UNDERSTANDING THE CHALLENGES OF REPRODUCING DEEP LEARNING BUGS

by

Mehil B. Shah

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
December 2024

I dedicate this thesis to my family - my mother, Shilpa Shah, and my father, Bimal Shah, who taught me about life and everything beyond, and my dear grandparents, the late Manorama Shah and the late Chandrakant Shah, whose memories I hold close to my heart. Every achievement of mine is a testament to their love, faith, and guidance.

Table of Contents

List of Tables

List of Figures

List of Abbreviations Used

API Application Programming Interface

BERT Bidirectional Encoder Representations from Transformers

CNN Convolutional Neural Network

CPU Central Processing Unit

CWE Common Weakness Enumeration

DCCNN Dual Channel Convolutional Neural Networks

DL Deep Learning

GAN Generative Adversarial Network

GLM Generalized Linear Model

GMM Gaussian Mixture Model

GPU Graphics Processing Unit

JDT Java Development Tools

JIT Just-In-Time

KL Kullback-Leibler

LLM Large Language Model

LSTM Long Short-Term Memory

MLP Multi-Layer Perceptron

MPEG Moving Picture Experts Group

PIL Python Imaging Library

RAM Random Access Memory

RCNN Region-based Convolutional Neural Network

RM-ANOVA Repeated Measures Analysis of Variance

RNN Recurrent Neural Network

SO Stack Overflow

SQL Structured Query Language

VGG Visual Geometry Group

Abstract

Software practitioners often face challenges when reproducing bugs from deep learning systems. Recent studies show that only 3% of deep learning bugs are reproducible, highlighting the inherent difficulties in their reproduction. Deep learning bugs can stem from many sources – training data, faulty code, hardware issues, framework, and environment configurations, which makes their reproduction challenging. Furthermore, the inherent non-determinism and data-driven nature of deep learning systems exacerbate the challenges of bug reproduction. One major challenge arises from faulty training data, which triggers data bugs. While existing literature documents symptoms for most types of deep learning bugs, such as API bugs, GPU bugs, Tensor bugs, and various training and model bugs, there remains a major gap in understanding the symptoms of data bugs. A better understanding of their symptoms will support any attempt to reproduce such bugs. Our first study addresses this gap by comprehensively investigating how data bugs manifest across three types of data. Through a systematic analysis of code-based, text-based, and metric-based benchmark datasets from the software engineering domain, we identify how data bugs manifest themselves through abnormal model behaviours, training dynamics, and the model's internal representations. Building on our understanding of the symptoms and manifestations of deep learning bugs, we investigate the practical challenges of reproducing deep learning bugs. In our second study, we collected 668 deep-learning bugs across three frameworks and 22 architectures. We then manually reproduced 148 bugs and identified ten edit actions and five types of component information that are essential for bug reproduction. We then use the Apriori algorithm to suggest useful information and edit actions required to reproduce specific bugs. Finally, our user study with 22 participants (e.g., developers and researchers) demonstrates that the suggested edit actions and component information can improve bug reproduction success rate by 22.92% and reduce reproduction time by 24.35%. Our research not only enhances the understanding of deep learning bugs (e.g., data bugs) but also offers concrete edit actions and component information to help reproduce them.

Acknowledgements

First and foremost, I am thankful to the Almighty for granting me the physical and mental capacities that empower me to pursue my endeavors with purpose and vigor. I would like to thank my supervisor, Dr. Masud Rahman, who, for the past two years, has not only been my advisor but also an excellent guide. You have taught me everything I know about research. When I joined Dalhousie two years ago, I was an empty book, and in the last two years, you have inspired me with your deep insights, critical thinking, ability to write and respond to any question or comment in the right sense, knack for finding the right research questions and directions, and the list goes on. Despite all the hurdles, the acceptance of my first paper is a testament to your unwavering support and motivation. I am also deeply thankful to my co-supervisor, Dr. Foutse Khomh, who has been a constant supporter of my research and whose expertise has been instrumental in my growth as a researcher. Despite his busy schedule, he consistently finds time to discuss important ideas, provide feedback, and check up on ongoing projects. His depth of knowledge and innovative thinking have contributed significantly to my growth and research. I am particularly grateful to both Dr. Rahman and Dr. Khomh for allowing me to serve on the organizing committee for SANER 2025; these experiences enabled me to network with leading experts in the field and learn more about life in academia. I extend my sincere thanks to Dr. Tushar Sharma, Dr. Hassan Sajjad, and Dr. Lizbeth Olivia Escobedo Bravo for their willingness to be on the examining committee and for their feedback on my work. Their comments helped improve my work significantly, and I deeply value their time and expertise.

I am profoundly grateful to my parents (Shilpa and Bimal) and my grandparents (Mammaji and Papaji) for everything they have done for me. They are my everything, and I aspire to make them proud one day. Words cannot capture my gratitude, and I doubt they ever will, so I simply say thank you for everything; this journey would not have been possible without your hard work and faith. I am especially thankful to my friends in Halifax, beginning with my closest friends, Usmi and Sigma. Usmi has

provided unwavering support throughout all my personal and professional challenges. She remains my first point of contact for major revisions, paper acceptances, and both joyful and difficult personal news. She is my cherished lunch break companion, making it one of the day's highlights. We have engaged in countless discussions on both technical and non-technical subjects. We have shared the most wonderful walks in the harshest weather, yet those walks remain among my fondest memories. I value her positive attitude, personality, and humour immensely. Most significantly, her deep understanding of me facilitates our communication, which is the greatest gift, followed closely by the ZARA perfume and the Bhindi. Sigma has stood beside me through every challenge and triumph, and we have created the most meaningful experiences together: exploring Nova Scotia's beauty, witnessing the Northern Lights, making coffee runs at Tims, TAing classes, debugging code, writing papers, serving on DAGS exec, helping each other move, celebrating festivals, and sharing birthdays. Her friendship is invaluable to me. I am grateful to my friends from the SMART Lab: Indranil, Saurabh, Gautam and Mootez. Indranil's engineering excellence is remarkable, yet it ranks below his exceptional personality, outstanding humor, and engaging company. Saurabh, our self-proclaimed world citizen, combines remarkable talent with unmatched humour and musical taste. His research approach continues to fascinate and inspire me. Gautam, acknowledged as our wisest member, brings joy to every gathering and maintains admirable composure in stressful situations. Mootez serves as my trusted advisor on academic and research matters. His brilliant ideas, knowledge, and work ethic motivate my daily improvement. Despite his expertise, he readily assists others with genuine warmth, and I have valued our idea exchanges and collaborative teaching experiences. I fondly recall the FIFA Nights, cricket match viewings, excellent meals, and wide-ranging discussions at Townhouse 26 with our former lab members and good friends, Parvez Mahbub and Ohiduzzaman Shuvo. I am thankful to Lulu, the extraordinary cat, for bringing pure joy to my days. Additionally, I extend my gratitude to my friends from RAISE Lab - Riasat Mahbub, Asif Samir, and Jitansh Arora - for their steadfast support throughout my journey.

My acknowledgments would be incomplete without mentioning my two closest friends - Bhanvi and Aditya. Our eight-year friendship has enriched my life beyond

words, despite my multiple attempts to express it. I have always maintained that if I had a sister, it would be Bhanvi. She has supported me consistently through both triumphs and challenges, and was the first to learn of my PhD acceptance. Her presence in my life extends far beyond what I can articulate. Regarding Aditya, the term 'friend' seems insufficient; he is my brother, and my presence in this PhD program is largely attributed to him. I have always considered him the more intelligent, dedicated, and superior version of myself, and I am fortunate to call him my brother. He introduced me to true hard work and preparation methods. Our collaborative exam preparation revealed proper work approaches to me, skills that have profoundly influenced my research capabilities. Beyond professional development, he remains my closest confidant, and despite our paths diverging after our Bachelor's, our connection has remained unbroken. While I could elaborate extensively on our friendship, I will simply state: our time in B3-138 and B1-38 represents some of my life's finest moments, and neither my undergraduate years nor life itself would have been the same without you. It has been an honour, my friend. Thank you.

Chapter 1

Introduction

1.1 Motivation

Software bugs are human-made errors in software systems that prevent them from functioning correctly. These bugs and failures cause the global economy to lose billions of dollars annually. Software developers also spend approximately 50% of their programming time tackling these issues [12, 14]. Recently, deep learning has gained significant momentum across many application domains, including software engineering, medicine, and finance. As software systems increasingly incorporate deep learning components in their workflow, managing their bugs and failures becomes more complex. Deep learning bugs originate from multiple sources, including training data, model architecture, and hyperparameters [139]. They pose unique challenges and can lead to severe consequences such as fatal accidents involving autonomous vehicles [135, 1]. Thus, to ensure the quality of software systems using deep learning, understanding these bugs and failures is essential. However, identifying, reproducing, and correcting the deep learning bugs remain a major challenge for software practitioners.

A key prerequisite to correcting deep learning bugs is to reproduce them. However, the reproduction requires a comprehensive understanding of their symptoms or manifestations. While existing studies [53, 137] have documented the symptoms of several types of deep learning bugs, including API, GPU, tensor, and various training and model bugs, there is a significant gap in the understanding of data bugs. Data bugs account for 26% of all deep learning bugs [56] and can originate from various sources such as incorrect labels, duplicates, and missing values [56, 54]. According to a recent study [34], many benchmark datasets contain up to 70% mislabeled data, causing the models trained on them to be faulty. Thus, understanding the data bugs, especially their symptoms and manifestations, is highly warranted to tackle them effectively.

An understanding of the manifestations and symptoms is definitely useful, but

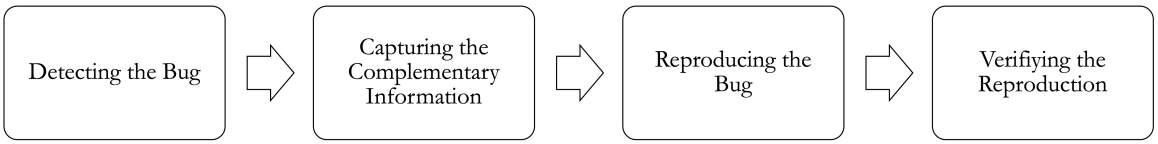


Figure 1.1: Bug reproduction for deep learning bugs

the bugs also need to be reproduced before their correction. Reproduction helps determine the presence or absence of a reported bug in a software system. However, according to existing investigations [93], only 3% of analyzed deep learning bugs are reproducible, which demonstrates the extreme challenges in their reproduction. Deep learning bugs are difficult to reproduce due to their multifaceted dependencies spanning data, hardware, libraries, frameworks, and client programs [21]. The inherent non-determinism of deep learning systems further complicates the reproduction, as the same code or model can produce different outcomes across multiple runs [94]. Moreover, deep learning systems suffer from a lack of interpretability [64], which makes the reproduction of deep learning bugs even more challenging. Recent studies have provided benchmark datasets containing reproducible bugs (e.g., gdefects4dl, Defects4ML) [71, 93], but they focus primarily on providing bug instances rather than their reproducibility. Thus, the reproduction of deep learning bugs, a multi-step, complicated process (Fig. 1.1), poses open challenges for software practitioners, AI engineers, and researchers, which warrants further investigation.

1.2 Problem Statement

Over the last few years, researchers have employed various methodologies to detect, understand, and resolve deep learning bugs automatically. However, despite these advancements, there exists a significant gap in the understanding and the reproduction of deep learning bugs. We discuss two such gaps in the literature as follows.

The lack in the understanding of symptoms of data bugs in deep learning models. Data bugs, the most prevalent type of deep learning bugs [56], pose unique challenges due to their hidden nature and implicit effects on model behaviour. Although existing studies [127, 82, 140] have investigated data quality issues, they have primarily focused on the quality of raw training data, overlooking two crucial

aspects. First, the errors in data preprocessing potentially propagate through the entire development process [10], but their impact on model performance remains understudied. Second, no existing work comprehensively examines how data bugs affect the training behaviours of deep learning models. This gap in the literature makes the understanding of data bugs incomplete, which is essential to reproduce and resolve them effectively.

The lack of guidance for reproducing deep learning bugs. An understanding of bug manifestations and symptoms is crucial, but reproducing deep learning bugs still remains challenging due to their complex dependencies on data, hardware, libraries, frameworks, and client programs [21]. Recent studies have provided benchmark datasets containing reproducible bugs [71, 93], but they primarily focus on dataset construction rather than establishing guidelines for bug reproduction. For example, while Morovati et al. [93] provided a benchmark dataset of 100 deep-learning bugs, they did not include crucial reproduction details such as necessary edit actions and component information for reproduction. As a result, their bugs cannot be reproduced using verbatim code snippets, indicating the need for additional information and specific edit actions. Such a lack of guidance or appropriate tools makes it challenging for software practitioners to effectively reproduce and resolve deep learning bugs. To address these gaps in the literature, we ask the following important research questions:

- RQ1: What are the manifestation patterns and symptoms of data bugs in deep learning systems?
- RQ2: What are the key challenges and requirements for reproducing deep learning bugs?

1.3 Contribution

We conduct two separate but complementary empirical studies to address the above gaps, enhancing the understanding and reproducibility of deep learning bugs. In the first study, we systematically investigate how data quality and preprocessing issues (a.k.a, data bugs) manifest themselves during the training of a deep learning model,

contributing to their overall understanding. First, we select three types of data for our analysis based on their frequent use in software engineering tasks: code-based, text-based, and metric-based data. Second, we select three state-of-the-art baseline models using these data types and train them with the buggy (containing data bugs) and bug-free datasets. Third, we capture the detailed training logs of each model using the Weights & Biases framework [138] and analyze the training metrics and model properties (e.g., gradients, weights, and biases) to derive meaningful insights. We compare the gradients, weights, and biases of faulty and fault-free models to identify symptoms and manifestations of data bugs. When training a model with code-based datasets (Devign [155] and BigVul [37]), we discover that data bugs manifest through gradient instability and reduced learning capacity of the model. We also analyze the model's attention weights and note significant degradation in its code comprehension abilities. When using text-based datasets (Eclipse [66] and Hadoop [13]), their data bugs lead to abnormal weight distributions and overfitting in a model. We also analyze the representations of the model using t-SNE plots and note that data bugs in text-based datasets can lead to poor feature representations, which ultimately affect the downstream task (e.g., duplicate bug report detection). In metric-based datasets (OpenStack [87] and QT [87]), the symptoms of data bugs manifest as vanishing gradients and poor optimization. We also perform the GradCAM [114] analysis for the model and observe that the models trained on faulty data focus on irrelevant tokens and struggle with generalization. All these findings enhance the current understanding of the symptoms and manifestations of data bugs across different contexts in software engineering. Our findings were also validated by repeating the experiments with new datasets.

Building upon the understanding of bug manifestations and symptoms, our second study focuses on improving the reproducibility of deep-learning bugs. First, we construct a dataset of 668 deep learning bugs, i.e., 568 from Stack Overflow and 100 from Defects4ML [93]. These bugs span three frameworks and 22 architectures, categorized into five types: 167 model, 213 tensor, 145 training, 113 GPU, and 30 API bugs. Second, by using stratified sampling, we select 165 of these bugs and attempt to reproduce them using the code snippets and complementary information from Stack Overflow and the benchmark dataset. We found that none of them can be reproduced

using the verbatim code provided. Through manual reproduction of the bugs, and by the analysis of bug reports, we identify ten crucial edit actions and five types of essential component information for the successful reproduction of deep learning bugs. Third, using the Apriori algorithm, we establish clear associations between the types of bugs and the edit actions or component information required for the reproduction of the bugs. These associations deliver important insights and enable us to suggest items for bug reproduction. Finally, to assess the practical value of our findings, we conduct a user study involving 22 developers from industry and academia. In our controlled experiment, the experimental group was equipped with our suggested guidelines for bug reproduction, whereas the control group worked without such guidance. The experimental group successfully reproduced 22.92% more bugs and spent 24.35% less time compared to the control group. All these results demonstrate the benefits of our suggested information supporting the reproducibility of deep learning bugs.

Our research makes several *novel* and significant contributions to the understanding and reproduction of deep learning bugs. First, we are the first to systematically analyze and characterize how data bugs manifest during model training across different types of software engineering data (code-based, text-based, and metric-based), providing detailed insights into models' gradient behaviours, weight distributions, and representations. These insights into data bugs are crucial for their reproduction and thus fill a critical gap in the literature. Second, leveraging our comprehensive understanding of bug symptoms, we develop the first systematic framework to support the reproduction of deep learning bugs. Our use of the Apriori algorithm to establish associations between bug types and key edit actions or component information is novel and provides actionable insights for practitioners. Third, our work is the first to quantitatively demonstrate the effectiveness of reproduction guidelines through controlled experiments with developers, showing significant improvements in reproduction success rates and spent time. Finally, our analysis spans multiple frameworks and architectures, making our findings more generalizable than previous studies focusing on a specific framework or types of bugs [53, 126].

1.4 Related Publications

Our first study is under major revision, and the second study has been accepted by Empirical Software Engineering Journal. We provide a list of relevant publications here. In each publication, I am the primary author and have conducted studies under the supervision of Dr. Masud Rahman and Dr. Foutse Khomh. While I wrote these papers, the co-authors took part in advising, editing, and reviewing them.

- Mehil Shah, M. Masudur Rahman, and F. Khomh, *Towards Understanding the Impact of Data Bugs on Deep Learning Models in Software Engineering*, Empirical Software Engineering Journal (EMSE), pp. 34, 2024 (Under Major Revision).

- Mehil Shah, M. Masudur Rahman, and Foutse Khomh. *Towards Enhancing the Reproducibility of Deep Learning Bugs: An Empirical Study.* Empirical Software Engineering Journal (EMSE), pp. 57, October 2024

I also contributed to other studies on deep learning bugs, which are under revision at the top conferences and journals.

- S. Jahan, M. Shah, and M. Masudur Rahman, *Towards Understanding the Challenges of Bug Localization in Deep Learning Systems",* Empirical Software Engineering Journal (EMSE), pp. 52, 2024 (Under Revision).

- S. Jahan, M. Shah, P. Mahbub, and M. Masudur Rahman, *Improved Detection and Diagnosis of Faults in Deep Neural Networks using Hierarchical and Explainable Classification"*, 47th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 12, 2025 (Under Revision).

1.5 Outline of the report

The report contains five chapters in total. We conduct two separate but complementary studies to better understand the challenges of reproducing deep learning bugs. This section outlines the chapters as follows:

- Chapter 1 discusses the motivation, problem statements, and research contributions and provides an overview of the report structure.
- Chapter 2 provides a comprehensive background required to follow the rest of the report, covering topics such as deep learning bugs, deep learning bug reports, data quality issues, and model explanation techniques.
- Chapter 3 discusses our first study on understanding the symptoms of data bugs and their impacts on model training.
- Chapter 4 discusses our second study on enhancing the reproducibility of deep learning bugs.
- Chapter 5 concludes the report with a list of directions for future works.

Chapter 2

Background

In this chapter, we discuss several important concepts required to follow the rest of the report. First, we discuss deep learning bugs and their different categories: Training, Model, Tensor and Input, API, and GPU bugs. Then, we discuss bug reports, root causes and prevalence of bugs, affected stages in the deep learning pipeline, and bug resolution patterns. We also discuss data quality challenges in software engineering datasets, including label noise, class imbalance, and data obsolescence. Finally, we provide an overview of model explanation techniques, including attention-based analysis, t-SNE visualization, and GradCAM.

2.1 Deep Learning Bugs

Developers often introduce implementation bugs and violate the best practices when developing deep learning software. While bugs represent actual defects triggering system failures or incorrect behaviour, the best practice violations are suboptimal choices impacting performance without causing outright failures. According to existing literature [54, 56], these issues can be classified into five categories. Table 2.1 shows their prevalence in deep learning systems.

2.1.1 Training Issues

Training Bugs: Training bugs represent implementation flaws that directly impair the training process. Common bugs include incorrect loss functions that compute wrong gradients, faulty data augmentation that corrupts training samples, and memory leaks during batch processing. Implementation errors in optimization algorithms can lead to incorrect weight updates or gradient calculations. Some bugs manifest as runtime crashes due to memory mismanagement during training loops, while others silently corrupt the training process, making models fail to converge [33].

Training Best Practice Violations: The violations of the best practices in training lead to suboptimal model performance. These include poor hyperparameter choices such as inappropriate learning rates or batch sizes, insufficient data preprocessing like missing normalization, and inadequate training strategies. Using imbalanced or insufficient training data, failing to implement early stopping, or choosing inappropriate optimization algorithms are common violations. While these issues don't cause system failures, they result in models with poor generalization or unnecessary computational overhead [54].

2.1.2 Model Issues

Model Bugs: Model bugs are implementation errors in neural network architecture. These include incorrect layer connectivity causing broken computational graphs; dimension mismatches between consecutive layers, and implementation errors in custom layers. Such bugs often manifest as runtime or numerical errors in the model's operations. For instance, an incorrect implementation of backpropagation in the custom layers or broken skip connections in residual networks can trigger model bugs, preventing a model from functioning.

Model Best Practice Violations: The violations of the best practices in model construction involve suboptimal architectural choices limiting model effectiveness. Examples include selecting inappropriate model types (e.g., MLPs for image tasks), choosing poor layer configurations, or implementing inefficient network depths. While these models may train and operate without errors, they fail to capture the underlying patterns from data effectively. Other violations include unnecessary model complexity, poor choice of activation functions, or inefficient layer arrangements that impact model performance without causing outright failures.

2.1.3 Tensor and Input Issues

Tensor and Input Bugs: Tensor bugs refer to implementation errors in data handling, such as incorrect reshaping operations that corrupt data structure, wrong indexing that accesses invalid memory locations, or improper type casting causing numerical errors. Input bugs include implementation flaws in data loading pipelines that corrupt input data, wrong channel ordering, or incorrect data type conversions

that cause runtime errors.

Tensor and Input Best Practice Violations: The best practice violations in tensor and input handling include inefficient data preprocessing pipelines, suboptimal tensor operations that increase computational overhead, and poor input validation strategies. While these don't cause system failures, they may result in unnecessary memory usage, slower processing times, or reduced model robustness to input variations.

2.1.4 API Issues

API Bugs: API bugs can be triggered by the uses of incorrect implementation of framework interfaces, such as using incompatible API versions that cause compilation errors, wrong parameter ordering in function calls, or improper error handling in API interactions. These bugs often result in runtime errors or undefined behaviour due to mismatched interfaces or incorrect API usage patterns.

API Best Practice Violations: API best practice violations include using deprecated but still functional API versions, adopting inefficient API usage patterns, or failing to utilize framework optimizations. While these issues don't cause explicit system failures, they may result in reduced performance, poor maintainability, or technical debt.

2.1.5 GPU Issues

GPU Bugs: GPU bugs, as suggested by Jahan et al. [59], are implementation errors in GPU-related code. These include memory access violations, race conditions in parallel operations, and incorrect synchronization implementations. Such bugs often cause system crashes, memory corruption, or incorrect computational results. Implementation errors in GPU device mapping, parallel execution, or memory transfers represent actual bugs posing significant challenges [112].

GPU Best Practice Violations: GPU best practice violations involve suboptimal usage of GPU resources without causing outright failures. These include poor memory management strategies and inefficient parallelization approaches. Moreover, violation of the best practices can lead to suboptimal data transfer patterns between CPU and GPU, which results in suboptimal performance [97]. While these violations don't

Table 2.1: Prevalence of different types of deep learning issues

Issue Category	**Training**	**Model**	**Tensor and Input**	**API**	**GPU**
Prevalence	52.5%	19.7%	19.5%	5.3%	2.9%

cause system crashes, they result in performance issues such as increased number of required memory operations, and reduced computational efficiency [131].

2.2 Deep Learning Bug Reports

To date, only limited investigation has been conducted to understand the patterns of deep learning (DL) bug reports. Long et al. [77] performed the first exploratory study to analyze the bug reporting trends and patterns for deep learning frameworks. Their work revealed several key themes that provide insight into the nature and lifecycle of these bug reports:

Root Causes and Prevalence: Their study has found that low training speed is the most common symptom for submitting performance-related bug reports, ranging from 27% to 67% across the different DL frameworks. However, no consistent pattern was observed for the root causes of accuracy-related reports. This suggests that performance issues, especially those manifesting as slow execution speed, are a major pain point encountered by the users of DL frameworks.

Affected Stages of DL Pipeline: Across the frameworks studied, the training stage was found to be the most prevalent in performance and accuracy bug reports, ranging from 38% to 77%. This finding is not surprising, as training is typically the most computationally intensive and time-consuming stage of the DL pipeline [117]. Performance bottlenecks or accuracy issues encountered during training can significantly impact the overall usability and efficiency of the framework.

Report Quality and Resolution: Their study found that a majority of the closed reports (69% to 100%) were either not classified or their titles, labels, and content did not match the actual bugs reported. These findings highlight inefficiencies in bug reporting and difficulties in the resolution process.

2.3 Data Quality Issues in Software Engineering Datasets

Label Noise: Label noise refers to errors in the labels assigned to data instances in a dataset. These errors can stem from various sources, including insufficient information, annotator mistakes, subjective judgments, and data encoding issues [39]. Label noise is prevalent in real-world datasets and thus can significantly impair the DL models using those datasets [40]. This issue is also relevant in software engineering datasets since they are often used to train DL models supporting code search, vulnerability detection, and program understanding [136, 142, 98]. For example, in a dataset for vulnerability detection, non-vulnerable code could be mistakenly labelled as vulnerable or vulnerable code could be labelled as non-vulnerable. Such label errors often result in inaccurate, biased, or flawed models [98]. In this work, we train multiple deep learning models using datasets containing label noise to determine its impact on the models.

Class Imbalance: Class imbalance refers to a disproportionate representation of different classes within a dataset. It is also highly prevalent in various software engineering datasets, including API recommendation, code review automation, and defect prediction [55, 132, 119]. For example, in defect prediction, there are often significantly fewer defective instances than non-defective ones. Existing studies [87, 60, 61, 81] show that only 5%-26% of the files contain the defective instances. When trained on imbalanced datasets, DL models tend to be biased towards the majority class and demonstrate poor performance in identifying the minority class [45].

Data Obsolescence: Data obsolescence, also known as concept drift, refers to the evolution of data over time and is a significant challenge for software engineering datasets. As software systems evolve, the characteristics of their data change, which leads to concept drift. This phenomenon is prevalent in many datasets, including the ones used for log-level prediction, anomaly detection and duplicate bug report detection [80, 99, 151]. Recent research by Zhang et al. [151] revealed that most duplicate bug report detection techniques were only evaluated using data up to January 2014. When these same techniques were applied to more recent data, their performance decreased significantly, highlighting the impact of data obsolescence on the effectiveness of deep learning models.

2.4 Model Explanation

Attention-Based Analysis: Attention mechanisms help deep learning models focus on the most relevant parts of the input [134]. Their inherent ability to assign importance weights to different input elements can also be leveraged to interpret the behaviours of the models. Recently, attention weights of input have been used to investigate the explainability of the deep learning models solving software engineering tasks [42, 105, 41]. For example, Fu et al. [42] leveraged the self-attention mechanism to explain the predictions of their proposed technique for vulnerability detection. In this study, we employ their dataset and attention-based analysis to examine how data quality and preprocessing issues affect a DL model's training behaviour and learning capacity.

t-SNE: t-Distributed Stochastic Neighbor Embedding (t-SNE) [133] is a technique that can reduce high-dimensional feature representations learned by neural networks and visualize them in 2D or 3D plots. These plots are often used to inspect how well a model has learned to differentiate among different classes in the data. By comparing t-SNE visualizations for different models and examining their separation of classes, we can evaluate their capacities to learn feature representations for a given task. This technique has also been used in several software engineering tasks, including duplicate bug report detection [88]. As duplicate bug report detection is one of our selected tasks, we utilise t-SNE to explain and visualize the predictions of our trained model for this task.

GradCAM: GradCAM (Gradient-weighted Class Activation Mapping) is a technique that visualizes the input features contributing the most to a neural network's outputs [114]. By analyzing the feature weights in the final convolutional layer of a convolutional neural network, GradCAM produces a heatmap highlighting the important regions in the input relevant to the model's output. In our study, we employ DeepJIT [52] as a baseline technique for defect prediction. Since DeepJIT is a CNN-based technique, we use GradCAM to understand and visualize the impact of input features on the model output.

2.5 Summary

This chapter provided an overview of key concepts essential for understanding the report. We discussed different types of deep learning bugs (e.g., Training, Model, Tensor and Input, API, and GPU bugs) and their prevalence. We then explored bug reports and their characteristics, including root causes and affected stages in the DL pipeline. We also examined three major data quality challenges in software engineering datasets: label noise, class imbalance, and data obsolescence. Finally, we described model explanation techniques, including attention-based analysis, t-SNE visualization, and GradCAM. The next chapter provides our first study on understanding the symptoms of data bugs in deep learning systems.

Chapter 3

Towards Understanding the Impact of Data Bugs on Deep Learning Models in Software Engineering

In this chapter, we investigate how data bugs manifest themselves during the training of deep learning models targeting software engineering tasks (e.g., defect prediction). Existing studies suggest that data bugs account for 26% of all deep learning bugs [56], with some datasets containing up to 70% mislabeled data entries [33]. However, the current understanding of the symptoms and manifestations of data bugs is limited, posing significant challenges to their reproduction. In this chapter, we fill this gap in the literature through a systematic investigation of data bugs from code-based, text-based, and metric-based data captured from 12 software engineering datasets.

