Towards Understanding the Faults of JavaScript-Based Deep Learning Systems

Lili Quan
College of Intelligence and Computing, Tianjin University
Tianjin, China

Qianyu Guo
Zhongguancun Laboratory
Beijing, China

Xiaofei Xie
Singapore Management University
Singapore

Sen Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China

Xiaohong Li*
College of Intelligence and Computing, Tianjin University
Tianjin, China

Yang Liu
Nanyang Technological University
Singapore

ABSTRACT

Quality assurance is of great importance for deep learning (DL) systems, especially when they are applied in safety-critical applications. While quality issues of native DL applications have been extensively analyzed, the issues of JavaScript-based DL applications have never been systematically studied. Compared with native DL applications, JavaScript-based DL applications can run on major browsers, making the platform- and device-independent. Specifically, the quality of JavaScript-based DL applications depends on the 3 parts: the application, the third-party DL library used and the underlying DL framework (e.g., TensorFlow.js), called JavaScript-based DL system. In this paper, we conduct the first empirical study on the quality issues of JavaScript-based DL systems. Specifically, we collect and analyze 700 real-world faults from relevant GitHub repositories, including the official TensorFlow.js repository, 13 third-party DL libraries, and 58 JavaScript-based DL applications. To better understand the characteristics of these faults, we manually analyze and construct taxonomies for the fault symptoms, root causes, and fix patterns, respectively. Moreover, we also study the fault distributions of symptoms and root causes, in terms of the different stages of the development lifecycle, the 3-level architecture in the DL system, and the 4 major components of TensorFlow.js framework. Based on the results, we suggest actionable implications and research avenues that can potentially facilitate the development, testing, and debugging of JavaScript-based DL systems.

CCS CONCEPTS

• **Software and its engineering** → *Software post-development issues*; • **Computing methodologies** → *Artificial intelligence.*

*Sen Chen (senchen@tju.edu.cn) and Xiaohong Li (xiaohongli@tju.edu.cn) are the corresponding authors.

KEYWORDS

JavaScript, Deep Learning, TensorFlow.js, Faults

ACM Reference Format:
Lili Quan, Qianyu Guo, Xiaofei Xie, Sen Chen, Xiaohong Li, and Yang Liu. 2022. Towards Understanding the Faults of JavaScript-Based Deep Learning Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560427

1 INTRODUCTION

Deep learning (DL) has been widely applied into various applications such as image classification [50], natural language processing [84], and speech recognition [44]. To support the DL-based applications, many DL libraries and frameworks such as TensorFlow [4], PyTorch [72], and Keras [1] have been developed and widely used. However, DL systems have been demonstrated to be vulnerable (e.g., adversarial attack [10, 11, 14, 24]), which can cause serious consequences when they are applied to some safety-critical applications such as healthcare [64] and autonomous driving [9]. Hence, quality assurance of DL systems is required.

Recently, extensive researches have been conducted from various communities including AI, software engineering, and security to study the quality issues of DL systems. For example, a lot of adversarial attack techniques [7, 25, 71] have been proposed to evaluate the model robustness. The quality of DL frameworks is also important for DL systems. Some works including bug analysis [16, 51, 53, 85] and framework testing [47, 73, 82] have been studied for DL frameworks. In addition to the model and DL frameworks, some studies are conducted on the programming bugs of DL applications (e.g., programming bugs with TensorFlow [87], bugs on the model architectures [86]). However, most of the studies focus on the native applications that can run on specific environments (e.g., Android and iOS) and DL frameworks.

A main drawback of the native applications is that they are often platform-specific (e.g., Windows, iOS, and Android) and device-specific (e.g., PC, mobile phones, and IoT devices). Considering the requirements for easy deployment and migration, JavaScript-based DL applications are becoming more and more popular. Compared to native DL applications, JavaScript-based applications are platform-agnostic and device-agnostic because they can easily run on major browsers such as Chrome, Firefox, and Safari on different platforms and devices [66]. Various JavaScript-based DL frameworks and

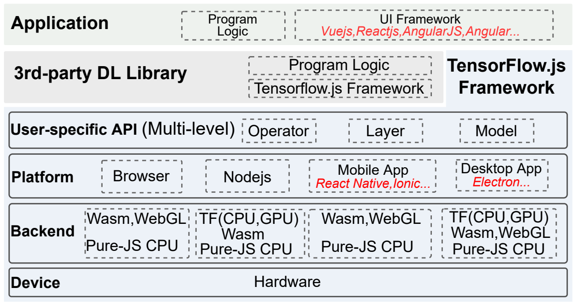


Figure 1: The typical architecture of JavaScript-based DL system

libraries (e.g., TensorFlow.js [29], Keras.js [81], and ML5.js [2]) have also been developed. While the quality issues of native applications have received a lot of attention, the quality study of JavaScript-based DL applications is still less touched, which also motivates this work to study the faults in JavaScript-based DL systems and their unique characteristics compared with native applications.

To fill this knowledge gap, in this paper, we conduct the first empirical study towards understanding the faults in JavaScript-based DL applications that run on multiple platforms (i.e., browsers, Node.js, mobile apps, and desktop apps). The quality of a JavaScript-based DL application typically depends on three parts: the application itself, the 3rd-party DL libraries used in the application and the underlying JavaScript-based DL framework. As shown in Figure 1, DL software including the 3-level architecture (i.e., application level, 3rd-party DL library level, and framework level) is called the JavaScript-based DL system. The faults in any of the three levels can significantly affect the quality of the entire system. Hence, in this paper, we conduct a comprehensive study on faults of all the *three levels*. In particular, for the DL framework, this paper focuses on TensorFlow.js which is the most popular JavaScript-based DL framework. As shown in Figure 1, TensorFlow.js contains *four major components* (i.e., API, Platform, Backend, and Device). Note that it contains several DL backends (e.g., WebGL and Wasm) specific to JavaScript, compared to native DL frameworks (e.g., TensorFlow).

On the other hand, we can observe the faults from the development lifecycle of JavaScript-based DL systems. As shown in Figure 2, the lifecycle usually includes *6 stages* [13, 18, 48, 53]. Specifically, *Environment Integration* refers to integrating DL framework (i.e., TensorFlow.js) into the applications. *Data Processing* mainly focuses on preprocessing the input and post-processing the model inference results. *Model Training* aims to build and train the model. *Model Conversion* converts models taken from other platforms into the target format. *Model Loading* loads the models through relevant APIs. *Model Inference* performs the prediction.

Specifically, we collected 72 relevant GitHub repositories including 1 official TensorFlow.js, 13 3rd-party DL libraries, and 58 JavaScript-based DL applications that cover the 3-level architecture shown in Figure 1. We collected 700 faults in total from these repositories. Based on these 700 faults, we perform a comprehensive analysis to investigate their symptoms, root causes, and fix patterns. We also highlight the unique characteristics of JavaScript-based DL systems compared to the bugs of native applications.

From these 700 faults, we summarized 26 symptoms, 17 root causes, and 16 fix patterns. Furthermore, we study the distribution

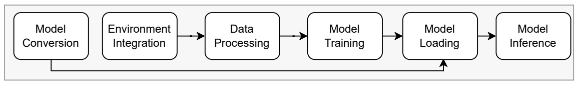


Figure 2: The typical developing stage of JavaScript-based DL system

of symptoms on the *6 stages* of the developing lifecycle; the distribution of root causes on the *3-level* architecture, and *4 components* of TensorFlow.js. The classification results (i.e., symptoms, root causes, and fix patterns) and the distribution results can help developers and researchers better understand the various faults and their characteristics, providing insights for developing different testing, debugging, and repairing techniques on JavaScript-based DL systems.

In summary, we make the following main contributions:

- To the best of our knowledge, this is the first empirical study towards understanding the characteristics of the faults in JavaScript-based DL systems. We constructed the taxonomies for fault symptoms, root causes, and fix patterns respectively, and further discussed the different characteristic of faults between the native DL systems and the JavaScript-based DL systems.
- We studied the fault distributions of symptoms and root causes on the 6 stages of the lifecycle of DL system, the 3-level architecture in the DL system, and the 4 components of TensorFlow.js.
- We provided a series of findings that benefit multiple stakeholders such as application developers, 3rd-party DL library developers, framework developers, and researchers in JavaScript-based DL ecosystems.
- We collected a dataset of real faults from a wide spectrum of sources, including the official TensorFlow.js repository, the 3rd-party DL libraries based on TensorFlow.js, and the high-level applications, which can be a valuable benchmark for further analyzing and testing the JavaScript-based DL ecosystems. We have made the fault dataset publicly available to facilitate the new research field. More details can be found on our website [3].

2 EMPIRICAL STUDY METHODOLOGY

2.1 Study Design

To characterize issues in JavaScript-based DL systems, we first collect and analyze relevant repositories from GitHub. As JavaScript-based DL systems can be built on top of various JavaScript-based DL frameworks, in this work, we mainly focus on the DL systems developed with TensorFlow.js [29], which is the most popular JavaScript-based DL framework. The overview of the methodology is illustrated in Figure 3. We first collect popular Github repositories through keyword search, including the official TensorFlow.js repository, the 3rd-party DL library repositories that wrap TensorFlow.js, and the repositories of DL-based web applications based on TensorFlow.js. For each repository, we crawl issues that may be related to fixing/discussing relevant problems and construct the candidate dataset for further analysis.

