Static information makes the most substantial contribution to *FL4Deep*’s performance, with its removal causing a 67% performance drop and showing a statistically significant difference. In contrast, KG link prediction has the least impact on *FL4Deep*’s performance, possibly due to the small size of the training dataset.

Table 7.5 Results of *FL4Deep* sensitivity analysis by removing Static Information (SI), Dynamic Information (DI), and KG Link Prediction (LP) components

Issue Type	# samples	<i>FL4Deep</i>	Removed Component		
			<i>SI</i>	<i>DI</i>	<i>LP</i>
Data	19	16	5	16	15
Mismatch framework/libraries	20	20	0	20	20
Loss function	16	11	5	8	9
Insufficient iteration	15	8	8	2	6
Optimization function	10	3	3	1	2
Activation function	26	24	6	22	22

7.4.4 RQ3. What are the characteristics of faults misclassified by *FL4Deep*?

Given that precision and recall are widely used metrics for comparing the effectiveness of different methods [322], we also assess the performance of the studied approaches using these metrics. Table 7.6 presents the *False Positives* (FP), *False Negatives* (FN), *Precision* (PR), and *Recall* (RC) for identifying various ML-related faults.

For data-related faults in DL-based systems, *FL4Deep* and *UMLAUT* are the only ones capable of detecting such faults. *FL4Deep* achieves the highest performance with precision and recall values of 100% and 84%, respectively in detecting data-related faults, while *UMLAUT* stays behind with precision and recall values of 23% and 44%, respectively. It is also worth mentioning that we do not observe any FP for *FL4Deep*, where the results of *UMLAUT* show 27 FP. In terms of identifying faults in the loss function, *FL4Deep*, *DeepFD*, and *DeepLocalize* outperform other methods. Notably, although *DeepFD* and *DeepLocalize* exhibit opposite trends in precision and recall, *FL4Deep* performs better on both metrics (with 85% for precision and 69% for recall). In other words, *FL4Deep* with only 2 FP in localizing faults regarding loss function shows better performance. When it comes to issues caused by insufficient training iterations, *DeepFD* and *FL4Deep* demonstrate nearly identical performance, with *FL4Deep* showing slightly better results, in terms of both precision and recall. Besides, *FL4Deep* performs better concerning 1 FP, in comparison with 2 FP for *DeepFD*. For detecting and localizing issues in the optimization function, *DeepFD* performs the best, with precision and recall of 36% and 50%, respectively, while *FL4Deep* ranks second. However, it should be taken into account that *FL4Deep* results in fewer FP for localizing faults regarding optimization functions. For faults related to the activation function, although *UMLAUT* achieves the highest recall, its precision is just 26%, significantly lower than its recall. This large gap between precision and recall is due to *UMLAUT*'s tendency to report activation issues for all tested faults. *AutoTrainer* also identifies activation function issues, achieving a precision of 91%, but its recall is 38%, again showing a significant disparity between the two metrics. In contrast, *FL4Deep* stands out as the best-performing method in this category, with a well-balanced precision of 89% and recall of 92% for localizing activation function faults. Besides, *FL4Deep* with 3 FP in localizing faults related to the activation function stays at the second rank, with respect to the number of FP.

For data-related faults in DL-based systems, *FL4Deep* and *UMLAUT* are the only methods capable of detecting such faults. *FL4Deep* delivers the best performance, achieving precision and recall rates of 100% and 84%, respectively, while *UMLAUT* lags behind with precision and recall values of 23% and 44%, respectively. Notably, *FL4Deep* produces no FP, whereas *UMLAUT* records 27 FPs. In terms of identifying faults in the loss function,

FL4Deep, *DeepFD*, and *DeepLocalize* outperform other approaches. Although *DeepFD* and *DeepLocalize* display opposite trends in precision and recall, *FL4Deep* surpasses both, with 85% precision and 69% recall, and only 2 FPs for loss function-related faults. For issues arising from insufficient training iterations, both *DeepFD* and *FL4Deep* demonstrate similar performance, though *FL4Deep* slightly edges out in both precision and recall. Additionally, *FL4Deep* gives only 1 FP, compared to 2 FPs for *DeepFD*. When it comes to detecting and localizing issues in the optimization function, *DeepFD* performs the best, with precision and recall values of 36% and 50%, respectively, while *FL4Deep* ranks second. However, *FL4Deep* produces fewer FPs in localizing faults related to the optimization function. For faults related to the activation function, while *UMLAUT* achieves the highest recall, its precision is just 26%, significantly lower than its recall. This disparity stems from *UMLAUT*'s tendency to flag activation issues for all tested faults. *AutoTrainer* also detects activation function issues with 91% precision, but its recall is only 38%, indicating a notable gap between the two metrics. In contrast, *FL4Deep* emerges as the top-performing method in this category, with balanced precision and recall values of 89% and 92%, respectively. Additionally, *FL4Deep*, with 3 FPs in localizing activation function faults, ranks second in terms of the number of FPs.

Finding 3. According to the *Precision* and *Recall* of localizing DL faults, *FL4Deep* has better performance for issues related to the data, mismatch between installed libraries on the training and deployment environments, loss function and insufficient training iterations. Although regarding faults within activation functions *AutoTrainer* and *UMLAUT* have the best precision and recall respectively, *FL4Deep* has better performance with respect to the balance between precision and recall. Besides, *FL4Deep* produces fewer FP in localizing all types of faults, except faults regarding the activation function that its number of recorded FP stays at the second place.

As the results indicate, *DeepFD* outperforms *FL4Deep* in identifying faults related to ‘insufficient iterations’ and ‘optimization function’. This difference can largely be attributed to the fact that *DeepFD* executes the buggy scripts 10 times, whereas *FL4Deep* relies on a single execution. It is well-acknowledged that faults related to ‘insufficient iterations’ and ‘optimization function’ are best identified by analyzing and monitoring the dynamic information during model training [24, 323]. Given the inherent randomness in DL, it is reasonable that results based on multiple executions tend to be more accurate than those from a single execution. However, it is important to consider that running multiple training executions can be highly costly and time-consuming [324]. On the other hand, *UMLAUT* shows the

Table 7.6 Comparison of False Positive (FP), False Negative (FN), Precision (PR), and Recall (RC) of various approaches in localizing ML-related faults

Issue Type	<i>DeepFD</i>				<i>DeepLocalize</i>				<i>AutoTrainer</i>				<i>UMLAUT</i>				<i>FL4Deep</i>			
	FP	FN	PR	RC	FP	FN	PR	RC	FP	FN	PR	RC	FP	FN	PR	RC	FP	FN	PR	RC
Data	0	19	0	0	0	19	0	0	0	19	0	0	27	10	0.23	0.44	0	3	1.00	0.84
Mismatch framework/libraries	0	20	0	0	0	20	0	0	0	20	0	0	0	20	0	0	0	0	1.00	1.00
Loss function	3	11	0.63	0.31	20	6	0.33	0.63	18	11	0.22	0.31	0	16	0	0	2	5	0.85	0.69
Insufficient iteration	2	6	0.82	0.60	0	15	0	0	0	15	0	0	0	15	0	0	1	5	0.89	0.62
Optimization function	9	5	0.36	0.50	15	9	0.06	0.10	0	10	0	0	8	10	0	0	7	7	0.30	0.30
Activation function	3	24	0.40	0.08	18	24	0.10	0.08	1	16	0.91	0.38	74	0	0.26	1.00	3	2	0.89	0.92

best performance in localizing faults related to the ‘activation function’, with an 8% higher accuracy than *FL4Deep*. However, this result can be explained by the fact that *UMLAUT* always reports ‘activation function’ faults for all tested scripts. As such, this higher performance should not be interpreted as better fault localization. In fact, *UMLAUT*’s very low precision (0.26%) in identifying ‘activation function’ faults confirms that its approach lacks accuracy for this type of fault.

Finding 4. Although *DeepDF* outperforms *FL4Deep* in localizing faults related to ‘insufficient training iteration’ and ‘optimization function’, it is more computationally expensive in terms of execution time and resource consumption. Furthermore, while *UMLAUT* surpasses *FL4Deep* in identifying faults in the ‘activation function,’ it consistently reports this fault for all tested DL codes, regardless of the presence of any fault in the ‘activation function’.

7.5 Discussion

7.5.1 Challenges in Fault Localization of DL-based Systems

Due to the specific challenges in testing and debugging DL-based systems, fault localization in these systems also presents unique difficulties. One major challenge is the reliance on test cases, which are fundamental to traditional software fault localization [36]. However, the DL testing community widely accepts that generating effective test cases for DL software systems is particularly challenging [325]. This difficulty stems from the inherent stochasticity in DL systems, which introduces uncertainty in their outputs. Additionally, the design of training data is crucial for both the performance and accuracy of DL models. Consequently, creating accurate test cases for DL software systems requires expertise not only in ML but also in related fields like data management [326].

DL frameworks (such as *Keras*, *TensorFlow*, and *PyTorch*), which are designed to simplify the development of DL-based systems, play a significant role in modern ML development [14].

