Demystifying Dependency Bugs in Deep Learning Stack

Kaifeng Huang[*]
Fudan University
China

Bihuan Chen[*] [†]
Fudan University
China

Susheng Wu[*]
Fudan University
China

Junming Cao[*]
Fudan University
China

Lei Ma[‡]
The University of Tokyo
Japan

Xin Peng[*]
Fudan University
China

ABSTRACT

Deep learning (DL) applications, built upon a heterogeneous and complex DL stack (e.g., Nvidia GPU, Linux, CUDA driver, Python runtime, and TensorFlow), are subject to software and hardware dependencies across the DL stack. One challenge in dependency management across the entire engineering lifecycle is posed by the asynchronous and radical evolution and the complex version constraints among dependencies. Developers may introduce dependency bugs (DBs) in selecting, using and maintaining dependencies. However, the characteristics of DBs in DL stack is still under-investigated, hindering practical solutions to dependency management in DL stack.

To bridge this gap, this paper presents the first comprehensive study to characterize symptoms, root causes and fix patterns of DBs across the whole DL stack with 446 DBs collected from StackOverflow posts and GitHub issues. For each DB, we first investigate the symptom as well as the lifecycle stage and dependency where the symptom is exposed. Then, we analyze the root cause as well as the lifecycle stage and dependency where the root cause is introduced. Finally, we explore the fix pattern and the knowledge sources that are used to fix it. Our findings from this study shed light on practical implications on dependency management.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

dependency bug, deep learning stack, empirical study

ACM Reference Format:
Kaifeng Huang, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. 2023. Demystifying Dependency Bugs in Deep Learning Stack. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616325

[*]K. Huang, B. Chen, S. Wu, J. Cao, and X. Peng are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
[†]B. Chen is the corresponding author.
[‡]L. Ma is also with University of Alberta, Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616325

1 INTRODUCTION

The significant breakthroughs in deep learning (DL) have brought great success to many DL-enabled applications, e.g., machine translation [30], medical diagnosis [43], voice assistants [21] and autonomous vehicles [12]. Such DL applications are built upon a heterogeneous and complex DL stack, including hardware (e.g., Nvidia GPU), OS (e.g., Linux), drivers (e.g., CUDA and cuDNN), runtime (e.g., Python) and libraries (e.g., TensorFlow). In other words, engineering DL applications requires software and hardware in the DL stack as prerequisite dependencies. One common challenge in engineering DL applications is dependency management across the DL stack [4, 13, 74], i.e., to properly manage versions and configurations of the software and hardware dependencies in the entire DL stack.

Motivation. Dependency management is challenging for three main reasons. First, *software and hardware dependencies are complex, and evolve quickly in an asynchronous and radical way.* Dependency complexity originates from two sources, i.e., deep stack and rich vendors. For example, many vendors provide DL libraries, e.g., Google's TensorFlow, Facebook's PyTorch and Microsoft's CNTK. Besides, dependency evolution is performed at the vendor's own pace and may introduce incompatible changes. For example, the micro-architecture of Nvidia GPU has evolved several generations over the years, from old versions such as Tesla to new versions such as Ampere. In the meantime, CUDA has evolved from version 1.0 to 11.6 to support different GPUs distinguished by compute capability, which ranges from 1.0 to 9.0. Therefore, developers may miss some dependencies and build an incomplete stack, or have troubles in selecting, updating and migrating dependency versions.

Second, *software and hardware dependencies need to satisfy complex version constraints to work together properly.* For example, each TensorFlow version only works compatibly with certain cuDNN versions, CUDA versions and Nvidia GPU versions. A developer set up an environment with TensorFlow gpu version 1.2.0rc0, Python 3.5.2, CUDA 8.0.61, cuDNN 8.0 and a GPU card with compute capability 2.1 on Windows 7 [52]. The setup failed to recognize a valid GPU card as this TensorFlow version required a GPU card with compute capability 3.0 or higher. These version constraints are scattered across documentations of software and hardware. Therefore, developers may build an incompatible stack, or introduce incompatibilities when updating versions or deploying to a new environment.