With the labeled issues, we study 4 research questions (i.e., the symptoms, root causes, fix patterns, and the differences from native DL systems). For the analysis of the symptoms in **RQ1**, we first summarize the taxonomy of fault symptoms and then analyze the distribution of symptoms on the 6 stages involved in the

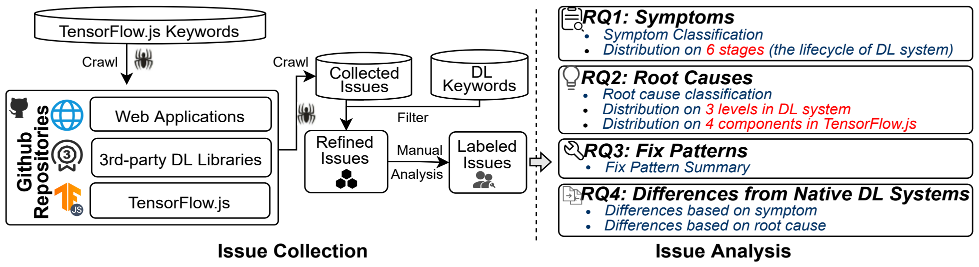


Figure 3: Overview of our methodology

development of JavaScript-based DL systems. The findings from **RQ1** can provide insights for understanding and detecting bugs for JavaScript-based DL systems. The root cause analysis in **RQ2** aims to characterize the fundamental reasons for these faults. We first summarize the different types of root causes, and further analyze the distribution of these root causes on the 3-level architecture and the components of TensorFlow.js, respectively. We summarize the *fix pattern* in **RQ3** aiming to characterize the solutions to fix these faults. Finally, in **RQ4**, we analyze the different features of the fault symptoms and fault root causes between the native DL systems studied in the previous work and JavaScript-based DL systems.

2.2 Data Collection

Following existing work [13, 16, 45, 51, 53], we first use the GitHub search API [22] to collect repositories that are related to JavaScript-based DL systems, including the DL framework TensorFlow.js [29], 3rd-party DL libraries, and web applications using DL. In HTML and JavaScript code, TensorFlow.js is usually imported using the `script tag`[1] and the statement `import * as tf from @tensorflow/tfjs` respectively. Therefore, we use "tfjs" and "TensorFlow.js" as the keywords to search the repositories. For each repository, we also collect the attributes such as links, number of stars [20], number of forks [21], number of issues, and language type. In total, we collected 924 candidate repositories. We filter out non-JavaScript-based and unpopular repositories based on the following criteria: **1)** the language type is not HTML, JavaScript or TypeScript [67]; **2)** there are no issues; and **3)** the total number of stars and forks is less than 10. In addition, we manually check the remaining repositories to exclude irrelevant repositories that are not real DL systems, e.g., some tutorials, books, or repositories that contain the keyword but do not actually use the TensorFlow.js. In the end, 72 repositories are selected, including 1 official TensorFlow.js framework, 13 3rd-party DL libraries, and 58 web applications. The details of the repositories can be found on our website [3].

Based on the repositories, we then collect issues before Dec. 2021 for the following study because these issues contain more detailed information such as original reports, discussions between users and developers, and the fix strategy. Table 1 shows the number of issues collected under each type of repository, where AF and MF stand for *Automatic Filtering* and *Manual Filtering*, respectively. A total of 3,859 issues are crawled initially, of which 2,374 are from the official TensorFlow.js repository, 1,194 are from 13 3rd-party DL libraries, and 291 are from 58 web applications. We exclude the issues with the corresponding label (e.g., *stat:awaiting response*) or without answers. Note that, considering that manually analyzing

[1]<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs/dist/tf.min.js"> </script>

Table 1: TensorFlow.js-related issues on GitHub

Repository Type	#Repositories	#Issues Crawled	After AF	After MF
Official TensorFlow.js	1	2,374	463	359
3rd-Party DL Library	13	1,194	724	291
Web Applications	58	291	106	34
Total	72	**3,859**	**1,293**	**684**

bugs is time-consuming, we cannot analyze all the historical issues of TensorFlow.js. Therefore, to balance scale and cost, we filter out the issues related to deprecated versions of TensorFlow.js (before 2020-01-01).

For issues from 3rd-party DL libraries and web applications, we adopt the similar filtering strategies used in previous work [51] to discard the DL irrelevant issues. Specifically, in [51], a vocabulary of relevant words (e.g., "epoch") related to native DL frameworks (e.g., TensorFlow) are defined, and all issues without these words are excluded. Considering the difference between TensorFlow.js and native DL frameworks, we update the vocabulary with TensorFlow.js-specific keywords (e.g., dispose, WebGL). The vocabulary finally contains 147 relevant words. As a result, 1,293 refined issues are selected as shown in column *After AF* of Table 1. Furthermore, during the manual labeling process (see Section 2.3), we discard issues with unclear descriptions and false positives. For example, some issues contain the keywords but are not errors. We finally obtain 684 issues, of which 359, 291, and 34 are from the TensorFlow.js, 3rd-party DL libraries, and web applications, respectively.

2.3 Manual Labeling

To answer the research questions, we manually label the faults in the 684 issues from 6 aspects: (1) *symptoms*, which show what the fault looks like, (2) *development stages*, which show at which stage the error happens, (3) *root causes*, which explain why the faults occur, (4) *3-level architecture*, showing at which level of the DL system a root cause comes from, (5) *components of TensorFlow.js*, indicating which component the root cause of a framework-related failure comes from and (6) *fix patterns*, which describes how a fault is resolved. Note that we need to construct the labels for the *symptoms*, *root causes* and the *fix patterns* in our study. The labels about *development stages*, *3-level architecture* and *components of TensorFlow.js* are fixed (see details in Figure 1), which are used to perform the distribution analysis of the symptoms and the root causes. The classification and the distribution analysis can help researchers and developers better understand, detect and fix different kinds of faults.

Regarding the labeling, we first randomly sample 50% of issues for pilot labeling. The first two authors label each fault of the issue following an open coding procedure [74]. Specifically, they carefully read each issue's title, descriptions and inter-developer discussions to understand the context, and construct the taxonomies for symptoms, root causes and fix patterns by grouping similar faults together into categories. The taxonomies are adjusted continuously in the construction process. During the labeling process, any disagreement is resolved by an arbitrator, who has more than five years of experience in DL-relevant research. All labels and taxonomies are finally discussed and finalized by all participants.

Second, the first two authors independently label the faults in the remaining issues based on the taxonomies generated in the pilot labeling. The issues that cannot be classified into the current taxonomies are labeled with a new category. The labeling process involves five rounds, and 20% of the remaining issues are labeled in each round. Following existing work [13, 45], we adopt the Cohen's Kappa coefficient [17] to measure the inter-rater agreement of the independent labeling. After the first round, the Cohen's Kappa coefficient is just about 39% and then the two authors discuss all these inconsistent results with the arbitrator. After the second round, the Cohen's Kappa coefficient reached 70%. Through further discussion with the arbitrator on inconsistencies, the Cohen's Kappa coefficients are over 90% after all the subsequent rounds. After manual labeling, we identify 700 faults from 684 issues collected, of which 16 issues contain 2 faults.

3 SYMPTOMS (RQ1)

3.1 Symptom Classification Results

Figure 4 shows the hierarchical taxonomy of fault symptoms in JavaScript-based DL systems. It is grouped into 5 high-level categories (i.e., *Crash*, *Build & Initialization Failure*, *Poor Performance*, *Incorrect Functionality*, and *Document Error*), 15 inner categories, and 15 leaf categories specific to *Reference Error*, *Data&Model Error*, and *Poor Performance*. Note that the blue and pink rectangles indicate symptoms specific to JavaScript-based and native DL systems respectively, which will be further detailed in Section 6.1.

3.1.1 ***Crash (A)***. This category indicates the functionality of DL systems is terminated unexpectedly with error messages like "undefined" or "uncaught Error", accounting for the largest proportion (45.43%) of faults in this study, including 318/700 faults and 5 subcategories. Note that non-functional terminations like performance issues (e.g., out of memory) are not included in this category.

Among them, *Fetch Failure (A.3)* and *Browser & Device Error (A.4)* mainly appear in the browser-based DL tasks, accounting for 26.42% of all crashes in this study. *Browser & Device Error* refers to the crashes showing messages that browsers or devices are problematic. For example, WebGL (i.e., a JavaScript API for rendering high-performance graphics on browsers, which can be used to accelerate DL tasks) is not supported on a Macbook Pro 2018 [30]. JavaScript-based DL systems need to request model files or data via the web API (i.e., `Fetch` [68]). *Fetch Failure* occurs during this process, which refers to crashes with error messages showing the fetch failure, i.e., it cannot directly access the local file system due to the same-origin policy [69] that browsers follow. These 2 subcategories jointly reveal that JavaScript-based DL systems are affected by the limitations of browsers and devices.