The remainder of this chapter is organized as follows. Section 3.1 provides an overview of the research problem and our conducted study. Section 3.2 describes our study methodology and experiment design. Section 3.3 presents our findings on how data quality issues and preprocessing errors manifest differently across code-based, text-based, and metric-based data. Section 3.4 discusses the implications of our findings for bug reproduction and verification strategies. Section 3.5 addresses the threats to validity. Section 3.6 reviews related work, and finally, Section 3.7 concludes the chapter with key takeaways and future directions.

3.1 Introduction

Deep Learning (DL) solutions have been widely adopted in many applications, including speech recognition [11], software testing [84, 35], autonomous driving [49], and software development [144]. Compared to traditional logic-driven software, DL-based software adopts a data-driven computing paradigm where its models are trained using data. However, the training data can be noisy, biased, or incomplete, which may lead to unexpected or erroneous behaviours in the DL models [139]. Besides,

deep learning systems can be very complex and opaque, making it difficult to understand their reasoning process. Thus, the poor quality of training data and the lack of transparency in the DL models could pose significant challenges to the reliability and trustworthiness of the DL-based systems.

Software systems are prone to different bugs, which manifest as system failures, erroneous outputs, or unpredictable behaviours [32]. Similarly, deep learning systems, while subject to conventional software bugs, also face unique challenges due to their complex architectures and data-driven paradigms [16]. Their bugs can originate from various sources, including training data, hyperparameters, and model parameters, and can lead to system crashes and unexpected runtime behaviour [56]. They can also result in severe consequences, as evidenced by the fatal accidents involving self-driving cars [135, 1]. Hence, bugs must be identified and fixed before deploying a deep-learning model in production. However, DL bugs are complex due to the multifaceted dependencies and non-determinism of DL systems. Their non-determinism leads to different outcomes across multiple runs, making debugging challenging [94]. Moreover, deep learning systems suffer from a lack of interpretability, which exacerbates the challenge in debugging [64].

According to existing literature [56], DL bugs can be divided into five categories: Data, Model, Structural, Non-Model Structural and API Bug. Among these categories, data bugs have been reported as the most prevalent ones, accounting for 26% of all bugs in deep learning systems [56]. They originate from the errors in training data, such as incorrect labels, duplicates, out-of-distribution records, and missing values [56, 54, 31]. These errors can significantly affect the model performance [70, 111]. However, resolving the data bugs is highly challenging [147, 136]. Data bugs are hidden in the dataset and implicitly affect a model's behaviour. They also propagate to the model parameters during training, which makes their detection difficult. Furthermore, according to an existing work [34], many existing benchmark datasets constructed by human annotators contain up to 70% mislabeled data, which leads to data bugs in the DL models relying on those datasets [34].

Moreover, the lack of understanding of how data bugs manifest during model training makes it particularly challenging to verify their reproduction. While practitioners can replicate the environmental conditions and code-level issues, they often

lack reliable indicators to confirm whether they have successfully reproduced data-related bugs. Understanding the symptoms and manifestations of data bugs during training can provide crucial verification mechanisms for bug reproduction, enabling more effective debugging and resolution strategies.

Existing studies [127, 82, 62, 140, 141, 33] have investigated the impact of data quality issues (e.g., label noise, class imbalance) on model performance. However, they primarily focus on the quality of the raw training data and overlook two crucial aspects: the preprocessing stage and the model training process. First, data preprocessing is essential for preparing the data in a suitable format for training. Errors, biases, or loss of information during preprocessing can propagate through the whole development steps of a model, significantly degrading the model's performance [10]. Despite its importance, the impact of preprocessing on model performance has not been thoroughly investigated by the existing literature. Second, training is a crucial step in model development. Observing the training process and monitoring a model's internal state, such as gradients, weights, and biases, can offer valuable insights into how data quality issues affect the model's learning process. For example, if the model is experiencing vanishing or exploding gradients, they might indicate poor feature scaling or outliers in the data [46]. Similarly, analyzing the weights and biases of the model during training can indicate whether the model is learning meaningful patterns or is being misled by noisy or corrupted data points [63]. Such insights can help practitioners take appropriate actions to mitigate their impact. However, no existing work examines the impact of data bugs on the training behaviours of DL models. Our study aims to address these critical gaps in the literature.

In this chapter, we conduct an empirical study to investigate the impact of data quality and preprocessing issues on the training of deep learning models used in software engineering tasks. First, we select three types of data for our analysis based on their frequent use in software engineering tasks: code-based, text-based, and metric-based. Second, we select state-of-the-art baseline models using these data types and compare their faulty models (containing data bugs) with corresponding bug-free versions. Third, we capture the detailed training logs using the Weights & Biases framework and analyze the training metrics and model properties (e.g., gradients,

weights, and biases) to derive meaningful insights. We performs quantitative analyses of the gradients, weights, and biases to identify symptoms and manifestations of data quality and preprocessing issues. Finally, we validate our findings with new datasets to ensure the generalizability of our results. Thus, we answer four important research questions as follows:

RQ1: *How do data quality and preprocessing issues in code-based data affect the training behavior of deep learning models?*

We investigate the impact of data quality issues from two code-based datasets, Devign and BigVul [155, 37]. These datasets contain C/C++ functions from multiple projects and have known quality issues [33]. Our analysis reveals that these issues cause gradient instability during training and significantly impair a model's capacity to capture complex patterns. Furthermore, our attention analysis (extracting token-level attention patterns, measuring their consistency, validating across samples) reveals how data quality affects model behaviour, which shows that data quality issues in the code-based data can reduce the code comprehension abilities of the deep learning model.

RQ2: *How do data quality and preprocessing issues in text-based data affect the training behaviour of deep learning models?*

We assess the impact of data quality issues from two text-based datasets, Eclipse, maintained in Bugzilla and Hadoop, maintained in JIRA. Our study reveals that quality issues in text data lead to reduced abnormal weight distributions and overfitting of models to noisy patterns during model training. Our analysis using the t-SNE plots highlights the deep learning models' difficulty learning consistent feature representations from noisy data.

RQ3: *How do data quality and preprocessing issues in metric-based data affect the training behavior of deep learning models?*

We investigate the impact of data quality issues from two metric-based datasets, Openstack and QT, which frequently suffer from class imbalance problems [87, 52]. Our analysis reveals that quality issues in metric-based data lead to vanishing gradients and poor optimization during the training process. Additionally, we perform post-hoc analysis using GradCAM, an explainable AI technique for visual analysis of model inputs. Our analysis demonstrates that models trained on imbalanced data

focus on irrelevant tokens and struggle to generalize to unseen data.

RQ4: *How well do our findings on data quality and preprocessing issues generalize to other code-based, text-based, and metric-based datasets?*

We evaluate the generalizability of our findings using six new datasets: D2A and Juliet (code-based), Spark and Mozilla (text-based), and Go and JDT (metric-based). Our analysis reveals that the data quality and processing issues in the training data lead to reduced model capacity, gradient instability, overfitting, and biased learning of the models, which align with our above findings. In contrast, the models trained on cleaned datasets show none of these issues. Such observations increase confidence in our findings and underscore the challenges of data bugs in deep learning models used in software engineering tasks.

3.2 Methodology

Fig. 3.1 shows the schematic diagram of our empirical study. We discuss different steps of our study as follows.

3.2.1 Data Type Selection

Selecting different types of data is crucial for our study since each data type has unique characteristics and quality issues. By examining multiple types of data, we can comprehensively investigate how data bugs manifest and affect deep learning models. Based on the prevalence in software engineering datasets, Yang et al. [144] identified the three most prevalent types of data: code-based, text-based, and metric-based. These data types have unique characteristics as follows.

Code-based data: Code-based data is frequently used in training deep learning models that target various software engineering tasks, such as code clone detection, code generation, program repair, and vulnerability detection [42, 149, 68, 129]. This data type encompasses source code files, test cases, and code changes [144].

Text-based data: Natural language texts play a crucial role in numerous software engineering tasks. Existing studies have used text-based data in deep learning techniques for software engineering tasks [144]. Text-based data includes requirements specifications, design documents, code comments, commit messages, bug reports, user reviews, and question-answer posts from forums like Stack Overflow [51, 99, 90].

Metric-based data: Metric-based data comprises various statistics derived by static analysis tools (e.g., SonarQube, Understand) from various software repositories. They quantify different aspects of source code, software design, development process, and software quality. Metric-based datasets have been used in several software engineering tasks such as defect prediction, effort estimation, code smell detection, and software maintainability assessment [52, 74, 29].

Thus, based on the prevalence of data types and their relevance to software engineering tasks, we consider *code-based*, *text-based* and *metric-based* data for our study (Step ①, Fig. 3.1).

3.2.2 Study Design

Task Selection

Selecting representative tasks for each type of data above is a critical step in our analysis (Step ②, Fig. 3.1). By carefully choosing the tasks that use our selected data types, we can examine how data bugs manifest and affect the training of deep learning models. In particular, we have selected the following representative tasks for our analysis, as suggested by Yang et al. [144]:

(a). Vulnerability Detection: Detection of vulnerabilities in software code is one of the key applications where deep learning models show promising results. These models often use source code or binary for their detection task.

(b). Duplicate Bug Report Detection: Duplicate bug report detection is a prominent use case of text-based data from software engineering that is leveraged by DL models. The task's objective is to identify bug reports that describe the same underlying issue from previous bug reports.

(c). Defect Prediction: Predicting defects in code components using software metrics is a common application of deep learning in software engineering. This task involves predicting the defect proneness of software modules based on various code metrics (e.g., complexity, coupling, cohesion).

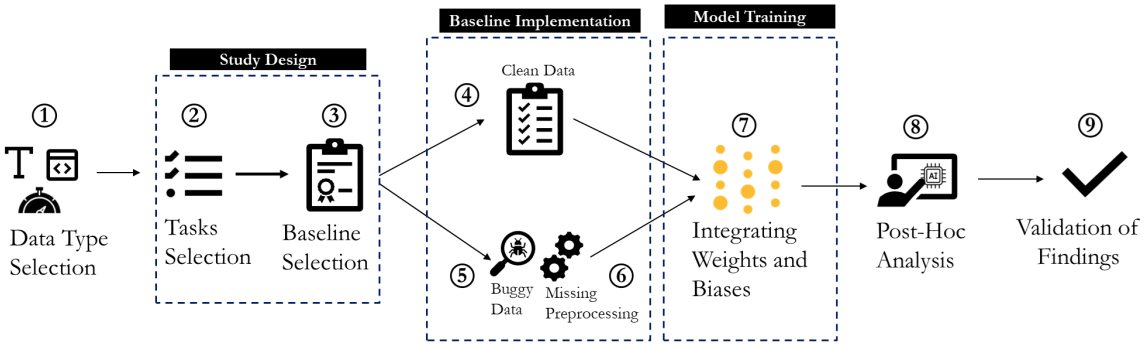


Figure 3.1: Schematic diagram of our study

Baseline Selection

After selecting the representative tasks, we choose multiple baseline approaches for the selected tasks. These approaches provide baseline deep learning models, which are used in our subsequent analysis. To select baselines for each of the types of data, we used the following criteria: (a) the technique should use a neural network model to perform one of the three tasks above, and (b) the technique should provide a comprehensive replication package with source code, datasets, and instructions for reproducibility. Furthermore, we prioritized the baselines with clean and buggy versions in their replication packages. Based on these filtration criteria, we selected the following state-of-the-art baselines for our study.

(a) Code-based data: LineVul [42] for vulnerability detection. LineVul is a state-of-the-art transformer-based model for line-level vulnerability detection in code. The model takes code tokens as input and learns to predict the vulnerability status of each line.

(b) Text-based data: DCCNN [51] for duplicate bug report detection. DCCNN, short for Duplicate bug report detection using Convolutional Neural Networks, employs a CNN-based architecture to identify duplicate bug reports from their textual descriptions. The CNN model learns to extract relevant features and patterns from a pair of bug reports and then predict whether they are duplicates or not.

(c) Metric-based data: DeepJIT [52] for defect prediction. DeepJIT is an end-to-end deep learning framework for Just-In-Time (JIT) defect prediction. This technique computes software metrics from datasets of code changes and extracts salient features from commit messages. The learned features from commit messages

Table 3.1: Summary of datasets used in our study

Data Type	Dataset	Size	Description
Code-Based	Devign [155]	27,318 functions	Vulnerable functions from 5 open-source projects in C/C++
	BigVul [37]	188,636 functions	Vulnerable code snippets from 211 projects in multiple languages
Text-Based	Eclipse [66]	74,376 bug reports	Bug reports from Eclipse project (2001-2007)
	Hadoop [13]	14,016 bug reports	Bug reports from Hadoop project (2006-2013)
Metric-Based	OpenStack [87]	66,065 source files	21 code metrics per file from OpenStack project
	QT [87]	95,758 source files	21 code metrics per file from QT project

and code changes are encoded into numerical matrices and then processed by separate CNN layers to predict whether the commit will likely introduce a defect.

Besides the relevance to the selected data types, these baselines have demonstrated state-of-the-art performance in their respective tasks, which justifies their selection for our study (Step ③, Fig. 3.1).

Datasets

We utilized six diverse datasets that were used by deep learning models targeting our tasks above: Devign and BigVul (code-based), Eclipse and Hadoop (text-based), and OpenStack and QT (metric-based). These datasets vary significantly in size, composition, and specific characteristics, allowing for comprehensive training and evaluation of deep learning models across different tasks. For each type of data, we chose two datasets to perform data source triangulation, per the guidelines by Runeson et al. [110], and to have multiple sources of evidence as recommended by Yin et al. [146]. Table. 3.1 provides a detailed summary of each dataset, including their sizes and brief descriptions, offering a clear overview of the data used in our experiments.

3.2.3 Experimental Setup

This section describes our experimental methodology, including how we configure our system, implement the baselines, and observe training behaviours.

System Configuration

To reflect the original environments of our selected baselines, we use the following setup:

(a) Code Editors: We use Visual Studio Code v1.79.0, which is a popular code editor for building DL-based applications [43].

(b) Dependencies: To automatically detect the API libraries used in the baseline techniques, we use the pipreqs package [1]. We also install the dependencies for each baseline into a separate virtual environment using the venv [2] module.

(c) Frameworks: We use Tensorflow and PyTorch for our experiments, as DeepJIT [52] and LineVul were originally developed in PyTorch, whereas DCCNN was originally developed in Tensorflow.

(d) Python Version: For our experiments, we use the same versions of Python originally used by the authors when they published their work.

(e) Hardware Config: Our experiments were run on a Compute Canada node having a Linux (CentOS 7) Operating System with 64GB primary memory (i.e., RAM) and 16GB GPU Memory (NVIDIA V100 Tensor Core GPU).

Baseline Preparation

To prepare the baselines, we utilized the replication packages provided by the authors of the original studies [42, 52, 51]. This allowed us to accurately reproduce the baseline setups and maintain the integrity of their original implementations. We prepared three variants of the original baselines for our analysis, which are described below.

1. Baseline with Clean Data: First, we obtained clean datasets from existing studies [33, 151, 148] and prepared the original techniques to be trained using this high-quality data (Step ④, Fig. 3.1). This variant represents the ideal setup for the models as it is prepared with clean and high-quality data.

2. Baseline with Buggy Data: To investigate the impact of data quality on model performance, we created a second variant prepared with data quality issues. We obtained buggy datasets from the same existing studies [33, 151, 148] and configured the baseline models to be trained using this buggy data (Step ⑤, Fig. 3.1). This step will allow us to observe how bugs in the training data might affect the model's learning process and subsequent performance.

3. Baseline with Missing Preprocessing: For our third variant, we focused

[1] https://pypi.org/project/pipreqs/
[2] https://docs.python.org/3/library/venv.html

on preprocessing faults, which are among the most prevalent types of faults in deep learning, according to the existing taxonomy [54]. We removed the preprocessing operations from the training pipeline in preparation for training (Step ⑥, Fig. 3.1). By setting up the model to simulate the missing preprocessing operation, we aim to understand how the absence of these crucial steps might impact a model's performance and robustness.

Integrating Weights and Biases

To better understand the training process and model behaviour, we integrated Weights and Biases (W&B) logging [138] into the existing baselines and their variants (Step ⑦, Fig. 3.1). After successfully integrating W&B, we trained the models and monitored and captured their higher-level training, such as the model's loss and accuracy during the training. Additionally, we logged lower-level statistics such as gradients, weights, and biases from all model layers. This allowed us to track a model's progress and performance over time. Furthermore, we logged the runtime configurations to capture the computational environment in which the experiments were conducted. Finally, to account for the variability and stochasticity inherent in neural networks, we ran each baseline model and corresponding variants five times [15].

3.2.4 Quantitative Analysis

To quantitatively analyze the impact of data quality and preprocessing issues on the deep learning models, we leveraged the data from Weights and Biases (W&B) captured during the training process. We use the gradients, weights, and biases over multiple runs and calculate aggregated statistics for each layer of the models (Step ⑦, Fig. 3.1). Table. 3.2 discusses the aggregate metrics used for analysis. By examining these aggregated statistics, we identified symptoms and manifestations of data quality and preprocessing issues and quantified their impact on deep learning models. We analyzed the statistical differences between models trained on clean datasets and those trained on buggy datasets to identify if there were any differences in their performance. We observed how data bugs and preprocessing issues affected different layers' weights, biases, and gradients. By quantifying the impact on specific parts of the models, such as attention layers, convolutional layers, and fully connected layers,

we gained insights into how data quality and preprocessing issues can degrade the performance and reliability of deep learning models in software engineering tasks.

3.2.5 Post-Hoc Analysis

To gain deeper insights into the impact of data bugs on our deep learning models, we conducted a post-hoc analysis using explainable AI techniques (Step ⑧, Fig. 3.1). These techniques help us understand how the models make predictions and identify the influential parts of the input data.

For the code-based data, we analyzed LineVul's attention weights. Since the LineVul model is attention-based, an analysis of its attention weights helps us identify the parts of the input code that the model focuses on when making predictions about vulnerabilities. By comparing the attention weights from models trained on clean and buggy data, we observed how data bugs affect a model's attention and potentially lead to incorrect predictions.

For the text-based data, we utilized t-SNE (t-Distributed Stochastic Neighbor Embedding) visualization [133]. t-SNE is a dimensionality reduction technique that helps us visualize high-dimensional data in a lower-dimensional space. By applying t-SNE to the representations learned by the DCCNN model at different layers, we visualized the decision boundary for duplicate and non-duplicate bug reports. This helps us understand how the learned representations differ between models trained on clean and buggy data, providing insights into the impact of data bugs on the model's learning process.

For the metric-based data, we implemented GradCAM [114]. In the context of defect prediction using DeepJIT, GradCAM helps us visualize which metrics and parts of the input features are most influential in the model's decision. By contrasting the GradCAM outputs of models trained on clean data against those trained on buggy data, we identify how data bugs affect the model's focus and potentially lead to incorrect defect predictions.

3.2.6 Validating the Derived Findings

To assess the generalizability of our findings from RQ1, RQ2, and RQ3, we conducted additional experiments using six new datasets not included in our initial analysis

Table 3.2: Aggregate metrics for model analysis

Operator	**Description**
max	The maximum value of the property in the layer.
min	The minimum value of the property in the layer.
median	The median value of the property in the layer.
mean	The average value of the property in the layer.
var	The variance of the property in the layer.
std	The standard deviation of the property in the layer.
skew	A measure of the asymmetry of the distribution of the property in the layer.
kurt	A measure of the peakedness and tail heaviness of the property's distribution in the layer.
spar	The fraction of properties in a layer that are zero or close to zero.

Table 3.3: Summary of datasets used for validation and generalization

Data Type	**Dataset**	**Size**	**Description**
Code-Based	Juliet	253,002 test cases	C/C++ and Java test cases covering 181 CWEs
	D2A	1,295,623 samples	Samples from six open-source projects such as OpenSSL, FFmpeg etc.
Text-Based	Mozilla	193,587 bug reports	Bug reports from Mozilla projects
	Spark	9,579 bug reports	Bug reports from Apache Spark project
Metric-Based	Go	61,224 files	Metrics from Go programming language project
	JDT	13,348 files	Metrics from Eclipse Java Development Tools

(Step ⑨, Fig. 3.1). In particular, we chose six new datasets representing three types of data: two code-based datasets (Juliet and D2A), two text-based datasets (Mozilla and Spark), and two metric-based datasets (Go and JDT). We collect the buggy and clean versions of these datasets from the same studies [151, 148, 33], which were used in our previous steps. Table 3.3 summarises our validation datasets, including their sizes, compositions, and brief descriptions.

In our validation, we retrained each of the baseline models (LineVul, DCCNN, and DeepJIT) with their corresponding datasets containing clean and buggy data. We performed similar quantitative analyses, including monitoring training behaviour and examining model components. Moreover, we also perform the post-hoc analyses using the same techniques (attention weight analysis, t-SNE visualization, and GradCAM) as in our original study. By comparing the results from these validation experiments with our initial findings, we aimed to determine whether the observed impacts of data quality issues are consistent across different datasets within each data type, and this assessed the generalizability of our conclusions.

3.3 Study Findings

In our study, we examined the effects of four common data bugs: label noise, class imbalance, concept drift, and missing preprocessing. To address each research question, we constructed and trained a comprehensive dataset of 120 buggy models, with 30 models dedicated to each of the four bug types. Additionally, we trained 30 models on clean data to serve as a baseline, helping us establish the expected training behaviour. We used the W&B logging framework [138] to capture the training behaviours of both faulty and baseline models. This section presents our findings, focusing on the most common symptoms of these data bugs across three data types: code-based, text-based, and metric-based.

3.3.1 RQ1: How do data quality and preprocessing issues in code-based data affect the training behaviour of deep learning models?

Impact of Data Quality

(a) Inconsistent Learning: Inconsistent learning in neural networks refers to the phenomenon where different layers or components of a network learn at varying rates or effectiveness, leading to suboptimal overall performance [18]. We observe inconsistent learning across the neural network layers when models were trained on low-quality data. As shown in Table. 3.4, 80% of the models trained on data with label noise and 56.67% of the models trained on data with concept drift demonstrated inconsistent learning. Our manual analysis of these models shows that ≈20% of their layers were affected, especially the attention and output layers. Due to the data bugs, these layers learn very slowly, as shown by their near-zero biases (< 0.01) [95]. In contrast, the layers unaffected by data quality issues learn quickly (bias $>= 0.5$). Attention and the output layers of the models also showed normal bias values (bias $>= 0.5$). Thus, the disparity in the bias values across the different layers of a neural network trained on low-quality code data results in inconsistent learning.

(b) Reduced Model Capacity: We found that models trained on low-quality data exhibited a reduced capacity to capture complex patterns from data. As shown in Table 3.4, 86.67% of models trained on data with label noise and 63.33% of models

Table 3.4: Manifestations of data bugs in code-based models

Data Quality Issues	Inconsistent Learning	Reduced Model Capacity	Gradient Instability
Label Noise	80.00%	86.67%	73.33%
Class Imbalance	6.67%	23.33%	53.33%
Concept Drift	56.67%	63.33%	16.67%

trained with concept drift demonstrated reduced capacity. Our analysis of these models revealed that 90-95% of their layers were affected, particularly embedding, attention, and dense layers. These layers exhibited notably small weights (e.g., $\mu \approx 0.011$, $\mu \approx 0.047$ for models trained with label noise and concept drift, respectively). These values were significantly lower than the typical weight values of 0.1 to 0.3, which were observed when the models were trained on clean data. The small and tightly clustered weights suggest a limited range in the representational power of a model's neurons, which indicates the reduced capacity of the corresponding model [75, 18].

(c) Gradient Instability: We also noted gradient instability when models were trained on low-quality data. As shown in Table 3.4, 73.33% of models trained on data with label noise and 53.33% with class imbalance exhibited gradient instability. Our analysis of these models revealed that 30-40% of their layers were affected, especially the attention and dense layers. These layers demonstrated extreme gradient values, ranging from 1×10^{-9} to 1×10^{1}, which was a significant deviation from the range of 1×10^{-3} to 1×10^{-1}, as observed in the baseline models trained on clean data. This wide range of gradient magnitudes leads to unstable learning, where some layers update their weights rapidly, and others remain almost static. This gradient instability results in suboptimal model performance, as the model fails to develop a coherent understanding of code data across all of its layers [100].

Impact of Missing Preprocessing Operations

(a) Slow Convergence: In code-based data, preprocessing operations such as tokenization and stopword removal are crucial, as they enable the accurate parsing and analysis of source code. Our analysis reveals that 66.67% of the models trained with missing preprocessing operations had slower convergence. We also found that ≈15% of the layers in these models suffered from slow convergence, especially the output layers. The slow convergence of the parameters of the output layers indicates that

no class was probable enough, which affects the decision-making ability of a model. Their weights and gradients were close to zero ($< 10^{-6}$) as opposed to 1×10^{-3} to 1×10^{-1} from the baseline models [24]. Such low gradients and weights often result in inefficient gradient flow, which affects a model's ability to converge towards the optimal parameter values.

(b) Incorrect Learning: We noted that the lack of preprocessing operations in code-based data led to inconsistent or incorrect learning. Our analysis showed that the models trained on clean datasets exhibited a uniform and bounded distribution in their bias values (e.g., ranging from 0.34 to 0.72). Such a distribution enables better learning and generalization to unseen data [120]. On the contrary, we noted unbounded bias values in the models trained on data without preprocessing. Some layers displayed bias values near zero ($\approx$0), while others had bias values exceeding 1.0. This behavior was prevalent in 53.33% of the models trained without preprocessing, affecting $\approx$60% of their layers. These inconsistent bias patterns indicate that the models struggle to leverage biases effectively, resulting in a limited set of learned weights primarily clustered near the origin ($\approx$0). Consequently, this leads to poor generalization and reduced "guessing power", degrading the model's ability to comprehend and analyze source code effectively [19].

(c) Distorted Distributions: Missing preprocessing steps can significantly distort the distribution of weights from deep learning models, as suggested by our analysis. We found that 70% of the models trained without preprocessing were affected. Our analysis of these models showed that 80% of their layers exhibit distorted distributions, particularly in embedding, attention, and dense layers. They demonstrated a skewness of > 1 and a kurtosis of $|k| >>> 3$, which are significantly higher than the skewness (0.4) and kurtosis (1.3) of the model weights trained on clean data. Such high skewness and extreme kurtosis indicate that the weights and gradients of an affected model are concentrated at a point farther right from the baseline counterparts. That is, small changes in input can cause a large change in output, which makes the model highly sensitive to input changes, reducing its robustness and reliability.

Impact analysis of Label Noise using Attention Mechanisms

Attention weights have been effectively used in software engineering research to understand the correlation between the input and a model's predictions (outputs) [25]. In our study, we also analyze the attention weights of a model to understand the impact of data quality issues on the model's training. Our study examined various data quality issues, and we found that label noise had the strongest impact on models trained with code-based data (see Table 3.4). Based on this finding, we decided to conduct a post-hoc analysis to gain a deeper understanding of how label noise affects code-based data. To investigate this issue, we trained 60 models in total. We divided these models into four groups of 15 models each: (a) models trained on a clean Devign dataset, (b) models trained on a clean BigVul dataset, (c) models trained on a Devign dataset with label noise, and (d) models trained on a BigVul dataset with label noise. For each group, we calculated the average attention weights across all 15 models. This approach helps us identify consistent patterns and mitigate the impact of non-determinism in deep learning. We discuss our findings below.

Bug-free Data: Models trained on bug-free datasets on vulnerability detection demonstrate a strong focus on tokens critical for identifying potential security flaws. This focus is evident in the BigVul dataset, which contains source code from Chrome, Linux, Android, and Tcpdump. When analyzing models trained on this dataset, we observe high attention weights on key functions and data structures related to security vulnerabilities. For instance, `rds6_inc_info_copy` (87.3796) and `struct rds_incoming` (64.1523) receive significant attention, indicating their relevance to network-related vulnerabilities. The models also heavily emphasize network fields such as `laddr` (85.7431) and `faddr` (86.2221), demonstrating their awareness of potential security issues in network communications. The Devign dataset contains source code from Linux, FFmpeg, Qemu, and Wireshark. Models trained on this dataset focus on memory-related operations and type conversions, which are one of the primary causes of security vulnerabilities [123]. Tokens like `uint16_t` (88.8645) and `uint64_t` (86.9365) receive very high attention weights, which reflects their importance to proper data handling. Critical function calls for cross-platform compatibility, such as `le16_to_cpu` (82.2068) and `le64_to_cpu` (80.8885), are also strongly emphasized. Models trained on both datasets show significant attention to

error-checking patterns and memory operations like `sizeof` (30.5034) and `if(!conn)` (32.1079). These attention weights suggest the models have developed a comprehensive understanding of code elements related to security flaws from the datasets.

Buggy Data: In contrast, the models trained on buggy data showed their inability to focus on tokens crucial for vulnerability detection. In BigVul, the model assigns significant weights to non-vulnerable tokens or variables such as `int` (37.6882), `rt` (36.5832), and `nl` (41.7406). Similarly, in Devign, the model focuses its attention on various elements like `static` (50.7911), `CCIDBus` (55.7003), and `Device` (43.4922). In other words, the model fails to prioritize the tokens that might signal potential vulnerabilities, such as unsafe operations or improper error handling. Moreover, the models trained on data with label noise pay considerable attention to non-informative tokens, such as newline characters (26.1465 in BigVul, 54.1878 in Devign) and empty tokens (8.2194 in BigVul, 10.1193 in Devign). This diffused attention pattern, evident across both datasets, suggests that these models struggle to distinguish between benign code patterns and those that may introduce vulnerabilities. This limits their effectiveness in identifying security-critical code sections.