However, creating test cases based on these frameworks can be particularly challenging due to internal faults [176, 327], rapid changes in their APIs [328, 329], and alterations in their internal operations. For example, Islam et al. [95] reported that approximately 26% of *TensorFlow* operations were modified between versions 1.10 and 2.0.

Beyond the previously mentioned challenges, the inherent randomness and uncertainty in DL can lead to different results with each execution of a DL software system [14]. To address this issue, some fault localization techniques for DL-based systems involve running the application multiple times. For instance, DeepFD [108] localizes faults by analyzing data from 10 runs of the application. However, since model training is a highly time-consuming process, this approach can significantly impact the performance of the fault localization technique.

FL4Deep tackles these challenges by leveraging both static and dynamic information, which reduces the impact of randomness in the fault localization process and eliminates the need to execute DL-based systems multiple times to localize faults. Furthermore, *FL4Deep* is implemented using *Keras 2.8* and *TensorFlow 2.8*, ensuring compatibility with all *Keras* and *TensorFlow* versions in the 2.x series.

7.5.2 Most Fitted Fault Localization Techniques for DL-based systems

In traditional software systems, there are several approaches to localize faults. We discuss here their applicability to DL-based software systems. Spectrum-Based Fault Localization (SBFL) [40] relies on analyzing the dynamic behavior of the software during execution. However, as highlighted in the previous section, the inherent stochasticity of DL systems poses challenges for generating test cases, a known issue referred to as the oracle problem in DL testing [330]. This challenge significantly impacts the effectiveness of SBFL. Learning-to-rank fault localization techniques, which build on SBFL, also encounter similar issues due to the reliance on the dynamic behavior of the system and test case generation. Mutation-Based Fault Localization (MBFL) [43] faces challenges related to the time required for fault localization. Given that model training (the core aspect of DL-based systems) is a time-intensive process, this can severely hinder the efficiency of MBFL approaches, where we need to run applications against various mutants. Techniques such as predicate switching and fault localization through program slicing, which rely on program code or control flow analysis, are not well-suited for DL-based systems [331]. This is because all lines of code within DL components are typically executed during each run of the DL software system, making these techniques less effective. Moreover, fault localization methods based on stack trace analysis may be inadequate for DL-based systems, as they often lack access to low-level traces from DL framework operations.

In contrast, history-based fault localization techniques present fewer challenges. These approaches analyze the probability of specific faults being linked to various elements of the application by examining the history of fault occurrences, making them more adaptable to DL-based systems. Information Retrieval (IR)-based techniques may also encounter fewer difficulties compared to others when applied to DL-based systems, in case they work based on the static information of the DL-based systems. The challenges of extracting relevant static information from DL software systems can be managed without significantly affecting the efficiency of the fault localization process.

FL4Deep provides a fault localization approach for DL-based systems by combining IR-based and history-based techniques, both of which are well-suited to the unique characteristics of DL-based systems. This approach enhances the accuracy and performance of the fault localization process for DL-based systems.

7.6 Threats to validity

7.6.1 Construct Validity

A potential limitation of this study stems from the selection of buggy scripts used to train DL models, which were central to both our proposed approach and comparison with prior approaches. To select buggy DL code, we relied solely on real-world samples sourced from SO posts and GitHub repositories. Specifically, we used publicly available benchmarks of DL bugs [28, 101, 108] to extract relevant samples, and supplemented them with additional buggy scripts identified through a review of SO posts related to DL faults. To compare *FL4Deep* with existing fault localization approaches for DL-based systems, we used buggy DL scripts previously employed by other studies to demonstrate the effectiveness of their approaches. Moreover, we applied several mutation operators on them [19, 20] to generate additional, unseen samples for a more comprehensive evaluation. Furthermore, to minimize bias in the evaluation, we employed a set of buggy samples that had been used in previous approaches for comparison.

7.6.2 Internal Validity

The first internal threat to the validity of this study is the limited number of buggy scripts used to train the ML models in various components of *FL4Deep*. This constraint arises because we only utilized real-world buggy scripts sourced from SO and GitHub. Moreover, we focus solely on reproducible buggy samples, filtering out non-reproducible ones which further limits the size of the training dataset. Bug reproducibility is a well-known challenge

in testing DL-based software systems, considering it has been reported that only about 3.34% of DL bugs reported on SO are reproducible [28, 239]. Another potential threat to internal validity is the selection of KNN, Random Forest (RF), and Decision Tree models for analyzing dynamic information. Although these are relatively simple ML models, they are efficient and widely used [298, 300]. Moreover, given the limited amount of data available for training, these models are likely more effective than more complex alternatives.

7.6.3 External Validity

The primary threat to the external validity of *FL4Deep* is its limitation to DL software systems developed using *Keras* or *TensorFlow*. These frameworks were selected due to their popularity as the two most widely used DL frameworks [241]. It is also worth mentioning that *FL4Deep* can be extended to other DL frameworks such as *PyTorch*. Since we implemented a Keras callback function to extract the necessary data for fault identification and localization in ML-based systems, extending this approach to other ML frameworks would simply require developing a strategy to extract the required data based on the capabilities of the targeted framework. Another potential threat to the external validity of this research is our focus on DL-based systems implemented using *Python*. Given that *Python* is the most commonly used language for developing DL software systems [121, 122], we believe that the results and conclusions of this study can be generalized to the majority of DL-based systems. Scalability could be another potential threat to the validity of the proposed approach, as we need to capture the required information during the training phase of ML-based systems. However, it is important to note that this is a known challenge in fault localization for ML-related software systems, and existing approaches also face this issue.

7.6.4 Reliability validity

We explained the methodology used in *FL4Deep* in detail and provided a replication package [295] allowing others to reproduce our results and expand our proposed methodology.

7.7 Chapter Summary

In this study, we introduced *FL4Deep*, a fault localization technique designed for DL-based systems. Unlike existing fault localization methodologies that primarily focus on the DL model and its training, *FL4Deep* takes a system-level approach, addressing the entire DL system pipeline. Additionally, *FL4Deep* leverages both static and dynamic information from DL-based systems to enhance the accuracy of fault identification and localization. Our

comparison of *FL4Deep* with four previously published fault localization techniques for DL-based systems demonstrates that *FL4Deep* outperforms these methods in three out of six fault categories, based on accuracy. Moreover, *FL4Deep* shows superior performance in four out of six fault types when evaluated by precision and recall. For future work, we aim to expand our training dataset by incorporating more real-world buggy DL samples, thereby improving the performance of the ML models used within *FL4Deep*. Additionally, we plan to enrich the fact extraction step by gathering more comprehensive information and developing additional rules to support the identification and localization of a broader range of DL faults.

CHAPTER 8 CONCLUSION

The primary objective of this thesis was to enhance reliability assessment methods for ML-based systems by examining four key aspects that play an important role in the reliability assessment of ML-based systems.

In Chapter 4, we conducted an in-depth investigation of bugs in ML-based systems to achieve a comprehensive understanding of the characteristics of both ML and non-ML bugs in these systems. We analyzed the top 300 ML-based system repositories on GitHub and manually reviewed 386 sampled issues raised within these repositories. Our findings revealed that ML components are more prone to errors compared to non-ML components, with nearly half of the identified bugs being related to ML components. Furthermore, we observed that existing classifications of ML bug types, symptoms, and root causes have become outdated, as ML components are now utilized across a broader range of areas and systems. While our results indicated that fixing ML bugs is generally more complex than non-ML bugs, as evidenced by metrics such as the number of commits and lines of code changed, the resources required (e.g., time-to-fix and developer expertise) to fix them are comparable for both ML and non-ML bugs. Additionally, we found that developers tend to approach ML bug fixes with greater caution, as evidenced by the lower number of bug-fix PRs merged without review for ML-related bugs.

Chapter 5 explored the common challenges that ML developers encounter during the development of ML-based systems, which have negative impacts on the reliability of the final ML applications. In this chapter, we focus specifically on DRL, a subdomain of ML. Despite the growing attention DRL has received in recent years, the challenges associated with developing DRL applications remain unclear. To address this gap, we analyzed 927 DRL-related SO posts, manually labeled them, and developed a taxonomy of common challenges in DRL application development. Our findings revealed that *Python* is the most frequently used programming language for DRL development, with *TensorFlow*, *Keras*, *PyTorch*, and *OpenAI Gym* being the most commonly used frameworks. Through our analysis, we identified five main categories of challenges in DRL development: *DRL issues*, *parallel processing & multi-threading*, *DRL libraries/frameworks*, *DL issues*, and *general programming*. While many challenges in DRL development overlap with those in ML applications, DRL also presents unique challenges (such as ones related to the environment, action, reward, etc.) which are captured in the *DRL issues* category. Our results further showed that *DRL comprehension*, *API usage in DRL frameworks*, and *problem design using DRL algorithms* are the most

significant challenges in DRL development. Additionally, we found that issues related to *parallel processing/multi-threading* and *DRL libraries/frameworks* tend to take longer to receive accepted answers.