Reference Error is the most common crash type with 154 faults (see A.1 in Figure 4), referring to that certain objects (i.e., function, variable, and training argument) are not implemented, defined, or found. Specifically, the function reference errors include DL-related function exceptions (i.e., *A.1.1*) and traditional function exceptions (i.e., *A.1.2*). Variable reference errors refer to disposed tensors and undefined variable properties/function return values are accessed by program (i.e., *A.1.3 and A.1.4*). The remaining 24 faults are *Training Argument Exception* (*A.1.5*). Among them, *A.1.1* (DL Operator Exception) is the most common, indicating the implementations of current JavaScript-based DL systems are still in fragile status, which can easily bring errors when using a lot of DL functions.

The *Data & Model Error (A.2)* refers to the crashes that data or model is reported to be problematic. Specifically, data errors represent the incorrect data types, shapes, and values, involving both DL-related tensors (see A.2.1 in Figure 4) and JavaScript variables (see A.2.2 in Figure 4). For example, the invalid data shape reports "*tensor should have 131072 values but has 14636*" [77], and the invalid data type reports "expected input to be of type HTMLImageElement" [57]. The model errors usually show failure occurs on model usage or structure construction with messages like "*model needs to be complied before used*". The remaining 14 crashes with low frequency (occurring only once or twice) are thus categorized in *Others*.

Finding: Crash is the most common symptom, accounting for 45.43% of all faults. In particular, *Fetch Failure* and *Browser & Device Error* mainly appear when performing DL tasks on the browsers. Meanwhile, the 154 *Reference Error* indicates that the JavaScript-based DL systems are still in fragile status.

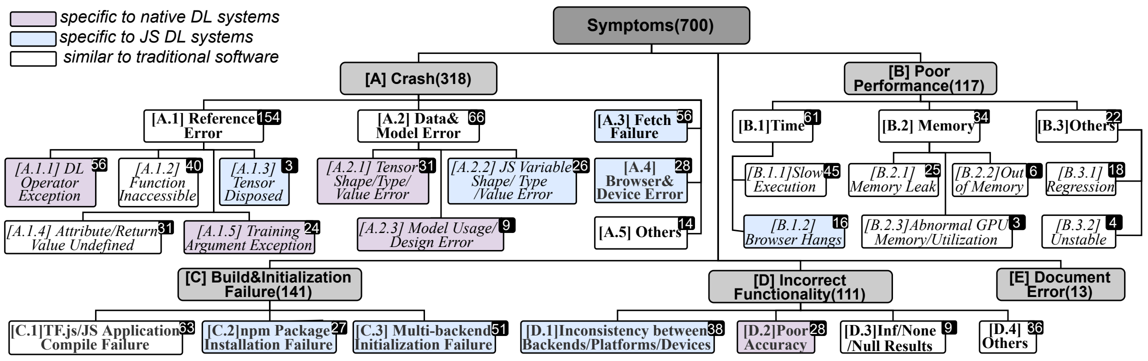
3.1.2 ***Build & Initialization Failure (C)***. At the start of developing JavaScript-based DL applications, developers need to build and initialize the necessary environments. 141/700 (20.14%) faults occurred in this process, which consists of 3 typical symptoms. Specifically, 63 faults belong to the building failure when developers compile TensorFlow.js from source code and further compile the DL applications (i.e., *C.1* in Figure 4). Alternatively, developers can install compiled TensorFlow.js via NPM (i.e., the package manager for JavaScript), during which 27 faults were found (*C.2* in Figure 4).

Apart from the explicit building errors mentioned above, there may still be exceptions even after the application has been successfully compiled. Consider the *Multi-backend Initialization Failure (C.3)*, 51 faults show that JavaScript-based DL systems fail to initialize certain DL backends (e.g., Wasm) even though they have already installed TensorFlow.js successfully [31]. The main reasons include: the device/browser used is incompatible with the backend, and some errors in the implementation of TensorFlow.js. More than 20% of the faults occurring during the building or initialization process indicate the complexity of the JavaScript-based DL system.

Finding: 20.14% of all faults are introduced when building and initializing the necessary environments for JavaScript-based DL systems, which is the second most common symptom.

3.1.3 ***Poor Performance (B)***. *Poor Performance* is another typical symptom for JavaScript-based DL systems, which slow down the execution processes, consume excessive resources, and bring bad user experiences. 117/700 faults belong to this category, covering 16.71% of all faults. It is organized into 3 inner categories (i.e., *Time*, *Memory*, and *Others*) and 7 leaf categories, as shown in Figure 4.

Time (B.1). This category covers the performance faults exhibiting high time cost, which accounts for the largest portion of



specific to native DL systems
specific to JS DL systems
similar to traditional software
Symptoms(700)
[A] Crash(318)
[B] Poor Performance(117)
[A.1] Reference Error 154
[A.2] Data& Model Error 66
[A.3] Fetch Failure 56
[A.1.1] DL Operator Exception 56
[A.1.2] Function Inaccessible 40
[A.1.3] Tensor Disposed 3
[A.2.1] Tensor Shape/Type/ Value Error 31
[A.2.2] JS Variable Shape/ Type /Value Error 26
[A.4] Browser& Device Error 28
[A.1.4] Attribute/Return Value Undefined 31
[A.1.5] Training Argument Exception 24
[A.2.3] Model Usage/ Design Error 9
[A.5] Others 14
[B.1]Time 61
[B.2] Memory 34
[B.3]Others 22
[B.1.1]Slow Execution 45
[B.2.1] Memory Leak 25
[B.2.2]Out of Memory 6
[B.3.1] Regression 18
[B.1.2] Browser Hangs 16
[B.2.3]Abnormal GPU Memory/Utilization 3
[B.3.2] Unstable 4
[C] Build&Initialization Failure(141)
[D] Incorrect Functionality(111)
[E] Document Error(13)
[C.1]TF.js/JS Application Compile Failure 63
[C.2]npm Package Installation Failure 27
[C.3] Multi-backend Initialization Failure 51
[D.1]Inconsistency between Backends/Platforms/Devices 38
[D.2]Poor Accuracy 28
[D.3]Inf/None /Null Results 9
[D.4] Others 36

Figure 4: Symptom taxonomy of faults in JavaScript-based DL systems

Poor Performance, i.e., 57.26%. Particularly, 38.46% of performance faults show *Slow Execution Time* when performing DL tasks, including data processing, model building, training, and prediction. The systems can still work but are extremely slow. 13.68% of the performance faults result in a more severe symptom (i.e., *Browser Hangs*) that JavaScript-based DL systems cease to respond to inputs. For example, the desktop browsers hang and cannot respond over a long period of time [58].

Memory (B.2). This category covers 29.06% of the performance faults which consume RAM/GPU memory abnormally. It contains 3 subcategories: the *Memory Leak (B.2.1)*, *Out of Memory (B.2.2)*, and *Abnormal GPU Memory/Utilization (B.2.3)*. Specifically, *B.2.2* is the most severe symptom and can cause the JavaScript-based DL systems to terminate unexpectedly. Moreover, *B.2.1* is the most common symptom in this category, which can lead to out of memory in severe cases. The remaining 3 memory faults show unexpectedly high or low GPU memory usage, i.e., *B.2.3*.

Others (B.3). We also summarize two special types of performance faults, i.e., the *Regression* and *Unstable*, covering 18.80% of all performance faults in this study. Specifically, *Regression* refers to the faults occurring after the TensorFlow.js upgradation. For example, the fragment shader compilation fails after TenorFlow.js upgrading from version 3.5.0 to 3.6.0 [79]. *Unstable* means the inference results of JavaScript-based DL systems are unstable. For example, when the portrait in front of the camera remains still, the face recognition results are constantly changing [59].

Poor Performance accounts for a considerable proportion of all faults in this study, and it can directly affect the user experience. There are two main reasons for the poor performance: 1) Web applications inherently suffer from low performance due to the use of DOM [83] tree in the browser. 2) The explicit memory management can easily introduce memory performance issues, e.g., manually releasing memory. It is especially required on the WebGL backend, because the browser does not automatically recycle WebGLTextures, a variable where tensor data is ultimately stored.

Finding: As a kind of non-functional fault, *Poor Performance* covers 16.71% of all faults in this study. It has various symptoms, e.g., more than one-third of the performance faults slow down JavaScript-based DL systems, and nearly 30% of the performance faults consume extremely high memory.

3.1.4 *Incorrect Functionality (D).*

We find another kind of faults that can run normally without crashes/failures, but the final results are incorrect. We refer to these faults as the *Incorrect Functionality*, covering 111/700 (15.86%) faults in this study. Specifically, 38 faults show that JavaScript-based DL systems produce different results under multiple DL backends, platforms, or devices (*D.1*). Models may give wrong inference results under some data in 28 faults (known as *Poor Accuracy (D.2)*) and even non-numerical outputs in 9 faults, such as the infinity and Null/None results (*D.3*). Besides, there are other discrete cases (36 faults) that JavaScript-based DL systems provide incorrect functionality. For example, TensorFlow.js can not properly switch the WebGL backend to the CPU backend [78].