Summary of RQ1: Models trained on code-based datasets containing data bugs exhibit many issues, including inconsistent learning across layers, reduced model capacity and gradient instability. Furthermore, missing preprocessing can lead to slow convergence, incorrect learning patterns, and distorted distributions, impairing the models' ability to understand and process code effectively.

3.3.2 RQ2: How do data quality and preprocessing issues in text-based data affect the training behaviour of deep learning models?

Impact of Data Quality Issues

(a) Abnormal Weight Distribution: We observed abnormal layer weights in the models trained on low-quality text data. As shown in Table 3.5, 93.33% of the models trained on data with concept drift exhibited this phenomenon. Our analysis of these models revealed that 80-85% of their layers were affected, particularly the

convolutional and dense layers. These layers demonstrated unusually high weight variances (e.g., $\sigma^2 \approx 2.3$), significantly deviating from their baseline counterparts (e.g., mean weight variance $\sigma^2 \approx 0.64$). The observed abnormal weight distribution can be attributed to the *dynamic nature* of text data in software engineering. As software and technology evolve, the corresponding text-based data also changes, which causes shifts in the underlying data distribution. This evolution in software and text can affect how the text is represented, and the relevant weights and biases are learned by the models. As a consequence, older and newer data points might require different sets of optimal parameters. During model training, the text data with concept drift generate large gradient signals to push the parameter values towards the optimal settings for their respective time periods, which could have led to abnormal weight distribution in the models.

(b) Gradient Skewness: We also observed significant gradient skewness in the models trained on low-quality text data. As shown in Table 3.5, 76.67% of the models trained on data with concept drift and 56.67% with label noise exhibited this phenomenon. Our analysis revealed that 80-90% of the layers in these models were affected, particularly the convolutional layers and batch normalization layers. The gradient distribution of these layers demonstrated substantial skewness ($\gamma \approx 2.9$ for models trained with concept drift, and $\gamma \approx 1.2$ for models trained with label noise), far exceeding the baseline skewness values ($\gamma \approx 0.2$). As discussed above, concept drift in text-based data can lead to large gradient signals during training. This not only affects weight distribution but also could make the gradients skewed. Large gradient signals shift the overall distribution towards higher values, causing positive skewness. Similarly, label noise can also cause gradient skewness. In our datasets with label noise, similar data points have different labels due to incorrect labelling. When a model encounters these mislabeled examples during training, it tries to learn a decision boundary that separates these similar points with different labels, which is theoretically impossible. This leads to large prediction errors, as the model's output significantly differs from the (incorrectly assigned) target labels. These large errors, in turn, result in large gradient updates during backpropagation. As the training process continues, these inconsistent and large gradient updates accumulate, causing the overall gradient distribution to be skewed.

Table 3.5: Manifestations of data bugs in text-based models

Data Quality Issues	Abnormal Weight Distribution	Gradient Skewness	Overfitting
Label Noise	16.67%	56.67%	3.33%
Class Imbalance	36.67%	23.33%	86.66%
Concept Drift	93.33%	76.67%	43.66%

(c) Overfitting: We noticed overfitting in the models trained on low-quality text data. As shown in Table 3.5, 86.66% of models trained on data with class imbalance exhibited overfitting. Our analysis revealed that 60-65% of the layers in these models were affected, particularly the dense layers. The median bias value of these layers ($med \approx 1.1$) is much higher than their baseline counterparts ($med \approx 0.07$). When a dataset has a disproportionate number of samples in one class compared to the others, the model tends to become biased towards the majority class. As a result, the model's biases become skewed towards the majority class, which results in the model overfitting the training data.

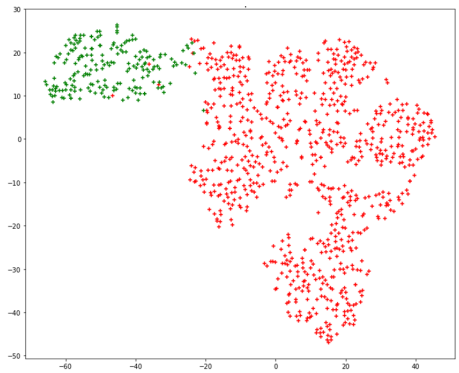
Impact of Missing Preprocessing Operations

(a) Numerical Instability: Models trained on raw text data showed clear signs of numerical instability. According to our analysis, 63.33% of the models exhibited this issue. From our analysis, we observed that 80-90% of their layers were affected, particularly in the embedding layers and convolutional layers. These layers showed extreme weight ranges (-2.16 to 2.54), significantly deviating from the typical ranges of -0.1 to 0.1 for weights observed in the models trained on the clean dataset. These extreme weight parameter values indicate potential instability in the models and can be traced to the lack of proper text preprocessing and tokenization [23]. Without these steps, the model encounters a high frequency of out-of-vocabulary words, including typos, rare technical terms, and inconsistent word forms (e.g., different tenses or misspellings of the same word). As the model attempts to learn representations for these out-of-vocabulary words, it often resorts to *extreme weight adjustments*, particularly in the embedding and convolutional layers, which are responsible for feature extraction. This leads to high volatility and significant variance in the layer weights, which results in exploding gradients and numerical instability in the neural networks [18, 47].

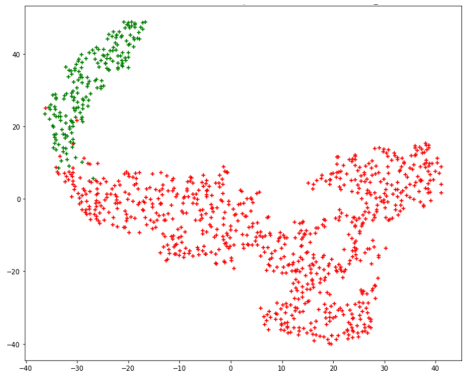
(b) Skewed Bias Distributions: Models trained on raw texts demonstrated significant skewness in their bias distributions. Our analysis revealed that 53.33% of the models exhibited this issue, with 50-55% of the layers affected, particularly the convolutional layers. These layers demonstrated high skewness (> 1) and moderate kurtosis ($|k| > 3$) in their bias values, which are significantly higher than their baseline counterparts (skewness = 0.34, kurtosis = 1.82), collected from the models trained on clean, preprocessed data. This phenomenon can be attributed to the imbalanced representation of the text data due to a lack of preprocessing operations. Without tokenization and stop word removal, certain stop words or phrases may appear disproportionately in the dataset. This imbalance directly affects the bias values in the model. As certain words or phrases occur more frequently, the neurons associated with these common patterns are activated more often. Consequently, the biases for these neurons receive more frequent updates during training, pushing them further from their initial values. In contrast, biases for neurons associated with less common words or patterns receive fewer updates. This disparity in update frequency leads to a spread in the bias values, with some shifting significantly and others remaining closer to their starting points. This results in a skewed distribution of bias values, which reflects the imbalanced nature of the input data.

Impact of Concept Drift on Decision Boundary

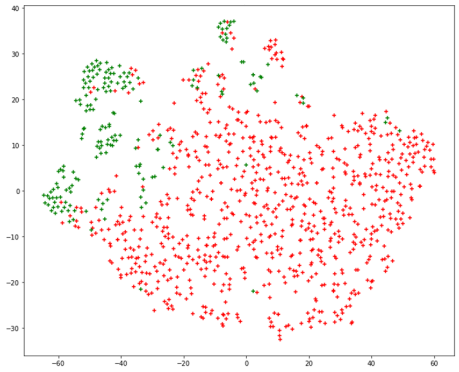
To assess the impact of data quality issues on deep learning models' training behaviour, we also performed a post-hoc analysis using t-SNE plots. First, we trained 30 models on clean datasets and 30 models on datasets containing concept drift. We selected concept drift for our qualitative analysis, as it affected the highest number of models (see Table. 3.5). Then, we generated the t-SNE plots for the dense layer representations, as shown in Fig. 3.2(a) and Fig. 3.2(c). We focus on dense layers since their decision boundaries capture complex, non-linear relationships between features and directly contribute to the final classification. Furthermore, existing literature [88] has analyzed the t-SNE plots of dense layers in their analysis. We discuss the outcome of our analysis below. From Fig. 3.2(a) and 3.2(c), we note the impact of concept drift on the dense layers of our analyzed deep learning models for duplicate bug report detection. In the t-SNE plots, the red points in the graph indicate the representation of



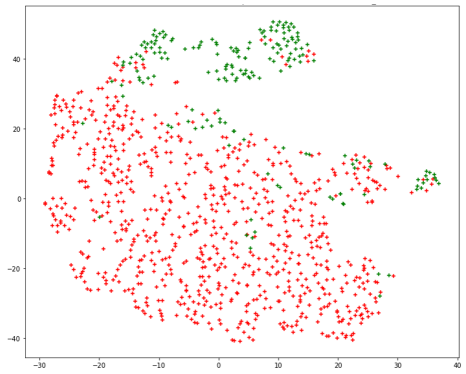
(a) First dense layer of models trained on bug-free data



(b) Second dense layer of models trained on bug-free data



(c) First dense layer of models trained on buggy data



(d) Second dense layer of models trained on buggy data

Figure 3.2: t-SNE plots for models trained on bug-free and buggy data

non-duplicate bug reports, and the green points indicate the representation of duplicate bug reports. Moreover, the axes of the graph represent the distance metrics that result from the t-SNE optimization process. In the first dense layer (Fig. 3.2(a)(a)), models trained on clean data show a clear separation between the two classes, indicating effective capture of the patterns. On the other hand, models trained on buggy data (Fig. 3.2(c)(a)) exhibit complex representations with ambiguous decision boundaries. This observation is supported by an analysis of KL Divergence, which serves as the cost function of the t-SNE plot. As the cost function, KL Divergence is automatically calculated and optimized during the generation of these visualizations, with lower KL Divergence values indicating a better representation of the high-dimensional data in the 2D space. For the first dense layer, the mean KL divergence for buggy models (0.32) is higher than that of bug-free models (0.26), indicating more noise in the layer representation of models affected by concept drift. The second dense layer (Fig.3.2(b)(b) and Fig.3.2(d)(b)) demonstrates similar trends, with bug-free models showing well-defined decision boundaries and buggy data models displaying unclear decision boundaries and fragmented clusters. This trend is further corroborated by the KL divergence analysis, where buggy models exhibit a mean KL divergence of 0.24 compared to 0.14 for bug-free models in the second dense layer. The increase in KL divergence for buggy models across both layers (23% in the first layer and 71% in the second layer) underscores the concept drift's impact on the network. It also highlights how concept drift progressively affects a model's ability to learn meaningful representations and make accurate predictions, ultimately impairing its performance in duplicate bug report detection.

Summary of RQ2: Quality issues in text-based data introduce many challenges for deep learning models, including abnormal weight distribution, gradient skewness, and overfitting. Moreover, missing preprocessing techniques can lead to numerical instability and skewed bias distributions, worsening a models' ability to learn effectively from the bug reports.

Table 3.6: Manifestations of data bugs in metric-based models

Data Quality Issues	Sparse Parameter Updates	Vanishing Gradients	Poor Optimization
Label Noise	33.33%	36.67%	23.33%
Class Imbalance	76.67%	83.33%	66.67%
Concept Drift	13.33%	16.67%	10.00%

3.3.3 RQ3: How do data quality and preprocessing issues in metric-based data affect the training behaviour of deep learning models?

Impact of Data Quality

(a) Sparse Parameter Updates: We observed sparse updates to the parameters of neural networks trained on low-quality metric data. As shown in Table 3.6, 76.67% of models trained with class imbalance exhibited this phenomenon. Our analysis revealed that 50-55% of their layers were affected, particularly the fully connected layers. In these layers, up to 85% of the neurons showed no change in their gradients. That is, 85% of the neurons in a particular layer have not received any updates and have effectively stopped learning. This phenomenon can be explained by the *disproportionate representation of classes* in imbalanced datasets. In such datasets, the model is exposed to significantly more examples from the majority classes, causing neurons associated with these classes to receive frequent and substantial gradient updates. Conversely, neurons detecting minority classes receive updates far less frequently and with smaller magnitudes. As a consequence, a significant portion of neurons, particularly those responsible for minority class features, become 'dead' and do not contribute to the learning process. Thus, the model struggles to learn representations for minority classes, leading to poor performance in these underrepresented categories.

(b) Vanishing Gradients: We also observed the problem of vanishing gradients in models trained on imbalanced datasets. As shown in Table 3.6, 83.33% of the models trained on imbalanced datasets exhibited this phenomenon. In these models, gradient magnitudes decreased exponentially from the output layer towards the input layer. Similarly, models trained on corrupted data experienced gradient drops

by a factor of 10^6 between the output and input layers. In contrast, models trained on clean data showed a moderate drop in gradients, with only a 10^2 factor decrease from output to input. As discussed above, class imbalance leads to sparse parameter updates, which in turn contributes to the vanishing gradient problem. As neurons associated with minority classes receive infrequent and smaller updates, their gradient signals become progressively weaker as they propagate backwards from the output layer through the network. Consequently, the early layers of the network (e.g., input layers), crucial for learning fundamental features, receive negligible updates for minority class patterns and suffer from vanishing gradients. This phenomenon severely impairs a model's ability to learn from minority classes, further exacerbating the issues caused by class imbalance.

(c) Poor Optimization: We also noticed poor optimization in the models trained on low-quality metric data. As shown in Table 3.6, 66.67% of models trained on datasets with class imbalance suffered from this issue. Our analysis revealed that the models trained on low-quality data led to a stagnation of loss values after the first epoch. This contrasts sharply with models trained on balanced datasets, which exhibited steady and consistent decreases in loss throughout training. The average training and validation loss for models trained on imbalanced datasets was approximately 1.5 times higher than those trained on clean datasets. This trend was consistent across different software projects. For instance, in the Openstack project, the training loss value increased from ≈75 for clean data to ≈115 for buggy data. Similarly, in the QT project, the training loss value rose from ≈110 for clean data to ≈170 for buggy data. When a dataset has a class imbalance, the model's corresponding optimization process is severely affected. The stagnation in loss values, coupled with the vanishing gradient problem, suggests that the model might get stuck in local minima. As a result, the model struggles to learn representations for minority classes or refine its decision boundaries, leading to poor performance.

Impact of Missing Preprocessing Operations

(a) Exploding Gradients: Models trained on raw numerical data showed signs of exploding gradients. According to our analysis, 66.67% of the models suffered from this issue. We observed that 70-80% of their layers were affected, particularly

in the output layers of the network. These layers showed extreme gradient values (e.g., from -4.23 to 2.76), significantly deviating from the typical gradient ranges of -1 to 1 observed in the models trained on properly scaled datasets. These extreme gradient values can be explained by the lack of appropriate feature scaling and normalization [46]. Without these preprocessing steps, the model encounters features with vastly different scales, leading to disproportionate weight updates. This leads to high volatility and significant variance in the gradient magnitudes, which results in exploding gradients and potential divergence from the normal behaviours in the neural networks [100, 47].

(b) High Variance in Weight Distribution: Models trained on unprocessed numerical data demonstrated significant variance in weight distribution. Our analysis revealed that 53.33% of the models exhibited this issue, with 85-90% of their layers affected, particularly the input and early hidden layers. These layers demonstrated high variance in weight magnitudes ($\sigma^2 \approx 5.76$), which is significantly higher than their baseline counterparts ($\sigma^2 \approx 0.89$) collected from the models trained on normalized data. This high variance in weight distribution stems from the interaction within a neural network and the diverse scales inherent in software metrics. For example, lines of code might be in thousands, while cyclomatic complexity might be in single digits. During training, the neural network compensates for these scale disparities to ensure a fair representation of all features in the learning process [67]. It assigns larger weights to small-scale features (e.g., cyclomatic complexity) to amplify their effect and smaller weights to large-scale features (e.g., lines of code) to prevent them from dominating the output. This compensation mechanism results in a wide range of weight values. The input and early hidden layers are particularly susceptible to this issue as they directly interact with the raw feature values [67], contributing to the observed weight variance issue.

Impact of Class Imbalance on Feature Importance

To analyze the impact of class imbalance on the buggy and bug-free models, we apply the GradCAM (Gradient-weighted Class Activation Mapping) technique [114]. This approach has been used in previous studies to understand the effectiveness and efficacy of defect prediction models [148]. For our GradCAM analysis, we used datasets

specifically designed for defect prediction, focusing on software metrics. Our dataset exhibited an imbalanced distribution, with the minor class comprising non-defective instances and the major class consisting of defective instances. By examining the GradCAM outputs, we investigate which metrics the models prioritize and how this focus differs between the two types of models trained on two distinct codebases: Openstack and QT.

Buggy Models: Our analysis of the GradCAM output for the buggy models reveals a disproportionate focus on obscure or overly specific tokens across both codebases. In the OpenStack model, tokens like 'audit_location_generator' (0.9987) and 'i62ce43a330d7ae94eda4' (0.9962) have probabilities very close to 1, while more general programming concepts like 'if' (0.0023) and 'return' (0.0041) have probabilities near 0. Similarly, the Qt model exhibits high probabilities for tokens such as 'Q_OBJECT_FAKE' (0.9976) and 'qAsConst' (0.9953), while potentially bug-indicative tokens like 'nullptr' (0.0037) or 'delete' (0.0052) have very low probabilities. This pattern suggests that the severe class imbalance has likely led to overfitting, causing the models to associate rare, noise tokens, or codebase-specific terms with software defects in the training data. Consequently, these buggy models may fail to generalize and capture the broader semantics and context of software defects across different codebases.

Bug-free Models: In contrast, according to GradCAM output, the bug-free models focus on a wider variety of relevant technical terms and phrases in both OpenStack and Qt codebases. For OpenStack, tokens such as 'vendor-data' (0.9871), 'xenapinfs-glance-integration' (0.9923), 'live_migr-ate' (0.9905), 'deprecationwarning' (0.9889), 'regressions' (0.9917), and 'backward' (0.9894) have probabilities close to 1. These terms intuitively relate to areas where bugs occurred in OpenStack code, such as migrations, deprecations, regressions, and vendor integrations. Similarly, for Qt, we observe a focus on both framework-specific and general programming concepts. Tokens like 'QObject' (0.9934), 'connect' (0.9912), 'emit' (0.9901), 'virtual' (0.9887), and 'override' (0.9923) have probabilities close to 1. Additionally, memory-related terms such as 'new' (0.9876), 'delete' (0.9908), and 'shared_ptr' (0.9892) also show high probabilities, reflecting the importance of memory management in C++ code.

This observation suggests that the bug-free models learn to focus on both codebase-specific concepts and general programming patterns that might have historically been associated with software defects. By mitigating the data quality issues, particularly class imbalance, these models have developed more generalizable representations that meaningfully capture the semantics of software defects across different codebases and frameworks.

The stark contrast in token focus between the buggy and bug-free models highlights the significant impact of class imbalance on model behavior. While the buggy models emphasize seemingly arbitrary or overly specific tokens to learn about software defects, the bug-free models focus on relevant technical concepts and terms.

Summary of RQ3: The presence of data quality issues in metric-based data introduces several challenges for deep learning models, including sparse parameter updates, vanishing gradients, and poor optimization. Moreover, inadequate preprocessing techniques can lead to exploding gradients and high variance in weight distribution, further hindering the models' ability to learn from the data effectively.

3.3.4 RQ4: How well do our findings on data quality and preprocessing issues generalize to other code-based, text-based, and metric-based datasets?

To evaluate the generalizability of our findings, we conducted experiments on six additional datasets: D2A and Juliet (code-based), Spark and Mozilla (text-based), and Go and JDT (metric-based). We present our findings below.

In code-based datasets, we found that 83.33% of models (25 out of 30) exhibited inconsistent learning, 86.67% (26 out of 30) showed reduced model capacity, and 76.67% (23 out of 30) experienced gradient instability when trained on data with label noise or concept drift. For text-based datasets, 93.33% of models (28 out of 30) showed abnormal weight distribution, 76.67% (23 out of 30) demonstrated gradient skewness, and 86.67% (26 out of 30) exhibited overfitting when trained on data with concept drift or class imbalance. In metric-based datasets, 76.67% of models (23 out of 30) showed sparse parameter updates, 83.33% (26 out of 30) experienced vanishing

gradients, and 73.33% (22 out of 30) demonstrated poor optimization when trained on data with label noise or class imbalance. When models were trained without appropriate preprocessing of data, we also observed similar patterns across all dataset types. In code-based datasets, 66.67% of models (20 out of 30) trained without proper preprocessing showed slow convergence, and 56.67% (17 out of 30) exhibited incorrect learning. For text-based data, 66.67% of models (20 out of 30) trained on unprocessed data showed numerical instability, while 56.67% (17 out of 30) exhibited skewed bias distributions. In metric-based datasets, 73.33% of models (22 out of 30) trained without preprocessing demonstrated exploding gradients, and 56.67% (17 out of 30) showed high variance in weight distribution. All percentages observed in these new datasets align closely with our original findings, varying within a 1-5% margin. These consistencies in observations across diverse datasets underscore the generalizability of our findings, highlighting the persistent impact of data quality issues and the importance of proper preprocessing in model training for software engineering tasks.

Model's Behaviour on Clean Datasets

After addressing data quality issues and retraining the above models, we observed significant improvements in model behaviour and performance across all types of datasets: code-based, text-based, and metric-based. The impacts of data bugs were significantly diminished, and the models exhibited stable and normal behaviours during their training.

In code-based datasets, we found only 3.33% of models (1 out of 30) showing signs of inconsistent learning, compared to 83.37% previously. Model capacity improved substantially, with only 6.67% of models (2 out of 30) failing to capture complex patterns. Gradient stability also increased, with only 10.00% of models (3 out of 30) demonstrating instable gradients, down from 76.67% in the presence of data quality issues.

For text-based datasets, the chance of models having abnormal weight distribution decreased dramatically, with only 6.67% of models (2 out of 30) showing this behavior, compared to 93.33% before. Gradient skewness was largely prevented, with only 10.00% of models (3 out of 30) demonstrating skewed gradient distributions. Overfitting was significantly reduced, with only 6.67% of models (2 out of 30) showing this

issue, down from 86.67% previously.

In metric-based datasets, sparse parameter updates were nearly prevented, with only 6.67% of models (2 out of 30) showing any signs of this issue. The vanishing gradient problem was significantly mitigated, with only 10.00% of models (3 out of 30) demonstrating this issue. Poor optimization was largely addressed, with only 13.33% of models (4 out of 30) showing this problem, compared to 73.33% previously.

The preprocessing issues, once addressed, also led to substantial improvements across all dataset types. In code-based datasets, only 6.67% of models (2 out of 30) showed slow convergence or incorrect learning patterns. For text-based data, only 10.00% of models (3 out of 30) demonstrated numerical instability or skewed bias distributions. In metric-based datasets, only 13.33% of models (4 out of 30) had exploding gradients or high variance in weight distribution.

3.3.5 Statistical Signifiance Tests

To assess the generalizability of our findings, we performed statistical significance tests with the following hypotheses.

$$H_0 : \text{The presence of data quality issues and model symptoms are independent}$$
$$H_1 : \text{The presence of data quality issues and model symptoms are dependent}$$

We selected McNemar's test [102] for the statistical tests due to our experimental design and paired nominal data, as it specifically evaluates changes in binary outcomes (presence/absence of symptoms) within the same models. While McNemar's test establishes statistical significance to test our null hypothesis that data quality issues and model symptoms are independent, the odds ratio measures the strength of their dependence [124]. In our analysis, an odds ratio of 1 would support our null hypothesis, which indicates that there is no association between data quality issues and model symptoms. Conversely, odds ratios greater than 1 would support our alternative hypothesis, suggesting that the presence of data quality issues increases the likelihood of observed symptoms. Moreover, odds ratios below 1 would indicate that data quality issues decrease the likelihood of model symptoms.

McNemar's test results consistently rejected the null hypothesis across all symptoms ($p < 0.01$). The odds ratios, measuring the strength of association between data quality issues and symptoms, were substantially greater than 1 in all cases, ranging from 15.53 for abnormal weight distribution to 806.10 for gradient skewness, strongly supporting our alternative hypothesis. The most pronounced dependencies were observed in inconsistent learning, gradient instability, and overfitting ($p < 0.0001$), indicating that these symptoms were exclusively present when data quality issues existed. Even the weakest association, found in the symptom "high variance in weight distribution" ($p = 0.0098$), demonstrated a statistically significant association between the symptom and data quality issues.

These results demonstrate that addressing data quality issues and applying appropriate proper preprocessing operations to training data can significantly improve model training for various software engineering tasks. The consistent improvements observed across different dataset types further validate the generalizability of our findings and underscore the critical importance of data quality and preprocessing in machine learning applications for software engineering tasks.

Summary of RQ4: Our findings on data quality and preprocessing issues generalize consistently across diverse software engineering datasets, including D2A and Juliet (code-based), Spark and Mozilla (text-based), and Go and JDT (metric-based). The repeated emergence of similar patterns and enhancements after addressing the data quality issues across multiple datasets increases confidence in our study findings.

3.4 Implications for Bug Reproduction

Our study on how data quality issues manifest during model training provides several actionable insights for verifying bug reproduction in deep learning systems:

Training Behavior Validation: Our findings reveal specific manifestation patterns that can serve as verification criteria during bug reproduction attempts. By monitoring gradient flows, weight distributions, and learning stability during training, practitioners can validate whether their reproduction exhibits the same behavioural

patterns as the original bug. These patterns provide concrete, measurable indicators to confirm the successful reproduction of data quality issues.

Model State Analysis: The manifestation of data quality issues can be verified by systematically analysing model states during training. Key indicators include changes in model capacity, abnormal weight evolution patterns, and unexpected optimization behaviour. By tracking these model state characteristics, practitioners can determine whether their reproduction accurately reflects the original data quality problems.

Automated Verification Systems: Our identified manifestation patterns enable the development of automated monitoring systems for bug reproduction verification. These systems can track critical training indicators such as gradient stability, feature representations, and learning dynamics. When implemented as part of the reproduction workflow, these monitoring systems objectively validate whether a reproduced bug matches the characteristics of the original issue.

3.5 Related Work

Data quality is crucial for the performance and reliability of deep learning models in software engineering. Previous studies have investigated the effects of data quality issues like label noise [40, 39, 98, 33], class imbalance [69, 76], data duplication [151, 33, 78], and concept drift [151, 65, 113] on deep learning models.

Recent studies have also analyzed the impact of data quality issues on deep learning models for software engineering tasks. Wu et al. [140] investigated mislabeled instances in five publicly available datasets commonly used for security bug report prediction, including Chromium, Ambari, Camel, Derby, and Wicket, where they identified 749 security bug reports incorrectly labelled as non-security. They also reported significant performance improvements for a retrained model that was trained on correctly labelled data. Tantithamthavorn et al. [127] analyzed over 3,900 issue reports from Apache projects and demonstrated that label noise can significantly impact the recall of a model. They also reported a significant improvement in the recall ($\approx$60%) for a retrained model on cleaned datasets. Kim et al. [62] evaluated the impact of intentionally injected noise into the datasets for defect prediction models

through controlled experiments. They found that when ≈30% of the data was mislabeled, the performance of defect prediction models decreased significantly, which highlights the models' sensitivity to label noise. Fan et al. [38] investigated mislabeled changes in just-in-time defect prediction datasets and found that certain labelling approaches can lead to performance reduction of up to 5%. Xu et al. [141] leveraged adversarial learning to improve the data quality of obsolete comment datasets, which led to an improvement in the accuracy of multiple existing models by 18.1%. Croft et al. [33] conducted a systematic analysis of data quality in software vulnerability datasets and found out that 71% of the labels are incorrect and 17-99% of data points are duplicated across four state-of-the-art datasets. Furthermore, they also analyzed the impact of the data quality issues on vulnerability detection. They found that the model's performance dropped by 65% when trained on clean data, which shows how the duplication and mislabelling of the data points can lead to inflated results.

In summary, the existing studies have examined the impact of data quality on model performance. However, a clear understanding of how these issues affect the training behaviours of a DL model remains limited. Most existing studies focus on specific data types or individual quality issues, preventing a comprehensive understanding of how data quality impacts deep learning models. Besides, how missing preprocessing operations can affect model behaviour is not well understood to date. Our study addresses these gaps in the literature through a systematic analysis of 900 models and 12 datasets targeting software engineering tasks.

Our work investigates how data quality and preprocessing issues affect the training behaviours of deep learning models. We adopt a comprehensive approach by examining three major issues - label noise, class imbalance, and concept drift - across software engineering data in code, text, and metric formats. We also demonstrate the generalizability of our findings through the reproduction of our findings on separate datasets, which were not used in our primary analysis. This multifaceted approach informs us of the impact of data bugs on deep learning models targeting software engineering tasks, addresses the gaps in the existing literature and encourages future efforts for appropriate debugging solutions.