In Chapter 6, we examined the importance of designing a standardized benchmark of bugs for the reliability assessment of ML-based systems, given that bugs and their life cycles are central to software reliability evaluation. Despite the development of various reliability assessment tools and techniques for ML-based systems (e.g., NeuraLint, DRLinter), there remains a significant gap, a lack of standard benchmarks for comparing different methodologies. To address this gap, we developed a benchmark of bugs specifically for ML-based systems, called *defect4ML*, which includes 100 bugs. Given that *TensorFlow* and *Keras* are the two most popular ML frameworks, we used ML scripts implemented using these frameworks to construct *defect4ML*. Our provided benchmark meets all the criteria of a standard benchmark by providing bugs from various ML frameworks and versions, detailed information on dependencies and required data, thorough explanations of bug types, and links to the original bug reports. To identify suitable bugs that align with benchmark standard criteria, we reviewed 1,777 bugs from GitHub and 1,296 from SO related to ML-based systems. Our analysis revealed that only 3.48% of GitHub bugs and 2.93% of SO bugs are reproducible. Furthermore, we found that 13.3% of the reproducible GitHub bugs could be verified using test cases provided by repository developers, whereas SO posts typically do not include test cases for verifying bug fixes.

Chapter 7 presented a system-level fault localization approach for DL-based systems. While existing fault localization approaches for DL-based systems primarily focus on the DL model and its training process, they often overlook other critical components of the DL development pipeline. To address this gap, we introduced *FL4Deep*, which covers the entire DL-based system development pipeline. *FL4Deep* leverages both static and dynamic information about DL-based systems and utilizes the KG to generate a ranked list of faults and their root causes that degrade system performance. To evaluate *FL4Deep*'s effectiveness, we tested it on 100 buggy samples (20 real-world bugs and 80 synthetic ones generated by DL mutation operators) and compared its performance with four state-of-the-art fault localization approaches for DL-based systems: *DeepLocalize*, *DeepFD*, *AutoTrainer*, and *UMLAUT*. The results demonstrated that *FL4Deep* outperformed these tools in four out of six DL bug categories in terms of accuracy. Moreover, in terms of precision and recall, *FL4Deep* achieved superior fault localization in five out of six bug categories compared to other approaches.

This thesis primarily aimed to improve reliability assessment methods for ML-based systems, from various perspectives deeply affecting the performance of reliability assessment methods.

8.1 Limitations

While this thesis aims to address key aspects of improving the reliability assessment of ML-based systems, it does not cover all dimensions of the field. For instance, as noted in Section 4.6, our study focuses on extracting buggy code from systems built with popular ML frameworks including *TensorFlow*, *Keras*, and *PyTorch*. However, other frameworks, such as JAX, which are gaining traction, may introduce novel fault types that were not previously observed. Additionally, although we sought to enhance testing methods for ML-based systems, our efforts did not encompass all categories of ML systems. For example, as explained in Section 6.6, the benchmark we developed concentrates on bugs in code developed for Supervised Learning. Beyond these broader limitations, several other factors could have influenced the results of our studies. These points warrant further exploration and could provide valuable insights for future research.

- *Programming language and ML frameworks:* In this thesis, we focused on software systems developed using *Python*, which is widely regarded as the most popular programming language for developing ML components and ML-based systems. However, it is important to consider that ML-based systems developed using other programming languages may introduce different types of faults, which might not be directly applicable to systems implemented in *Python*. Furthermore, we concentrated on the most commonly used ML frameworks—*TensorFlow*, *Keras*, and *PyTorch* for exploring bugs in ML-based systems. Moreover, while we relied on *Keras* and *TensorFlow* in the implementation of *FL4Deep*, it can be easily adapted to localize faults in ML systems developed with other frameworks like *PyTorch*.
- *Labeling and categorization process:* Although we employed manual labeling, which can introduce potential bias, we mitigated this by involving at least three expert raters with substantial experience in testing and developing ML-based systems. To further reduce bias and ensure consistency, we adhered to a systematic approach previously validated in related studies, resolving any disagreements among raters through a structured consensus-building process [4]. This helped to enhance the reliability and accuracy of the labeling and categorization results used in various analyses throughout this thesis.

- *Sampling of extracted data:* Since we applied sampling to the extracted data (repositories and buggy code), it is important to note that sampling is a common practice in the SE research, especially when handling large datasets. For our sampling process, we adopted a 95% confidence level and a 5% confidence interval, which are standard values frequently used in related studies [20, 140]. Also, in Subsection 4.3.2, instead of randomly selecting repositories, we prioritized the most popular ones, based on metrics such as the number of stars, forks, and other popularity indicators. This ensures that our sample represents widely-used projects. Additionally, in Subsection 4.3.3, we employed Stochastic Universal Sampling (SUS) to minimize bias when selecting buggy code samples, ensuring a more balanced and representative selection.
- *Limited training data:* Since we utilized ML models in various components of *FL4Deep*, sufficient training data is essential to achieve optimal model performance. However, our dataset was limited to real-world buggy samples, which restricted the total number of available examples. Furthermore, we exclusively used reproducible samples with confirmed solutions, verified by the users who reported the bugs, to ensure the accuracy and reliability of the data.

8.2 Future Research

Although several approaches have been proposed to enhance the reliability assessment of ML components and ML-based systems, much work remains to ensure these systems are tested thoroughly enough to guarantee their trustworthiness. The findings and approaches presented in this thesis represent just one step toward that goal, and we hope they inspire other researchers to build upon them. We conclude with the following points as possible directions for future research.

- *Expanding generalizability of the proposed approaches:* A key area for improvement is adapting the proposed approaches to other ML frameworks. Currently, *FL4Deep* is designed to work with *Keras* and *TensorFlow* (see Chapter 7). However, it could be extended to support other widely-used frameworks such as *PyTorch* and *JAX*. Since our approach relies on a Keras callback function to capture the necessary data for identifying and localizing faults in ML-based systems, extending this approach to other ML frameworks will require leveraging the capabilities of those frameworks to extract the required information and serving the extracted data as the input to the analysis part. Additionally, all buggy samples in *defect4ML* are currently implemented using *Keras* and *TensorFlow*, as they are among the most popular ML frameworks (see Chapter 6).

Expanding to other frameworks would increase the tool’s applicability as well.

- *Providing new taxonomy of bugs in ML programs:* As the use of ML and its subfields continues to expand rapidly into new areas, the current taxonomy of ML bugs has become outdated [28]. New types of faults have emerged that cannot be effectively classified using the existing taxonomy. Moreover, the rise of new ML frameworks, such as *JAX*, which has recently gained significant attention, may introduce bugs that are not included by the current taxonomy. For example, conducting a comprehensive investigation of the most popular ML frameworks, based on their usage in GitHub repositories and discussions on Q&A platforms (e.g., SO), and subsequently providing a revised taxonomy of bugs reported during the development of ML-based applications using these frameworks would be highly valuable. Therefore, developing an updated taxonomy of bugs in ML programs would enable researchers to identify and address these issues more effectively and accurately.
- *Leveraging capabilities of pretrained models:* Given that our approaches rely on ML models in several components and there is a lack of datasets containing reproducible buggy ML samples, leveraging pretrained models could enhance the performance of our reliability assessment methods. For instance, we trained models in *FL4Deep* (see Chapter 7) using only 75 buggy real-world samples. In contrast, some models have been trained on much larger, more general-purpose datasets, such as FB15K, which contains nearly 600,000 triples. As a result, fine-tuning a general-purpose trained model using the existing bug dataset could enhance the performance of the proposed fault localization approach. Additionally, the growing popularity of Large Language Models (LLMs) in software testing presents another promising opportunity to improve the performance of the ML models used in our approaches.

REFERENCES

- [1] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau *et al.*, “An introduction to deep reinforcement learning,” *Foundations and Trends® in Machine Learning*, vol. 11, pp. 219–354, 2018.
- [2] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, “Hidden technical debt in machine learning systems,” *Advances in neural information processing systems*, vol. 28, 2015.
- [3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, “Software engineering for machine learning: A case study,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.
- [4] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.
- [5] L. Traini, “Exploring performance assurance practices and challenges in agile software development: an ethnographic study,” *Empirical Software Engineering*, vol. 27, p. 74, 2022.
- [6] R. S. Pressman, *Software engineering: a practitioner’s approach*. Palgrave macmillan, 2005.
- [7] M. R. Lyu, “Software reliability engineering: A roadmap,” in *Future of Software Engineering (FOSE’07)*. Minneapolis: IEEE, 2007, pp. 153–170.
- [8] D. Galin, *Software quality assurance: from theory to implementation*. England: Pearson education, 2004.
- [9] *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*, IEEE Std., 2010.
- [10] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, “Testing machine learning based systems: a systematic mapping,” *Empirical Software Engineering*, vol. 25, pp. 5193–5254, 2020.