Notably, we clarify the incorrect functionality involves two levels. 1) The DL system level. That is, JavaScript-based DL systems give an incorrect inference result. For example [23], when there is a hand/phone in the camera, the *blazeface* infers that there is a face, indicating the inference of the DL system is not robust. 2) The DL operator level. Namely, the DL operators in TensorFlow.js give wrong calculation results without crashing. For example [32], with an input `NaN`, the operator `tf.isNaN` outputs `FALSE`, indicating the implementation of this operator is incorrect. We emphasize this is a severe symptom that should arouse more attention from TensorFlow.js vendors. Due to the statistical characteristics of DL models in decision-making, the inference outputs show more uncertainty and uninterpretability than traditional software, which require carefully-designed oracles to capture the unexpected results.

Finding: *Incorrect Functionality* accounts for 15.86% of all faults in this study. This symptom category appears not only in the DL system but also in the specific operator. Moreover, we need to design test oracles for capturing these faults.

3.1.5 *Document Error (E).*

Document Error refers to the faults related to TensorFlow.js official documents/tutorials, including invalid links, incorrect instructions, and missing tutorials. Although the 13 documental faults only account for 1.86% in this study, they will not only bring bad experiences to TensorFlow.js users but also may cause implementation bugs or even security vulnerabilities for the entire DL systems. Similar implementation problems caused by poor-quality docs have been extensively studied in other fields [12]. As the foundation of DL development in JavaScript ecosystems, the rigorousness and correctness of the TensorFlow.js guidance docs should also be seriously paid attention to.

Finding: Although the *Document Error* only accounts for a small proportion (i.e., 1.86%), it will bring bad experiences to the TensorFlow.js users.

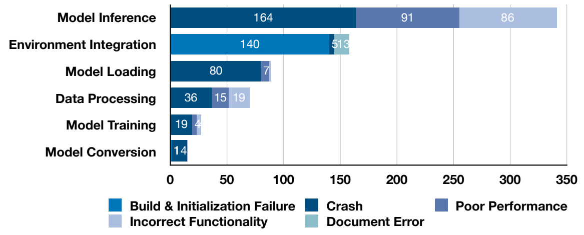


Figure 5: Symptom distribution in each stage

3.2 Symptom Distribution

We further investigate the symptom distributions in the 6 stages (See Figure 2) to understand how the faults differ across different stages. Figure 5 shows the distribution in each stage where faults are exposed. As we can see, the symptoms at different stages are different. Particularly, *Crash* and *Poor Performance* are the common symptoms in all stages (except *Environment Integration*), especially the *Model Inference* stage. Build & Initialization Failure mainly appear in *Environment Integration* stage.

In terms of stages, all faults are scattered throughout the life cycle of JavaScript-based DL system, including model development (e.g., model training) and model deployment. *Model Inference* and *Environment Integration* are the most error-prone stages, accounting for 71.28% of all faults. By contrast, *Model Conversion* is the stage with the least errors. This fault distribution is quite different from that on mobile DL applications [16], e.g., most faults (i.e., 48.4%) in mobile DL applications occur during the *Model Conversion* stage. This is caused by the differences between JavaScript DL applications and mobile DL applications. For example, the models in mobile DL applications are usually pretrained on servers and then converted for mobile usage, while models in JavaScript-based DL applications can be directly trained on browsers or servers with Node.js. Besides, the complex and diverse runtime environment (e.g., different backends) in TensorFlow.js makes that more faults belong to Environment Integration, which is also different from native DL frameworks.

Finding: Faults exposed at different stages present different symptoms. *Crash* and *Poor Performance* are the top two symptoms that go through the entire life cycle of JavaScript-based DL systems. *Model Inference* and *Environment Integration* are the most error-prone stages, covering 71.28% of all faults, due to the complexity of the architecture of the JavaScript-based DL system (multi-level and multi-component).

4 ROOT CAUSES (RQ2)

4.1 Root Cause Classification Results

The root cause taxonomies of the studied faults are shown in Figure 6, which is organized into 5 high-level categories (i.e., *Incorrect Programming*, *Execution Environment Error*, *Configuration & Dependency Error*, *Data/Model Error* and *Unknown*) and 17 inner categories. The number of faults assigned to each category is in the top right corner. Note that, there are 46 (6.57%) faults that are difficult to analyze their root causes, and are therefore classified as *Unknown*. The blue and pink rectangles indicate root causes specific to JavaScript-based DL systems and native DL systems respectively, which will be further detailed in Section 6.2.

*4.1.1 **Incorrect Programming (A)**.*

This category covers faults caused by program code, which is the most common category and accounts for 357 (51.00%) of the faults. It contains 7 subcategories i.e., *Unimplemented Operator*, *Inconsistent Modules in TF.js*, *Incorrect Code Logic*, *Incompatibility between 3rd-party DL Library and TF.js*, *API Misuse*, *Import Error*, and *Improper Exception Handling*.

Incorrect Code Logic causes the most faults, accounting for 23.29% of all faults in this study, which can be divided into three subcategories based on the code functionality. ① Incorrect DL-specific algorithms due to incorrect implementation of DL-specific functions in TensorFlow.js (e.g., basic DL operators [33]), 3rd-party DL libraries (e.g., face recognition algorithm [60]), and incorrect code logic in web applications (e.g., [70]). The most *Incorrect Code Logic* faults (119 faults) belong to this subcategory. ② Incorrect memory management algorithm due to improper memory management, accounting for 24 (3.43%) of all faults. It mainly appears when explicit memory management is used on the WebGL backend. For example [34], there is a memory leak in operator `tf.signal.stft`, because some intermediate tensors are not released in time. The TensorFlow.js vendors explained it as *"Complex components cannot be released if there are multiple references on the components and those references are disposed before the complex tensor is disposed."* ③ Poor environmental adaptability due to the incorrect/missing condition checking (i.e., if-else blocks) required to handle different environments (e.g., specific browsers). It accounts for 20 (2.86%) of all faults. Such faults mainly occur in 3rd-party libraries and TensorFlow.js. For example [35], the application works well on Chrome and Edge but fails on Opera. The TensorFlow.js vendors explained that *"In certain cases (e.g. in a webworker running in Opera), the window is not defined, which will fail the isMobile function. This PR adds a fallback check which uses navigator.userAgentData.mobile."*

46 faults are caused by *Unimplemented Operator*, i.e., the DL operators used in DL systems are not yet supported or implemented by TensorFlow.js. For example [36], the operation `tf.mod` is not supported by the Wasm backend of TensorFlow.js. Besides, there are many modules in TensorFlow.js (e.g., the `tfjs-core` and `tfjs-tflite`), which cooperate with each other to complete various DL tasks and adapt to various environments. Inconsistent implementations between these modules lead to 21 faults in this study (see *A.2* in Figure 6). For example [37], `tfjs-core` does not support tensor of type Int8Array provided by another module `tfjs-tflite`.

Apart from the faults caused by TensorFlow.js implementations, 43 faults are introduced by the flaws in 3rd-party DL libraries, i.e., *A.5* in Figure 6. It refers to faults caused by the wrong TensorFlow.js versions, as the 3rd-party DL library requires the specific version of TensorFlow.js. For example [61], the 3rd-party DL library `face-api.js` executes based on outdated versions of TensorFlow.js (i.e., 2.x), resulting in an incompatibility error "`t.toFloat` not being a function".

Finding: *Incorrect Programming* is the most common root cause category and covers 7 subcategories, accounting for 357 (51.00%) of all faults. Among them, *Unimplemented Operator*, *Inconsistent Modules in TF.js*, and *Incorrect Code Logic* are related to the implementations of TensorFlow.js. The most common subcategory is *Incorrect Code Logic*, especially the incorrect implementation of DL-specific algorithms.