3.6 Threats to Validity

One potential threat to the *internal* validity of our findings is the choice of baseline models and datasets. Although we followed systematic criteria for selection (Section 3.1, Section 3.2), there may be other relevant models or datasets not considered. We mitigate this threat by carefully documenting our selection criteria and validating our findings across diverse datasets within each data type category (code-based, text-based, and metric-based). Another threat is the potential for errors or biases in our analysis techniques. We mitigate this threat by using analysis techniques which are well-established in the literature and triangulating our findings through post-hoc analysis (using XAI techniques) and quantitative analysis (using gradients, weights, and biases). Another threat is the subjective nature of our post-hoc analysis, particularly in interpreting attention weights for code comprehension, t-SNE visualizations, and GradCAM results. We mitigated this by following established guidelines for analysis from previous studies [148, 42, 88].

The main threat to *external* validity is the generalizability of our findings to software engineering tasks or data types that were not those considered. Our analysis is based on three tasks – vulnerability prediction, defect prediction and duplicate bug report detection. The impact of data quality and preprocessing issues may manifest differently in other tasks or application domains. Additionally, we focused on specific types of data quality issues, which may limit the generalizability of our findings. To mitigate these threats, we selected prevalent tasks, data types, and data quality issues based on comprehensive surveys and prior studies [144, 151, 33, 69, 76] and used established benchmark datasets from different projects and domains.

The primary threat to *conclusion* validity comes from the inherent stochasticity in deep learning model behaviours. To account for this, we run each model multiple times and collectively analyze our findings in an aggregate manner across different data types and model behaviours, examining patterns in training dynamics (through gradients, weights, and biases). Our conclusions are drawn from these consistent, aggregate patterns observed in model behaviours when encountering data quality issues.

The primary threat to *conclusion* validity is the potential for statistical errors or violations of assumptions in our quantitative analysis. While we followed best

practices and ran multiple trials to account for stochasticity, there may be inherent limitations or biases that could affect the accuracy or generalizability of our conclusions. Additionally, our conclusions are based on specific experiments and analysis techniques, and other approaches or methodologies could yield different or complementary insights.

3.7 Summary

Data bugs in deep learning systems present a significant challenge, as their hidden nature and implicit effects make them difficult to identify and reproduce. While existing studies have examined data quality issues, they have primarily focused on raw training data, overlooking the critical aspects of data bugs (e.g., preprocessing errors). To address these gaps, we conducted an extensive investigation into how data quality and preprocessing errors manifest during the training of deep learning models across various software engineering tasks. Our analysis captures data bugs from three prevalent data types: code-based, text-based, and metric-based data, and examines their manifestation patterns in vulnerability detection, duplicate bug report detection, and defect prediction tasks. Through a systematic methodology combining state-of-the-art baselines and explainable AI techniques (e.g., GradCAM), we identified distinct symptoms of data bugs for each type of data: gradient instability and biased learning in code-based data, abnormal weight distributions and overfitting in text-based data, and vanishing gradients with poor optimization in metric-based data.

Our findings not only advance the current state of knowledge but also have significant implications for deep learning researchers and software practitioners. While an understanding of how bugs manifest is useful, they also need to be reproduced before correction, which could be highly challenging. Thus, Chapter 4 focuses on enhancing the reproducibility of deep learning bugs through an empirical study.

Chapter 4

Towards Enhancing the Reproducibility of Deep Learning Bugs: An Empirical Study

Bug reproduction verifies the presence or absence of a reported bug in a software system. Hence, deep learning bugs must be reproduced before their correction. In this chapter, we investigate the challenges of reproducing deep learning bugs. Existing studies suggest that only 3% of deep learning bugs are reproducible, indicating difficulty in their reproduction. However, not many studies were conducted to reproduce the deep learning bugs or to understand their reproducibility challenges. This chapter addresses those gaps by conducting an empirical study and designing effective guidelines to enhance the reproducibility of deep learning bugs.

The rest of the chapter is organized as follows: Section 4.1 provides an overview of the research problem and our conducted study. Section 4.2 presents a motivating example that highlights these challenges. Section 4.3 describes our study methodology. Section 4.4 presents our study findings, and Section 4.5 discusses key findings and actionable insights. Section 4.6 discusses threats to the validity of our study, and Section 4.7 reviews related literature. Finally, Section 4.8 concludes the chapter.

4.1 Introduction

Deep learning (hereby DL) has been widely used in many application domains, including natural language processing, finance, cybersecurity [116, 7, 20], autonomous vehicles [101, 79], and healthcare systems [36].

Like any software system, deep learning systems are prone to bugs. Bugs in deep learning systems could arise from various sources, including errors in the dataset, incorrect hyperparameters, and incorrect structure of the deep learning model [21]. These bugs could lead to program failures, poor performance, or incorrect functionality, as reported by existing literature [57]. They can also lead to serious consequences, as shown by the fatal crash involving Uber's self-driving car [135]. Thus, we must

fix the bugs before deploying a deep learning model in production. However, one must reproduce the bugs before fixing them to verify their presence/absence in the system. Unfortunately, reproducing DL bugs is challenging due to deep learning systems' multifaceted nature and dependencies, which encompass data, hardware, libraries, frameworks, and client programs. Furthermore, deep learning systems are inherently non-deterministic (i.e., random weight initialization), which leads to different outcomes across multiple runs and thus makes the reproduction of deep learning bugs challenging [94]. Moreover, they also suffer from a lack of interpretability [64], which makes the reproduction of deep learning bugs challenging in comparison to traditional software bugs. According to existing investigations [93], only 3% of their analyzed deep learning bugs were reproducible, further demonstrating the challenges in reproducing DL bugs.

Existing literature investigates the challenges of reproducing programming errors or bugs from various sources. Mondal et al. [89] investigate the challenges in reproducing programming issues reported on Stack Overflow and suggest several edit actions to help reproduce them. Rahman et al. [109] conduct a multi-modal study to understand the factors behind the non-reproducibility of software bugs. They identify 11 significant factors behind bug non-reproducibility, including missing information, bug duplication, false positive bugs, and intermittency. Overall, these studies highlight the challenges in reproducing traditional software bugs but do not deal with any deep learning bugs, which warrants further investigation.

Recently, a few studies have attempted to tackle the challenges of deep learning bugs. Liang et al. [71] provide a dataset of 64 deep learning bugs collected from GitHub issues. They classify these bugs into six categories according to the taxonomy of Humbatova et al. [54]. Moravati et al. [93] constructed a benchmark dataset containing 100 deep learning bugs collected from StackOverflow and GitHub; they reproduced each of them. Although these studies offer benchmark datasets containing reproducible bugs, their primary focus is dataset construction. They do not report any detailed instructions (e.g., sequence of actions) essential to the reproduction of the deep learning bugs. Our work attempts to fill this important gap in the literature.

In this chapter, we conduct an empirical study to better understand the challenges in reproducing deep learning bugs. First, we collect a total of 568 DL bugs from Stack

Overflow posts. Then, we extend them with 100 DL bugs from the benchmark dataset of Moravati et al. [93], which makes up our final dataset of 668 DL bugs. Using the taxonomy of Humbatova et al. [93], we divide these bugs into five categories: 167 model, 213 tensor, 145 training, 113 GPU, and 30 API bugs. Second, by using stratified sampling, we select 165 of these bugs and determine their reproducibility status by attempting to reproduce them using the code snippet and complementary information from Stack Overflow and the benchmark dataset. Third, we manually analyzed various artefacts such as bug reports, reproduction scripts, follow-up questions, and thread discussions to determine the categories of information that are useful for reproducing deep learning bugs. Fourth, we use the information gathered during the bug reproduction and create a dataset of transactions that link the type of bug with edit actions and component information. Then, our study establishes connections among the component information, edit actions, and the type of bugs by employing the Apriori algorithm [8]. Finally, to further validate our findings, we conducted a developer study with 22 participants from industry and academia. Half of the participants were asked to reproduce bugs using our identified information, while the other half were asked to reproduce the same bugs without access to that information. Results from the user study show that our recommended edit actions and component information can reduce the time to reproduce the bug by 24.35%. Thus, we answer three important research questions as follows:

- **RQ1: Which edit actions are useful for reproducing deep learning bugs?**

  Determining the key edit actions that are crucial for reproducing deep learning bugs is important since almost none of the bugs can be reproduced using the verbatim code snippet. By manually reproducing 148 deep learning bugs, we identify ten key edit actions that could be useful to reproduce deep learning bugs (e.g., input data generation, neural network construction, hyperparameter initialization).

- **RQ2: What types of component information and edit actions are useful for reproducing specific types of deep learning bugs?**

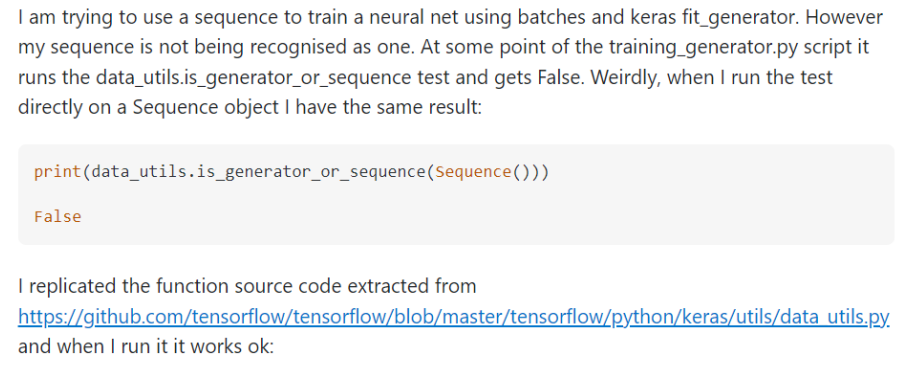
  Different types of DL bugs need different types of information or edit actions, which warrants further investigation. Using the Apriori algorithm, we have

determined the top 3 pieces of information and the top 5 edit actions that can help one reproduce each type of bug. These insights can be used not only to detect the missing information in a submitted bug report but also to formulate the follow-up questions soliciting the missing information.

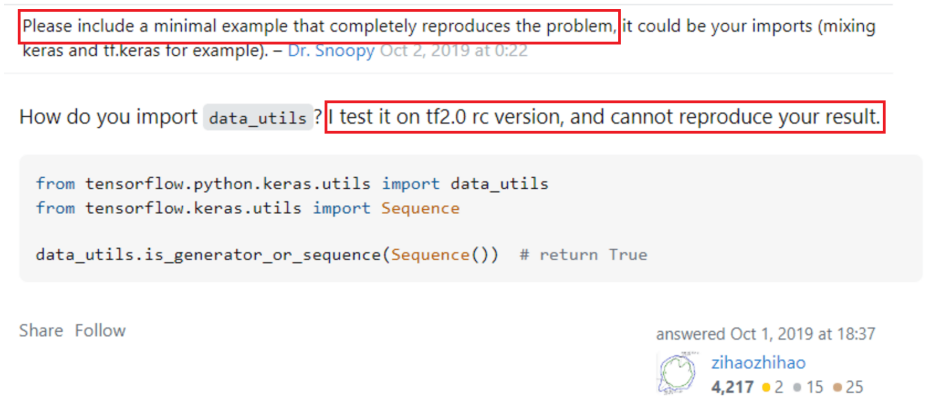
- **RQ3: How do the suggested edit actions and information affect the reproducibility of deep learning bugs?**
  To assess the effectiveness of our suggested edit actions and component information in improving bug reproducibility, we conducted a user study involving 22 professional developers. In our developer study, the participants assigned to the experimental group used our suggested edit actions and component information to reproduce deep learning bugs. In contrast, the participants in the control group reproduced the bugs without access to the suggested edit actions and component information. We found that our recommended edit actions and component information (a) helped the developers reproduce 22.92% more bugs and (b) decreased their time to reproduce the deep learning bugs by 24.35%.

4.2 Motivating Example

Bug reports or programming Q&A posts might not always provide sufficient information to reproduce a deep learning bug. Let us consider the example question shown in Fig. 4.1(a) [6]. Here, the user attempts to train a neural network using Keras. The input data type is a *Sequence* object, which is used as the base object for fitting a sequence of data, such as a dataset [128] The user aims to pass the training dataset as a *Sequence* object to the $fit\_generator()$ method. However, s/he discovers that their *Sequence* object is not recognized by the Keras library. The user also provides a code snippet to aid the reproducibility of the bug. Unfortunately, the issue cannot be reproduced since the provided information does not contain the required dependencies and imports. In Stack Overflow, this question has failed to receive a precise response. Even though the above code (Fig. 4.1(a)) could be made compilable or executable using edit actions, the bug cannot be reproduced due to its complex nature. Since the earlier study [89] does not deal with any deep learning bugs, their suggested edits might also not be effective.



(a) Bug reported by the Stack Overflow User



(b) Answers highlighting the non-reproducibility of the bug

Figure 4.1: An irreproducible bug from Stack Overflow

Let us consider another example question shown in Fig. 4.2(a) [5]. Here, the user wants to obtain the confusion matrix from a multi-class classification model. Unfortunately, s/he runs into a runtime error, as the $confusion\_matrix()$ method does not support multiple output labels. To reproduce this bug, we first generate a synthetic multi-class dataset since the training data is missing from the question. We also add the required import statement for the 'confusion_matrix' function. Then, we initialize the model hyperparameters based on the code snippet. When we run the code on this synthetic data, it triggers the same runtime error due to the wrong data shape being passed to 'confusion_matrix()'. Thus, by extracting the key information and applying the necessary edit actions, we were able to reproduce the bug. Similarly, this issue was reproduced by other users of Stack Overflow, and the question received a correct solution within two hours of its submission, as shown in Fig. 4.2(b). To summarize,

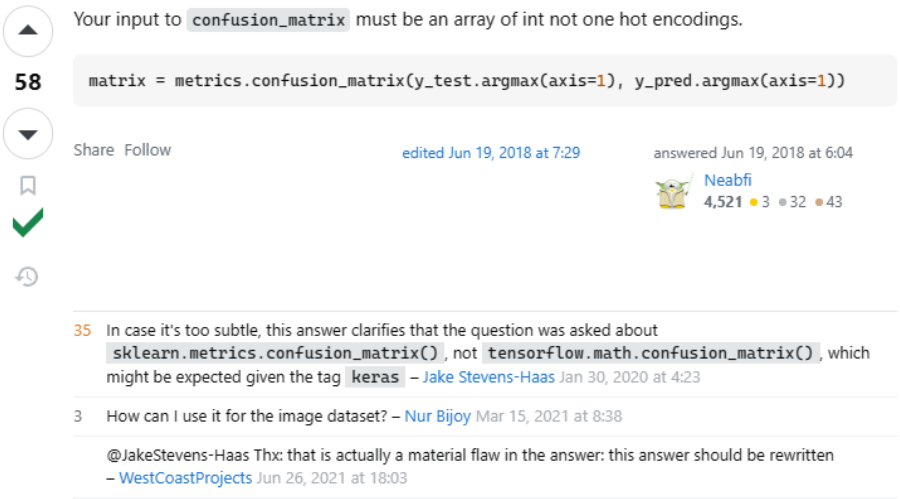
I did the following to get the confusion matrix.

```
y_pred = (predictions > 0.5)

confusion_matrix(y_test, y_pred)
Traceback (most recent call last):

  File "<ipython-input-38-430e012b2078>", line 1, in <module>
    confusion_matrix(y_test, y_pred)

  File "/Users/abrahammathew/anaconda3/lib/python3.6/site-packages/sklearn/metrics/c
    raise ValueError("%s is not supported" % y_type)

ValueError: multilabel-indicator is not supported
```

However, I am getting the above error.

How can I get a confusion matrix when doing a multiclass neural network in Keras?

(a) Bug reported by the Stack Overflow User



(b) Accepted Answer

Figure 4.2: A reproducible bug from Stack Overflow

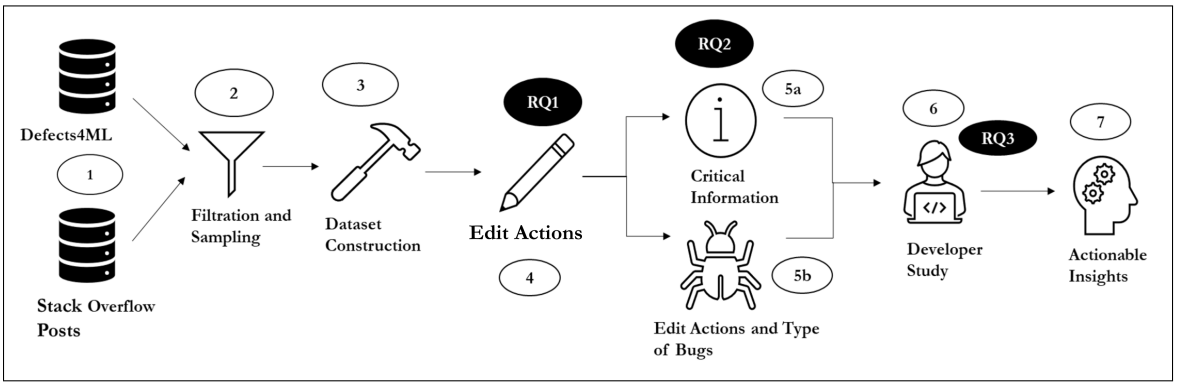


Figure 4.3: Schematic diagram of our empirical study

by generating a synthetic dataset, adding the necessary import statement, and initializing the model hyperparameters, we can reproduce the example deep learning bug. Our study in this article proposes the methodology to extract the information and determine the edit actions necessary for systematically reproducing deep learning bugs.

4.3 Study Methodology

Fig. 4.3 shows the schematic diagram of our empirical study. We discuss different steps of our study as follows.

4.3.1 Selection of Data Sources

We select Stack Overflow as a primary data source for our study. It is the largest programming Q&A site for programming topics containing over 24 million questions and 35 million answers [121]. Thus, Stack Overflow could be a potential source for deep learning (DL) bugs. Developers often submit their encountered problems on Stack Overflow, when building their deep learning applications. According to a recent work [154], Stack Overflow contains at least 30 topics and 80K posts related to deep learning issues, which makes it a potential source for our data. We also select the benchmark dataset of Moravati et al. [93] (Defects4ML in Fig. 4.3) containing 100 DL bugs as our second source of data. Specifically, we use 63 GitHub issues and 37 Stack Overflow posts from the Defects4ML dataset in our study.

To gather relevant posts from Stack Overflow, we use the Stack Exchange Data

Explorer platform[1]. We employ multiple filters to capture relevant posts discussing deep learning bugs. First, we select the recent posts (i.e., submitted between May 2020 and May 2023) with the following tags: 'tensorflow', 'keras', and 'pytorch'. These tags were selected since they represent the most frequently used frameworks for deep learning [150]. This filtration resulted in 14,065 posts for PyTorch, 14,971 posts for Tensorflow, and 3,152 posts for Keras. We then applied the following four filtration criteria to remove conceptual questions from our collection:

- **Keyword Filtering**: This filtration removes the *how-to* questions and the questions requesting installation instructions. To filter these questions, we use appropriate keywords ('how', 'install', and 'build'), as recommended by Humbatova et al. [54].

- **Code Snippet**: Given our focus on the reproducibility of reported issues, the questions must include code segments. Therefore, we consider such questions that contain at least one line of code, as suggested by Mondal et al. [89].

- **Accepted Answer**: We only select the questions with accepted answers, ensuring that each reported issue (e.g., DL bug) was reproduced and fixed, as advised by Humbatova et al. [54].

- **Negative Question Score**: This filtration helps us discard questions that the community has found to be of low quality, as proposed by Ponzanelli et al. [104].

We combine all four filtration criteria above, construct a SQL query, and then execute the query against the Stack Exchange Data Explorer [122]. The SQL query is available in our replication package [115]. After this filtration step, we obtained a total of 279 posts for Keras, 1,433 posts for Tensorflow, and 1,700 posts for PyTorch. To ensure the validity of our chosen filtration criteria and confirm that the posts are related to DL bugs, we conducted a follow-up manual analysis on a representative sample of the filtered posts. We determined the appropriate sample size using Cochran's sample size formula [30]:

$$n = \frac{Z^2 \cdot p \cdot (1-p)}{e^2}$$

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

Where:

- n = sample size
- Z = Z-value for the desired confidence level (1.645 for 90% confidence)
- p = population proportion (assumed to be 0.5 for maximum variability)
- e = margin of error (0.1 or 10%)

After plugging in the values:

$$n = \frac{1.645^2 \cdot 0.5 \cdot (1 - 0.5)}{0.1^2} = 67.65$$

Therefore, a representative sample of 67 posts was needed to achieve 90% confidence with a 10% margin of error. We chose these parameters to balance the precision of the estimates with the manual effort required for the analysis. Moreover, higher confidence levels or lower margins of error would have necessitated significantly larger sample sizes and analysis time as follows:

- 95% confidence, 5% margin of error: $n = 346$ posts (86.5 hours)
- 99% confidence, 1% margin of error: $n = 2,832$ posts (708 hours)

For each of the 67 sampled posts, we analyzed the entire Q&A discussion thread from Stack Overflow and the corresponding code changes in GitHub. This manual analysis took approximately 15 minutes per post, resulting in a total of around 17 hours of effort. We found that ≈96% of the sampled posts discussed deep learning bugs, confirming the validity of our filtration criteria (Steps 1-2, Fig. 4.3).

4.3.2 Dataset Construction

The above filtration step resulted in 3,412 Stack Overflow posts. To construct our final dataset, we apply another filter to these posts where we leverage the tags from Humbatova et al.[54]. We aimed to select representative samples from different types of deep learning bugs. To reduce the risk of miscategorizing bug types, we selected tags from Table 1 that exactly matched the leaves of the bug taxonomy proposed by Humbatova et al. [93]. We identified several Stack Overflow tags that partially

Table 4.1: Tags used for filtering different types of bugs

Type of Bug	**Tags**
Model	layer, model, activation-function
Tensor	tensor
Training	loss-function, training-data, optimization, loss, data-augmentation, performance, learning-rate, hyperparameters, initialization, imbalanced-data, nan
GPU	gpu, nvidia, cuda
API	typeerror, valueerror, attributeerror, importerror, compilererrors, syntaxerror, modulenotfounderror

Table 4.2: Summary of the constructed dataset

Type of Bugs	**Total Number of Bugs**
Model	113
Training	95
GPU	101
API	24
Tensor and Input	193
Mixed	42
Total	**568**

matched the leaves of the taxonomy. However, to eliminate confusion about bug types, we only used tags that had an exact match with the taxonomy. Through this manual analysis of posts, bugs and tags, we selected the tags that best represented each bug taxonomy/sub-taxonomy from Humbatova et al. [54]. Table. 4.1 shows our selected 3 tags for model bugs, 1 for tensor bugs, 11 for training bugs, 3 for GPU bugs, and 7 for API bugs. We look for these tags in the above Stack Overflow posts and collect 113 Model bugs, 193 Tensor Bugs, 95 Training Bugs, 101 GPU Bugs, and 24 API Bugs. We also discover 42 bugs belonging to multiple categories (e.g., 5 Model & Tensor Bugs, 5 Tensor & GPU Bugs, 3 Training & API Bugs). Thus, the final dataset contains a total of 668 bugs (568 from our dataset + 100 from Defects4ML [93]) and captures a balanced representation from different types of DL bugs, as shown in Table 4.2 (Step 3, Fig. 4.3).

4.3.3 Environment Setup

For our experiments, we use the following environment setup:

- **Code Editors**: We use *Visual Studio Code v1.79.0*[2] and *PyCharm 2023.1.1* [3] to execute code snippets and reproduce bugs from our dataset. Visual Studio Code and PyCharm are popular code editors for building DL-based applications [44].

- **Dependencies**: To detect the API libraries adopted by the code snippets, we use the *pipreqs* package[4]. We also install the dependencies for each bug into a separate virtual environment using the *venv*[5] module.

- **Frameworks**: Since our dataset contains bugs from *Tensorflow*, *Keras*, and *PyTorch*, we used all three frameworks in our experiments.

- **Libraries**: For generating the random inputs and visualizing the training metrics, we leveraged several scientific computing libraries such as *numpy*, *pandas*, and *matplotlib*.

- **Python Version**: During our experimentation, we used *Python v3.10* to reproduce the deep learning bugs, the latest stable version. If the Stack Overflow issue reported a Python version other than v3.10, we reproduce the bug using the mentioned version instead.

- **Hardware Config**: Our experiments were run on a desktop computer having a 64-bit Windows 11 Operating System with 16GB primary memory (i.e., RAM) and 8GB GPU Memory (Intel(R) Iris XE Graphics).

4.3.4 Manual Classification of Posts

Before conducting any further analysis, we first determine if each post targets a deep learning bug. If a post is not related to a deep learning bug, we exclude it from further analysis. We also excluded the posts not reporting the evaluation metrics of the buggy model from our study. Without these metrics, it was impossible to determine if a bug was successfully reproduced or not. This two-step filtering process helped our analysis focus on relevant and measurable deep-learning bug reports.

[2]https://code.visualstudio.com/
[3]https://www.jetbrains.com/pycharm/
[4]https://pypi.org/project/pipreqs/
[5]https://docs.python.org/3/library/venv.html

Once we confirm that the post is relevant to a DL bug, we check if each selected post accurately represents the type of bug it is assigned to. To validate the bug categorization, we manually analyze each selected Stack Overflow post and review the issue description, stack trace, observed behaviour, expected behaviour, and accepted answer. This ensures that the tag-based selection of posts from Stack Overflow does not affect our results' validity. We perform this manual verification since the Stack Overflow posts might lack an independently verified category label. In contrast, the GitHub issues in the benchmark dataset by Morovati et al. [93] already contain a validated category label.

Reproducing Deep Learning Bugs from Stack Overflow

Once we confirm the category of each post or issue, we follow a two-step approach to reproduce the deep-learning bugs from Stack Overflow posts. First, we gather complementary information about each bug, such as the dataset, code snippet, library versions, and the framework used. Second, we attempt to reproduce the bug using the code snippet and supporting data (e.g., dataset information, environment configurations, hyperparameters, and training logs).

Reproducing Bugs from Github Issues

To reproduce the bugs from GitHub issues, we employed a systematic approach. First, we located the bug-inducing commit of a bug using the commit ID provided by the benchmark. Next, we cloned the corresponding repository and checked out the buggy version of the code using the commit ID. We then set up the development environment according to the reported bug, which may include installing the required version of the programming language and necessary dependencies or libraries, configuring environment variables and setting up any necessary databases, services, or external dependencies the code requires. If applicable, we applied necessary edit actions to the code, such as modifying specific lines or adding/removing code snippets, to trigger the buggy behaviour. We then ran the updated code snippet or the specific part of the codebase where the bug was expected to occur. To verify the presence of a bug, we compared the observed behaviour with the reported buggy behaviour and ran the test cases from the existing benchmark (Defects4ML). Finally, if the bug was

reproduced, we recorded the edit actions used and critical information necessary for the bug reproduction.

Agreement Analysis

The first author and one independent collaborator conducted the bug reproduction process. We first reproduced 10 bugs from our dataset and achieved a Cohen Kappa of 54.5%. Then, we had two meetings to identify the main reasons for our disagreements and resolved them. In the next round, we reproduced 10 more bugs and achieved a Cohen Kappa of 89.1%, which is considered an almost perfect agreement [86]. After achieving an almost perfect agreement, the first author reproduced the remaining bugs (i.e., 128) while the independent collaborator checked the reproduction, achieving an average Cohen Kappa of 85.4%. If the first author fails to reproduce the bug within 60 minutes, the independent collaborator also attempts to reproduce the bug. If both of them failed to reproduce the bug, the bug was marked as irreproducible. We spent ≈280 person-hours on the manual bug reproduction process.

4.3.5 Verification of Bug Reproduction

To verify the successful reproduction of bugs, we employed different strategies depending on the nature of the bugs.

Explicit Bugs

An explicit bug results in an error message or exception. To verify the reproduction of this bug, we adopted a straightforward approach. We extracted the error message from the bug report and attempted to reproduce the bug under the reported conditions. We considered the bug reproduction successful when the observed error message matched the error message in the bug report.

Silent Bugs

Silent bugs are also referred to as functional or numerical errors. They do not result in system crashes or hangs and do not display error messages, but they lead to incorrect behaviour [126]. We adopted a comprehensive approach to verify the reproduction of

silent bugs, as they often manifest subtly through silent issues like slow training or low accuracy.

We reproduced the silent bugs from Stack Overflow posts and GitHub issues. Subsequently, we used the evaluation metrics reported in the original posts or issues as the ground truth to verify the buggy behaviour. In cases where the code snippet was incomplete, we applied our edit actions to make it compilable, executable, and runnable.

To determine if a bug was successfully reproduced, we followed these steps:

- We executed the modified code snippet five times, each time with a different random seed, and calculated the average evaluation metric across these runs.
- We compared the average evaluation metric to the reported evaluation metric of the buggy model.
- If the average evaluation metric was within a 5% error margin of the reported metric, we considered the bug to be reproduced.

We selected the 5% threshold for error margin based on the existing literature [103, 9]. Pham et al.[103] found that implementation-level non-determinism could account for ≈3% variance in the training and evaluation metrics of DL models, often caused by factors such as parallel processing issues, automatic selection of primitive operations, task scheduling, and floating-point precision differences. Furthermore, Alahmari et al.[9] demonstrated that the variance in evaluation metrics could vary from 3% to 7% for models trained using the same dataset and code. Thus, our 5% threshold accounts for the inherent variability in deep-learning models while still maintaining a reasonable standard for bug reproduction. This systematic approach allows us to verify the successful reproduction of silent bugs, ensuring the reliability of our findings.