- [11] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
- [12] J. McCarthy, “What is artificial intelligence?” 2007.
- [13] J. Bell, *Machine learning: hands-on for developers and technical professionals*. John Wiley & Sons, 2020.
- [14] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine learning testing: Survey, landscapes and horizons,” *IEEE Transactions on Software Engineering*, 2020.
- [15] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2018.
- [16] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, “State-of-the-art in artificial neural network applications: A survey,” *Heliyon*, vol. 4, p. e00938, 2018.
- [17] K.-K. Lau and S. di Cola, *An Introduction to Component-Based Software Development*. World Scientific Publishing Co Pte Ltd: World Scientific, 2017. [Online]. Available: <https://www.worldscientific.com/doi/abs/10.1142/10486>
- [18] S. Carta, A. Corrigan, A. Ferreira, A. S. Podda, and D. R. Recupero, “A multi-layer and multi-ensemble stock trader using deep learning and deep reinforcement learning,” *Applied Intelligence*, vol. 51, pp. 889–905, 2021.
- [19] N. Humbatova, G. Jahangirova, and P. Tonella, “Deepcrime: Mutation testing of deep learning systems based on real faults,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 67–78. [Online]. Available: <https://doi.org/10.1145/3460319.3464825>
- [20] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, “A comprehensive study on challenges in deploying deep learning based software,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 750–762. [Online]. Available: <https://doi.org/10.1145/3368089.3409759>
- [21] B. Vlasic and N. E. Boudette. (2016) Self-driving tesla was involved in fatal crash, us says. The New York Times Company New York,

- NY, USA. [Online]. Available: <https://www.nytimes.com/2016/07/01/business/self-driving-tesla-fatal-crash-investigation.html>
- [22] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.
 - [23] M. Morales, *Grokking Deep Reinforcement Learning*. Manning Publications, 2019.
 - [24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
 - [25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, pp. 529–533, 2015.
 - [26] D. Gandhi, L. Pinto, and A. Gupta, “Learning to fly by crashing,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3948–3955.
 - [27] M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, “Deepstack: Expert-level artificial intelligence in heads-up no-limit poker,” *Science*, vol. 356, pp. 508–513, 2017.
 - [28] M. M. Morovati, A. Nikanjam, F. Khomh, and Z. M. Jiang, “Bugs in machine learning-based systems: a faultload benchmark,” *Empirical Software Engineering*, vol. 28, p. 62, 2023.
 - [29] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, “An empirical study on tensorflow program bugs,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2018, pp. 129–140.
 - [30] *Information technology, Systems and software engineering, FiSMA 1.1 functional size measurement method*, Std., 2010.
 - [31] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.
 - [32] M. Vieira, H. Madeira, K. Sachs, and S. Kounev, “Resilience benchmarking,” in *Resilience assessment and evaluation of computing systems*. Springer, 2012, pp. 283–301.

- [33] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, “How to build a benchmark,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 333–336.
- [34] K. Huppler, “The art of building a good benchmark,” in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2009, pp. 18–30.
- [35] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm,” *Information and Software Technology*, vol. 99, pp. 164–176, 2018.
- [36] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, “An empirical study of fault localization families and their combinations,” *IEEE Transactions on Software Engineering*, vol. 47, pp. 332–347, 2019.
- [37] W. E. Wong and T. Tse, *Handbook of software fault localization: foundations and advances*. John Wiley & Sons, 2023.
- [38] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.
- [39] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, “Spectrum-based fault localization in software product lines,” *Information and Software Technology*, vol. 100, pp. 18–31, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S095058491730188X>
- [40] L. Naish, H. J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, aug 2011. [Online]. Available: <https://doi.org/10.1145/2000791.2000795>
- [41] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 609–620.
- [42] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Does mutation testing improve testing practices?” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921.
- [43] M. Papadakis and Y. Le Traon, “Metallaxis-fl: Mutation-based fault localization,” *Softw. Test. Verif. Reliab.*, vol. 25, p. 605–628, aug 2015. [Online]. Available: <https://doi.org/10.1002/stvr.1509>

- [44] M. Harman and R. Hierons, “An overview of program slicing,” *software focus*, vol. 2, pp. 85–92, 2001.
- [45] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–36, 2005.
- [46] X. Zhang, N. Gupta, and R. Gupta, “A study of effectiveness of dynamic slicing in locating real faults,” *Empirical Software Engineering*, vol. 12, pp. 143–160, 2007.
- [47] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, “Locating faults with program slicing: an empirical analysis,” *Empirical Software Engineering*, vol. 26, pp. 1–45, 2021.
- [48] L. Gong, H. Zhang, H. Seo, and S. Kim, “Locating crashing faults based on crash stack traces,” *arXiv preprint arXiv:1404.4100*, 2014.
- [49] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 272–281.
- [50] C. Manning, P. Raghavan, and H. Schütze, “Introduction to information retrieval,” *Natural Language Engineering*, vol. 16, pp. 100–103, 2010.
- [51] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, pp. 707–740, 2016.
- [52] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, and K. Yin, “A review of process fault detection and diagnosis: Part iii: Process history based methods,” *Computers & Chemical Engineering*, vol. 27, pp. 327–346, 2003. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S009813540200162X>
- [53] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, “Bugcache for inspections: hit or miss?” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 322–331.
- [54] X. Yu, K. E. Bennin, J. Liu, J. W. Keung, X. Yin, and Z. Xu, “An empirical study of learning to rank techniques for effort-aware defect prediction,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 298–309.
- [55] X. Chen, S. Jia, and Y. Xiang, “A review: Knowledge reasoning over knowledge graph,” *Expert Systems with Applications*, vol. 141, p. 112948, 2020.

- [56] A. Singhal, “Introducing the knowledge graph: things, not strings,” May 2012. [Online]. Available: <https://blog.google/products/search/introducing-knowledge-graph-things-not/>
- [57] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, and A. Wahler, *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Springer International Publishing, 2020. [Online]. Available: <https://books.google.ca/books?id=1qnNDwAAQBAJ>
- [58] J. Barrasa, J. Webber, and J. Webber, *Building Knowledge Graphs: A Practitioner’s Guide*. O’Reilly, 2023. [Online]. Available: <https://books.google.ca/books?id=Ztb5zgEACAAJ>
- [59] “Rdf 1.1 primer,” June 2014. [Online]. Available: <https://www.w3.org/TR/rdf11-primer/>
- [60] J. Z. Pan, “Resource description framework,” in *Handbook on ontologies*. Springer, 2009, pp. 71–90. [Online]. Available: <https://jena.apache.org/>
- [61] “Notation3 language,” <https://w3c.github.io/N3/spec/>, accessed: 2024-08.
- [62] “Notation3 language,” May 2024. [Online]. Available: <https://w3c.github.io/N3/spec/>
- [63] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, “Freebase: a collaboratively created graph database for structuring human knowledge,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1247–1250.
- [64] N. Shirvani-Mahdavi, F. Akrami, M. S. Saeef, X. Shi, and C. Li, “Comprehensive analysis of freebase and dataset creation for robust evaluation of knowledge graph link prediction models,” in *International Semantic Web Conference*. Springer, 2023, pp. 113–133.
- [65] R. West, E. Gabrilovich, K. Murphy, S. Sun, R. Gupta, and D. Lin, “Knowledge base completion via search-based question answering,” in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 515–526.
- [66] A. Rossi, D. Barbosa, D. Firmani, A. Matinata, and P. Merialdo, “Knowledge graph embedding for link prediction: A comparative analysis,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, pp. 1–49, 2021.

- [67] L. Katz, “A new status index derived from sociometric analysis,” *Psychometrika*, vol. 18, pp. 39–43, 1953.
- [68] L. Page, S. Brin, R. Motwani, T. Winograd *et al.*, “The pagerank citation ranking: Bringing order to the web,” 1999.
- [69] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer, 2018, pp. 593–607.
- [70] S. Vashishth, S. Sanyal, V. Nitin, and P. Talukdar, “Composition-based multi-relational graph convolutional networks,” *arXiv preprint arXiv:1911.03082*, 2019.
- [71] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, “Translating embeddings for modeling multi-relational data,” *Advances in neural information processing systems*, vol. 26, 2013.
- [72] M. Galkin, M. Berrendorf, and C. T. Hoyt, “An open challenge for inductive link prediction on knowledge graphs,” *arXiv preprint arXiv:2203.01520*, 2022.
- [73] J. Baek, D. B. Lee, and S. J. Hwang, “Learning to extrapolate knowledge: Transductive few-shot out-of-graph link prediction,” *Advances in neural information processing systems*, vol. 33, pp. 546–560, 2020.
- [74] M. Galkin, E. Denis, J. Wu, and W. L. Hamilton, “Nodepiece: Compositional and parameter-efficient representations of large knowledge graphs,” *arXiv preprint arXiv:2106.12144*, 2021.
- [75] R. S. S. Kumar, D. O. Brien, K. Albert, S. Viljöen, and J. Snover, “Failure modes in machine learning systems,” *arXiv preprint arXiv:1911.11034*, 2019.
- [76] S. Beyer, C. Macho, M. Di Penta, and M. Pinzger, “What kind of questions do developers ask on stack overflow? a comparison of automated approaches to classify posts into question categories,” *Empirical Software Engineering*, vol. 25, pp. 2258–2301, 2020.
- [77] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, “Why is developing machine learning applications challenging? a study on stack overflow posts,” in *2019 acm/ieee international symposium on empirical software engineering and measurement (esem)*. IEEE, 2019, pp. 1–11.