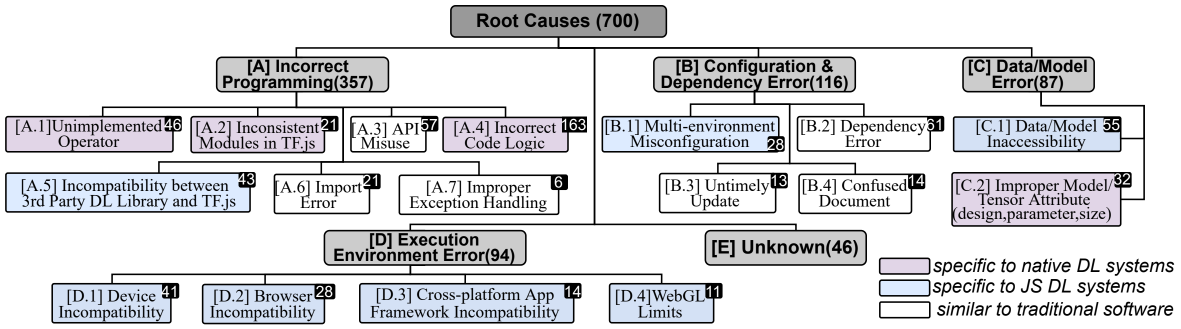

Root Causes (700)
[A] Incorrect Programming(357)
[B] Configuration & Dependency Error(116)
[C] Data/Model Error(87)
[A.1]Unimplemented Operator 46
[A.2] Inconsistent Modules in TF.js 21
[A.3] API Misuse 57
[A.4] Incorrect Code Logic 163
[B.1] Multi-environment Misconfiguration 28
[B.2] Dependency Error 61
[C.1] Data/Model Inaccessibility 55
[A.5] Incompatibility between 3rd Party DL Library and TF.js 43
[A.6] Import Error 21
[A.7] Improper Exception Handling 6
[B.3] Untimely Update 13
[B.4] Confused Document 14
[C.2] Improper Model/Tensor Attribute (design,parameter,size) 32
[D] Execution Environment Error(94)
[E] Unknown(46)
[D.1] Device Incompatibility 41
[D.2] Browser Incompatibility 28
[D.3] Cross-platform App Framework Incompatibility 14
[D.4]WebGL Limits 11
specific to native DL systems
specific to JS DL systems
similar to traditional software

Figure 6: Root cause taxonomy of faults in JavaScript-based DL systems

The remaining 3 root causes are common in traditional software. Specifically, *API Misuse* refers to the faults due to the users' misunderstanding of APIs, including missing or redundant calls to an API, wrong API names, and invalid API input/parameters (i.e., type/shape/value error). It contains 57 faults, accounting for 8.14% of all faults. Our further analysis finds 52.63% (30/57) of *API Misuse* faults are due to invalid usage of inputs/parameters, indicating that in JavaScript-based DL systems, developers are confused about the types and shapes of parameters/inputs supported by the APIs, especially for data types that can only be used in specific environments. *Import Error* refers to the faults (21 faults) caused by the missing/incorrect import of TensorFlow.js, and the import of multiple versions of TensorFlow.js at the same time. The remaining 6 faults are due to *Improper Exception Handling*, including missing exceptions, suspicious exceptions, and confusing error messages.

Finding: For the root causes that are also common in traditional software, API Misuse accounts for a considerable amount, especially the invalid API input/parameters.

4.1.2 *Configuration & Dependency Error (B).*

116/700 (16.57%) faults are caused by the incorrect configuration and dependencies, which is the second most common category. In particular, 28 *Multi-environment Misconfiguration* faults are specific to JavaScript-based DL systems. They are caused by incorrect bundler configurations that are used to ensure the same implementation of a JavaScript-based DL system can be deployed on heterogeneous environments (e.g., browser and Node.js), regardless of the underlying hardware types (i.e., PC, smartphones, and wearable devices) and the operating systems (e.g., Windows, iOS, and Android). For example, a fault is caused by not marking "os" as the external attribute in the bundler configuration for the browser target [38].

74 faults are caused by dependency-related problems, of which 61 faults suffer from the missing/redundant dependency, the dependency version mismatch, and dependencies with security vulnerabilities [39] (see *Dependency Error*); the remaining 13 faults are caused by the untimely updates of *tensorflow.so* [40] and npm packages (see *Untimely Update*). These dependency faults are closely related to the characteristics of TensorFlow.js, which relies on various libraries. For example, XNNPACK [43] is a highly optimized library for floating-point neural network inference that can be used on ARM, WebAssembly, and x86 platforms. Such complicated dependencies will inevitably introduce fragility during the configuration and runtime. The dependencies that are not updated in time or the relevant properties are not given correctly may cause serious problems in the entire system. Another 14 faults in this category are due to *Confused Document* in TensorFlow.js. Although the number is small, it brings bad experiences to users.

Finding: *Configuration & Dependency Error* is the second most common root cause, covering 116 (16.57%) of all faults. *Multi-environment Misconfiguration* and *Dependency Error* constitute two notable root causes, which are closely determined by the characteristics of TensorFlow.js (i.e., it depends on various libraries and can be used on multiple environments/platforms).

4.1.3 *Data/Model Error (C).*

DL model and data introduce 87 (12.43%) faults. Particularly, 55/87 faults are caused by the *Data/-model Inaccessibility*, due to 1) the browser limitations, i.e., local data/model cannot be accessed because of the same-origin policy [69] that browsers follow; 2) the UI framework limitations, i.e., model is not placed in specified folders (e.g., public/asset) as required by the UI framework; 3) the incorrect model path or extension. If developers are unfamiliar with the features of browsers and UI frameworks, it is easy to introduce the inaccessibility of models or data.

The remaining 32 faults are caused by *Improper Model/Tensor Attribute*, including the poor model design (e.g., incorrect inference due to the poor quality of models provided by the 3rd-party DL libraries [52]), improper model parameter (e.g., long inference time due to the large input size [62]), and improper model size (e.g., ssd mobilenetv1 model cannot run on Android because it requires a lot of resources [63]). This shows that the quality of the pre-trained models provided in the 3rd-party DL libraries needs to be improved, and some models with large size cannot work well due to the limited computing power of the web platform.

Finding: 63.22% of *Data/Model* faults are caused by the Data/-Model Inaccessibility. Such faults are mainly related to the limitations of browsers and UI frameworks.

4.1.4 *Execution Environment Error (D).*

As stated before, TensorFlow.js is designed to execute on various environments, such as different backends (e.g., WebGL and Wasm) for browsers, and cross-platform applications [29]. In this study, 94/700 (13.43%) faults are caused by imperfect support of TensorFlow.js for some hardware/software environments, which can be further divided into 4 subcategories, i.e., the *Device Incompatibility*, *Browser Incompatibility*, *Cross-platform App Framework Incompatibility*, and *WebGL Limits*.

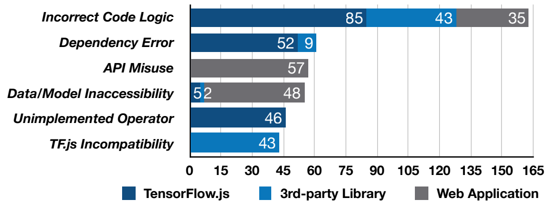


Figure 7: The distributions of root causes on 3 levels of DL systems

Device Incompatibility is the major environmental root cause, covering the largest number of faults (41/94 faults). Such issues persist when TensorFlow.js executes on specific hardware (e.g., graphics card) and operating systems (e.g., Android). For example [26], the DL system gives abnormal results on devices equipped with Intel HD Graphics. Compared with native DL frameworks [13], such faults are more prominent in JavaScript-based DL frameworks, indicating the implementation of DL backends specific to JavaScript usage needs to be improved in order to fit more diverse devices.

Browser Incompatibility (i.e., PC browser and mobile browser) is another major cause, covering 28 faults. Such faults are reasonable because the DL inference on browsers is executed in JavaScript and relies on the browser engine for interpretation. There are also 14 faults caused by *Cross-platform App Framework Incompatibility*. Specifically, TensorFlow.js can be integrated into mobile/desktop applications via cross-platform application frameworks (e.g., React Native). However, some of them are not compatible with the underlying libraries (e.g., *expo-gl*) on which TensorFlow.js depends. Apart from the incompatibility factors mentioned above, the inherited limitations of WebGL (a set of JavaScript APIs which can be used for accelerating DL tasks) can also bring faults in some browser-based DL scenarios (14 faults). For example, the GUI is blocked due to the management mechanism of GPU resources in WebGL [41].

Finding: *Execution Environment* is a common root cause that is quite specific to JavaScript-based DL systems, due to the complex software/hardware environments they execute on. *Device Incompatibility* and *Browser Incompatibility* are top two environmental causes, accounting for 73.40% of this category.

4.2 Root Cause Distribution

4.2.1 *Distribution on the 3 levels of DL systems.*

We further analyze the distribution of root causes to understand how these faults present on the 3 levels of JavaScript-based DL systems (i.e., Web application, 3rd-party DL library, and TensorFlow.js). As shown in Figure 7, the top 6 common root causes (425/700 faults, 60.71%) are considered due to the space limit. Note that the *TF.js Incompatibility* stands for *A.5* in Figure 6.

In terms of root cause, *Incorrect Code Logic* is the most common type, which distributes over all of the 3 levels. Such faults should be a concern for all developers and researchers. The remaining root causes are distributed over a specific level. For example, 85.25% of *Dependency Errors* and all of the *TF.js Incompatibility* faults appear in TensorFlow.js and 3rd-party library, respectively. In terms of system levels, most faults are caused by TensorFlow.js. Different levels present different fault distribution tendencies. Specifically, faults on TensorFlow.js are closely related to low-level implementation, e.g., *Incorrect Code Logic* and *Unimplemented Operator*, indicating that TensorFlow.js is still at the early stages of development. The vendors need to enrich DL operators and check the libraries on which TensorFlow.js depends in time. As a comparison, faults on 3rd-party libraries are mainly caused by *TF.js Incompatibility*, and faults on web applications are more about the high-level usage of TensorFlow.js, e.g., the *API Misuse* and *Data/Model Inaccessibility*. In these cases, the 3rd-party library vendors need more effort to keep compatible with TensorFlow.js, and web application developers should carefully use DL-related APIs and handle model/data.