Information Collection During Verification

Following the successful reproduction of any bug above, we capture various information related to each bug, such as the deep learning architecture involved, edit actions used to reproduce the bug, time taken to reproduce the bug, type of bug reproduced,

and the type of information present in the bug report. All these data gathered during bug reproduction helped us identify key edit actions and information components to reproduce specific types of deep learning bugs (Step 4, Fig. 4.3).

4.3.6 Identifying Type Specific Information and Edit Actions

Algorithm Selection

To establish a relationship among the bug types, component information, and the key editing actions necessary for bug reproduction, we use the Apriori [8] algorithm. Apriori is a well-known algorithm for mining frequent itemsets from a list of transactions. It helps one identify common patterns and associations between different elements.

The Apriori algorithm exploits the principle that if an itemset is frequent across the transactions, then all of its subsets must also be frequent. It starts by identifying frequent individual items in the dataset and extends them to larger and larger itemsets as long as they meet certain constraints (e.g., support threshold). The algorithm terminates when no further successful extensions are found. More specifically, the Apriori algorithm consists of two main steps: the join step and the prune step. In the join step, the algorithm generates new candidate itemsets by joining the frequent itemsets found in the previous iteration. In the prune step, the algorithm checks the support count of each candidate itemset and discards the itemsets that do not meet the minimum support threshold. This process is repeated until no more frequent itemsets are generated. In our context, we apply the Apriori algorithm to analyze the information gathered during bug reproduction. Our goal was to determine the frequent combinations of bug types and component information, as well as edit actions during bug reproduction. We leveraged the Apriori algorithm's systematic pattern-mining capabilities to establish these relationships. By identifying the frequent itemsets, we can gain insights into the common patterns and associations among bug types, component information, and edit actions, which can help us understand and improve the bug reproduction process.

Generating the Transactions

To create our datasets for the Apriori algorithm, we employed a character encoding, where we encoded all labels into a unique character (e.g., 'Training Bug' was encoded as 'T', 'Model Bug' was encoded as 'M', 'Obsolete Parameter Removal' was encoded as 'O') and converted the data into transactions using the following format.

Bug Type → Information Category

Example: $T \rightarrow DH$
Description: The transaction above indicates that the reproduced bug is a training bug (T). The corresponding bug report contains useful information about the bug, such as the dataset used for training (D) and hyperparameters used by the model (H).

Bug Type → Edit Action

Example: $M \rightarrow OLN$
Description: The transaction above indicates that the reproduced bug is a model bug (M). To reproduce the bug, we performed three edit actions, as described below:

- Obsolete Parameter Removal (O): We removed some of the parameters that were absent in the recent library and framework version to ensure that the code compiles.
- Logging (L): We logged various intermediate program states to verify the bug's presence.
- Neural Network Definition (N): We reconstructed the neural network based on the information provided in the bug report.

Following the specified format, we created two datasets of transactions that associate bug types with crucial information and edit actions, respectively. After creating the transactions, we use the Apriori algorithm to compute the support and confidence for our generated itemsets and association rules. We talk about these metrics in detail below.

Metrics for Apriori Algorithm

Support is the proportion of transactions in the dataset that contain a particular itemset. Mathematically, the support of an itemset X is defined as the ratio of the number of transactions containing X to the total number of transactions. It is expressed as:

$$\text{Support}(X) = \frac{\text{Transactions containing } X}{\text{Total number of transactions}}$$

For a rule $X \Rightarrow Y$, where X and Y are two itemsets, the support is calculated for the combined itemset $X \cup Y$.

Confidence measures the likelihood that a rule $X \Rightarrow Y$ holds. It is defined as the ratio of the support of the combined itemset $X \cup Y$ to the support of the antecedent itemset X. Mathematically, confidence is expressed as:

$$\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}$$

Confidence values range from 0 to 1. A high confidence value indicates a strong association between antecedents and consequent itemsets.

Definitions for Apriori Algorithm

- **Itemset:** An itemset is a set of one or more items. In our context, an item can be a bug type, an information category, or an edit action. For example, {T, D, H} is an itemset containing three items: bug type T (training bug), information category D (dataset), and information category H (hyperparameters).

- **Transaction:** A transaction is a record that contains one or more items. In our study, we have two types of transactions as follows:

  - Bug Type → Information Category: These transactions associate a bug type with the useful component information for reproducing that bug.

  - Bug Type → Edit Action: These transactions associate a bug type with the edit actions performed to reproduce that bug.

- **Rule:** A rule is an implication of the form $X \Rightarrow Y$, where X and Y are itemsets. It suggests that if itemset X is present in a transaction, then itemset Y is likely to be present as well. In our study, rules are generated from the transactions to establish associations between bug types and component information or edit actions.

Association Rule Generation

We conducted two separate association rule mining operations in our study - one focused on component information while the other focused on edit actions used to reproduce deep learning bugs. The Apriori algorithm generates rules by first identifying frequent itemsets and then creating rules from them. The steps below explain the process of rule generation with an example.

- **Identify frequent itemsets:** The algorithm scans the transactions to find itemsets that occur frequently while satisfying the minimum support threshold. For example, if the item T, D appears in 20% of the transactions and the minimum support threshold is 10%, it is considered a frequent item.

- **Generate rules:** Once frequent itemsets are identified, the algorithm generates rules from them. For each frequent itemset, the algorithm creates rules by splitting the itemset into antecedent (left-hand side) and consequent (right-hand side). For example, from the item T, D, H, the following rules can be generated:

  - $T \Rightarrow D, H$
  - $D \Rightarrow T, H$
  - $H \Rightarrow T, D$
  - $T, D \Rightarrow H$
  - $T, H \Rightarrow D$
  - $D, H \Rightarrow T$

- **Calculate confidence:** For each generated rule, the algorithm calculates the confidence value. Confidence measures the likelihood that the consequent itemset appears in a transaction given that the antecedent itemset is present. Rules

with confidence values above a minimum threshold are considered strong associations.

In particular, we extracted 27 itemsets and 34 rules for component information, highlighting the useful information for reproducing deep learning bugs. Similarly, we extracted 126 itemsets and 284 rules for edit actions, capturing the edit actions needed to reproduce bugs. We generated association rules based on the entries in our dataset and did not filter or remove any rules before determining confidence values. We then calculate the confidence values for all generated rules to identify the most influential ones for connecting bug types with edit actions and useful information.

Computation of Confidence Values for the Generated Association Rules

As discussed earlier, support indicates how frequently a rule occurs, while confidence indicates the generality of the rule. To compute the confidence values for each association rule, we performed the following steps:

- **Calculate Support for Antecedent (X):** We calculate the support for the antecedent, which is the bug type in our case (e.g., 'T' for Training Bug or 'M' for Model Bug). Support for X is the proportion of all transactions that contain the specific bug type.

- **Calculate Support for Combined Itemset ($X \cup Y$):** We then calculate the support for the combined itemset, $X \cup Y$, which includes both the bug type and the information category or edit action (e.g., 'T $\cup$ H' for Training Bug associated with Hyperparameter Information, or 'M $\cup$ O' for Model Bug associated with Obsolete Parameter Removal).

- **Compute Confidence for the Rule ($X \Rightarrow Y$):** The confidence of the rule $X \Rightarrow Y$ is computed by dividing the support of the combined itemset $X \cup Y$ by the support of the antecedent X. This step gives us the confidence value, which indicates how often the information category or edit action Y is associated with the bug type X in our transactions.

For example, consider the rule $T \Rightarrow D$. This rule indicates that if the type of bug is a 'Training Bug' (T), it can be reproduced by the edit action 'Input Data Generation'

(D). To calculate the confidence of this rule, we count the number of transactions in which the training bug is reproduced by using the edit action 'Input Data Generation' and divide this count by the total number of transactions involving training bugs in the dataset.

Identification of High Confidence Associations

We use high confidence and support values to detect the rules that reliably capture the core factors necessary to reproduce specific types of deep learning bugs. Based on these high-confidence rules, we identify the top 3 pieces of useful information and the top 5 edit actions used to reproduce each bug type. The decision to select three useful pieces of information and five edit actions was influenced by two key factors. First, we adhere to the Parsimony Principle [48], which suggests that selecting the simplest set of rules is preferable when multiple rules can predict or describe the same phenomenon. We thus concentrate on the most significant factors by selecting the top 3 and top 5 rules for edit actions and useful information, respectively. Second, we filter the rules based on minimum confidence values of 30%, as suggested by Liu et al. [72]. With our limited dataset, a 30% confidence threshold can indicate a substantial pattern since finding associations in 30% of cases points to a meaningful correlation given the data size. Furthermore, the 30% minimum confidence helps filter out spurious correlations with small datasets that can occur by chance. Therefore, for our dataset, the selected threshold strikes an effective balance - it is high enough to identify meaningful associations in the data while eliminating noise from false correlations. Overall, this filtration left us with 23 rules for edit actions and 20 for useful information. Focusing on these high-confidence, high-support rules can reveal the patterns that reproduce deep learning bugs (Step 5a, 5b, Fig. 4.3).

4.3.7 User Study

To assess the benefits and implications of our findings in a real-life setting, we conduct a user study involving 22 developers (10 from academia + 12 from industry) (Step 6, Fig. 4.3). We discuss our study setup, including instrument design and participation selection, as follows:

Instrument Design: We used Opinio, an online survey tool recommended by our

institution, to construct and distribute our questionnaire. Opinio enabled us to track the time spent by the participants on each individual question, which proved to be useful for our further analysis. The use of Opinio also did not require any additional effort from the participants, which made it a suitable choice for our user study. We divided our questionnaire into three sections. We discuss them in detail below:

- **Introduction**: We first summarize our findings on the reproducibility of deep learning bugs to provide the participants with the necessary context and background information. For the survey itself, we do not give the respondents a fixed time to complete it, but we specify that it should take ≈60 minutes on average; this number was derived from our pilot study. This was done to ensure that the respondents do not work under time pressure. Since the participants have different levels of experience, allocating a fixed time for bug reproduction might affect our results.

- **Demographic Information**: After providing the contextual information about our study, we collect demographic information from developers (e.g., experience bug fixing in deep learning frameworks). We then ask the developers to elaborate on the challenges that they face when reproducing deep learning bugs in their daily lives.

- **Questionnaire Preparation**: First, we select eight bugs (2 Tensor, 2 API, 2 Model, and 2 Training Bugs) from our dataset constructed during manual bug reproduction and dataset creation. During this process, we categorized the bugs by difficulty level and type based on the number of edit actions and critical information required to reproduce them. We use stratified random sampling to pick 2 bugs from each type; one of the bug types is relatively easy to reproduce with only 1 edit action, and the other type is relatively difficult to reproduce warranting multiple edit actions. We pick 8 bugs following this approach and, then randomly assign them to four sets, each containing 1 easy and 1 difficult bug. Second, we provide the users with the issue description and the code snippet from the original Stack Overflow post. We also provide a Google Colaboratory notebook containing the code for sample edit operations to aid the bug reproduction process. Finally, we include a free-text box in the

form to allow users to share any additional information about edit actions not covered in our study. Our complete study form, which contains the instructions provided to the participants, is present in Appendix B.

Study Session: During our study session, each participant completes the following five tasks. First, the participant provides their demographic information. Second, the participant explains their daily challenges when reproducing deep learning bugs. Third, each participant reproduces two deep learning bugs and self-reports the edit actions and information they used to reproduce the bugs. Fourth, the participant also provides the rationale behind their self-reported edit actions and component information used to reproduce the bugs. Finally, the participant provides information about any other edit actions that they might have used to reproduce the bugs but are not covered in our study.

Participant Selection: We first conducted a pilot study with two researchers and two developers. Based on their feedback, we rephrased ambiguous questions and added sample code to aid the manual bug reproduction by the users. Incorporating this constructive feedback enabled us to refine and improve the quality of our final questionnaire. Then, we invite professional developers and researchers with relevant deep learning experience to our study. We send our invitations to the potential participants using direct correspondences, organization mailing lists (e.g., Mozilla Firefox), and public forums (e.g., LinkedIn and Twitter). A total of 22 participants responded to our invitations. Out of them, 10 (45%) came from academia, and 12 (55%) came from industry. In terms of bug-fixing experience with deep learning bugs, 14 participants (63.63%) had 1-5 years of experience, 4 (18.18%) had 5-10 years of experience, and the remaining 4 (18.18%) had less than one year of experience. In terms of deep learning frameworks, 19 participants (86.34%) reported having working experience with Tensorflow, 21 (95.45%) reported experience with PyTorch, and 20 (90.91%) reported experience with Keras. All these statistics indicate a high level of cross-framework expertise within our participants.

Defining Control and Experimental Groups: We carefully divided the study participants into control and experiment groups, as shown in Table. 4.3. Using these two groups, we wanted to assess the benefit of our recommended information in the context of bug reproduction. The control group receives no hints about how

to reproduce a bug. In contrast, the experimental group receives hints (e.g., useful information, edit actions) that could help them reproduce a bug.

Ensuring Similar Experience Levels: Our guiding principle was to ensure that both groups had a similar distribution in their relevant experience. Specifically, we surveyed all the developers about their experience and used a stratified random sampling approach to assign them to the two groups. This randomization allowed us to minimize potential bias and confounding factors across the groups.

Table 4.3: Distribution of participants in the control and experimental groups

DL Experience	Developers in Control Group	Developers in Experimental Group
<1 Year	3 (27.27%)	3 (27.27%)
1-5 Years	6 (54.54%)	5 (45.45%)
5-10 Years	2 (18.18%)	3 (27.27%)

Leveraging Associations to Produce Hints for User Study: We use high-confidence associations from our RQ2 (Step 3.6.6) to develop hints for our user study. These associations revealed the edit actions and critical information needed to reproduce specific types of bugs. We use the steps below to construct the hints in a systematic way for our user study.

1. **Bug Categorization:**
   (a) Analyze the information found in a bug report to identify the specific type or category of the bug.
   (b) Use the identified bug category to retrieve relevant component information and edit actions.
2. **Retrieve component information and edit actions:**
   (a) Based on the identified bug category, retrieve the top 3 components and top 5 edit actions from the findings of RQ2.
   (b) The component information includes the most important aspects required to reproduce or understand a bug (e.g., shape of input data, training code, error messages).
   (c) The edit actions refer to the most frequently associated actions to a specific bug category.

3. **Determining the Most Relevant Statement:** We also collect the most relevant statement from the bug description as follows.

   (a) Split the bug report's text into a list of sentences using '.' as the delimiter.

   (b) Collect a list of key phrases for the components retrieved in Step 2(a). These key phrases are derived from our qualitative analysis of the bug reports and can be found in the replication package [115].

   (c) Generate the Sentence-BERT embeddings for every statement in the bug report and the keywords above using the 'sentence-transformers/all-MiniLM-L6-v2' pre-trained model.

   (d) For each statement in the bug report, calculate the cosine similarity between the statement embedding and the embeddings of the keywords.

   (e) Identify the relevant statement for the most prevalent component (i.e., top component from Step 2(a)) based on their cosine similarity score.

   (f) Prepare a context-specific hint that guides the user to a particular statement using the following template: 'Focus on the statement: <Statement extracted in Step 3(e)>'

4. **Hint Formulation:**

   (a) Combine the retrieved component information and edit actions from the research findings with the context-specific hint to formulate the complete hint for the user study.

   (b) Use the template below to formulate the hint for each bug.

Template for Hint Generation

Hints

1. <CI1>, <CI2>, <CI3> can be useful information for reproducing the bug.
2. <EA1>, <EA2>, <EA3>, <EA4>, <EA5> can be useful edit actions for reproducing the bug.
3. Focus on the Statement: "<**Most Relevant Statement from the SO Post**>"

We ensure a systematic formulation of hints for our study by following the steps above. This approach incorporates high-confidence associations from Step 3.6.6 and the bug description for individual bug. With this intervention, we plan to measure if and how our recommended information help participants reproduce the bugs more accurately or quickly. The code and results for the hint formulation are available in our replication package [115].

4.3.8 User Study Results Analysis

To assess the effectiveness of our edit actions, we analyze the participants' responses and open-ended feedback. Specifically, we analyze the edit actions used by the participants and compare them with those used in our manual bug reproduction process. If similar edit actions were used, it would indicate that our recommended edit actions and component information were effective in the reproduction of the bugs.

In our user study, the participants first reproduce their assigned bugs and report the information or actions they used to reproduce their bugs. To analyze the effectiveness of our recommended information, we compare the control and experimental groups in terms of their success rates in bug reproduction and the time taken to reproduce the bugs. We select bug reproducibility rate and time taken for bug reproduction as our key metrics to evaluate the effectiveness of our findings. These quantitative metrics directly measure how successful and efficient our recommended information is in assisting the developers to reproduce deep learning bugs. Higher reproducibility rates and reduced reproduction times in the experimental group compared to the control group would indicate that our findings are effective for improving deep learning bug reproducibility. We also analyze the open-ended feedback from the participants using thematic analysis to determine if our provided hints were useful or not. We also analyze them to uncover new insights into deep learning bug and their reproduction.

By analyzing the results from our user study, we establish quantifiable patterns and trends in deep learning bug reproducibility. Through developers' open-ended feedback, we identify key challenges and practical solutions that improve debugging practices. The user study results and feedback also provide information about the effectiveness of our recommended actions for bug reproduction. This enables us to

Table 4.4: Summary of the reproduced bugs

Type of Bug	**Bugs Reproduced**	**Model Architectures Covered**
Training (T)	50	CNN, LSTM, AutoEncoder, MLP, RCNN, ResNet
Model (M)	42	BERT, CNN, GMM, LSTM, MLP VGG16, Transformers
API (A)	20	CNN, GAN, MLP, Transformers, VGG19, Variational RNN
GPU (G)	3	-
Tensor and Input (I)	29	CNN, GAN, Logistic Regression, MLP, ResNet
Mixed (X)	4	CNN, BERT, MLP

deliver novel and actionable insights regarding the current state of deep learning bug reproducibility based on empirical evidence (Step 7, Fig. 4.3)[6].

4.4 Study Findings

In this section, we present the findings of our study by answering three research questions as follows.

4.4.1 RQ1: Which edit actions are crucial for reproducing deep learning bugs?

To answer the first research question, we worked with 165 bugs and successfully reproduced 148 bugs of different types and architectures. Table. 4.4 briefly summarizes our reproduced bugs. Through our comprehensive reproduction process, we identify ten edit actions that are crucial for reproducing deep learning bugs. Table. 4.5 shows the identified actions from our qualitative analysis. We explain these actions as follows.

Input Data Generation ($\mathbf{A_1}$) is one of the key edit actions for reproducing deep learning bugs. This action involves programmatically generating synthetic input data that closely matches the characteristics of the original data used for training the model. The key objective of input data generation is to simulate representative data that can trigger or reproduce the erroneous model behavior described in the bug report. This allows for the reproduction of issues that manifest only in the presence of specific data properties or distributions. To perform input data generation, we leverage any details about the data that are provided in the bug report, such as data

[6]This study was approved by Dalhousie's Research Ethics Board (REB Approval #2023-6890)

types, value ranges, shapes, distributions, preprocessing steps, etc. For example, for image data, the report may specify that inputs are RGB images of size 224x224x3 with pixel values normalized to [0,1]. Similarly, for text data, the description may indicate sequences of 512 tokens processed using a particular tokenizer. Using this information, we can systematically generate synthetic data matching the properties through appropriate library functions. For images, we can use libraries like OpenCV [2] or PIL [4] to construct random images of the required size and channels. For text, we can sample token sequences from a standard corpus or use specialized generative models like GPT-2 [107]. Our manual bug reproduction shows that ≈73% of our collected posts from Stack Overflow have the relevant data characteristics. Let us consider the issue reported in the Stack Overflow post (Issue #61781193). In this post, the reporter suspects that the model is not learning - as evidenced by the constant training loss across the epochs. The following text from the post shows how the reporter might submit the input data distribution.

R: My training data has input as a sequence of 80 numbers in which each represent a word and target value is just a number between 1 and 3.

Using this information, we generated the random input data as follows and were able to reproduce the corresponding bug.

```
train_data = torch.utils.data.TensorDataset(torch.randint(0, 200, (1000, 80)), torch.
    randint(1, 3, (1000,)))
```

Neural Network Construction (A_2) was one of the most used edit actions during our bug reproduction. In this edit action, we construct a neural network based on the architecture provided by the reporter. Similar to the data characteristics, the information about the neural network is present in ≈65% of the reproducible issue reports. Using the neural network description from the issue reports, we were able to construct the models. Let us consider the issue reported in the Stack Overflow post (Issue #63204176). In this post, the reporter submits an issue where a CrossEntropy loss function within a loop is overwritten with a Tensor, causing a TypeError in later iterations. When we analyze the post, we find that the reporter mentions that they have used a logistic regression model (1-layer neural network with a sigmoid activation function [85]). However, the reporter does not provide the code snippet necessary for reproducing the bug.

Table 4.5: Edit actions for reproducing deep learning bugs

Edit Action	**Overview**
A_1: *Input Data Generation (D)*	Generating input data that simulates the data used for training the model.
A_2: *Neural Network Construction (N)*	Reconstructing or modifying the neural network based on the information provided.
A_3: *Hyperparameter Initialization (H)*	Initializing the hyperparameters for training, such as batch size and number of epochs.
A_4: *Import Addition and Dependency Resolution (R)*	Determining the dependencies in the code snippet and adding the missing import statements.
A_5: *Logging (L)*	Adding appropriate logging statements to capture relevant information during reproduction.
A_6: *Obsolete Parameter Removal (O)*	Removing outdated parameters or functions to match the parameters of the latest library versions.
A_7: *Compiler Error Resolution (C)*	Debugging and resolving compiler errors that arise due to syntactic errors in the provided code snippet.
A_8: *Dataset Procurement (P)*	Acquiring the necessary datasets and using them to train the model.
A_9: *Downloading Models & Tokenizers (M)*	Fetching pre-trained models and tokenizers from external sources.
A_{10}: *Version Migration (V)*	Updating code to adapt to changes introduced in newer versions of libraries and frameworks.

R: I am trying to write a simple multinomial logistic regression using mnist data.

To reproduce the bug, we constructed multiple logistic regression models for the MNIST dataset, as highlighted in the post. We added the import statements, wrote the code to load the MNIST dataset, and completed the code. Using the training loop's code snippet from the original code and our edit actions, we could complete the code snippet and reproduce the bug successfully. Using this partial information, we constructed multiple multinomial logistic regression models, and despite the lack of relevant code examples, we successfully reproduced the corresponding bug.

Hyperparameter Initialization ($\mathbf{A_3}$) is one of the core edit actions for reproducing deep learning bugs. As a part of this edit action, we initialize various hyperparameters (e.g., number of epochs, batch size, optimizer) for training the neural networks. Sometimes, the reporter did not provide these configurations, and we had to initialize them using default parameters to reproduce the bug. This is further evidenced by the fact that only ≈53% of our collected posts from Stack Overflow have information about the hyperparameters. Let us consider the issue reported in the Stack Overflow post (Issue #31880720), where the author gets a poor test accuracy of 13.9% when training a neural network on a synthetic binary classification dataset. After manual analysis of the post and the code snippet, we observe that some of the hyperparameters in the code snippet have not been initialized.

```
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch, show_accuracy=
    True, verbose=2, validation_data=(X_test, Y_test))
```

To reproduce the bug, we initialized the batch size with commonly-used values $\{32, 64, 128\}$ and the number of epochs ranging from 1 to 10. Since we initialized these hyperparameters, we ran the edited code snippet five times to confirm the effectiveness of our edit operation. We were able to reproduce the bug in all five iterations. We performed the steps mentioned above for all 56 bugs involving hyperparameter initialization.

Import Addition and Dependency Resolution ($\mathbf{A_4}$) are one of the most common edit actions for all types of bugs. In this edit action, we analyze the code manually and determine the dependencies required for the code snippet to run. Then, we install the dependencies and manually import them to complete the code snippet. This edit action was also used to reproduce traditional software bugs, as reported by Mondal et al. [89]. In the Stack Overflow posts we collected, ≈47% of the issue reports lacked the required dependency information. This significant absence of the dependency information further highlights the need for our suggested edit action. Let us consider the issue reported in the Stack Overflow post (Issue #50306988). In this post, the neural network model with softmax activation struggles to fit a simple 2-feature classification dataset, converging extremely slowly compared to logistic regression, which achieves 100% accuracy. However, the code snippet reported in the post was incomplete, and the dependency details were missing. Hence, we resolved the

dependencies (keras, numpy and random) and added the required import statements to the code snippet. With the resolved dependencies, we were able to reproduce the bug successfully.

Logging ($\mathbf{A}_5$) plays a crucial role in the reproduction of different types of bugs. This action involves adding log statements to the provided code snippet to verify the reproduction of a bug. Let us consider the issue reported in the Stack Overflow question (Issue #70546468). The reporter provides a numpy array with shape (1, 3). The reporter then vertically concatenates (stack depthwise) multiple copies of this array to create tensor groups with shapes (2, 3) and (1, 3). After this vertical concatenation process, the reporter expects the tensor's shape to be (2, None, 3). Unfortunately, the tensor's shape was (2, None, None), according to the reporter. To verify the claim made by the reporter, we introduce a log statement within the code snippet. When we run the modified code snippet, we observe the shape of the stacked tensor to be (2, None, None), thereby confirming the reproduction and validity of the bug.

Obsolete Parameter Removal ($\mathbf{A}_6$) is a crucial edit action that enhances bug reproducibility when differences in library and framework versions cause compatibility issues. It involves removing obsolete parameters no longer supported by newer versions. Frequent updates to deep learning frameworks and APIs can cause breaking changes, i.e., the code written for older versions of the framework is incompatible with newer versions. Developers reproducing these bugs often face challenges when the bug report's environment is significantly older than their current working environment, and downgrading to match those outdated versions is not always feasible due to various constraints. In some cases, the bug report uses significantly outdated versions that are no longer supported or maintained by the framework developers. Additionally, the developer's current project may have dependencies that require newer versions of Python or the frameworks, making it impractical to downgrade. Furthermore, organizations or development teams may have policies in place that mandate using the latest stable versions for security, performance, and maintainability reasons, preventing the use of older versions.

In such cases, developers must port the bug-reproducing code to their current environment, where Obsolete Parameter Removal edit action proves highly beneficial.

It makes the code compatible with newer framework versions while preserving the bug-reproducing behaviour. Consider the issue reported in the Stack Overflow post (Issue #65992364). In this post, the reporter attempts to optimize an object detection model using 'pytorch-mobile', but the code snippet fails to optimize the model file size. The code snippet in the issue also contains the following line of code.

```
script_model_vulkan = optimize_for_mobile(script_model, backend=``Vulkan")
```

However, according to the API documentation of PyTorch 1.6.0 [106], the `backend` parameter was no longer supported. We removed the obsolete parameter and thus were able to reproduce the bug.

Compiler Error Resolution (A_7) is one of the extensively employed edit actions in reproducing bugs. In this edit action, we resolve compiler errors to get the code running and reproduce the bug. While downgrading the compiler or library versions can sometimes resolve errors or enable the bug reproduction with deprecated functionality, it may not always be feasible. For example, major framework releases often remove support for older versions, making downgrading impossible. Additionally, Python version mismatches between the original buggy code and the current development environment can prevent successful downgrading. In cases where downgrading is infeasible due to such constraints, correction of compiler errors enables reproducing deep learning bugs despite incompatible environments.

For example, in the Stack Overflow post (Issue #71514447), the reporter states that the training loss is significantly increasing after every epoch. They suspect that the bug might be due to the incorrect computation of loss values. Furthermore, to help the developers reproduce the bug, they provide a detailed description of the bug and a complementary code snippet. However, the code snippet provided by the user does not compile, as the `criterion` function has not been defined. Hence, we replaced the function parameter with the default value of cross-entropy loss, as shown below.

```
def train_model(model, optimizer, train_loader,  num_epochs, criterion = nn.
    CrossEntropyLoss()):
```

Using the edit action above, we resolved the compiler errors and thus were able to reproduce the corresponding bug.

Dataset Procurement (A_8) is a critical edit action to reproduce bugs that

require specific datasets. In this edit action, we analyze the issue report and attempt to procure the dataset mentioned in the report. For instance, in the Stack Overflow post (Issue #73966797), the reporter mentioned that they used the CIFAR-10, a well-known dataset for object detection. To reproduce the bug, we downloaded the dataset from its original source[7], and using the dataset and the training code, we could reproduce the bug.

Downloading Models & Tokenizers (A_9) is one of the edit actions that was frequently used to reproduce bugs in Large Language Models and Transformer-based architectures. In this edit action, we download the pre-trained models and tokenizers for reproducing a bug. Let us consider the issue reported in the Stack Overflow post (Issue #69660201). In this post, the reporter faces a ValueError when fitting a text classification model with a BERT tokenizer, due to a mismatch between the model's expected input and the tf.data.Dataset created from the text corpus and labels. For example, in the Stack Overflow post (Issue #69660201), the reporter has provided the following information.

R: In the preprocessing layer, I'm using a BERT preprocessor from TF-Hub.