- [78] A. A. Bangash, H. Sahar, S. Chowdhury, A. W. Wong, A. Hindle, and K. Ali, “What do developers know about machine learning: a study of ml discussions on stackoverflow,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 260–264.
- [79] H. Jelodar, Y. Wang, C. Yuan, X. Feng, X. Jiang, Y. Li, and L. Zhao, “Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey,” *Multimedia Tools and Applications*, vol. 78, pp. 15 169–15 211, 2019.
- [80] A. Hamidi, G. Antoniol, F. Khomh, M. Di Penta, and M. Hamidi, “Towards understanding developers’ machine-learning challenges: A multi-language study on stack overflow,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 58–69.
- [81] X. Zhang, J. Zhai, S. Ma, and C. Shen, “Autotrainer: An automatic dnn training problem detection and repair system,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.
- [82] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, “Automatic fault detection for deep learning programs using graph transformations,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, sep 2021.
- [83] E. Schoop, F. Huang, and B. Hartmann, “Umlaut: Debugging deep learning programs using program structure and model behavior,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–16.
- [84] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, “Faults in deep reinforcement learning programs: a taxonomy and a detection approach,” *Automated Software Engineering*, vol. 29, pp. 1–32, 2022.
- [85] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.
- [86] A. Kuhnle, M. Schaarschmidt, and K. Fricke, “Tensorforce: a tensorflow library for applied reinforcement learning,” Web page, 2017. [Online]. Available: <https://github.com/tensorforce/tensorforce>
- [87] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, “Dopamine: A research framework for deep reinforcement learning,” *arXiv preprint arXiv:1812.06110*, 2018.

- [88] M. Plappert, “keras-rl,” <https://github.com/keras-rl/keras-rl>, 2016.
- [89] A. H. Yahmed, A. A. Abbassi, A. Nikanjam, H. Li, and F. Khomh, “Deploying deep reinforcement learning systems: A taxonomy of challenges,” *arXiv preprint arXiv:2308.12438*, 2023.
- [90] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.
- [91] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, “Bugbench: Benchmarks for evaluating bug detection tools,” in *Workshop on the evaluation of software defect detection tools*, vol. 5. Chicago, Illinois, 2005.
- [92] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, pp. 1236–1256, 2015.
- [93] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, “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, 2019, pp. 468–478.
- [94] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, “Jacontebe: A benchmark suite of real-world java concurrency bugs (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 178–189.
- [95] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, “Repairing deep neural networks: Fix patterns and challenges,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1135–1146.
- [96] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, “Theano: A python framework for fast computation of mathematical expressions,” *arXiv e-prints*, pp. arXiv–1605, 2016.
- [97] R. Collobert, S. Bengio, and J. Mariéthoz, “Torch: a modular machine learning software library,” Idiap, Tech. Rep., 2002.
- [98] Torch, “Torch official github repository,” <https://github.com/torch/torch7>, accessed: 2021-9-1.

- [99] <https://github.com/dpressel/baseline/commit/4dad463>, 2016, accessed: 2021-11-01.
- [100] <https://github.com/albu/albumentations/commit/fec1f3b>, 2018, accessed: 2021-11-01.
- [101] M. Wardat, W. Le, and H. Rajan, “Deeplocalize: Fault localization for deep neural networks,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.
- [102] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, “Detecting numerical bugs in neural network architectures,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 826–837.
- [103] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.
- [104] J. R. Shewchuk, “Concise machine learning,” May 2022.
- [105] A. Karpathy. (2019) A recipe for training neural networks. [Online]. Available: <https://karpathy.github.io/2019/04/25/recipe/>
- [106] H. Ben Braiek and F. Khomh, “Testing feedforward neural networks training programs,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, pp. 1–61, 2023.
- [107] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.
- [108] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, “Deepfd: Automated fault diagnosis and localization for deep learning programs,” *arXiv preprint arXiv:2205.01938*, 2022.
- [109] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, “Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 561–572.
- [110] —, “An effective data-driven approach for localizing deep learning faults,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.08947>

- [111] “Write complete json log after training,” <https://github.com/snorkel-team/snorkel/pull/1445>, 2019.
- [112] “Interactive learning has server error hosting on docker,” <https://github.com/RasaHQ/rasa/issues/4142>, 2019.
- [113] “Pps: fix of ispixelhit,” <https://github.com/cms-sw/cmssw/pull/35089>, 2021.
- [114] P. Grubb and A. A. Takang, *Software maintenance: concepts and practice*. World Scientific, 2003.
- [115] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, “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*, 2021, pp. 968–980.
- [116] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, “Exposing numerical bugs in deep learning via gradient back-propagation,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 627–638. [Online]. Available: <https://doi.org/10.1145/3468264.3468612>
- [117] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. Savannah, GA, USA: USENIX, 2016, pp. 265–283.
- [118] F. Chollet *et al.*, “Keras: The python deep learning library,” *Astrophysics Source Code Library*, pp. ascl–1806, 2018.
- [119] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” Ithaca, NY, United States, 2019.
- [120] S. Li, Y. Wu, Y. Liu, D. Wang, M. Wen, Y. Tao, Y. Sui, and Y. Liu, “An exploratory study of bugs in extended reality applications on the web,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2020, pp. 172–183.

- [121] C. Voskoglou, “What is the best programming language for machine learning,” 2017, accessed: 2021-10-06. [Online]. Available: <https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7>
- [122] S. Gupta, “What is the best language for machine learning?” <https://www.springboard.com/blog/data-science/best-language-for-machine-learning>, 2021, accessed: 2021-10-06.
- [123] GitHub, “Github graphql api documentation,” <https://docs.github.com/en/graphql>, 2021, accessed: 2021-7-27.
- [124] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, “What is the connection between issues, bugs, and enhancements?” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2018, pp. 306–315.
- [125] H. Hata, R. G. Kula, T. Ishio, and C. Treude, “Same file, different changes: The potential of meta-maintenance on github,” in *Proceedings of the 43rd International Conference on Software Engineering*, IEEE. 3 Park Avenue, New York NY 10016-5997, USA: IEEE Press, 2021, p. 773–784. [Online]. Available: <https://doi.org/10.1109/ICSE43902.2021.00076>
- [126] HanLP, “Hanlp: Han language processing,” <https://github.com/hankcs/HanLP>, 2021, accessed: 2021-11-01.
- [127] Tensorflow, <https://github.com/tensorflow/models>, 2021, accessed: 2021-11-01.
- [128] NVIDIA, “Nvtabular,” <https://github.com/NVIDIA-Merlin/NVTabular>, 2021, accessed: 2021-11-01.
- [129] ultralytics, “Yolov3,” <https://github.com/ultralytics/yolov3>, 2021, accessed: 2021-11-01.
- [130] Tensorpack, <https://github.com/tensorpack/tensorpack>, 2021, accessed: 2021-11-01.
- [131] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Transactions on software engineering*, vol. 25, pp. 557–572, 1999.
- [132] Keras, “Formal documentation of keras apis,” <https://keras.io/api/models/>, 2022.
- [133] “Replication package,” <https://github.com/ML-Bugs-2022/Replication-Package>, 2023.

- [134] R. Falotico and P. Quatto, “Fleiss’ kappa statistic without paradoxes,” *Quality & Quantity*, vol. 49, pp. 463–470, 2015.
- [135] Y. Yang, T. He, Y. Feng, S. Liu, and B. Xu, “Mining python fix patterns via analyzing fine-grained source code changes,” *Empirical Software Engineering*, vol. 27, pp. 1–37, 2022.
- [136] S. Quach, M. Lamothe, Y. Kamei, and W. Shang, “An empirical study on the use of szz for identifying inducing changes of non-functional bugs,” *Empirical Software Engineering*, vol. 26, pp. 1–25, 2021.
- [137] L. Hartling, M. Hamm, A. Milne, B. Vandermeer, P. L. Santaguida, M. Ansari, A. Tsertsvadze, S. Hempel, P. Shekelle, and D. M. Dryden, “Validity and inter-rater reliability testing of quality assessment instruments,” 2012.
- [138] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, “Faults in deep reinforcement learning programs: a taxonomy and a detection approach,” *Automated Software Engineering*, vol. 29, pp. 1–32, 2022.
- [139] G. Joshua, F. Yang, S. Junjie, A. Sumaya, X. Yuan, Chen, and Q. Alfred, “A comprehensive study of autonomous vehicle bugs,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 385–396. [Online]. Available: <https://doi.org/10.1145/3377811.3380397>
- [140] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, “An empirical study of common challenges in developing deep learning applications,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2019, pp. 104–115.
- [141] E. Wirsansky, *Hands-on genetic algorithms with Python: applying genetic algorithms to solve real-world deep learning and artificial intelligence problems*. Packt Publishing Ltd: Packt Publishing Ltd, 2020.
- [142] RasaHQ, <https://github.com/RasaHQ/rasa/issues/8541>, 2021, accessed: 2021-11-01.
- [143] —, <https://github.com/RasaHQ/rasa/issues/4730>, 2021, accessed: 2021-11-01.
- [144] “some tests fails with pytorch 1.6,” <https://github.com/speechbrain/speechbrain/issues/248>, 2020.