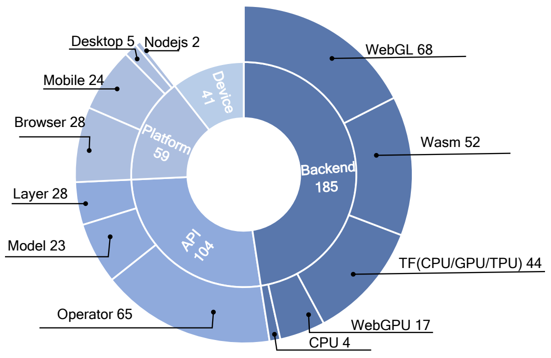


Figure 8: The distribution of faults across TensorFlow.js components

4.2.2 *Distribution on the framework components.*

TensorFlow.js contains 4 components (see Figure 1), i.e., API, Platform, Backend, and Device. To evaluate the quality of them, we further analyze the distribution of faults on each component, as shown in Figure 8. The inner layer represents the 4 components and corresponding faults number. The outer layer represents the sub-component corresponding to the component (the inner node of the same color) and the number of errors caused by each sub-component.

As we can see, 389/700 (55.57%) faults are introduced by the 4 TensorFlow.js components. DL-backend-related faults cover the most faults (i.e., 185/389 faults), of which the JavaScript-specific backends (i.e., WebGL, Wasm, WebGPU, and Pure-JS CPU) account for the majority of cases (141 faults), especially the WebGL (68/141) and Wasm (52/141). A large scale of faults brought by DL backends indicates that JavaScript-based DL systems are still at dawn for the goal of conducting DL across multiple environments. In particular, JavaScipt-specific backend (e.g., WebGL) are more fragile than the native backend. API component introduces the second most faults (104/389 faults), which consists of 3 levels of APIs, i.e., the operator-level, layer-level, and model level. Among them, the operator-level APIs bring the most faults (65 faults). Since existing testing techniques [13, 47, 73, 82] suffer from low operator coverage, we emphasize this is a challenge for detecting API errors on JavaScript-based DL systems. Regarding the Platform component, it causes 59 faults, which mainly occur on browsers and mobile applications. The remaining 41 faults are due to device components.

Finding: In terms of the 3 system levels, the root causes present certain tendencies on different levels. *Incorrect Code Logic* is the most common one that affects all of the 3 levels. In terms of the components in TensorFlow.js, most faults are caused by the DL backends, especially the JavaScript-specific backends (e.g., WebGL), which calls for new testing techniques and debugging methods for detecting such errors.

Table 2: Fix patterns of the faults

Object	Subject	Fix Pattern	#
Environment	Developer	Changing version	99
	Developer, Vendor	Modifying dependency configuration	63
	Developer	Changing device/browser	39
	Developer	Changing backend	31
	Developer	Modifying the value of environment variable	19
	Developer	Fix import confusion in program	13
Model	Developer	Modifying model file path/extension	37
	Developer	Changing model	32
Data	Developer	Add data processing	27
	Developer, Vendor	Replace data shape/type	27
Program & API	Vendor	Add unsupported operator	40
	Developer, Vendor	Replace API with another effective one	29
	Developer	Modify API parameter usage	25
	Developer, Vendor	Fix environment adaptability	20
	Developer, Vendor	Adjust API invocation sequence	19
	Developer, Vendor	Add API usage for memory management	19
Total	-	-	539

5 FIX PATTERNS (RQ3)

A fix pattern means that the subject fixes these faults by using them to modify the object. Table 2 shows the 16 common fix patterns with 539 faults, which involve 4 major objects (i.e., Environment, Model, Data, and Program & API) and 2 subjects (i.e., application developers and framework/3rd-party library vendors). Note that we consider the subject to be the framework/3rd-party library vendors if the fault is fixed via PR in the framework/third-party library repository, otherwise it is the application developers.

Environment. Modifying the environment configuration is the most common, which resolve 264 faults and is typically used by developers. Specifically, changing versions of the TensorFlow.js, 3rd-party DL libraries, and compiler/installer (e.g., Typescript, Node.js, and NPM) can solve most faults (i.e., 99). 89 faults can be resolved by changing the browser/device, backend, and values of environment variables. Besides, 13 import confusion in the program can be fixed by adding/removing imports and changing import statements. Note that the pattern of modifying the dependency configuration includes adding dependencies, removing dependencies, replacing mismatched dependencies, and modifying related configuration options, which can be used by both developers and vendors.

Model. Developers fix 69 faults by modifying the models. Specifically, modifying the model file path/extensions solves 37 faults, which are mainly model inaccessibility faults. Another 32 faults are fixed by model reconstruction, including retraining the model, reconverting the model, and replace with another similar model.

Data. There are two patterns that act on the data, including adding data processing and replacing data shape/types. Specifically, developers fix 27 faults by adding preprocessing of inputs and post-processing of model predictions. Another 27 faults are fixed by developers and vendors by replacing the shape/type of related data.

Program & API. 152 faults are resolved by modifying the related programs and API usage. For this object, there are 6 fix patterns, 1 of which is typically used by the vendors (i.e., add unsupported operator), 1 by the developer (i.e., modify API parameter usage), and the remaining 4 can be used by both the developers and the vendors. Specifically, vendors solve 40 faults (i.e., unimplemented operators) by adding unsupported operators. Developers solve 25 API Misuse faults by modifying the usage of the API parameters. The remaining 4 patterns resolve 87 faults that can be introduced by any of TensorFlow.js, 3rd-party library, and web application.

Note that, the fix pattern applied to one object can also repair faults caused by other objects. For example, for the pattern of changing versions in the environment, developers can also use it to resolve incorrect program logic errors in TensorFlow.js in addition to faults caused by the environment, because vendors fix bugs inside TensorFlow.js and update the TensorFlow.js version frequently. Additionally, faults caused by the same root cause can be fixed by different patterns. For example, for an error caused by the Wasm backend not supporting a certain operator, the developer can bypass the faults by changing the backend into WebGL backend, however, the vendor can add support for the operator to solve the faults. Although both methods can solve the faults, adding support for the operator is the most direct and effective method.

Finding: We summarize 16 common fix patterns based on 2 subjects and 4 objects. Modifying the environment is the most common pattern, especially changing versions of the TensorFlow.js, 3rd-party DL libraries, and compiler/installer. We find that faults caused by the same root cause can be fixed by different patterns in practice.

6 DIFFERENCES FROM NATIVE DL SYSTEMS (RQ4)

In this section, we discuss the differences between the taxonomies proposed in this study and that proposed by previous work on native DL systems from 2 aspects: the symptom and the root cause.

6.1 Differences Based on Symptom

As highlighted by the pink rectangles in Figure 4, 5 symptoms (e.g., *DL Operator Exception*) involving the characteristics of DL are shared with the existing symptom taxonomies for native DL (e.g., TensorFlow and TensorFlow Lite) faults [13, 16, 45, 51]. Despite all this, their symptoms on different DL frameworks are not exactly the same. For example, we found some cases where a basic DL operator/training argument is problematic in TensorFlow.js, but it is supported in TensorFlow (see [42]). This shows that TensorFlow.js needs to be aligned with the native framework in providing complete implementations of DL operators and training parameters.

Moreover, 8 symptoms are closely related to the characteristics of JavaScript-based DL systems, as shown by the blue rectangles in Figure 4, covering 35% faults. In Particular, the *Multi-backend Initialization Failure* and *Inconsistency between Backends/Platforms/Device* are determined by the characteristics of TensorFlow.js, which provide several parallel DL backends (e.g., WebGL and Wasm) specific to the different JavaScript execution environment. Similarly, The *Fetch Failure* and *Browser&Device Error* are determined by the fact that JavaScript-based DL systems mainly run on heterogenous browsers, including both PC browsers and mobile browsers. Such new symptoms cover more than one-third of the faults in this study, indicating that the quality of JavaScript-based DL systems deserves a comprehensive investigation.

The remaining symptoms are similar to traditional software, as shown by the white rectangles in Figure 4. The performance faults account for a large proportion in this study. Considering the low performance of JavaScript DOM manipulation and the explicit memory management of the WebGL backend, JavaScript-based DL systems are more prone to performance issues, which should arouse developers' attention and be analyzed exclusively in the future.

6.2 Differences Based on Root Cause

JavaScript-based DL systems experience some common issues as is for native DL systems, which have been extensively studied in prior work [13, 56, 85]. As shown by the pink rectangles in Figure 6, these issues are primarily caused by incorrect code logic (163 faults), unimplemented operators (46 faults), improper model/tensor attribute (32 faults), and inconsistent implementations between different modules in TensorFlow.js (21 faults). Although the aforementioned problems are expected, it is crucial to stress that a total of 262 faults (over 37%) brought on by typical DL-specific problems illustrate the incompleteness of JavaScript-based DL systems in supporting core DL functionalities.