Based on the above information, we added the relevant URLs and configured the code to download the preprocessor and encoder. After downloading the preprocessor and encoder, we successfully reproduced the corresponding bug.

```
tfhub_handle_preprocess = ‘‘https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3
    "
tfhub_handle_encoder = ‘‘https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4
    _H-512_A-8/1"
```

Version Migration (A_{10}) is a vital edit action for reproducing bugs in deep learning frameworks and libraries. It involves adapting code written for older versions to the latest version, confirming the same bug's presence in the current environment. Mondal et al. [89] introduced a similar concept called 'Code Migration'. Similar to Obsolete Parameter Removal, Version Migration aims to address compatibility issues caused by the updates in deep learning frameworks and APIs. While Obsolete Parameter Removal focuses on removing deprecated parameters, Version Migration contains additional modifications required to make the code follow the newer version's syntax, APIs, and functionalities. This may involve updating function calls, class

[7]https://www.cs.toronto.edu/ kriz/cifar.html

constructors, import statements, and other code elements affected by the version changes.

The necessity for Version Migration often arises when the versions used in the bug report are significantly outdated compared to the developer's current working environment. Bug reports may contain code snippets written for older framework versions that are no longer actively maintained or supported by the developers. In such cases, downgrading to the exact reported version is not feasible, as it would require reverting to unsupported and potentially insecure versions. Additionally, the developer's current project may have dependencies that require newer versions of Python or deep learning frameworks, making it impractical to downgrade. Furthermore, organizations or development teams may have policies in place that mandate using the latest stable versions for security, performance, and maintainability reasons, preventing the use of older versions. To demonstrate the utility of the Version Migration, let us consider the issue reported in the post on Stack Overflow (Issue #45711636). The post describes an issue with the CNN architecture constructed by the user. When the user passes the input through the CNN, they encounter a ValueError, which highlights a problem with the negative dimension value of the input. However, the code snippet provided in the issue uses Tensorflow 1.3.0. To reproduce this issue in TensorFlow 2.14.0, we carefully migrated the model built in TensorFlow 1.3.0 to the syntax of Tensorflow version 2.14.0. Specifically, we updated the Sequential, Conv2D and MaxPooling2D constructors to match the TensorFlow 2.14.0 API syntax. We also changed padding and pool size parameter schemes and removed the obsolete input shapes. Finally, we upgraded the model compilation and imported the required Tensorflow 2.14.0 modules. After modifying the code snippet, we successfully reproduced the bug in our runtime environment. We verified the reproduction by comparing the error message from our updated code snippet with the one reported in the Stack Overflow post.

Summary of RQ1: By manually reproducing **148** deep learning bugs, we identify **ten key edit actions** that could be useful to reproduce deep learning bugs (e.g., input data generation, neural network construction, hyperparameter initialization). These edit actions can help developers complete the code snippets and thus reproduce their deep learning bugs.

4.4.2 RQ2: What component information and edit actions are useful for reproducing specific types of deep learning bugs?

Identifying useful Information in Bug Reports

While reproducing deep learning bugs, we kept track of information from bug reports that helped us reproduce them. After reproducing 148 bugs successfully, we have identified five useful pieces of information that can improve the chance of reproducing a bug. We discuss these factors in detail below.

Data (F_1): Data is one of the essential factors in ensuring the reproducibility of deep learning bugs. Deep learning systems heavily rely on the data [22], and reproducing deep learning bugs becomes easier with access to the original data. Data helps us reproduce the deep learning bugs by providing the exact sample inputs that trigger the erroneous behaviour. By understanding the training data distributions and ranges, we can reconstruct the original training environment, which is crucial for reproducibility. However, the issue reports often lack direct information about the data. To address this problem, we collect information about the data by extracting the shape of the data, data distribution, type of variables and their corresponding ranges from the issue description. Leveraging this information and our proposed edit actions, namely Input Data Generation (A_1) and Dataset Procurement (A_8), we can generate or obtain the necessary data to reproduce deep learning bugs. According to our investigation, ≈77% of our reproduced bugs contained information about the data (see Table. 4.6) in their issue reports. For example, in the Stack Overflow post (Issue #43464835), the reporter has provided the dimensions and sample row of the training dataset, as shown below.

R: I have a train dataset of the following shape: (300, 5, 720)

Sample Input: [[[6. 11. 389. ..., 0. 0. 0.]]]]

Using this information and our proposed edit action Input Data Generation (A_1), we first generated a data frame of size (300, 5, 720) that contains values in the range from 0 to 400. Then, by leveraging the generated data and other useful information, we successfully reproduced the corresponding bug.

Model (F_2): The model architecture describes the components of a deep learning model and how they transform inputs into outputs. Understanding the model architecture is crucial for reproducing deep learning bugs, as it provides insights into the model's components, connectivity, and the required neural network architecture. According to our investigation, ≈58% of our reproduced bugs contained information about the model architecture (see Table. 4.6) in their issue reports. However, the complete source code implementing the model architecture might not always be available in the issue reports. To overcome this limitation, we carefully gather information about a model's architecture, i.e., the number of layers, layer properties and activation function from the issue description. Leveraging this information and our proposed edit action Neural Network Construction (A_2), we reconstruct the model and reproduce the deep learning bugs. For example, let us consider the following text from a Stack Overflow post (Issue #63204176) that mentions the use of multinomial logistic regression on the MNIST dataset, as shown below. Unfortunately, the reporter fails to provide the code snippet for the model.

R: I am trying to write a simple multinomial logistic regression using mnist data.

Despite the absence of the code snippet, the issue description provides useful hints about the model architecture and dataset. Using them and our edit action – Neural Network Construction (A_2), we successfully reproduced the bug.

Hyperparameters (F_3): Hyperparameters play a crucial role in controlling the learning process and model behaviour during training. They encompass parameters such as learning rate, batch size, number of epochs, optimizer, regularization techniques, and loss functions. The specific values chosen for these hyperparameters significantly impact a model's performance, training process, and bug manifestation. Thus, reporting the complete set of hyperparameters is essential to help reproduce deep learning bugs. According to our investigation, 48% of our reproduced bugs contain information about the hyperparameters used in their issue reports. For the 52% of bug reports that did not include hyperparameter information, we used the

Hyperparameter Initialization edit action (A_2) to reproduce the bugs by initializing the hyperparameters with default values. In the example Stack Overflow post (Issue #65993928), we can see how hyperparameters can play a crucial role in reproducing the bug.

```
loss2 = (2 * (log_sigma_infer - log_sigma_prior)).exp() \ +((mu_infer - mu_prior)/
    log_sigma_prior.exp()) ** 2 \ - 2 * (log_sigma_infer - log_sigma_prior) - 1

loss2 = 0.5 * loss2.sum(dim = 1).mean()
```

Since the reporter was using a custom loss function, it was vital for them to share the value of constants and the formula used to calculate the loss. The reporter's custom loss function, with constants 2 and 0.5 and its formula, was crucial for reproducing a bug in model training due to incorrect loss calculation, emphasizing the need to provide a complete set of hyperparameters.

Code Snippet (F_4): Code snippets are critical for reproducing bugs, as highlighted by the fact that 98% of professional developers consider them an essential component of bug reproducibility [118]. They include the data preprocessing, data splitting technique, code used for training the model, and implementation of the evaluation metrics. However, from our manual bug reproduction, we observe that even though code snippets are present in ≈82% of the bug reports, only 9.41% of them can be used verbatim for bug reproduction. To address this limitation, we use several of our proposed edit actions, such as Import Addition and Dependency Resolution (A_4), Logging (A_5), Obsolete Parameter Removal (A_6), Compiler Error Resolution (A_7) and Version Migration (A_{10}). These edit actions help us fix the errors in the code and make the code compileable and runnable.

To demonstrate the importance of submitting a complete code snippet, let us consider the Stack Overflow post with Issue #76186890. In this issue, a high-quality code snippet helps us reproduce the bug related to a complicated architecture (T5) without significant changes. The code snippet uses the T5 model from HuggingFace Transformers for text-to-text translation. It preprocesses the input text data, configures the target IDs to predict a specific answer, calculates loss and perplexity metrics,

Table 4.6: Prevalence of useful information in reproducible issue reports

Factor	Data	Model	Hyperparameters	Code Snippet	Logs
Prevalence	77.4%	58.1%	47.9%	82.1%	87.6%

and trains the model. Having a complete and runnable snippet was helpful in reproducing the bug. It provides sufficient details like data preparation, loss calculation, model training, and evaluation to improve the ease of reproducing deep learning bugs, which were essential for bug reproduction.

Logs ($\mathbf{F_5}$): Logs provide a real-time record of the model's behaviour during training and inference. Traditional software logs consist of information such as event logs and stack traces, whereas deep learning systems consist of compiler error logs, training error logs, and evaluation logs [27]. Sharing these logs is crucial for reproducing deep learning bugs as they allow us to verify if we can reproduce the same erroneous behaviour reported in the original issue. From our manual bug reproduction, we discover that ≈88% of our reproduced bugs contain the necessary logs in their issue reports. Such a high presence of logs in our dataset of reproduced bugs highlights the importance of logging in deep learning bug reproduction. By matching the logs from the original issue and the logs from reproduction on our local machines, we were able to confirm the presence of several bugs in the deep learning systems.

The Stack Overflow post (Issue #34311586) shows how logs can be used to confirm the presence of deep learning bugs. In this particular issue, the reporter shared the training logs and the code snippet. We modified the code snippet with our proposed edit actions to make it compilable and runnable. When executed, we observed the same anomalous behaviour in the training logs as described by the original reporter. This demonstrates the importance of sharing training and evaluation logs in the issue report. We also found them to be one of the most useful pieces of information to reproduce deep learning bugs.

Relationship between Useful Information and Type of Bugs

During our manual analysis (Section 4.4.2), we identify useful information for reproducing deep learning bugs. We used the Apriori algorithm to determine the relationship between the type of bug and the information required to reproduce the bug. The

Table 4.7: Top 3 useful component information for reproducing specific types of deep learning bugs

Training	**Model**
Code Snippet (0.86)	Logs (0.7857)
Data (0.82)	Code Snippet (0.7143)
Logs (0.76)	Model (0.6429)
Tensor	**API**
Data (0.9655)	Logs (0.85)
Logs (0.9310)	Code Snippet (0.75)
Code Snippet (0.7241)	Model (0.70)

insights from this analysis could serve two purposes – detecting missing information in submitted bug reports and formulating follow-up questions to obtain any missing information needed for reproducibility. Table. 4.7 summarizes our findings, and we discuss them in detail below.

Data: The accurate reproduction of bugs in deep learning systems heavily relies on the presence of data and its characteristics. They play a crucial role in reproducing two key categories of bugs - **training** bugs and **tensor** bugs.

For training bugs, the data has a confidence value of 82.00% according to our generated rules (check Table. 4.7). This high confidence highlights the significance of data in reproducing the numerous training issues that can manifest during model development. The details about the data, such as the number of samples, class distribution, feature distributions, data splitting ratios, and preprocessing steps, are instrumental in reproducing training bugs. For example, training a classification model on a highly skewed dataset is prone to overfitting. With the knowledge of the data distribution, we can recreate a representative dataset, which can help us reproduce the bugs, even if the actual dataset is not available.

For tensor bugs, data characteristics have an even higher confidence of 96.55%, according to our generated rules (Table. 4.7). This suggests that tensor attributes like shape, data type, sparsity, value ranges and origin are pivotal for reliably reproducing many bugs stemming from invalid dimensions or precision issues. For instance, bugs arising from tensor shape mismatches can emerge if the shape of the input data does not match the expected input shape. Furthermore, real-world data is often more vulnerable to human error and biases compared to synthetic data [125]. Therefore,

comprehensive documentation and availability of these salient data characteristics are imperative for the reliable reproduction of the numerous **training** and **tensor** bugs in deep learning systems.

Model: A neural network model's architecture and implementation details are crucial to reproducing **model** and **API** bugs. The model architecture has 78.57% and 70.00% confidence values for model and API bugs, respectively, according to our generated rules. These high values signify that access to model details is vital for reliably reproducing issues stemming from model capacity, connectivity, and API usage.

An access to the model architecture (e.g., layer types, layer connectivity, weight initialization schemes, etc.) can help us systematically reproduce model bugs. Insufficient learning capacity in specific layers and improper weight initialization might lead to exploding/vanishing gradients [50], whereas incorrect layer connectivity might lead to representational bottlenecks [130]. These issues usually manifest as model bugs during training and inference. Thus, information about the model architecture can help us localize and reproduce such bugs.

For reproducing the API bugs, model architecture can provide fundamental context. Details like layer dimensions, bottlenecks, parallelization needs, and memory requirements show how the model interacts with the API [145]. Bottlenecks within a model, areas where data processing slows down, and the need for parallel processing to handle large-scale computations shape how APIs are utilized. Additionally, the varying memory requirements of different architectures impact how the model leverages the system's resources via the API. This includes memory allocation for storing weights and activations and managing the flow of data through the network during operations like forward and backward propagation. Understanding the model architecture is vital for reproducing bugs, as it indicates whether issues stem from the model's design or from its interaction with the API. For example, a bug might arise due to the model's inability to handle certain operations efficiently, or it could be a result of the API not properly supporting specific architectural features. Hence, model architecture helps us reproduce the bugs triggered by incorrect usage of API. Therefore, the presence of model architecture allows the reproduction of both **model** bugs and **API** bugs in deep learning systems.

Code Snippet: Code snippets are invaluable for reproducing deep learning bugs since they isolate and encapsulate the core logic that triggers them. Developers can demonstrate and share the essence of buggy behaviour by creating a minimal reproducible example. Code snippets have very high confidence values of 86.0% for training bugs, 72.41% for tensor bugs, 71.43% for model bugs, and 75.0% for API bugs, according to our generated rules. These high values across all bug types signify that code snippets are critical for reliably reproducing deep learning bugs.

Each code snippet may contain the relevant model, data processing, training and evaluation scripts. Developers can use such a snippet to recreate the bug-inducing steps. For example, a snippet may compactly capture just a few lines, mishandling tensor shapes or performing incorrect gradient calculations, eliminating any confounding factors and enabling developers to reproduce these bugs. Developers can systematically execute, analyze, and debug the code to reproduce various bugs, including training, tensor, model, and API bugs.

Logs: Logs play a fundamental role in deep learning by facilitating the reproduction and resolution of bugs. They comprehensively record every detail during model training, capturing information about various conditions, configurations, and events before the failures. Logs have high confidence values of 76.00% for training bugs, 93.10% for tensor bugs, 78.57% for model bugs, and 85.00% for API bugs according to our generated rules. Such high confidence levels across all bug types highlight that comprehensive log recording is important for the reliable reproduction of deep-learning bugs.

These logs include essential components such as hyperparameters, dataset characteristics, hardware specifications, framework versions, and random number generator seed values. The true power of logs lies in their ability to recreate past training runs precisely. Developers can isolate and reproduce the environment that led to the original bugs using the logged hyperparameters, such as batch size, learning rate schedules, and gradient clipping thresholds. Logs can also help developers reproduce specific deep-learning bugs. For example, logged random seeds can help the developers recreate a specific weight initialization or data batching order, which can then be used to reproduce **training** bugs. Similarly, logs can be used to reproduce **Model**, **Tensor** and **API** Bugs. Thus, logs are invaluable for accurately recreating conditions

Table 4.8: Top 5 edit actions for reproducing specific types of deep learning bugs

Training	**Model**
Input Data Generation (0.5625)	Hyperparameter Initialization (0.5142)
Import Addition (0.4583)	Dataset Procurement (0.4390)
Compiler Error Resolution (0.3750)	Compiler Error Resolution (0.4146)
Dataset Procurement (0.3542)	Import Addition (0.4146)
Hyperparameter Initialization (0.3333)	Neural Network Construction (0.3659)
Tensor	**API**
Hyperparameter Initialization (0.5517)	Input Data Generation (0.40)
Input Data Generation (0.5172)	Hyperparameter Initialization (0.40)
Import Addition (0.5172)	Import Addition (0.35)
Dataset Procurement (0.4138)	Logging (0.25)
Obsolete Parameter Removal (0.3448)	Obsolete Parameter Removal (0.25)

that result in errors and enabling the reproduction of deep learning bugs.

Relationship between Edit Actions and Type of Bugs

After determining association between the useful information and the type of bug, we derived the relationship between the edit actions and the type of bug. We use the Apriori algorithm to determine the relationship, as done earlier. Table. 4.8 summarizes our findings, and we discuss them in detail below.

Training Bug: Input data generation (56.25% confidence) is the most frequently used edit action for reproducing training bugs. However, our analysis (Table. 4.6) shows that ≈77% of the bug reports/SO posts contain information about the data. Leveraging this information and our proposed edit action – Input Data Generation (A_1), we can reproduce the training bugs in deep learning systems. Furthermore, import addition (45.8% confidence) and compiler error resolution (37.5% confidence) are other edit actions which are often used to reproduce training bugs. Although 79% of bug reports contain code snippets, they are often incomplete. To reproduce the bug effectively, we thus need to complete these snippets by adding the necessary import statements and migrating them to the latest versions of libraries and frameworks. Finally, dataset procurement (35.42% confidence) and hyperparameter initialization (33.33% confidence) may also help reproduce training bugs by procuring the dataset required and initializing the missing hyperparameters.

Since the training bugs have specific sub-faults, we manually analyse the Stack Overflow posts and Github issue reports for different sub-types and discuss the edit actions, which can be used to reproduce the specific sub-faults below.

- **Optimisation:** Optimization bugs can be reproduced by employing a combination of edit actions, with a primary focus on three key areas: Hyperparameter Initialization (60.00% confidence), Input Data Generation (60.00% confidence), and Neural Network Construction (40.00% confidence). These actions involve configuring the optimizer and learning rate, creating representative training data, and building the model architecture, respectively. Additionally, logging (40.00% confidence) can help monitor the training process, while Import Addition (20.00% confidence), Compiler Error Resolution (20.00% confidence), and Version Migration (20.00% confidence) ensure compatibility within the environment. By systematically testing different optimizer configurations, generating appropriate input data, and constructing the relevant model architecture, developers can effectively reproduce optimization issues and gain insight into the underlying causes.

- **Loss Function:** Loss function bugs, which can arise from incorrect loss calculations or suboptimal loss function choices, can be reproduced through a combination of edit actions. Hyperparameter Initialization (66.67% confidence) plays a crucial role in configuring the loss function while Logging (58.33% confidence) captures the output of intermediate loss values during training. Since the design of the output layer in the neural network directly influences the calculation of loss, Neural Network Construction (41.67% confidence) may be necessary to reproduce the bugs related to the loss function. Furthermore, it is essential to generate appropriate training data through Input Data Generation (33.33% confidence) to effectively reproduce bugs related to the loss function.

- **Hyperparameters:** Hyperparameter sub-faults, such as suboptimal batch size, suboptimal number of epochs, and suboptimal learning rate, can be reproduced by utilizing the Hyperparameter Initialization edit action (64.29% confidence). This involves initializing various hyperparameters, such as batch

size, epochs, and learning rate, that may be missing or suboptimal in the provided code. By experimenting with different commonly used values for these hyperparameters, we were able to reproduce them. In addition to that, Input Data Generation (50.00% confidence) proves to be valuable for reproducing hyperparameter bugs. Generating representative input data is crucial for triggering hyperparameters-related issues during the training process. Furthermore, actions like Import Addition (42.86% confidence), Logging (35.71% confidence), and Compiler Error Resolution (35.71% confidence) are helpful in ensuring that the code runs smoothly and allows for testing different hyperparameter configurations. These actions contribute to creating a compatible environment to address hyperparameter-related concerns effectively.

Model Bug: Model bugs are often reproduced by using the edit action – hyperparameter initialization (51.4% confidence). From our manual analysis, we observed that the hyperparameters were not often reported for model bugs. Hence, hyperparameter initialization was often used to reproduce the model bugs. Additionally, dataset procurement (43.90% confidence) and compiler error resolution (41.46% confidence) are typical edit actions to reproduce the model bugs. Moreover, the code snippets for deep learning bugs are often incomplete, as observed in our manual analysis; hence, import addition (41.46% confidence) is a critical edit action for completing the code snippet and reproducing model bugs. Finally, model bugs might be reproduced by modifying a neural network's architecture, as shown by the moderate confidence value (36.6%) for the edit action – neural network construction. Since model bugs are primarily caused by errors in the neural network architecture [54], reconstructing the neural network might be the first step to reproduce them. Thus, model bugs can be reliably reproduced through several edits that initialize hyperparameters, resolve compiler errors, procure datasets, add imports, and construct neural networks.

Similar to training bugs, model bugs also have specific sub-faults. Hence, we report the edit actions for reproducing specific sub-faults below.

- **Layer Type & Properties:** To reproduce specific model bugs related to individual layers, such as incorrect layer types, suboptimal filter sizes, or inappropriate activation functions, it is necessary to carefully define the layers as

part of the Neural Network Construction (78.57% confidence) action. By accurately specifying the layers in the model architecture, developers can trigger the desired layer-specific bugs for further analysis and resolution. It is also crucial to initialize the appropriate hyperparameters through the Hyperparameter Initialization (57.14% confidence) action. Additionally, obtaining the required input data shapes via Input Data Generation (42.86% confidence) is essential for triggering layer-specific issues during the training process. To ensure compatibility within the project environment, actions such as Compiler Error Resolution (42.86% confidence) and Version Migration (42.86% confidence) are beneficial for updating the layer definitions and reproducing any compatibility issues that may arise.

- **Model Type & Properties:** Bugs associated with suboptimal model architecture, incorrect network structure, or missing layers can often be reproduced using the Neural Network Construction edit action (71.43% confidence). By carefully constructing the neural network based on the architecture details provided in a bug report, developers can reproduce sub-faults from the model bugs category. The interaction between the hyperparameters and the model architecture could be important for model bugs. Therefore, the Hyperparameter Initialization action (42.86% confidence) also plays a significant role in effectively reproducing model-related issues. To ensure compatibility with the latest library and framework version, actions such as Import Addition (35.71% confidence), Obsolete Parameter Removal (28.57% confidence), and Compiler Error Resolution (28.57% confidence) are essential. These actions help update the codebase and ensure that the necessary dependencies are met to define the desired model architecture accurately.

Tensor Bug: Data is the most important information for reproducing tensor bugs, as shown by the strong correlation between Tensor Bug & Data (see Table. 4.7). This phenomenon can be attributed to the fact that tensor bugs primarily relate to the data [54]. Since tensor bugs are so data-dependent, dataset procurement emerges as an important edit action to reproduce them. Bug reports reference the exact dataset used (if well-known) or describe the data type, shape and distribution. With this information, input data generation and dataset procurement can generate or procure

the data required for the reproduction of the tensor bugs. Like other deep learning bugs, import addition (51.72% confidence) and hyperparameter initialization (55.17% confidence) also play a key role in completing the code snippet and reproducing the tensor bugs. Thus, tensor bugs can be reliably reproduced through several edits that procure the datasets, generate input data, add logging, import the required dependencies, and initialize the hyperparameters.

API Bug: Since API Bugs stem from an improper usage of application programming interfaces (APIs), they do not focus as heavily on the model training process. As a result, bug reports might lack detailed information about the input data or hyperparameters used. This encourages the use of common edit actions such as input data generation (40.00% confidence) and hyperparameter initialization (40.00% confidence) when reproducing API bugs. Additionally, code snippets in bug reports are often incomplete, necessitating edit actions like import addition (35.00% confidence) and logging (25.00% confidence) to trace the intermediate states and outputs of the API. Thus, API bugs can be reliably reproduced through several edit actions that generate input data, import the required dependencies, add logging, initialize hyperparameters and remove obsolete parameters.

Summary of RQ2: By applying the Apriori algorithm on the data produced from our successful reproduction of **148** deep learning bugs, we identified the **top 3** most important pieces of information and the **top 5** edit actions needed to reproduce each category of deep learning bug. This provides insights into the missing information that should be solicited in bug reports as well as the edit actions required to reproduce specific bug types.

4.4.3 RQ3: How do the suggested edit actions and component information affect the reproducibility of deep learning bugs?

We conduct a developer study to determine if our suggested edit actions and information help one reproduce deep learning bugs. We prepared four sets of bugs, each consisting of two different types of bugs (see Section 4.6). We randomly assigned one of the four bug sets to each participant and instructed them to reproduce the bugs in their assigned set. We divided the participants into the control and experimental

Table 4.9: Percentage of bugs successfully reproduced by control and experimental group across different sets.

	% of bugs successfully reproduced by Control group	**% of bugs successfully reproduced by Experimental group**	% Increase
Set 1	75.00	100.00	25.00
Set 2	66.66	83.33	16.66
Set 3	83.33	100.00	16.66
Set 4	66.66	100.00	33.33
Average	**72.91**	**95.83**	**22.92**

groups using stratified random sampling (see Section 4.6). The developers in both the control and experimental groups reproduced the same bugs from our prepared set of bugs. The only difference was that the experimental group received hints based on our findings to help reproduce the bugs, while the control group did not receive these hints. Since both groups received randomly selected sets from the same pool of four identical bug sets, each bug was reproduced independently by at least five developers across both control and experimental groups.

Table. 4.9 shows the bug reproducibility rate of the control and experimental groups across different sets of bugs. We found that developers in the control group could reproduce **72.91%** bugs without hints. In contrast, the developers in the experimental group could reproduce **95.83%** of bugs with our hints - a **22.92%** increase. This significant increase in reproducibility rate demonstrates that our identified edit actions and information improved developers' ability to reproduce deep learning bugs.

We also collect open-ended feedback from the participants using the free-form text boxes and performed thematic analysis to better understand their pain points in bug reproduction. Their feedback provided new insights not covered in our original study. Notably, **31.78%** of developers highlighted the *lack of standardized debugging tools* as the primary challenge in bug reproduction. Furthermore, the developers mentioned *missing information (data, logs, hyperparameters, and dependencies)*, and *lack of unit testing or version control in DL systems* as the most challenging aspects of reproducing deep learning bugs, as shown below.

Question: What are the challenges of bug reproduction in day-to-day activities?
R1: lack of debugging support and missing information
R2: data quality issues and lack of standardized debugging procedures and tools.
R3: no standardized debugging practices, and lack of clarity on the information needed, also lot of dependencies (libraries, framework, data, infra and so on)
R4: the flaky nature of deep learning models, and the unclear expectations of how model is supposed to behave.
R5: memory issues, documentation issues (missing information in issues), weak debugging support
R6: version control of deep learning is tricky, because of multiple snapshots of models, model management and reproducibility is tricky. also, the lack of standardized debugging practices makes it more tricky.
R7: distributed computing makes it difficult to find and reproduce the bugs. test coverage is also a problem as we cannot find the bugs properly because of lack of coverage and that is a problem with the reproduction of bugs.

We also analyze how our suggested actions and information help the participants in bug reproduction. According to the qualitative responses, **40.91%** of developers found our suggested edit actions to be helpful for reproducing their assigned bugs. **54.55%** of developers report that our suggested hints about the useful information helped them narrow down where to look in the code. Thus, the qualitative feedback below highlights the benefits of our findings in real-life settings.

R1: I followed the guidance in the survey and used the hints to generate the appropriate training dataset, which then allowed me to reliably reproduce the problematic behavior during model training.
R2: had to add manual imports for torch and nn, and fixed the compiler errors related to imports, as highlighted by the hint provided
R3: Following the hint, I systematically generated all of the necessary input data that would be required in order to reliably reproduce the software bug during testing.
R4: The hint mentions the Iris dataset, so I used the edit action called “Dataset

Procurement" and got the dataset, downloaded it and edited the code snippet to reproduce the bug.
R5: with the given hint, I generated the input data required for the bug reproduction.
R6: The imports were missing and as per the hint, data frame was generated for specific columns, which helped the resolution of the bug.
R7: As hinted in the survey, explicitly specifying the columns when constructing the data frame helped in the bug reproduction

We also calculate the time the control and experimental groups took to reproduce their assigned bugs. This helps us assess our findings' benefits in reducing the time required for deep learning bug reproduction. Table. 4.10 show the time taken to reproduce bugs by the control and experimental groups. The experimental group outpaced the control group in bug reproduction across all sets, with the most notable difference in Set 4, where the experimental group was **30.16%** faster. On average, the control group lagged behind the experimental group by **24.35%** across all sets. These results demonstrate that our recommended edit actions and *component information* enabled the experimental group to reproduce deep learning bugs much faster than the control group that did not receive this information.

Impact of Hints on Bug Reproducibility

To determine the impact of our hints on bug reproducibility, we constructed a generalized linear model (GLM) using the Binomial family with a logit link function [96]. This allowed us to test the statistical significance of multiple independent factors on bug reproducibility (i.e, our dependent variable). The factors were experience with deep learning bug fixing, experience with deep learning, profession, and the presence/absence of hints, with 'Hints' being our main factor of interest. The DL-BugFixExp_* factors represent the participants' experience in fixing deep learning bugs, with levels 0, 1, and 2 corresponding to experience of 0-4, 5-9, and 10+ years of experience, respectively. We chose Odds Ratio as our effect size metric for two main reasons. First, it has been commonly used in similar studies within software engineering research, as demonstrated by Ceccato et al. [26]. Second, the Odds Ratio is appropriate for a logistic regression-based model with a binary target variable, which

Table 4.10: Time taken for bug reproduction by control and experimental groups

	Average time taken by Control Group (seconds)	**Average time taken by Experimental Group (seconds)**	**% Decrease**
Set 1	1437	1088	**24.28**
Set 2	1803	1563	**13.31**
Set 3	2548	1792	**29.67**
Set 4	2165	1512	**30.16**

Table 4.11: GLM model for assessing the impact of various factors on the reproducibility of deep learning bugs

Variable	**Estimate**	**Std. Error**	**z value**	**Pr > |z|**	**Effect Size (OR)**
Intercept	-3.4229	1.898	-1.803	0.071	-
DLBugFixExp_0	0.2251	0.856	0.263	0.793	1.252493
DLBugFixExp_1	-0.2449	0.564	-0.435	0.664	0.782786
DLBugFixExp_2	-3.4032	2.283	-1.491	0.136	0.033268
Hints	3.0899	0.991	3.118	**0.002**	21.974308
DLExp	2.5448	1.725	1.475	0.140	12.740844
Field	0.1915	1.145	0.167	0.867	1.211112

we used in our study.