Third, *each dependency version may contain bugs or need proper configuration.* While dependency version constraints are satisfied, there might be bugs in specific versions under certain circumstances. For example, a developer created a Seq2Seq model using TensorFlow 1.5 but encountered an error [57]. It was caused by a bug only in TensorFlow 1.5, and could be alleviated by upgrading to 1.6 or downgrading to 1.4. In addition, there might be misconfigurations during the installation of dependencies. For example, some kernel modules are required to be signed on Secure Boot enabled systems when the Nvidia driver is installed. However, this may cause unknown errors raised from CUDA [51], which could be fixed by disabling Secure Boot. Therefore, developers might use a buggy dependency version or misconfigure a dependency version.

In summary, developers may introduce various dependency management problems in selecting, using and maintaining dependencies in the DL stack during the entire engineering lifecycle (i.e., environment setup, development, deployment and maintenance). We refer to these problems as dependency bugs (DBs).

Literature. On the one hand, a lot of advances have been made to investigate DBs in different ecosystems, e.g., Java [23, 65], C/C++ [31], JavaScript [44], Python [40, 64], Go [63], and Debian and Red Hat [6]. They only consider DBs at the homogeneous library layer. However, DBs in the DL ecosystem are different because they can occur across all the heterogeneous layers in the DL stack. On the other hand, a lot of efforts have been made to investigate characteristics (e.g, symptoms, root causes and fix patterns) of general bugs [25, 27, 28, 42, 76] and specific bugs [10, 14, 62, 73, 75] in DL applications. However, these studies are not specifically designed for DBs, and thus only uncover partial characteristics of DBs in DL stack. Therefore, although it is necessary to understand the characteristics of DBs in DL stack, no systematic study exists yet.

Our Study. To bridge this gap, we present the first comprehensive study to characterize DBs in DL stack. An overview of our study is presented in Fig. 1. After introducing the DL stack (see Sec. 2), we first collect 446 DBs from StackOverflow posts and GitHub issues, and then analyze these DBs to answer three RQs (see Sec. 3).

- **RQ1 Symptom:** What are the symptoms of DBs? At which lifecycle stages and dependencies are they exposed?
- **RQ2 Root Cause:** What are the root causes of DBs? At which lifecycle stages and dependencies are they introduced?
- **RQ3 Fix Pattern:** What are the fix patterns of DBs? Which knowledge sources are used to fix DBs?

Through these research questions, we aim to provide useful findings for developers and researchers (see Sec. 4, 5 and 6). For example, 38.8% of the DBs manifest DL specific errors or anomalies in software and hardware dependencies, behavior, model and data, mostly leading to crashes. Violation of constraints among software and hardware dependencies causes 79.8% of the DBs. Development is the most bug-affecting lifecycle stage, which exposes 51.8% of the DBs, while environment setup is the most bug-prone lifecycle stage, which introduces 90.8% of the DBs. 227 (50.9%) of the DBs are not introduced and exposed in the same dependency. Changing dependency version and adding dependency are the most common fix patterns, which are leveraged to fix 70.0% and 11.9% of the DBs. Source code, documentation, issue tracker and other online resource are important knowledge sources of fixing DBs.

Our findings provide practical implications for developers and researchers on dependency management across the entire engineering lifecycle (see Sec. 7), e.g., construct dependency knowledge graph for the entire DL stack, recommend dependencies in the entire DL stack, detect, localize and fix dependency bugs, and upgrade and migrate dependencies. To demonstrate the usefulness of our implications, we design a prototype of DB detection and fixing.

In summary, our work makes the following contributions.

- We conduct the first comprehensive study to explore symptoms, root causes and fix patterns of 446 DBs in DL stack.
- We provide implications for developers and researchers on dependency management in engineering DL applications.

2 DEEP LEARNING STACK

Developers need to set up a DL environment before developing or deploying DL applications. The setup process often involves the following steps. First, developers need to choose a physical machine with GPUs and operating system installed. Besides, developers can use a virtual machine on the physical machine, or choose a virtual machine on the cloud supported by cloud service providers (e.g., Amazon SageMaker). Second, to fully empower upper libraries and DL applications, developers need to install the corresponding GPU drivers and GPU-accelerated SDKs (e.g., CUDA and cuDNN). Third, developers need to select a runtime environment based on the programming language that DL applications are developed with (e.g., Python and Java). Forth, a number of libraries should be leveraged to boost the development of DL applications from different perspectives. Finally, developers could develop and deploy DL applications on top of the software and hardware dependencies.