Besides, we summarized 220 faults (31.4% of all faults) whose root causes are specific to JavaScript-based DL systems, covering 7 major subcategories. As shown by the blue rectangles in Figure 6, 126 faults are caused by the incompatibility problem. These compatibility issues can be summarized into two levels: 1) the 3rd-party DL library level, i.e., the incompatibility between TensorFlow.js and the 3rd-party DL libraries that wraps TensorFlow.js (43 faults); 2) the environment level, i.e., the incompatibility of TensorFlow.js with various execution environments, including the devices (41 faults), browsers (28 faults), and cross-platform applications (14 faults). Notably, another 28 faults are caused by the unique multi-environment characteristic of JavaScript-based DL systems. Compared with other single-environment DL systems (e.g., mobile DL apps), the JavaScript-based DL systems are designed to run in a variety of environments that rely on different configurations (e.g., browsers and Nodejs). In addition, 55 faults are mainly caused by the limitations of the browser and UI framework (e.g., Local data/-models inaccessible due to the same-origin policy [69] in browsers), and 11 faults are caused by WebGL limitations (e.g., GUI block due to the WebGL management mechanism of GPU resources). Such new root causes introduce more than 30% faults, suggesting the differences between the JavaScript-based systems and others.

The remaining categories in Figure 6 are common causes in traditional software, as shown by the white rectangles. The dependency-related error (61 faults) and API misuse (57 faults) are two leading factors. Particularly, as for *API Misuse*, 52.6% API-misuse cases are due to the invalid inputs/parameters in JavaScript-based DL systems, which is quite different from traditional software [5, 6] and even native DL framework [13], where the API missing/redundancy, and incorrect API names are uppermost cases. Therefore, for the JavaScript-based DL systems, some existing problems in traditional software are still worth exploring in combination with the characteristics of these systems.

Finding: 8 symptoms (35% of all faults) in our study are specific to the JavaScript-based DL systems, suggesting that the quality of JavaScript-based DL systems deserves a comprehensive investigation. 7 root causes (nearly one-third of all faults) in our study are specific to the JavaScript-based DL systems. Particularly, the incompatibility issues are prominent (18%) in TensorFlow.js usage. More efforts are needed for TensorFlow.js vendors to adapt to more environments.

7 DISCUSSION

7.1 Implications

7.1.1 ***For Application Developers.*** *Data/Model Inaccessibility* and *API Misuse* are major fault causes for the JavaScript-based DL applications, as shown in Section 4.2.1. Therefore, we conclude some tips: **1)** Avoid data/model inaccessibility. Since data and models are the foundation of DL applications, developers should first ensure their accessibility. Depending on the causes of such issues, we recommend that developers avoid such errors by ① putting the model on the Internet and using TensorFlow.js to load the online model via the URL to bypass the limitation that the browsers cannot directly access local files; ② checking if the model is placed in the folder required by the UI frameworks (e.g., *Angular*); ③ checking if the model path and extension are correct. **2)** Use API carefully. Quite a few faults are caused by the confusion on the API parameters/inputs in TensorFlow.js. Therefore, developers should use the APIs carefully, such as understanding the usage of the API based on the official documentation before programming.

7.1.2 ***For 3rd-party DL Library Developers.*** **1)** Improve the environmental adaptability. Section 4.1.1 reveals that faults caused by poor environmental adaptability mainly appear in 3rd-party DL libraries. Developers are expected to conduct cross-platform testing before releasing a library, ensuring the library adapts to any platform, especially the browsers and Node.js. **2)** Enhance compatibility with TensorFlow.js. Only a specific version of TensorFlow.js can work with 3rd-party DL libraries, which brings great inconvenience to users. Such issues should arouse the attention of developers.

7.1.3 ***For Framework Developers.*** **1)** Enhanced testing of the implementations of DL backends specific to JavaScript (e.g., WebGL). JavaScript-specific DL backends present more faults compared to native DL backends (see Section 4.2.2). Such faults should be noted and detected promptly. **2)** Do more unit testing. *Incorrect Code Logic* is the most common root cause (see Section 4.2.1). It causes various fault symptoms and is difficult to locate, so developers should focus on detecting such faults before releasing a new version to ensure that the implementation logic of each module is correct. **3)** Check and update dependencies in time. Many faults in TensorFlow.js are related to the libraries on which it depends, especially the libraries with real vulnerabilities. To avoid these errors, we recommend that developers check and update dependencies in time. **4)** Expand DL operators. Many operators have not been supported by TensorFlow.js, developers should support as many operators as possible to align with the mature native frameworks (e.g., TensorFlow).

7.1.4 ***For Researchers.*** **1)** Call for testing techniques for performance faults. In terms of symptoms, poor performance faults are prominent in JavaScript-based DL systems and the reasons for such faults are difficult to analyze. Therefore, testing and debugging techniques for such faults are desired. **2)** Focus on testing the framework. For the 3 levels in the JavaScript-based DL system, framework introduces the most faults. Particularly, JavaScript-specific DL backends present more faults compared to native DL backends, but there is currently no testing techniques for such errors. Thus, how to design effective testing methods according to the characteristics of JavaScript-based DL framework is a challenge for future research.

7.2 Threats to validity

The external threat to validity lies in the dataset. First, the selection of our study subjects (i.e., TensorFlow.js, 3rd-party DL libraries, and web applications) may be biased. To mitigate this threat, we choose the most popular and representative JavaScript-based DL framework (i.e., TensorFlow.js) as a base. The selected 3rd-party DL libraries and web applications are all built on TensorFlow.js. The findings based on TensorFlow.js-related systems can be largely applied to other JavaScript-based DL frameworks (e.g., Paddle.js [27] and WebDNN [28]) as most of them can run on DL backends (e.g. WebGL) specific to JavaScript like TensorFlow.js. Second, we identify relevant GitHub repositories and issues related to TensorFlow.js based on keyword matching. Some candidates may be ignored due to the predefined keywords, which would introduce biased in the data construction. To mitigate such a threat, we follow the previous work [51] to carefully select effective keywords to ensure most of the relevant repositories and issues can be identified. The internal threat to validity lies in our manual labeling process. To minimize the subjectivity of researchers, two authors conducted the labeling process independently, and another arbitrator with 3-year DL development experience helps to reach an agreement through discussions. Moreover, we leveraged Cohen's Kappa coefficient to measure the inter-rater agreement of independent labeling. The high kappa value indicates a high agreement between researchers.

8 RELATED WORK

A number of empirical studies have emerged recently on analyzing the bugs relevant to DL/ML frameworks. Thung et al. [80] first targeted ML systems (i.e., Apache Mahout, Apache Lucene, and Apache OpenNLP) and analyzed 500 bug reports. They focused on the frequencies, types, severity&impact, fixing effort&duration of these bugs. Sun et al. [75] focused on ML frameworks (i.e., Scikitlearn, Paddle, and Caffe) and manually analyzed 329 real bugs to study the bug types and bug evolution. Several studies focused on the bugs in DL frameworks and applications, which generally collected real faults from Stack Overflow and GitHub, and applied taxonomic methods for fault summarizing. Specifically, [49, 51, 53, 54, 65, 87] studied the bugs symptoms, root causes, and effects of DL applications under PC platform, which rely on popular native DL frameworks (e.g., TensorFlow, Keras, and PyTorch). [19, 55, 56] analyzed the implementation bugs of TensorFlow itself in terms of the symptom, root cause, and fix pattern. Chen et al. [13] extended to four DL frameworks (i.e. TensorFlow, PyTorch, MXNet, and DL4J), and analyzed the current testing status of DL frameworks. Moreover, there are also some empirical studies focusing on specific bug types. For example, Gu et al. [45] studied the training issues of developers in DL software. Cao et al. [8] characterized the performance bugs in DL systems. Tambon et al. [76] studied the silent bugs existed in DL frameworks. Deploying DL techniques onto mobile platforms has currently become another trend. Chen et al. [15] built a taxonomy of specific challenges that developers encounter during the deployment of DL software. They further studied the deployment faults of mobile DL applications in terms of the symptoms and fix patterns [16].

The aforementioned studies target the DL systems on the PC or mobile platforms, which are built on top of native DL frameworks (e.g., TensorFlow and TensorFlow Lite). Different from them, firstly, we target the JavaScript-based DL systems built on TensorFlow.js, which is totally different from native frameworks in terms of the implementations of DL backends and the execution environments. Secondly, previous studies analyzed the faults on a specific level (i.e., framework-level or application-level), while this study analyzes the faults over a 3-level architecture of JavaScript-based DL systems, including the web applications, 3rd-party DL libraries, and TensorFlow.js. Thirdly, different from the existing studies [15, 16] that are related to the deployment faults of DL models on mobile devices, we mainly focus on the faults related to both the development (e.g., model training) and deployment of DL models on multi-environments (i.e., browsers, Node.js, and cross-platform apps). Apart from the symptom and fix pattern analyzed in [16], we also classify the root cause in detail and analyze the fault distribution on the 3 levels of JavaScript-based DL system and 4 major TensorFlow.js components. We further detail the different features of fault symptoms and root causes between other DL systems [16] and JavaScript-based DL system in Section 3.