Based on the regression results in Table 4.11, we can see that the presence of hints had a statistically significant positive effect on the reproducibility of deep learning bugs ($p = 0.002 < .05$). The effect size, as measured by the odds ratio, indicates that the presence of hints increases the odds of reproducing a deep learning bug by a factor of 21.97 compared to the absence of hints. The intercept term, with an estimate of -3.4229 ($p = 0.071$), represents the baseline probability of reproducing a deep learning bug when no hints are provided, the participant has no experience with deep learning bug fixing, deep learning in general, and their profession is not considered. Moreover, as shown in Table. 4.11, factors like experience with deep learning bug fixing, general deep learning experience, and profession (academia vs industry) did not significantly influence reproducibility.

Overall, the above results suggest that the presence of targeted hints positively impacts the reproducibility of deep learning bugs with a statistically significant margin and a large effect size. Other factors, such as experience and profession, do not play a significant role in bug reproducibility.

Summary of RQ3: In the user study, developers were assigned to either the experimental or control group where the experimental group received our suggested edit actions and component information. The experimental group **reproduced 22.92% more deep learning bugs** and **decreased reproduction time by 24.35%** compared to the control group. This demonstrates that the identified edit actions and component information substantially improve the reproducibility rate and reduce the time needed to reproduce deep learning bugs.

4.5 Discussions

In this section, we first provide actionable insights about the reproducibility of deep learning bugs and recommend potential directions for further research (see Section 6.1). We then demonstrate how our findings can improve the reproducibility of the DL bugs with the use of large language models (e.g., Llama 3) (see Section 6.2).

4.5.1 Reproducibility of Deep Learning Bugs

From our manual reproduction and user study, we observe that *API*, *Model*, and *Tensor* bugs are relatively more straightforward to reproduce. This behaviour can be explained by the fact that these bugs are more specific to the location in the code where they originate. For example, faulty input data usually triggers tensor bugs, whereas incorrect usage of a framework's API causes API bugs. In contrast, training bugs cover multiple issues related to deep learning model training, and GPU bugs relate to GPU devices for deep learning and manifest across GPU interactions.

This behaviour is further supported by the reproducibility rates and efforts involved in reproducing different types of bugs. The reproducibility rates of Training bugs and GPU bugs were 89.65% and 42.85%, respectively. On the other hand, the reproducibility rates for API, Model and Tensor bugs were 81.81%, 88.46%, and 80.75%, respectively. Even though the training bugs could be reproduced reliably, the efforts involved in reproducing the training bugs were significantly more than those of other bugs. The average time to reproduce the API, Model and Tensor bugs was 45.5, 43.9 and 43.8 minutes, respectively. On the other hand, the average time to reproduce

the Training bugs was 52.5 and 57.33 minutes, respectively. These statistics highlight that we need to put more effort into reproducing the Training and GPU bugs, which calls for further research into the specific nature of Training and GPU bugs.

To assist future research in the reproducibility of deep learning bugs, we provide directions for future research below:

Understanding Training Bug Reproducibility: Training bugs are the most common type of bug in deep learning systems, accounting for 52.5% of all bugs [54]. Training bugs have high reproducibility rates but also require significant effort to reproduce. This behaviour presents an opportunity to understand better what factors make training bugs more reproducible or harder to reproduce. By analyzing the training procedures, model architectures, optimizers, hyperparameters, and other elements that either aid or hinder reproducibility, we can uncover insights to guide the diagnosis and repair of training bugs. There is also room to develop improved tools and methodologies explicitly focused on efficiently reproducing the nuanced nature of training bugs.

Analyzing the GPU Bug Reproducibility Gap: The lower reproducibility rate for GPU bugs highlights a gap in understanding the interactions between deep learning code and the underlying GPU hardware or drivers. By further studying irreproducible GPU bugs and quantifying the aspects that impede reproducibility, such as hardware differences, software dependencies, and environmental factors, we can work towards solutions to increase reproducibility. Opportunities exist to build infrastructure, leverage containerization or virtualization, and create testing tools to better control for sources of non-determinism that impact GPU bug reproduction.

Reproducible Testbeds for Deep Learning: Ultimately, the variability in reproducing different categories of deep learning bugs motivates the need for reproducible testbeds. Shared sets of reproducible deep learning bugs with associated test cases, model architectures, training configurations, dependencies, and environmental contexts will accelerate future research. Constructing such testbeds requires systematic characterization of how these factors influence reproducibility. Reproducible testbeds also support developing specialized techniques for efficiently reproducing and debugging deep learning bugs.

4.5.2 Challenges in Reproducing Deep Learning Bugs: A Comparison between Stack Overflow and GitHub

Reproducing deep learning bugs from Stack Overflow posts and GitHub repositories presents different challenges and requires varying levels of effort. The main differences lie in the availability of code and data, environmental setup, context, completeness, and reproducibility expectations. We explain them briefly as follows.

First, GitHub repositories often provide the complete codebase and sometimes the accompanying datasets, whereas Stack Overflow posts typically include small code snippets without the broader context and data. Hence, we need more effort in reconstructing the code and generating synthetic data when reproducing bugs from Stack Overflow. Second, GitHub repositories commonly include setup instructions, dependency files, and development environment configurations, making it easier to recreate the original environment accurately. On the contrary, Stack Overflow posts rarely provide such details, requiring guesswork about software versions, dependencies, and environment settings. Finally, GitHub issues tend to have more contextual information, such as detailed problem descriptions, steps to reproduce a bug, and error logs supporting root cause analysis of the issue. Stack Overflow posts may lack this level of detail, making it harder to comprehend and reproduce a bug accurately.

Despite the additional resources available in GitHub repositories, reproducing bugs from them still presents several challenges:

- *Incomplete or outdated repositories*: Critical files, dependencies, or code changes may be missing or outdated, making it difficult to reproduce the exact environment and conditions under which a bug occurred.

- *Large and complex codebases*: Deep learning projects often have extensive and intricate codebases, requiring significant time and expertise to set up and navigate them.

- *Proprietary or sensitive data*: Many projects may involve proprietary data that cannot be shared publicly, making it challenging to reproduce data-related bugs without access to the original data.

- *Insufficient documentation*: Many GitHub repositories fail to provide clear documentation on project architecture, installation steps, and expected behaviours. The lack of clear documentation makes it difficult for developers to reproduce bugs, as they face challenges in understanding the codebase and configuring the project locally.

- *Dependency management*: Resolving dependency conflicts or managing compatibility issues across different project versions can be a significant challenge when reproducing bugs.

To address these challenges, we recommend several changes or adaptations to the status quo as follows. First, Stack Overflow should leverage its user base and search functionality for discussing common deep-learning bugs and quick solutions, while GitHub should be the primary platform for in-depth bug reproduction and resolution. Second, Stack Overflow should introduce a specialized format for deep learning bug reports, expanding on its traditional minimum working example approach to include dataset details, framework versions, hardware specifications, and reproducible code snippets. GitHub repositories should be kept up-to-date, well-documented, and complete. Finally, both platforms should incentivize active participation in bug reporting and resolution through specialized badges, reputation points, or recognition. Stack Overflow could introduce "Deep Learning Debug Rooms" for real-time, collaborative problem-solving, and GitHub could highlight top contributors in bug resolution. By combining Stack Overflow's community-driven approach with GitHub's comprehensive project management features, we can develop a more efficient ecosystem for deep learning bug resolution, leading to more robust deep learning systems.

4.6 Threats to Validity

Threats to *external validity* relate to the generalizability of our findings. We have reproduced multiple bugs from five different types to mitigate this threat. Moreover, we have reproduced bugs from 14 different architectures, ranging from Logistic Regression, CNN, Transformers and BERT-based architectures. Finally, we have also reproduced the bugs from the past six years (2017-2023), and our findings align with

and extend the previous findings of software bug reproducibility [89], possibly indicating the generalizability of our study.

Threats to *internal validity* pertain to experimental errors and confounding variables during the reproduction of bugs. The possibility of introducing new bugs into the code exists during bug reproduction. We have implemented several strategies and precautions to mitigate these threats in our experimental design. For instance, bug reproduction attempts are carried out in controlled and isolated Python virtual environments. We refrain from using hardcoded or fixed values in our code snippets to prevent potential errors during bug reproduction. For instance, during the bug reproduction, we utilize random hyperparameter initialization, construct multiple neural networks, and generate randomized input data that aligns with the original distribution. This methodology mitigates the influence of unforeseen variables on the outcomes of our experiments. To further mitigate this threat, we ran the updated code snippets five times for all the edit actions. Following bug reproduction, we undertake rigorous manual analysis to identify any discrepancies or anomalies in the outcomes.

A potential threat to the validity of our findings arises from the inclusion of context-specific, natural language hints from bug reports. This could potentially introduce bias when assessing the experimental group's performance. However, our analysis of qualitative feedback from participants indicates that participants primarily benefited from the hints derived from our Apriori algorithm (RQ2), with minimal mention of context-specific hints from the bug report. In particular, the participants reported using the guidance to generate appropriate datasets, address import issues, create necessary input data, and apply specific edit actions, which are closely tied to our algorithm's outputs rather than the natural language hints. Thus, the threat posed by the inclusion of natural language hints might be negligible.

Another threat to validity is the incomplete or incorrect information in the bug report, which could impact the applicability of certain edit actions suggested by our technique. In particular, edit actions that require additional context from the report, such as those involving input data generation, neural network construction, dataset procurement, and downloading models/tokenizers, may be limited by missing details in the reports. However, the majority of our suggested edit actions focus on code

modifications that can be derived from the provided code snippets without relying heavily on external context. Therefore, while missing information may constrain our capability for data and model-driven actions, our core technique and findings remain widely applicable for many code-focused edit recommendations. To mitigate this threat further, we apply extensive filtration based on the criteria suggested by Moravati et al. [93] and Humbatova et al. [54], which leads to a clean and generalizable dataset.

Finally, the way in which the contextual information is presented to the developers could potentially threaten the validity of the control and experiment groups in the user study. To mitigate this threat, we ensure that the contextual information is presented in an identical format to the control and experiment groups.

Threats to *construct validity* relate to the use of appropriate metrics for evaluating the results of a user study. In the user study, we used the reproducibility rate and time to reproduce the deep learning bugs as the metrics to evaluate the benefits of our findings. To measure these accurately, we had participants justify why and how they used the edit actions or found certain information important. We verified these justifications against our ground truth to determine if the bug was successfully reproduced. After checking the user justifications, we calculated the reproducibility rate and spent time. By including user justifications and verifying them, we ensured that the metrics directly measured how easily bugs could be reproduced. We also had clear criteria for determining reproduction success and used Opinio for precise timing data [3], mitigating threats related to measurement errors or subjective assessments.

Another threat to the validity might be the accuracy of tags used to categorize the bugs. The incorrect categorization of bugs might lead to incorrect conclusions for different bugs. To mitigate this threat, we only select the tags that are present in the taxonomy and the sub-taxonomies defined by Humbatova et al. [54]. For example, we specifically chose the "loss-function" as one of the tags to distinguish the training bugs since the Loss Function falls within the sub-taxonomy of Training bugs. Thus, the threats to construct validity were effectively mitigated.

4.7 Related Work

There have been several studies that focus on the aspects of reproducibility of software bugs [89, 91, 92], or focus on why some bugs cannot be reproduced [109, 108]. Many studies attempt to learn the nature of deep learning bugs [152, 57, 54, 83], and how they can be localized automatically [58, 143, 153]. A few studies also attempt to learn about the state of reproducibility of deep learning in software engineering [28, 73], and reproduce deep learning bugs as a part of benchmark dataset creation [93]. Unfortunately, only a little research has been done to understand the challenges in reproducing deep learning bugs and how we can improve their reproducibility.

Mondal et al. [89] extensively investigated the reproducibility of programming issues reported on Stack Overflow. They identified several key edit actions required to reproduce programming errors and traditional software bugs. While their work is a source of inspiration, it does not provide the edit actions for reproducing deep learning bugs. Since their dataset is constructed in Java programming language with no questions related to deep learning, their findings might not apply to deep learning bugs.

Chen et al. [28] proposed a unified framework to train reproducible deep learning models. They also provide reproducibility guidelines and mitigation strategies for conducting a reproducible training process for DL models. However, the primary focus of their study was on the reproducibility of deep learning models, not deep learning bugs. Furthermore, the guidelines for deep learning models might not always extend to deep learning bugs.

Moravati et al. [93] constructed the faultload benchmark containing deep learning bugs, *defects4ML*. As a part of their benchmark dataset, they reproduced 100 bugs as a part of the reproducibility criterion for benchmark datasets. While their study was the first to reproduce deep learning bugs actively, they did not report the techniques and actions used to reproduce bugs. Furthermore, they did not report the useful information from the issue reports, which helped them reproduce the bugs.

Unlike many earlier studies above, we conduct an extensive empirical study to understand the challenges of deep learning bug reproduction and how we can improve the reproducibility of deep learning bugs. We construct a dataset of 668 bugs and manually reproduce 165 bugs, spanning three frameworks, five types of bugs, and 14

architectures. We not only (1) define ten edit actions which can be used to reproduce deep learning bugs but also (2) explore the associations among the type of bugs, the information required to reproduce them, and the specific edit actions that can be used to reproduce them, which makes our work novel. Our findings are also generalizable to Tensorflow and PyTorch bugs due to the diversity of our dataset. To ensure transparency and reproducibility, we have made our dataset and replication package publicly available[8].

4.8 Summary

The reproducibility of deep learning bugs remains a significant challenge for software practitioners, particularly given their low success rate of reproducibility (e.g., 3%) and their potential for severe consequences. Through our empirical study, we reproduced 148 deep learning bugs across three frameworks and 22 architectures, identifying ten key edit actions and component information essential for bug reproduction. Our investigation revealed important associations among bug types, component information, and edit actions influencing reproduction success. The practical value of our findings was assessed through a user study involving 22 developers, where participants using our recommended edit actions and information reproduced 22.92% more bugs, spending 24.35% less time. Our work advances the current state of knowledge by providing actionable insights into deep learning bug reproduction and also empowers software practitioners with concrete strategies to tackle the challenges of bug reproduction and subsequent correction. Chapter 5 concludes our report and discusses the avenues for future work.

[8]https://github.com/mehilshah/Bug_Reproducibility_DL_Bugs

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Software practitioners and AI engineers encounter significant challenges when reproducing bugs in deep learning systems. Recent studies reveal that only 3% of deep learning bugs are reproducible, demonstrating the challenges in their reproduction. These bugs can originate from various sources, including mislabelled training data, faulty code, hardware problems, framework incompatibilities, and environment configurations. Moreover, the inherent non-determinism and data-driven nature of deep learning systems further complicate the reproduction of these bugs. Despite significant research efforts and advancements in recent years, we identify two major gaps in the literature. First, there is a lack of comprehensive understanding of how data bugs, one of the most prevalent types of deep learning bugs, manifest in deep learning systems. Such an understanding is crucial for the successful reproduction of data bugs. Second, no existing studies examine the reproducibility challenges of deep learning bugs. Addressing these knowledge gaps is essential for developing effective tools or methods supporting bug reproduction in deep learning applications.

In this RAD report, we conduct two empirical studies to address the above gaps from the literature. Our first study examines how data quality problems manifest themselves during the training of deep learning models. We also analyze how their symptoms and manifestation vary across three types of software engineering data – code, text, and metric. Our findings reveal that code-based issues lead to gradient instability, text-based problems cause abnormal weight distributions, and metric-based issues result in vanishing gradients. These findings advance our understanding of the data bugs in deep learning systems. Our second study focuses on bug reproduction, where we manually reproduce 148 deep learning bugs, spending ≈280 person-hours

and identify ten essential edit actions and five pieces of component information. Using the Apriori algorithm, we also establish associations among bug types, component information, and necessary edit actions, enabling automated recommendation. Then, we assess the recommended items through a developer study involving 22 participants, where developers using our recommended actions successfully reproduced 22.92% more bugs, spending 24.35% less time.

In summary, our research systematically analyzes how data bugs manifest during model training across different types of software engineering data (code-based, text-based, and metric-based) and provides detailed insights into models' gradient behaviours, weight distributions, and representations. Through our novel framework leveraging the Apriori algorithm, we provide comprehensive guidelines for reproducing deep learning bugs with specific edit actions and component information, which was not offered by any of the existing works.

5.2 Limitations

We discuss the limitations of our conducted studies as follows:

5.2.1 Limitations of Study 1: Understanding Data Bug Symptoms

Our first study on understanding data bug symptoms has the following limitations:

Dataset Coverage. While we analyzed three types of software engineering data (code-based, text-based, and metric-based), our datasets were primarily from open-source projects. The findings may vary for proprietary software systems or different application domains.

Model Selection. We focused on specific state-of-the-art models [42, 51, 52] for each data type. Though we validated our findings with alternative models, the manifestation patterns might differ for other model architectures or newer variants.

Hardware Environment. Our experiments were conducted using specific GPU configurations from Compute Canada clusters [17]. The training behaviours and bug manifestations might show slight variations under different hardware setups.

5.2.2 Limitations of Study 2: Enhancing Bug Reproducibility

Our second study on bug reproducibility has the following limitations:

Dataset Sample Size. Although we analyzed 668 deep learning bugs from Stack Overflow and GitHub, they represent a subset of all possible bug patterns in deep learning systems. The distribution of bug types might differ in other contexts or repositories.

Framework Versions. Our reproduction guidelines were developed based on specific versions of deep learning frameworks. Some details might need adjustment for newer framework versions as the ecosystem evolves.

User Study Scale. The user study involved 22 developers, which while informative, represents a modest sample size. A larger-scale study might reveal additional insights or variations in findings.

Time Frame. The bug reports we analyzed were collected within a specific time frame (2017 to 2023). As deep learning technologies evolve, new types of bugs and reproduction challenges might emerge.

These limitations present natural opportunities for future research and will encourage further work.

5.3 Future Work

Building upon our empirical studies on understanding and reproducing deep learning bugs, we anticipate several promising directions for future work. These directions target standardization and automation to tackle data quality issues and reproduction challenges in deep learning bugs.

5.3.1 Analysis and Verification of Bug Manifestations

Our investigation of data quality issues across code-based, text-based, and metric-based data revealed distinct manifestation patterns for each type. To leverage these findings, future work should target automated detection and verification of these patterns. This includes developing pattern recognition algorithms that can monitor a deep learning model under training and identify the gradient instability in code-based issues, abnormal weight distributions in text-based problems, and vanishing

gradients in metric-based cases. Based on our investigation across multiple datasets, we suggest that early warning systems could be useful in monitoring these indicators during training. Additionally, given our manual reproduction of 148 bugs, we believe that there is a need for appropriate metrics that can quantify observed patterns in various property distributions (e.g., gradients, weights, and biases), which can standardize and support the detection of specific bugs across different deep learning frameworks.

5.3.2 Improving Bug Reporting

Our user study involving 22 participants demonstrates that complementary information and guidance significantly improve bug reproduction (Chapter 4). To build on this success, future research can focus on developing effective templates for bug reporting and capturing essential debugging information. These templates should systematically document the specific component information required for bug reproduction, including model configurations, data preprocessing steps, and environment settings. They should also incorporate sections for recording the relevant edit actions identified in our study, such as parameter adjustments and data transformations. While our work provides initial key elements to formulate such a template, further research is needed to validate the template and its components across different frameworks and development contexts. They should focus on the usability and practicality of such templates. Moreover, additional investigation is needed to understand how these templates should be adapted for different types of deep learning bugs, given the distinct symptoms observed across code-based, text-based, and metric-based issues (Chapter 3).

5.3.3 Generating Reproducibility Scripts for Bug Reports

Our study found that developers reproduced 22.92% more bugs when provided with a combination of code snippets and structured guidance (e.g., relevant edit actions). Our findings highlight that code alone is insufficient; successful bug reproduction requires the synergy between clear instructions, environment specifications, and executable scripts. Given the complexity of the whole reproduction process, reproduction scripts might be an appropriate option to tackle the challenge. Our analysis indicates

that effective reproduction scripts should contain three key elements: (a) executable code that triggers the bug, (b) environment setup instructions, including dependencies and configurations, and (c) relevant edit actions and component information for reproduction. To ensure consistent reproduction across different development setups, the scripts should be designed to run in a containerized environment that can be automatically provisioned. Such an automation could help standardize the bug reproduction workflow and help the software practitioners and AI engineers reproduce their bugs effectively.

5.4 Our Future Research Plan

Based on our empirical findings and identified limitations, we plan to develop three interconnected automated solutions to enhance the reproducibility of deep learning bugs:

5.4.1 Minimal Working Example Generator

As our first step, we will develop an automated technique to generate or synthesize minimal working examples from bug reports. It will address the challenge of verbose and overcomplicated bug reports by automatically extracting and synthesizing essential code components. By leveraging natural language processing and code analysis techniques, our tool will distil bug reports into concise, self-contained examples demonstrating the core issue. This solution has the potential to significantly reduce debugging time and improve bug reproduction success rates across the developer community. The minimal working examples generated by this tool will serve as the foundation for our more comprehensive reproducibility solutions.

5.4.2 Reproducibility Script Generator

Building upon the minimal working examples, we will create an automated system for generating complete reproduction scripts from bug reports. This system will expand on the concise examples by incorporating all necessary context and setup information. By combining our minimal working examples with automated environment detection and configuration generation, the system will produce comprehensive

scripts that capture the complete reproduction workflow. The potential impact includes standardized bug reproduction processes, reduced manual effort, and improved communication between bug reporters and developers. These scripts will be designed to work seamlessly with our containerized environment, forming a complete reproduction pipeline.

5.4.3 Containerized Reproduction Environment

To complete our reproduction ecosystem, we will develop a standardized containerized solution that will execute our generated scripts in a controlled environment. This final piece will address the challenges of environment-dependent bugs and inconsistent reproduction results by providing a consistent platform for reproducing deep learning bugs across different hardware configurations and framework versions. The environment will be specifically designed to support both our minimal working examples and comprehensive reproduction scripts, ensuring reliable execution regardless of the user's local setup. The potential impact includes eliminating environment-related reproduction failures and enabling seamless sharing of reproduction environments within the development community. Together, these three solutions will form an end-to-end system for reliable deep learning bug reproduction, from initial bug report to verified reproduction.

Bibliography

[1] Federal transportation agency finds tesla's claims about feature don't match their findings and opens second investigation. https://www.theguardian.com/technology/2024/apr/26/tesla-autopilot-fatal-crash. Accessed on 25 Apr, 2024.

[2] Open computer vision library. https://opencv.org/. Accessed on 29 June, 2024.

[3] Opinio survey software. https://surveys.dal.ca/opinio/admin/folder.do. Accessed on 30 August, 2024.

[4] Python pillow official site. https://python-pillow.org/. Accessed on 27 June, 2024.

[5] Get confusion matrix from a keras multiclass model. https://stackoverflow.com/q/50920908, 2018. Accessed on December 28, 2023.

[6] data_utils.is_generator_or_sequence returns always false. https://stackoverflow.com/q/58190114, 2019. Accessed on January 3, 2024.

[7] Peter Martey Addo, Dominique Guegan, and Bertrand Hassani. Credit risk analysis using machine and deep learning models. *Risks*, 6(2):38, 2018.

[8] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[9] Saeed S Alahmari, Dmitry B Goldgof, Peter R Mouton, and Lawrence O Hall. Challenges for the repeatability of deep learning models. *IEEE Access*, 8:211860–211868, 2020.

[10] Alberto Amato and Vincenzo Di Lecce. Data preprocessing impact on machine learning algorithm performance. *Open Computer Science*, 13(1):20220278, 2023.

[11] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

[12] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. ICSE*, pages 361–370, 2006.

[13] Apache. Apache jira. https://issues.apache.org/jira/projects/HADOOP/issues. Accessed on April 28, 2024.

[14] A. Arcuri. On the automation of fixing software bugs. In *ICSE*, pages 1003–1006, 2008.

[15] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[16] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*, pages 50–59. IEEE, 2018.

[17] Susan Baldwin. Compute canada: advancing computational research. In *Journal of Physics: Conference Series*, volume 341, page 012001. IOP Publishing, 2012.

[18] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[19] God Bennett. Answer to 'what is the role of the bias in neural networks?". https://stackoverflow.com/a/47118013, Nov 2017. Accessed on September 3, 2024.

[20] Daniel S Berman, Anna L Buczak, Jeffrey S Chavis, and Cherita L Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4):122, 2019.

[21] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *Journal of Systems and Software*, 164:110542, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0164121220300248.

[22] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data validation for machine learning. In *MLSys*, 2019.

[23] Jason Brownlee. A gentle introduction to exploding gradients in neural networks. https://machinelearningmastery.com/exploding-gradients-in-neural-networks/, Dec 2017. Accessed on September 6, 2024.

[24] Richard L Burden and J Douglas Faires. Numerical analysis, brooks, 1997.

[25] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, et al. A systematic literature review on explainability for machine/deep learning-based software engineering research. *arXiv preprint arXiv:2401.14617*, 2024.

[26] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19:1040–1074, 2014.

[27] Boyuan Chen and Zhen Ming (Jack) Jiang. A survey of software log instrumentation. *ACM Computing Surveys*, 54(4):1–34, May 2022. URL: `https://dl.acm.org/doi/10.1145/3448976`.

[28] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming (Jack) Jiang. Towards training reproducible deep learning models. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2202–2214, New York, NY, USA, 2022. Association for Computing Machinery. URL: `https://doi.org/10.1145/3510003.3510163`.

[29] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Trang Pham, Aditya Ghose, and Tim Menzies. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 45(7):637–656, 2018.

[30] William Gemmell Cochran. *Sampling techniques*. john wiley & sons, 1977.

[31] Pierre-Olivier Côté, Amin Nikanjam, Nafisa Ahmed, Dmytro Humeniuk, and Foutse Khomh. Data cleaning and machine learning: a systematic literature review. *Automated Software Engineering*, 31(2):54, 2024.

[32] Domenico Cotroneo, Roberto Pietrantuono, Stefano Russo, and Kishor Trivedi. How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software*, 113:27–43, 2016.

[33] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 121–133. IEEE, 2023.

[34] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering*, 49(3):1044–1063, 2022.

[35] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 95–105, 2018.

[36] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24–29, 2019.

[37] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[38] Yuanrui FAN, Xin XIA, Daniel A COSTA, David LO, Ahmed E HASSAN, and Shanping LI. The impact of changes mislabeled by szz on just-in-time defect prediction.(2019). *IEEE Transactions on Software Engineering*, pages 1–26, 2019.

[39] Benoît Frénay, Ata Kabán, et al. A comprehensive introduction to label noise. In *ESANN*. Citeseer, 2014.

[40] Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.

[41] Michael Fu and Chakkrit Tantithamthavorn. Gpt2sp: A transformer-based agile story point estimation approach. *IEEE Transactions on software engineering*, 49(2):611–625, 2022.

[42] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[43] GeeksforGeeks. Best ides for machine learning. `https://www.geeksforgeeks.org/best-ides-for-machine-learning/`. Accessed on June 29, 2024.

[44] GeeksforGeeks. Best ides for machine learning. `https://www.geeksforgeeks.org/best-ides-for-machine-learning/`. Accessed on December 25, 2023.

[45] Görkem Giray, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur, and Bedir Tekinerdogan. On the use of deep learning in software defect prediction. *Journal of Systems and Software*, 195:111537, 2023.

[46] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[47] Ian Goodfellow. *Deep learning*, volume 196. MIT press, 2016.

[48] Marco Gori, Alessandro Betti, and Stefano Melacci. *Machine Learning: A constraint-based approach*. Elsevier, 2023.

[49] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[50] Roger Grosse. Lecture 15: Exploding and vanishing gradients. *University of Toronto Computer Science*, 2017.

[51] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 117–127, 2020.

[52] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[53] Shuo Hong, Hailong Sun, Xiang Gao, and Shin Hwei Tan. Investigating and detecting silent bugs in pytorch programs. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 272–283. IEEE, 2024.

[54] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20.

[55] Ivana Clairine Irsan, Ting Zhang, Ferdian Thung, Kisub Kim, and David Lo. Multi-modal api recommendation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 272–283. IEEE, 2023.

[56] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 510–520, Tallinn Estonia, August 2019. ACM. URL: `https://dl.acm.org/doi/10.1145/3338906.3338955`.

[57] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. ESEC/FSE 2019, page 510–520, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi.org/10.1145/3338906.3338955`.

[58] Foad Jafarinejad, Krishna Narasimhan, and Mira Mezini. Nerdbug: Automated bug detection in neural networks. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*, AISTA 2021, page 13–16, New York, NY, USA, 2021. Association for Computing Machinery. URL: `https://doi.org/10.1145/3464968.3468409`.