- [145] “Add typing for trainer.logger,” <https://github.com/Lightning-AI/lightning/pull/11114>, 2021.
- [146] “Loading a checkpoint that was saved in pl < 1.2 still breaks,” <https://github.com/Lightning-AI/lightning/issues/7400>, 2021.
- [147] “reduce_lr_on_plateau can’t find validation metrics in most recent release.” <https://github.com/scverse/scvi-tools/issues/1112>, 2021.
- [148] “Bug: Mismatch value with speechbrain.nnet.pooling.statisticalpooling,” <https://github.com/speechbrain/speechbrain/issues/1048>, 2021.
- [149] “Bug fix for static pulse shapes,” <https://github.com/cms-sw/cms-sw/pull/23001>, 2018.
- [150] “Fix docs for early stopping,” <https://github.com/Lightning-AI/lightning/pull/865>, 2020.
- [151] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, ser. eclipse ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 35–39. [Online]. Available: <https://doi.org/10.1145/1117696.1117704>
- [152] G. Long and T. Chen, “On reporting performance and accuracy bugs for deep learning frameworks: An exploratory study from github,” 2022. [Online]. Available: <https://arxiv.org/abs/2204.07893>
- [153] T. Menzies, “The five laws of se for ai,” *IEEE Software*, vol. 37, pp. 81–85, 2019.
- [154] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, “The symptoms, causes, and repairs of bugs inside a deep learning library,” *Journal of Systems and Software*, vol. 177, p. 110935, 2021.
- [155] “Loading saved destvi.from_rna_model doesn’t consider two anndatas used to create model,” <https://github.com/scverse/scvi\protect\discretionary{\char\hyphenchar\font}{\font}{\font}{\font}tools/issues/1087>, 2021.
- [156] “Fix model architecture for deployment to onnx,” <https://github.com/mehta-lab/microDL/pull/234>, 2023.
- [157] “tf.function not used for model inference,” <https://github.com/RasaHQ/rasa/issues/10728>, 2022.

- [158] “Fix default ckpt path when logger exists,” <https://github.com/Lightning\protect\discretionary{\char\hyphenchar\font}{\font}{AI/lightning/pull/771}>, 2020.
- [159] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. De Water, “Studying pull request merges: a case study of shopify’s active merchant,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2018, pp. 124–133.
- [160] K. Chaturvedi, P. Kapur, S. Anand, and V. Singh, “Predicting the complexity of code changes using entropy based measures,” *International Journal of System Assurance Engineering and Management*, vol. 5, pp. 155–164, 2014.
- [161] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability*, vol. 24, pp. 219–250, 2014.
- [162] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, “A systematic review of effect size in software engineering experiments,” *Information and Software Technology*, vol. 49, pp. 1073–1086, 2007. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584907000195>
- [163] N. Cliff, “Dominance statistics: Ordinal analyses to answer ordinal questions.” *Psychological bulletin*, vol. 114, p. 494, 1993.
- [164] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, “Cliff’s delta calculator: A non-parametric effect size program for two groups of observations,” *Universitas Psychologica*, vol. 10, pp. 545–555, 2011.
- [165] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, “Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen’sd indices the most appropriate choices,” in *annual meeting of the Southern Association for Institutional Research*. The Pennsylvania State University: Citeseer, 2006, pp. 1–51.
- [166] “Gensim library,” https://radimrehurek.com/gensim_3.8.3/index.html, 2022.
- [167] “Mallet library,” <https://mimno.github.io/Mallet/>, 2022.
- [168] A. Romano, X. Liu, Y. Kwon, and W. Wang, “An empirical study of bugs in webassembly compilers,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3 Park Avenue, New York NY 10016-5997, USA: IEEE, 2021, pp. 42–54.

- [169] A. Bosu and J. C. Carver, “Impact of developer reputation on code review outcomes in oss projects: An empirical investigation,” in *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, 2014, pp. 1–10.
- [170] P. Schober, C. Boer, and L. A. Schwarte, “Correlation coefficients: appropriate use and interpretation,” *Anesthesia & Analgesia*, vol. 126, pp. 1763–1768, 2018.
- [171] A. Tagra, H. Zhang, G. K. Rajbahadur, and A. E. Hassan, “Revisiting reopened bugs in open source software systems,” *Empirical Software Engineering*, vol. 27, pp. 1–34, 2022.
- [172] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, “Characterizing and predicting which bugs get reopened,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1074–1083.
- [173] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, “Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft,” *IEEE Transactions on Software Engineering*, vol. 43, pp. 56–75, 2016.
- [174] C. Maddila, C. Bansal, and N. Nagappan, “Predicting pull request completion time: a case study on large scale cloud services,” in *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 874–882.
- [175] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.
- [176] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, “Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow,” 2021.
- [177] X. Hu, L. Chu, J. Pei, W. Liu, and J. Bian, “Model complexity of deep learning: A survey,” *Knowledge and Information Systems*, vol. 63, pp. 2585–2619, 2021.
- [178] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, “Complexity vs. performance: empirical analysis of machine learning as a service,” in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 384–397.
- [179] K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution: a roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 73–87.

- [180] S. Aradi, “Survey of deep reinforcement learning for motion planning of autonomous vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, pp. 740–759, 2020.
- [181] Y. Xiao, J. Liu, J. Wu, and N. Ansari, “Leveraging deep reinforcement learning for traffic engineering: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 23, pp. 2064–2097, 2021.
- [182] C. Yu, J. Liu, S. Nemati, and G. Yin, “Reinforcement learning in healthcare: A survey,” *ACM Computing Surveys (CSUR)*, vol. 55, pp. 1–36, 2021.
- [183] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.
- [184] M. Bagherzadeh, N. Kahani, and L. Briand, “Reinforcement learning for test case prioritization,” *IEEE Transactions on Software Engineering*, vol. 48, pp. 2836–2856, 2021.
- [185] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.
- [186] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, pp. 26–38, 2017.
- [187] A. Zolfagharian, M. Abdellatif, L. Briand, M. Bagherzadeh *et al.*, “Search-based testing approach for deep reinforcement learning agents,” *arXiv preprint arXiv:2206.07813*, 2022.
- [188] F. Tambon, V. Majdinasab, A. Nikanjam, F. Khomh, and G. Antoniol, “Mutation testing of deep reinforcement learning based on real faults,” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 188–198.
- [189] S. Zhang, Z. Ding, and H. Dong, *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Springer, 2020.
- [190] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, “Deep reinforcement learning for multi-agent systems: A review of challenges, solutions, and applications,” *IEEE transactions on cybernetics*, vol. 50, pp. 3826–3839, 2020.

- [191] W. Du and S. Ding, “A survey on multi-agent deep reinforcement learning: from the perspective of challenges and applications,” *Artificial Intelligence Review*, vol. 54, pp. 3215–3238, 2021.
- [192] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester, “Challenges of real-world reinforcement learning: definitions, benchmarks and analysis,” *Machine Learning*, vol. 110, pp. 2419–2468, 2021.
- [193] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, “An empirical study of common challenges in developing deep learning applications,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.
- [194] Q. Rao and J. Frtunikj, “Deep learning for self-driving cars: Chances and challenges,” in *Proceedings of the 1st international workshop on software engineering for AI in autonomous systems*, 2018, pp. 35–38.
- [195] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic, “A taxonomy of software engineering challenges for machine learning systems: An empirical investigation,” in *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings 20*. Springer International Publishing, 2019, pp. 227–243.
- [196] E. de Souza Nascimento, I. Ahmed, E. Oliveira, M. P. Palheta, I. Steinmacher, and T. Conte, “Understanding development process of machine learning systems: Challenges and solutions,” in *2019 acm/ieee international symposium on empirical software engineering and measurement (esem)*. IEEE, 2019, pp. 1–6.
- [197] W. Zhu, H. Zhang, A. E. Hassan, and M. W. Godfrey, “An empirical study of question discussions on stack overflow,” *Empirical Software Engineering*, vol. 27, pp. 1–25, 2022.
- [198] StackExchange, “Stack exchange data dump,” <https://archive.org/details/stackexchange>, 2022, accessed: 2023-02-01.
- [199] M. M. Morovati, F. Tamboon, M. Taraghi, A. Nikanjam, and F. Khomh, “Paper replication package,” https://github.com/drlchallenges/drl_challenges, 2023, accessed: 2023-02-01.
- [200] J. He, Z. Xin, B. Xu, T. Zhang, K. Kim, Z. Yang, F. Thung, I. Irsan, and D. Lo, “Representation learning for stack overflow posts: How far are we?” *arXiv preprint arXiv:2303.06853*, 2023.