Moreover, various JavaScript-based DL frameworks have been released to enable DL tasks on browsers. To understand how well these frameworks behave in practice, Ma et al. [66] measured the performance gap of 7 JavaScript-based frameworks when running different DL tasks on Chrome. Guo et al. [46] aimed at the DL software deployment across different platforms, and investigated the performance gap when the trained models are migrated from the PC platform to mobile devices and Web browsers. Instead of focusing on the performance problems, we focus on the characteristics of faults in TensorFlow.js, 3rd-party DL libraries, and the web DL applications built on top of TensorFlow.js.

9 CONCLUSION

In this work, we conducted the first comprehensive study on faults in JavaScript-based DL systems by manually inspecting 700 related faults from 3 levels of GitHub repositories (i.e., TensorFlow.js, 3rd-party DL libraries wrapping TensorFlow.js, and web DL applications based on TensorFlow.js). We constructed taxonomies for fault symptoms, root causes, and fix patterns, respectively. Besides, we also analyzed the distribution of symptoms from the 6 stages involved in the lifecycle of JavaScript-based DL systems and analyzed the distribution of root causes based on the 3 levels in JavaScript-based DL systems and the 4 components of the TensorFlow.js. Additionally, we highlighted the different fault features between JavaScript-based DL systems and native DL systems. The symptoms, root causes, and fix patterns discovered by our study can be adopted to facilitate fault fix in JavaScript-based DL systems. Finally, we discussed the implications for different stakeholders based on our findings.

ACKNOWLEDGMENTS

This work was partly supported by the National Natural Science Foundation of China (No. 62102284, 61872262), the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033), the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001.

REFERENCES

[1] 2022. *Keras: The Python Deep Learning Library.* https://keras.io

[2] 2022. *ML5.js.* https://learn.ml5js.org

[3] 2022. *Website of this study.* https://sites.google.com/view/dl-fault-study4js

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI.* USENIX Association, 265–283.

[5] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th international conference on mining software repositories.* 464–467.

[6] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[7] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. In *Proceedings of the 6th International Conference on Learning Representations, ICLR.* OpenReview.net.

[8] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2021. Characterizing Performance Bugs in Deep Learning Systems. *arXiv preprint arXiv:2112.01771* (2021).

[9] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In *Proceedings of the IEEE International Conference on Computer Vision, ICCV.* IEEE Computer Society, 2722–2730. https://doi.org/10.1109/ICCV.2015.312

[10] Guangke Chen, Sen Chen, Lingling Fan, Xiaoning Du, Zhe Zhao, Fu Song, and Yang Liu. 2021. Who is real bob? adversarial attacks on speaker recognition systems. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 694–711.

[11] Guangke Chen, Zhe Zhao, Fu Song, Sen Chen, Lingling Fan, and Yang Liu. 2022. AS2T: Arbitrary source-to-target adversarial attack on speaker recognition systems. *arXiv preprint arXiv:2206.03351* (2022).

[12] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 1516–1527.

[13] Junjie Chen, Yihua Liang, Qingchao Shen, and Jiajun Jiang. 2022. Toward Understanding Deep Learning Framework Bugs. *arXiv preprint arXiv:2203.04026* (2022).

[14] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

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

[16] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE.* IEEE, 674–685. https://doi.org/10.1109/ICSE43902.2021.00068

[17] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[18] Mengnan Du, Fan Yang, Na Zou, and Xia Hu. 2020. Fairness in deep learning: A computational perspective. *IEEE Intelligent Systems* 36, 4 (2020), 25–34.

[19] Xiaoting Du, Guanping Xiao, and Yulei Sui. 2020. Fault triggers in the TensorFlow framework: An experience report. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 1–12.

[20] Github. 2022. *About Stars.* https://help.github.com/articles/about-stars/

[21] Github. 2022. *Forking a Repo.* https://help.github.com/articles/fork-a-repo/

[22] Github. 2022. *Github Serach API.* https://docs.github.com/cn/rest/search

[23] Github. 2022. *Issues.* https://github.com/tensorflow/tfjs/issues/5486

[24] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR.*

[25] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR.*

[26] google. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/299

[27] google. 2022. *face-api.js.* https://github.com/PaddlePaddle/Paddle.js

[28] google. 2022. *face-api.js.* https://github.com/mil-tokyo/webdnn

[29] Google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs

[30] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/4768

[31] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/4593

[32] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5800

[33] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5641

[34] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/4378

[35] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5334

[36] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5110

[37] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5700

[38] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/4745

[39] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5492

[40] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5702

[41] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/5454

[42] google. 2022. *TensorFlow.js.* https://github.com/tensorflow/tfjs/issues/4852

[43] google. 2022. *XNNPACK.* https://github.com/google/XNNPACK

[44] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP.* IEEE, 6645–6649. https://doi.org/10.1109/ICASSP.2013.6638947

[45] Diandian Gu, Zhenpeng Chen, Yuanqiang Liu, Zili Zhang, Yun Ma, Xin Jin, and Xuanzhe Liu. 2021. Demystifying Developers' Issues in Distributed Training of Deep Learning Software. *arXiv preprint arXiv:2112.06222* (2021).

[46] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE.* IEEE, 810–822. https://doi.org/10.1109/ASE.2019.00080

[47] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 486–498.

[48] Hannes Hapke and Catherine Nelson. 2020. *Building Machine Learning Pipelines.* O'Reilly Media.

[49] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, and Song Wang. 2022. Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries. *arXiv preprint arXiv:2203.06502* (2022).

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition, CVPR.* IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[51] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE.* ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395

[52] infinitered. 2022. *nsfwjs.* https://github.com/infinitered/nsfwjs/issues/16

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

[54] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE.* ACM, 1135–1146. https://doi.org/10.1145/3377811.3380378

[55] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *International Conference on Database Systems for Advanced Applications.* Springer, 604–620.

[56] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.

[57] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/788

[58] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/826

[59] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/59

[60] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/66

[61] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/794

[62] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/32

[63] justadudewhohacks. 2022. *face-api.js.* https://github.com/justadudewhohacks/face-api.js/issues/77

[64] Siqi Liu, Sidong Liu, Weidong Cai, Sonia Pujol, Ron Kikinis, and Dagan Feng. 2014. Early Diagnosis of Alzheimer's Disease with Deep Learning. In *Proceedings of the 11th IEEE International Symposium on Biomedical Imaging, ISBI.* IEEE, 1015–1018. https://doi.org/10.1109/ISBI.2014.6868045

[65] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of

deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[66] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. 2019. Moving Deep Learning into Web Browser: How Far Can We Go?. In *Proceedings of the 28th World Wide Web Conference, WWW*. ACM, 1234–1244. https://doi.org/10.1145/3308558.3313639

[67] Microsoft. 2022. *TypeScript*. https://www.typescriptlang.org/

[68] Mozilla. 2022. *Fetch API*. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[69] Mozilla. 2022. *Same-origin policy*. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[70] nsfwjs. 2022. *nsfwjs*. https://github.com/infinitered/nsfwjs/issues/461

[71] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2016. Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples. *CoRR* abs/1602.02697 (2016). arXiv:1602.02697 http://arxiv.org/abs/1602.02697

[72] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. *Openreview* (2017).

[73] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE/ACM, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[74] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[75] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An Empirical Study on Real Bugs for Machine Learning Programs. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference, APSEC*. IEEE Computer Society, 348–357. https://doi.org/10.1109/APSEC.2017.41

[76] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2021. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. *arXiv preprint arXiv:2112.13314* (2021).

[77] TensorFlow.js. 2022. *Issues*. https://github.com/tensorflow/tfjs/issues/5821

[78] TensorFlow.js. 2022. *Issues*. https://github.com/tensorflow/tfjs/issues/5632

[79] TensorFlow.js. 2022. *TensorFlow.js*. https://github.com/tensorflow/tfjs/issues/5246

[80] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE*. IEEE Computer Society, 271–280. https://doi.org/10.1109/ISSRE.2012.22

[81] Transcranial. 2022. *Keras.js*. https://github.com/transcranial/keras-js

[82] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 788–799. https://doi.org/10.1145/3368089.3409761

[83] Wikipedia. 2022. *Document Object Model*. https://en.wikipedia.org/wiki/Document_Object_Model

[84] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[85] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. 2018. Security Risks in Deep Learning Implementations. In *Procedings of the IEEE Security and Privacy Workshops, SP Workshops*. IEEE Computer Society, 123–128. https://doi.org/10.1109/SPW.2018.00027

[86] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 359–371.

[87] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 129–140. https://doi.org/10.1145/3213846.3213866