[59] Sigma Jahan, Mehil B Shah, and Mohammad Masudur Rahman. Towards understanding the challenges of bug localization in deep learning systems. *arXiv preprint arXiv:2402.01021*, 2024.

[60] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[61] Hossein Keshavarz and Meiyappan Nagappan. Apachejit: a large dataset for just-in-time defect prediction. In *Proceedings of the 19th international conference on mining software repositories*, pages 191–195, 2022.

[62] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 481–490, 2011.

[63] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International conference on machine learning*, pages 1885–1894. PMLR, 2017.

[64] Maya Krishnan. Against interpretability: a critical examination of the interpretability problem in machine learning. *Philosophy & Technology*, 33(3):487–502, 2020.

[65] Tuan Dung Lai. Towards the generation of machine learning defect reports. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1038–1042. IEEE, 2021.

[66] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Generating duplicate bug datasets. In *Proceedings of the 11th working conference on mining software repositories*, pages 392–395, 2014.

[67] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

[68] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 249–260. IEEE, 2017.

[69] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. Robust learning of deep predictive models from noisy and imbalanced software engineering datasets. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[70] Weixin Liang, Girmaw Abebe Tadesse, Daniel Ho, Li Fei-Fei, Matei Zaharia, Ce Zhang, and James Zou. Advances, challenges and opportunities in creating data for trustworthy ai. *Nature Machine Intelligence*, 4(8):669–677, 2022.

[71] Yunkai Liang, Yun Lin, Xuezhi Song, Jun Sun, Zhiyong Feng, and Jin Song Dong. gdefects4dl: a dataset of general real-world deep learning program defects. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 90–94, 2022.

[72] Bing Liu, Wynne Hsu, and Yiming Ma. Mining association rules with multiple minimum supports. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 337–341, 1999.

[73] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. On the reproducibility and replicability of deep learning in software engineering. *ACM Trans. Softw. Eng. Methodol.*, 31(1), oct 2021. URL: `https://doi.org/10.1145/3477535`.

[74] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. Deep learning based code smell detection. *IEEE transactions on Software Engineering*, 47(9):1811–1837, 2019.

[75] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, Online, November 2020. Association for Computational Linguistics. URL: `https://aclanthology.org/2020.emnlp-main.463`.

[76] Yanbin Liu, Wen Zhang, Guangjie Qin, and Jiangpeng Zhao. A comparative study on the effect of data imbalance on software defect prediction. *Procedia Computer Science*, 214:1603–1616, 2022.

[77] Guoming Long and Tao Chen. On reporting performance and accuracy bugs for deep learning frameworks: An exploratory study from github. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pages 90–99, 2022.

[78] José Antonio Hernández López, Boqi Chen, Tushar Sharma, and Dániel Varró. On inter-dataset code duplication and data leakage in large language models. *arXiv preprint arXiv:2401.07930*, 2024.

[79] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 120–131, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3238147.3238202`.

[80] Minghua Ma, Shenglin Zhang, Dan Pei, Xin Huang, and Hongwei Dai. Robust and rapid adaption for concept drift in software system anomaly detection. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 13–24. IEEE, 2018.

[81] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. Defectors: A large, diverse python dataset for defect prediction. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 393–397. IEEE, 2023.

[82] Zaheed Mahmood, David Bowes, Peter CR Lane, and Tracy Hall. What is the impact of imbalance on software defect prediction performance? In *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*, pages 1–4, 2015.

[83] Tarek Makkouk, Dong Jae Kim, and Tse-Hsun Peter Chen. An empirical study on performance bugs in deep learning frameworks. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46, 2022.

[84] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on cross-project predictive mutation testing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 160–171. IEEE, 2019.

[85] Machine Learning Mastery. Building a logistic regression classifier. `https://machinelearningmastery.com/building-a-logistic-regression-classifier-in-pytorch/`. Accessed on 21 June, 2024.

[86] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[87] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In *Proceedings of the 40th international conference on software engineering*, pages 560–560, 2018.

[88] Montassar Ben Messaoud, Asma Miladi, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Lobna Ghadhab. Duplicate bug report detection using an attention-based neural language model. *IEEE Transactions on Reliability*, 2022.

[89] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K. Roy. Can issues reported at stack overflow questions be reproduced? an exploratory study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 479–489, 2019.

[90] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K Roy. Can we identify stack overflow questions requiring code snippets? investigating the cause & effect of missing code snippets. *arXiv preprint arXiv:2402.04575*, 2024.

[91] Saikat Mondal, Mohammad Masudur Rahman, Chanchal K Roy, and Kevin Schneider. The reproducibility of programming-related issues in stack overflow questions. *Empirical Software Engineering*, 27(3):62, 2022.

[92] Saikat Mondal and Banani Roy. Reproducibility of issues reported in stack overflow questions: Challenges, impact & estimation. *Impact & Estimation*.

[93] Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming (Jack) Jiang. Bugs in machine learning-based systems: A faultload benchmark. *Empirical Softw. Engg.*, 28(3), apr 2023. URL: `https://doi.org/10.1007/s10664-023-10291-1`.

[94] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. The impact of nondeterminism on reproducibility in deep reinforcement learning. 2018.

[95] Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. A modern take on the bias-variance tradeoff in neural networks. *arXiv preprint arXiv:1810.08591*, 2018.

[96] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 135(3):370–384, 1972.

[97] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. Efficient nearest-neighbor data sharing in gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–26, 2020.

[98] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 52–63, 2023.

[99] Youssef Esseddiq Ouatiti, Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Ahmed E Hassan. The impact of concept drift and data leakage on log level prediction models. *Empirical Software Engineering*, 29(5):1–37, 2024.

[100] R Pascanu. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2013.

[101] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. *Commun. ACM*, 62(11):137–145, oct 2019. URL: `https://doi.org/10.1145/3361566`.

[102] Matilda QR Pembury Smith and Graeme D Ruxton. Effective use of the mcnemar test. *Behavioral Ecology and Sociobiology*, 74:1–9, 2020.

[103] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan 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*, pages 771–783, 2020.

[104] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software*, pages 343–352, 2014.

[105] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*, 49(1):84–98, 2022.

[106] PyTorch. Pytorch v1.6. `https://pytorch.org/docs/1.6.0/`. Accessed on 12 June, 2024.

[107] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[108] Mohammad M Rahman, Foutse Khomh, and Marco Castelluccio. Works for me! cannot reproduce–a large scale empirical study of non-reproducible bugs. *Empirical Software Engineering*, 27(5):111, 2022.

[109] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. Why are some bugs non-reproducible? : –an empirical investigation using data fusion–. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 605–616, 2020.

[110] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14:131–164, 2009.

[111] Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. "everyone wants to do the model work, not the data work": Data cascades in high-stakes ai. In *proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[112] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[113] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. 2015.

[114] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[115] Mehil Shah. mehilshah/bug_reproducibility_dl_bugs. `https://github.com/mehilshah/Bug_Reproducibility_DL_Bugs`. Accessed on January 3, 2024.

[116] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[117] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[118] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *Empirical Software Engineering*, 25:5255–5294, 2020.

[119] Qinbao Song, Yuchen Guo, and Martin Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269, 2018.

[120] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[121] Stack Exchange. All sites - stack exchange. `https://stackexchange.com/sites?view=list`, 2023. Accessed on December 12, 2023.

[122] Stack Exchange Inc. Stack Exchange Data Explorer. `https://data.stackexchange.com/`, 2023. Accessed on December 7, 2023.

[123] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX network programming. 1: The sockets networking API / W. Richard Stevens; Bill Fenner; Andrew M. Rudoff.* Addison-Wesley professional computing series. Addison-Wesley, Boston, 3rd ed edition, 2013.

[124] Magdalena Szumilas. Explaining odds ratios. *Journal of the Canadian academy of child and adolescent psychiatry*, 19(3):227, 2010.

[125] Deepak Talwar, Sachin Guruswamy, Naveen Ravipati, and Magdalini Eirinaki. Evaluating validity of synthetic data in perception tasks for autonomous vehicles. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 73–80. IEEE, 2020.

[126] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. *Empirical Software Engineering*, 29(1):10, 2024.

[127] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823. IEEE, 2015.

[128] Keras Team. Keras documentation: Python & numpy utilities. `https://keras.io/2.16/api/utils/python_utils/`. Accessed on 6 June, 2024.

[129] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 981–992, 2020.

[130] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW)*, page 1–5, April 2015. URL: `https://ieeexplore.ieee.org/abstract/document/7133169`.

[131] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[132] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, pages 2291–2302, 2022.

[133] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[134] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[135] Daisuke Wakabayashi. Self-driving uber car kills pedestrian in arizona, where robots roam. `https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html`, Mar 2018. Accessed on December 17, 2023.

[136] Wenhan Wang, Yanzhou Li, Anran Li, Jian Zhang, Wei Ma, and Yang Liu. An empirical study on noisy label learning for program understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. URL: `https://doi.org/10.1145/3597503.3639217`.

[137] Moshi Wei, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. Demystifying and detecting misuses of deep learning apis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[138] Weights and Biases. Weights and biases: An ai developer platform. `https://wandb.ai/site`, 2021. Accessed 21 February, 2024.

[139] Steven Euijong Whang and Jae-Gil Lee. Data collection and quality challenges for deep learning. *Proceedings of the VLDB Endowment*, 13(12):3429–3432, 2020.

[140] Xiaoxue Wu, Wei Zheng, Xin Xia, and David Lo. Data quality matters: A case study on data label correctness for security bug report prediction. *IEEE Transactions on Software Engineering*, 48(7):2541–2556, 2021.

[141] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Jingwei Xu, and Xiaoxing Ma. Data quality matters: A case study of obsolete comment detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 781–793. IEEE, 2023.

[142] Zhengkang Xu, Shikai Guo, Yumiao Wang, Rong Chen, Hui Li, Xiaochen Li, and He Jiang. Code comment inconsistency detection based on confidence learning. *IEEE Transactions on Software Engineering*, 2024.

[143] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan 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*, pages 627–638, 2021.

[144] Yanming Yang, Xin Xia, David Lo, and John Grundy. A survey on deep learning for software engineering. *ACM Comput. Surv.*, 54(10s), sep 2022. URL: `https://doi.org/10.1145/3505243`.

[145] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology*, 151:107004, 2022.

[146] Robert K Yin. *Case study research: Design and methods*, volume 5. sage, 2009.

[147] Yining Yin, Yang Feng, Shihao Weng, Zixi Liu, Yuan Yao, Yichi Zhang, Zhihong Zhao, and Zhenyu Chen. Dynamic data fault localization for deep neural networks. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1345–1357, 2023.

[148] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 427–438, New York, NY, USA, 2021. Association for Computing Machinery. URL: `https://doi.org/10.1145/3460319.3464819`.

[149] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 29–41, 2020.

[150] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–115. IEEE, 2019.

[151] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. Duplicate bug report detection: How far are we? *ACM Transactions on Software Engineering and Methodology*, 32(4):1–32, 2023.

[152] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 129–140, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3213846.3213866`.

[153] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao 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*, pages 826–837, 2020.

[154] Henghui Zhao, Yanhui Li, Fanwei Liu, Xiaoyuan Xie, and Lin Chen. State and tendency: an empirical study of deep learning question&answer topics on stack overflow. *Science China Information Sciences*, 64:1–23, 2021.

[155] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

Appendix A

Supplementary details

The following replication packages contain all datasets, code, and materials needed to reproduce the research findings from two of our studies on deep learning bugs.

A.1 Towards Understanding the Impact of Data Bugs on Deep Learning Models in Software Engineering

Replication Package: `https://shorturl.at/IvNTN`

A.2 Towards Enhancing the Reproducibility of Deep Learning Bugs: An Empirical Study

Replication Package: `https://shorturl.at/AtNOO`

Appendix B

Forms for the User Study

The following section contains the forms provided to the participants of the user study in Chapter 3.

B.1 Control Group Form

The first form, included here, was provided to the control group. This group did not receive any hints during the study, which allowed for the measurement of the reproducibility of deep learning bugs without our findings.

Questions

Menu

- Questions
- Branching
- Custom question numbers
- Set page breaks
- Remove page breaks
- Preview survey
- Reports and data
- Survey home

Recent items

- mehilshah
- Group - 4E
- Group - 3E
- Group - 2E
- Group - 1E
- raisedal
- main folder
- Answering Follow-up Quest
- usmimukherjee

Guide

The following steps are recommended for your survey:

- Create questions
- Add conditional branching
- Customize look and feel
- Set privacy and behavior
- Translate survey
- Publish survey
- Analyze collected data

Group - 1C

Section 1 — Edit

New text

Introduction

Hello there!
Welcome to this survey! We are a group of researchers from Dalhousie University, Canada. Recently, we conducted an empirical study involving 85 reproducible bugs from Stack Overflow posts. Our aim was to understand two main aspects: (1) the edit actions that can be employed to complete code snippets for bug reproduction and (2) the information that enhances the reproducibility of bug reports. Our investigation has yielded several interesting findings, and we are seeking your feedback on them.

We reproduced 85 bugs and discovered they could be reproduced using 10 edit actions. To enhance their reproducibility, 5 main information categories need to be present. The edit actions and information categories are described below.

Edit Actions
Input Data Generation: Generating input data which simulates the data used for training the model.
Neural Network Construction: Reconstructing or modifying the neural network based on the information provided
Hyperparameter Initialization: Initializing the hyperparameters for training, such as batch size and number of epochs
Import Addition and Dependency Resolution: Determining the dependencies in the code snippet and adding the missing imports.
Logging: Adding appropriate logging statements to capture relevant information during reproduction
Obsolete Parameter Removal: Removing outdated parameters or functions to match the parameters of the latest library versions
Compiler Error Resolution: Debugging and resolving compiler errors that arise due to the errors in the provided code snippet.
Dataset Procurement: Acquiring the datasets and using them to train the model
Downloading Models & Tokenizers: Fetching pre-trained models and tokenizers from external sources.
Version Migration: Updating the code to adapt the changes introduced in newer library or framework versions.

Information Categories
Data: Shape of the input data, type of data, data distribution.
Model: Neural network architecture, number of layers, neurons, activation function for layers.
Hyperparameters: Batch size, epochs, optimizers, loss function.
Code Snippet: Training code snippet, evaluation script, data preprocessing, and transformation operations.
Logs: Compiler error logs, training error logs

[Edit | Delete]

---------- page break ----------

Section 2 — Edit | Delete

New text

Demographics

New text | New question | New question from library / other surveys

Question 1 — Edit | Add to library | Delete

1. What is your relevant work experience with deep learning?

- ◯ <1 Year
- ◯ 1-5 Years
- ◯ 5-10 Years
- ◯ >10 Years

Split section | New text | New question | New question from library / other surveys

Question 2 — Edit | Add to library | Delete

2. What is your relevant experience with deep learning bug fixing?

- ◯ <1 Year
- ◯ 1-5 Years
- ◯ 5-10 Years
- ◯ >10 Years

Split section | New text | New question | New question from library / other surveys

Question 3 — Edit | Add to library | Delete

3. What is your current occupation?

- ◯ Software Practitioner (Software Engineer, Deep Learning Engineer, Machine Learning Engineer etc.)
- ◯ Researcher (Masters/Doctoral Student, PostDoc, Faculty)

Split section | New text | New question | New question from library / other surveys

Question 4 Edit | Add to library | Delete

4. What are the deep learning frameworks you have worked with?

- [ ] Tensorflow
- [ ] PyTorch
- [ ] Keras
- [ ] Other

Split section | New text | New question | New question from library / other surveys

Question 5 Edit | Add to library | Delete

5. What challenges are associated with reproducing deep learning bugs in your day-to-day activities?

New text

---------- page break ----------

Section 3 Edit | Delete

Bug #1

Given the issue description, and the code snippet. Please reproduce the bug, and select the most appropriate edit operations and critical information needed to reproduce this bug.

To help the reproduction process, we have provided the sample edit operations here.

Original Issue Report: https://stackoverflow.com/questions/59278771/super-low-accuracy-for-neural-network-model

Description:
I followed a tutorial on neural network model evaluation using cross-validation with code (given below). The accuracy was supposed to be around 95.33% (4.27%) but I got ~Accuracy: 34.00% (13.15%) on a few attempts. The model code seems exactly the same. I downloaded the data from here as instructed. What could go wrong? Thanks

Code Snippet: You can use this Colab notebook as the base notebook to start the reproduction process: https://colab.research.google.com/drive/1CH0EKq3Wc2ctcw1kWvAzxe3O7i-FA05f?usp=sharing

[Edit | Delete]

New text | New question | New question from library / other surveys

Question 6 Edit | Add to library | Delete

6. What are the edit operations that could be used to reproduce this bug?

- [ ] Input Data Generation
- [ ] Hyperparameter Initialization
- [ ] Logging
- [ ] Compiler Error Resolution
- [ ] Downloading Models and Tokenizers
- [ ] Neural Network Construction
- [ ] Import Addition and Dependency Resolution
- [ ] Obsolete Parameter Removal
- [ ] Dataset Procurement
- [ ] Version Migration

Split section | New text | New question | New question from library / other surveys

Question 7 Edit | Add to library | Delete

7. Why do you think these edit operations could prove useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 8 Edit | Add to library | Delete

8. What are the critical information components that could help the reproducibility of this bug?

- [ ] Data
- [ ] Hyperparameters
- [ ] Model
- [ ] Code Snippet
- [ ] Logs

Split section | New text | New question | New question from library / other surveys

Question 9 Edit | Add to library | Delete

9. How do you think the selected critical information could be useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 10 Edit | Add to library | Delete

10. Did you implement any additional operations or actions beyond those suggested by us? Please let us know your thoughts.

New text

--------- page break ---------

Section 4 Edit | Delete

Bug #2

Given the issue description, and the code snippet. Please reproduce the bug, select the most appropriate edit operations and critical information needed to reproduce this bug.

To help the reproduction process, we have provided the sample edit operations here.

Original Issue Report: https://stackoverflow.com/questions/39525358/neural-network-accuracy-optimization

Description:

I have constructed an ANN in keras which has 1 input layer(3 inputs), one output layer (1 output) and two hidden layers with with 12 and 3 nodes respectively, as shown below in the code.

The dataset has 4 columns: 3 columns with values in the range [60, 70] and the target variable is binary (0/1 output)

so after 150 epochs i get: **loss: 0.6932 - acc: 0.5000 - val_loss: 0.6970 - val_acc: 0.1429**

My question is: how could i modify my NN in order to achieve higher accuracy?

Code Snippet: You can use this Colab notebook as the base notebook to start the reproduction process: https://colab.research.google.com/drive/1O8y5vYDP7ODPcvi1cGNraMOP8iXxlLhM?usp=sharing

[Edit | Delete]

New question | New question from library / other surveys

Question 11 Edit | Add to library | Delete

11. What are the edit operations that could be used to reproduce this bug?

- [ ] Input Data Generation
- [ ] Hyperparameter Initialization
- [ ] Logging
- [ ] Compiler Error Resolution
- [ ] Downloading Models and Tokenizers
- [ ] Neural Network Construction
- [ ] Import Addition and Dependency Resolution
- [ ] Obsolete Parameter Removal
- [ ] Dataset Procurement
- [ ] Version Migration

Split section | New text | New question | New question from library / other surveys

Question 12 Edit | Add to library | Delete

12. Why do you think these edit operations could prove useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 13 Edit | Add to library | Delete

13. What are the critical information components that could help the reproducibility of this bug?

- [ ] Data
- [ ] Hyperparameters
- [ ] Model
- [ ] Code Snippet
- [ ] Logs

Split section | New text | New question | New question from library / other surveys

Question 14 Edit | Add to library | Delete

14. How do you think the selected critical information could be useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 15 Edit | Add to library | Delete

15. Did you implement any additional operations or actions beyond those suggested by us? Please let us know your thoughts.

New text

---------- page break ----------

Section 5 Edit | Delete

New text

Unique ID Generation

New text | New question | New question from library / other surveys

Question 16 Edit | Add to library | Delete

16. Please use this secure link, and enter the Unique ID generated in the following textbox.

If you want to withdraw from the survey, email us with this Unique ID at shahmehil@dal.ca, and we will delete your response.

Unique ID

New text | New question | New question from library / other surveys

---------- page break ----------

B.2 Experimental Group Form

The second form, included here, was used by the experimental group. This group received hints during the study, which aimed to explore the effects of these hints on the reproducibility of deep learning bugs.

Questions

Menu

- Questions
- Branching
- Custom question numbers
- Set page breaks
- Remove page breaks
- Preview survey
- Reports and data
- Survey home

Recent items

- mehilshah
- Group - 1C
- Group - 4E
- Group - 3E
- Group - 2E
- raisedal
- main folder
- Answering Follow-up Quest
- usmimukherjee

Guide

The following steps are recommended for your survey:

- Create questions
- Add conditional branching
- Customize look and feel
- Set privacy and behavior
- Translate survey
- Publish survey
- Analyze collected data

Group - 1E

Section 1 — Edit

New text

Introduction

Hello there!
Welcome to this survey! We are a group of researchers from Dalhousie University, Canada. Recently, we conducted an empirical study involving 85 reproducible bugs from Stack Overflow posts. Our aim was to understand two main aspects: (1) the edit actions that can be employed to complete code snippets for bug reproduction and (2) the information that enhances the reproducibility of bug reports. Our investigation has yielded several interesting findings, and we are seeking your feedback on them.

We reproduced 85 bugs and discovered they could be reproduced using 10 edit actions. To enhance their reproducibility, 5 main information categories need to be present. The edit actions and information categories are described below.

Edit Actions
Input Data Generation: Generating input data which simulates the data used for training the model.
Neural Network Construction: Reconstructing or modifying the neural network based on the information provided
Hyperparameter Initialization: Initializing the hyperparameters for training, such as batch size and number of epochs
Import Addition and Dependency Resolution: Determining the dependencies in the code snippet and adding the missing imports.
Logging: Adding appropriate logging statements to capture relevant information during reproduction
Obsolete Parameter Removal: Removing outdated parameters or functions to match the parameters of the latest library versions
Compiler Error Resolution: Debugging and resolving compiler errors that arise due to the errors in the provided code snippet.
Dataset Procurement: Acquiring the datasets and using them to train the model
Downloading Models & Tokenizers: Fetching pre-trained models and tokenizers from external sources.
Version Migration: Updating the code to adapt the changes introduced in newer library or framework versions.

Information Categories
Data: Shape of the input data, type of data, data distribution.
Model: Neural network architecture, number of layers, neurons, activation function for layers.
Hyperparameters: Batch size, epochs, optimizers, loss function.
Code Snippet: Training code snippet, evaluation script, data preprocessing, and transformation operations.
Logs: Compiler error logs, training error logs

[Edit | Delete]

---------- page break ----------

Section 2 — Edit | Delete

New text

Demographics

New text | New question | New question from library / other surveys

Question 1 — Edit | Add to library | Delete

1. What is your relevant work experience with deep learning?

- ◯ <1 Year
- ◯ 1-5 Years
- ◯ 5-10 Years
- ◯ >10 Years

Split section | New text | New question | New question from library / other surveys

Question 2 — Edit | Add to library | Delete

2. What is your relevant experience with deep learning bug fixing?

- ◯ <1 Year
- ◯ 1-5 Years
- ◯ 5-10 Years
- ◯ >10 Years

Split section | New text | New question | New question from library / other surveys

Question 3 — Edit | Add to library | Delete

3. What is your current occupation?

- ◯ Software Practitioner (Software Engineer, Deep Learning Engineer, Machine Learning Engineer etc.)
- ◯ Researcher (Masters/Doctoral Student, PostDoc, Faculty)

Split section | New text | New question | New question from library / other surveys

Question 4 Edit | Add to library | Delete

4. What are the deep learning frameworks you have worked with?

- [ ] Tensorflow
- [ ] PyTorch
- [ ] Keras
- [ ] Other

Split section | New text | New question | New question from library / other surveys

Question 5 Edit | Add to library | Delete

5. What challenges are associated with reproducing deep learning bugs in your day-to-day activities?

New text

---------- page break ----------

Section 3 Edit | Delete

Bug #1

Given the issue description, and the code snippet. Please reproduce the bug, and select the most appropriate edit operations and critical information needed to reproduce this bug.

To help the reproduction process, we have provided the sample edit operations here.

Original Issue Report: https://stackoverflow.com/questions/59278771/super-low-accuracy-for-neural-network-model

Description:
I followed a tutorial on neural network model evaluation using cross-validation with code (given below). The accuracy was supposed to be around 95.33% (4.27%) but I got ~Accuracy: 34.00% (13.15%) on a few attempts. The model code seems exactly the same. I downloaded the data from here as instructed. What could go wrong? Thanks

Code Snippet: You can use this Colab notebook as the base notebook to start the reproduction process: https://colab.research.google.com/drive/1CH0EKq3Wc2ctcw1kWvAzxe3O7i-FA05f?usp=sharing

Hints:

1. Code Snippet, Logs and Data can be useful information for reproducing the bug.
2. Input Data Generation, Import Addition, Version Migration, Hyperparameter Initialization and Compiler Error Resolution can be useful edit actions for reproducing the bug.
3. Focus on the statement: "I followed a tutorial on neural network model evaluation using cross-validation with code (given below)".

[Edit | Delete]

New text | New question | New question from library / other surveys

Question 6 Edit | Add to library | Delete

6. What are the edit operations that could be used to reproduce this bug?

- [ ] Input Data Generation
- [ ] Hyperparameter Initialization
- [ ] Logging
- [ ] Compiler Error Resolution
- [ ] Downloading Models and Tokenizers
- [ ] Neural Network Construction
- [ ] Import Addition and Dependency Resolution
- [ ] Obsolete Parameter Removal
- [ ] Dataset Procurement
- [ ] Version Migration

Split section | New text | New question | New question from library / other surveys

Question 7 Edit | Add to library | Delete

7. Why do you think these edit operations could prove useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 8 Edit | Add to library | Delete

8. What are the critical information components that could help the reproducibility of this bug?

- [ ] Data
- [ ] Hyperparameters
- [ ] Model
- [ ] Code Snippet
- [ ] Logs

Split section | New text | New question | New question from library / other surveys

Question 9 Edit | Add to library | Delete

9. How do you think the selected critical information could be useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 10 Edit | Add to library | Delete

10. Did you implement any additional operations or actions beyond those suggested by us? Please let us know your thoughts.

New text

---------- page break ----------

Section 4 Edit | Delete

Bug #2

Given the issue description, and the code snippet. Please reproduce the bug, select the most appropriate edit operations and critical information needed to reproduce this bug.

To help the reproduction process, we have provided the sample edit operations here.

Original Issue Report: https://stackoverflow.com/questions/39525358/neural-network-accuracy-optimization

Description:

I have constructed an ANN in keras which has 1 input layer(3 inputs), one output layer (1 output) and two hidden layers with with 12 and 3 nodes respectively, as shown below in the code.

The dataset has 4 columns: 3 columns with values in the range [60, 70] and the target variable is binary (0/1 output)

so after 150 epochs i get: **loss: 0.6932 - acc: 0.5000 - val_loss: 0.6970 - val_acc: 0.1429**

My question is: how could i modify my NN in order to achieve higher accuracy?

Code Snippet: You can use this Colab notebook as the base notebook to start the reproduction process: https://colab.research.google.com/drive/1O8y5vYDP7ODPcvi1cGNraMOP8iXxlLhM?usp=sharing

Hints:

1. Logs, Model and Code Snippet can be useful information for reproducing the bug.
2. Neural Network Construction, Import Addition, Hyperparameter Initialization, Logging, and Dataset Procurement can be useful edit actions for reproducing the bug.
3. Focus on the statement: "so after 150 epochs i get: loss: 0.6932 - acc: 0.5000 - val_loss: 0.6970 - val_acc: 0.1429"

[Edit | Delete]

New question | New question from library / other surveys

Question 11 Edit | Add to library | Delete

11. What are the edit operations that could be used to reproduce this bug?

- [ ] Input Data Generation
- [ ] Hyperparameter Initialization
- [ ] Logging
- [ ] Compiler Error Resolution
- [ ] Downloading Models and Tokenizers
- [ ] Neural Network Construction
- [ ] Import Addition and Dependency Resolution
- [ ] Obsolete Parameter Removal
- [ ] Dataset Procurement
- [ ] Version Migration

Split section | New text | New question | New question from library / other surveys

Question 12 Edit | Add to library | Delete

12. What are the critical information components that could help the reproducibility of this bug?

- [ ] Data
- [ ] Hyperparameters
- [ ] Model
- [ ] Code Snippet
- [ ] Logs

Split section | New text | New question | New question from library / other surveys

Question 13 Edit | Add to library | Delete

13. How do you think the selected critical information could be useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 14 Edit | Add to library | Delete

14. How do you think the selected critical information could be useful in reproducing the bug?

Split section | New text | New question | New question from library / other surveys

Question 15 Edit | Add to library | Delete

15. Did you implement any additional operations or actions beyond those suggested by us? Please let us know your thoughts.

New text

--------- page break ---------

Section 5 Edit | Delete

New text

Unique ID Generation

New text | New question | New question from library / other surveys

Question 16 Edit | Add to library | Delete

16. Please use this secure link, and enter the Unique ID generated in the following textbox.

If you want to withdraw from the survey, email us with this Unique ID at shahmehil@dal.ca, and we will delete your response.

Unique ID

New text | New question | New question from library / other surveys

---------- page break ----------