- [201] M. U. Haque, L. H. Iwaya, and M. A. Babar, “Challenges in docker development: A large-scale study using stack overflow,” in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.
- [202] M. Zahedi, R. N. Rajapakse, and M. A. Babar, “Mining questions asked about continuous software engineering: A case study of stack overflow,” in *Proceedings of the evaluation and assessment in software engineering*, 2020, pp. 41–50.
- [203] H. Lune and B. L. Berg, *Qualitative research methods for the social sciences*. Pearson, 2017.
- [204] Google, “How to use google sheets,” <https://support.google.com/docs/answer/6000292?hl=en&co=GENIE.Platform%3DDesktop>, 2020, accessed: 2023-02-01.
- [205] N. McDonald, S. Schoenebeck, and A. Forte, “Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, pp. 1–23, 2019.
- [206] G. Vijayaraghavan and C. Kaner, “Bug taxonomies: Use them to generate better tests,” *Star East*, vol. 2003, pp. 1–40, 2003.
- [207] S. E. Hove and B. Anda, “Experiences from conducting semi-structured interviews in empirical software engineering research,” in *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE, 2005, pp. 10–pp.
- [208] M. Aldhaen, “Interview versus questionnaire from the perspective of cbe members,” *International Journal of Education, Learning and Development*, vol. 8, pp. 21–41, 2020.
- [209] M. DeCarlo, *Scientific inquiry in social work*. Open Social Work Education, 2018.
- [210] H. Nekkanti and S. Reddy, “Surveys in software engineering: A systematic literature review and interview study,” Ph.D. dissertation, MSc thesis, Blekinge Institute of Technology, Sweden, 2016.
- [211] Qualtrics, “Qualtrics official website,” <https://www.qualtrics.com/>, 2023, accessed: 2023-04-01.
- [212] M. M. Morovati, A. Nikanjam, F. Tambon, F. Khomh, and Z. M. Jiang, “Bug characterization in machine learning-based systems,” *Empirical Software Engineering*, vol. 29, p. 14, 2024.

- [213] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, “Rllib: Abstractions for distributed reinforcement learning,” in *International Conference on Machine Learning*. PMLR, 2018, pp. 3053–3062.
- [214] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki, “LIFT: Reinforcement learning in computer systems by learning from demonstrations,” *CoRR*, vol. abs/1808.07903, 2018. [Online]. Available: <http://arxiv.org/abs/1808.07903>
- [215] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” *Empirical Software Engineering*, vol. 24, pp. 381–416, 2019.
- [216] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, “A graph-based approach to api usage adaptation,” *ACM Sigplan Notices*, vol. 45, pp. 302–321, 2010.
- [217] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, “Software engineering challenges of deep learning,” in *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2018, pp. 50–59.
- [218] J. J. Bruer, J. A. Tropp, V. Cevher, and S. R. Becker, “Designing statistical estimators that balance sample size, risk, and computational cost,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 9, pp. 612–624, 2015.
- [219] M. Panzer and B. Bender, “Deep reinforcement learning in production systems: a systematic literature review,” *International Journal of Production Research*, vol. 60, pp. 4316–4341, 2022.
- [220] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, “Deep reinforcement learning for autonomous driving: A survey,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, pp. 4909–4926, 2021.
- [221] U. Lorenz, *Reinforcement Learning From Scratch: Understanding Current Approaches - with Examples in Java and Greenfoot*. Springer International Publishing, 2022. [Online]. Available: <https://books.google.ca/books?id=bV2YEAAAQBAJ>
- [222] D. Reda, T. Tao, and M. van de Panne, “Learning to locomote: Understanding how environment design matters for deep reinforcement learning,” in *Proceedings of the 13th ACM SIGGRAPH Conference on Motion, Interaction and Games*, 2020, pp. 1–10.

- [223] Y. Li, “Deep reinforcement learning: An overview,” *arXiv preprint arXiv:1701.07274*, 2017.
- [224] A. Mojica-Hanke, A. Bayona, M. Linares-Vásquez, S. Herbold, and F. A. González, “What are the machine learning best practices reported by practitioners on stock exchange?” *arXiv preprint arXiv:2301.10516*, 2023.
- [225] J. D. Gibbons, *Nonparametric measures of association*. Sage, 1993, no. 91.
- [226] J. Frost, *Introduction to Statistics: An Intuitive Guide for Analyzing Data and Unlocking Discoveries*. Statistics By Jim, 2019.
- [227] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, “Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control,” in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7512–7519.
- [228] M. Pumperla, E. Oakes, and R. Liaw, *Learning Ray: Flexible Distributed Python for Machine Learning*. O’Reilly Media, 2023. [Online]. Available: <https://books.google.ca/books?id=vKj0zgEACAAJ>
- [229] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.
- [230] M. Openja, F. Majidi, F. Khomh, B. Chembakottu, and H. Li, “Studying the practices of deploying machine learning projects on docker,” in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, 2022, pp. 190–200.
- [231] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, “Ray: A distributed framework for emerging {AI} applications,” in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.
- [232] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, and X. Peng, “Demystifying dependency bugs in deep learning stack,” *arXiv preprint arXiv:2207.10347*, 2022.
- [233] D. Garg, P. Goel, G. Kandaswamy, A. Ganatra, and K. Kotecha, “A roadmap to deep learning: a state-of-the-art step towards machine learning,” in *Advanced Informatics for Computing Research: Second International Conference, ICAICR 2018, Shimla, India, July 14–15, 2018, Revised Selected Papers, Part I 2*. Springer, 2019, pp. 160–170.

- [234] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “Docter: documentation-guided fuzzing for testing deep learning api functions,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 176–188.
- [235] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, “Towards better dependency management: A first look at dependency smells in python projects,” *IEEE Transactions on Software Engineering*, 2022.
- [236] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, “An empirical study of the dependency networks of deep learning libraries,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 868–878.
- [237] A. Ayman, A. Aziz, A. Alipour, and A. Laszka, “Smart contract development in practice: trends, issues, and discussions on stack overflow,” *arXiv preprint arXiv:1905.08833*, 2019.
- [238] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, “How do i refactor this? an empirical study on refactoring trends and topics in stack overflow,” *Empirical Software Engineering*, vol. 27, p. 11, 2022.
- [239] M. B. Shah, M. M. Rahman, and F. Khomh, “Towards enhancing the reproducibility of deep learning bugs: An empirical study,” *arXiv preprint arXiv:2401.03069*, 2024.
- [240] M. Kim, Y. Kim, and E. Lee, “Denchmark: A bug benchmark of deep learning-related software,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 540–544.
- [241] O. G. Yalçın, “Top 5 deep learning frameworks to watch in 2021 and why tensorflow,” 2021, accessed: 2022-12-29. [Online]. Available: <https://towardsdatascience.com/top-5-deep-learning-frameworks-to-watch-in-2021-and-why-tensorflow-98d8d6667351>
- [242] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “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, 2019, pp. 589–593.
- [243] <https://github.com/vmelan/cifar-experiment/commit/561c82e>, 2018, accessed: 2022-06-01.
- [244] <https://stackoverflow.com/questions/53119432>, 2018, accessed: 2021-11-01.

- [245] <https://github.com/acflorea/keras-playground/commit/d44c90c>, 2018, accessed: 2022-06-01.
- [246] <https://github.com/keras-team/keras-tuner/commit/3758611>, 2018, accessed: 2022-06-01.
- [247] <https://github.com/hunkim/DeepLearningZeroToAll/commit/9f8fb94>, 2018, accessed: 2022-06-01.
- [248] <https://stackoverflow.com/questions/34311586>, 2016, accessed: 2021-11-01.
- [249] GitHub, “Github official website,” <https://github.com/about>, 2021, accessed: 2021-7-27.
- [250] G. developer guideline documentation, “Github rest api,” <https://developer.github.com/v3/>, 2021, accessed: 2021-7-27.
- [251] M. Abidi, F. Khomh, and Y.-G. Guéhéneuc, “Anti-patterns for multi-language systems,” in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, 2019, pp. 1–14.
- [252] M. Abidi, M. Grichi, F. Khomh, and Y.-G. Guéhéneuc, “Code smells for multi-language systems,” in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, 2019, pp. 1–13.
- [253] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, “Are multi-language design smells fault-prone? an empirical study,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, pp. 1–56, 2021.
- [254] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- [255] M. L. McHugh, “Interrater reliability: the kappa statistic,” *Biochemia medica*, vol. 22, pp. 276–282, 2012.
- [256] Keras, “Keras 2.1.5,” <https://github.com/keras-team/keras/releases/tag/2.1.5>, 2016, accessed: 2021-11-01.

- [257] StackOverflow, “Stack overflow annual developer survey,” <https://insights.stackoverflow.com/survey/2021>, 2021, accessed: 2022-04-01.
- [258] <https://stackoverflow.com/questions/44924690>, 2018, accessed: 2021-11-01.
- [259] <https://stackoverflow.com/questions/38080035>, 2017, accessed: 2021-11-01.
- [260] <https://stackoverflow.com/questions/58636087>, 2018, accessed: 2021-11-01.
- [261] <https://stackoverflow.com/questions/42264649>, 2017, accessed: 2021-11-01.
- [262] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.
- [263] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” 2009.
- [264] <https://stackoverflow.com/questions/56103207>, 2019, accessed: 2021-11-01.
- [265] <https://stackoverflow.com/questions/50079585>, 2018, accessed: 2021-11-01.
- [266] https://github.com/suchaoxiao/keras-frcnn_modify/commit/2f51f68, 2017, accessed: 2021-11-01.
- [267] <https://github.com/PhilippeNguyen/kinopt/commit/fdee16f>, 2018, accessed: 2021-11-01.
- [268] *IEEE Standard for System, Software, and Hardware Verification and Validation*, IEEE Std., 2017.
- [269] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, “Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1556–1560.
- [270] <https://github.com/vaclavcadek/keras2pmml/commit/4795ec6>, 2019, accessed: 2021-11-01.
- [271] M. Felderer and R. Ramler, “Quality assurance for ai-based systems: Overview and challenges (introduction to interactive session),” in *International Conference on Software Quality*. Springer, 2021, pp. 33–42.

- [272] L. Jia, H. Zhong, and L. Huang, "The unit test quality of deep learning libraries: A mutation analysis," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 47–57.
- [273] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 785–796.
- [274] L. Jia, H. Zhong, X. Wang, L. Huang, and Z. Li, "How do injected bugs affect deep learning?" in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 793–804.
- [275] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan, "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*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 771–783. [Online]. Available: <https://doi.org/10.1145/3324884.3416545>
- [276] M. Kirk, *Thoughtful machine learning: A test-driven approach*. " O'Reilly Media, Inc.", 2014.
- [277] J. Zhang, E. T. Barr, B. Guedj, M. Harman, and J. Shawe-Taylor, "Perturbed model validation: A new framework to validate model relevance," 2019.
- [278] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, pp. 1–12, 2004.
- [279] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std., 1990.
- [280] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.
- [281] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 132–142.
- [282] M. Xue, C. Yuan, H. Wu, Y. Zhang, and W. Liu, "Machine learning security: Threats, countermeasures, and evaluations," *IEEE Access*, vol. 8, pp. 74 720–74 742, 2020.

- [283] X. Liu, L. Xie, Y. Wang, J. Zou, J. Xiong, Z. Ying, and A. V. Vasilakos, "Privacy and security issues in deep learning: A survey," *IEEE Access*, vol. 9, pp. 4566–4593, 2020.
- [284] C. Zhu, W. R. Huang, H. Li, G. Taylor, C. Studer, and T. Goldstein, "Transferable clean-label poisoning attacks on deep neural nets," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7614–7623.
- [285] J. Brownlee, "Use early stopping to halt the training of neural networks at the right time," 2020, accessed: 2022-12-29. [Online]. Available: <https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/>
- [286] L. Rice, E. Wong, and Z. Kolter, "Overfitting in adversarially robust deep learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 8093–8104.
- [287] A. Chouldechova and A. Roth, "The frontiers of fairness in machine learning," *arXiv preprint arXiv:1810.08810*, 2018.
- [288] S. Barocas and A. D. Selbst, "Big data's disparate impact," *California Law Review*, vol. 104, pp. 671–732, 2016. [Online]. Available: <http://www.jstor.org/stable/24758720>
- [289] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery." *Queue*, vol. 16, pp. 31–57, 2018.
- [290] C. Dwork, "Differential privacy: A survey of results," in *International conference on theory and applications of models of computation*. Springer, 2008, pp. 1–19.
- [291] A. Nikanjam and F. Khomh, "Design smells in deep learning programs: An empirical study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 332–342.
- [292] *ISO/IEC/IEEE International Standard - Systems and software engineering-Vocabulary*, Std., 2017.
- [293] M. A. Alipour, "Automated fault localization techniques: a survey," *Oregon State University*, vol. 54, 2012.
- [294] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.
- [295] M. M. Morovati, A. Nikanjam, and F. Khomh, "Paper replication package," <https://github.com/mohmehmo/fl4deep>, 2024, accessed: 2024-07.

- [296] J. Kumar Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of computer science and technology*, vol. 25, pp. 1016–1029, 2010.
- [297] D. K. Lee, J. In, and S. Lee, "Standard deviation and standard error of the mean," *Korean journal of anesthesiology*, vol. 68, pp. 220–223, 2015.
- [298] S. J. Rigatti, "Random forest," *Journal of Insurance Medicine*, vol. 47, pp. 31–39, 2017.
- [299] B. Charbuty and A. Abdulazeez, "Classification based on decision tree algorithm for machine learning," *Journal of Applied Science and Technology Trends*, vol. 2, pp. 20–28, 2021.
- [300] S. Zhang, "Challenges in knn classification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, pp. 4663–4675, 2021.
- [301] C. Vens, J. Struyf, L. Schietgat, S. Džeroski, and H. Blockeel, "Decision trees for hierarchical multi-label classification," *Machine learning*, vol. 73, pp. 185–214, 2008.
- [302] Q. Wu, M. Tan, H. Song, J. Chen, and M. K. Ng, "Ml-forest: A multi-label tree ensemble method for multi-label classification," *IEEE transactions on knowledge and data engineering*, vol. 28, pp. 2665–2680, 2016.
- [303] M.-L. Zhang and Z.-H. Zhou, "A k-nearest neighbor based algorithm for multi-label classification," in *2005 IEEE international conference on granular computing*, vol. 2. IEEE, 2005, pp. 718–721.
- [304] V. R. Joseph, "Optimal ratio for data splitting," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 15, pp. 531–538, 2022.
- [305] J. Tan, J. Yang, S. Wu, G. Chen, and J. Zhao, "A critical look at the current train/test split in machine learning," *arXiv preprint arXiv:2106.04525*, 2021.
- [306] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.
- [307] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

- [308] L. Bottou, “From machine learning to machine reasoning: An essay,” *Machine learning*, vol. 94, pp. 133–149, 2014.
- [309] N. Duan, D. Tang, and M. Zhou, “Machine reasoning: Technology, dilemma and future,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, 2020, pp. 1–6.
- [310] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, “A review of relational machine learning for knowledge graphs,” *Proceedings of the IEEE*, vol. 104, pp. 11–33, 2015.
- [311] “RdfLib: a python library for working with rdf,” <https://github.com/RDFLib/rdfLib>, accessed: 2024-08.
- [312] R. Verborgh and J. De Roo, “Drawing conclusions from linked data on the web: The eye reasoner,” *IEEE Software*, vol. 32, pp. 23–27, 2015.
- [313] B. De Meester, D. Arndt, P. Bonte, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, E. Mannens, and R. Van de Walle, “Event-driven rule-based reasoning using eye,” in *ISWC2015*. CEUR, 2015.
- [314] “Cwm: a general-purpose data processor for the semantic web,” 2002. [Online]. Available: <http://www.w3.org/2000/10/swap/doc/cwm.html>
- [315] “Apache jena: a java framework for writing semantic web applications.” 2009. [Online]. Available: <https://jena.apache.org/>
- [316] M. Ali, M. Berrendorf, C. T. Hoyt, L. Vermue, S. Sharifzadeh, V. Tresp, and J. Lehmann, “PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings,” *Journal of Machine Learning Research*, vol. 22, pp. 1–6, 2021. [Online]. Available: <http://jmlr.org/papers/v22/20-825.html>
- [317] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, “Ablation studies in artificial neural networks,” *arXiv preprint arXiv:1901.08644*, 2019.
- [318] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, “Revisiting unreasonable effectiveness of data in deep learning era,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 843–852.
- [319] J. P. Klein and M. L. Moeschberger, *Survival analysis: techniques for censored and truncated data*. Springer Science & Business Media, 2006.

- [320] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020.
- [321] A. Hackshaw and A. Kirkwood, “Interpreting and reporting clinical trials with results of borderline significance,” *Bmj*, vol. 343, 2011.
- [322] C. Goutte and E. Gaussier, “A probabilistic interpretation of precision, recall and f-score, with implication for evaluation,” in *European conference on information retrieval*. Springer, 2005, pp. 345–359.
- [323] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” 2013. [Online]. Available: <https://arxiv.org/abs/1211.5063>
- [324] A. Panichella and C. C. Liem, “What are we really testing in mutation testing for machine learning? a critical reflection,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2021, pp. 66–70.
- [325] K. Sugali, “Software testing: Issues and challenges of artificial intelligence & machine learning,” *International Journal of Artificial Intelligence & Applications*, vol. 12, 2021.
- [326] J. Gao, C. Tao, D. Jie, and S. Lu, “What is ai software testing? and why,” in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 27–2709.
- [327] E. Rivera-Landos, F. Khomh, and A. Nikanjam, “The challenge of reproducible ml: an empirical study on the impact of bugs,” 2021.
- [328] Y. Yang, T. He, Z. Xia, and Y. Feng, “A comprehensive empirical study on bug characteristics of deep learning frameworks,” *Information and Software Technology*, vol. 151, p. 107004, 2022.
- [329] S. A. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, “Characterization and automatic updates of deprecated machine-learning api usages,” in *2021 IEEE International*

- Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 137–147.
- [330] D. Seca, “A review on oracle issues in machine learning,” *arXiv preprint arXiv:2105.01407*, 2021.
- [331] M. Usman, Y. Sun, D. Gopinath, R. Dange, L. Manolache, and C. S. Păsăreanu, “An overview of structural coverage metrics for testing neural networks,” *International Journal on Software Tools for Technology Transfer*, vol. 25, pp. 393–405, 2023.