This setup process is complicated by involving a wide scope of software and hardware dependencies. To reduce the complexity and provide a complete solution, a DL stack is proposed by organizing dependencies into layers. For example, Patterson shows a generic program stack consisting of modeling code, framework, storage, driver, operating system and hardware [2]. By following the setup process and referencing the DL stack at Patterson Consulting [2], Intel [26], Huawei [24] and Nvidia [1], we summarize a DL stack in Fig. 1. It consists of five layers. From top to bottom, they are *Application*, *Library*, *Runtime* and *Driver*, *OS/Container*, and *Hardware*.

Specifically, the *Application* layer contains DL applications from various domains, e.g., autonomous driving. The *Library* layer contains the dependencies the upper-layer DL applications directly or transitively depend on. It covers a wide range of libraries, including frameworks (e.g., TensorFlow, PyTorch and CNTK) which provide abstraction and generic functionality implementation for DL algorithms, front-end libraries providing high-level abstraction or language bindings (e.g., Keras, ktrain and NeuPy), and other libraries in the ecosystem. The *Runtime* layer includes interpreters for dynamically typed languages (e.g., Python and JavaScript) and virtual machines for statically typed languages (e.g., Java and .Net). The *Driver* layer contains the dependencies for interacting with GPUs, including GPU drivers, computing platforms and GPU-accelerated SDKs (e.g., Nvidia GPU driver, CUDA and cuDNN). The *Library* layer can directly interact with the *Runtime* and *Driver* layer, and thus they are put at the same layer. The *OS/Container* layer contains operating systems, containers and other virtual environments (e.g., Ubuntu, Windows, macOS, Docker, and Amazon SageMaker). The *Hardware* layer contains fundamental hardware like CPU, GPU, mobile chips, and vendor-specific chips (e.g., Google's TPU).

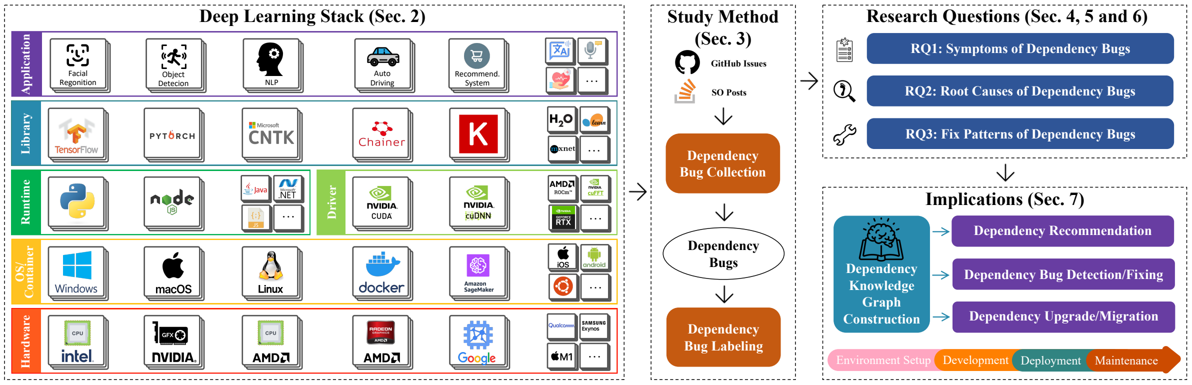

Deep Learning Stack (Sec. 2)
Application
Library
Runtime
Driver
OS/Container
Hardware
Facial Regonition
Object Detecion
NLP
Auto Driving
Recommend. System
Study Method (Sec. 3)
GitHub Issues
SO Posts
Dependency Bug Collection
Dependency Bugs
Dependency Bug Labeling
Research Questions (Sec. 4, 5 and 6)
RQ1: Symptoms of Dependency Bugs
RQ2: Root Causes of Dependency Bugs
RQ3: Fix Patterns of Dependency Bugs
Implications (Sec. 7)
Dependency Knowledge Graph Construction
Dependency Recommendation
Dependency Bug Detection/Fixing
Dependency Upgrade/Migration
Environment Setup
Development
Deployment
Maintenance

Figure 1: An Overview of Our Empirical Study on Dependency Bugs in DL Stack

3 EMPIRICAL STUDY METHODOLOGY

We first introduce the design of our empirical study, and then present our process of data collection and data labeling.

3.1 Study Design

Our *symptom analysis* in **RQ1** aims to characterize the observable consequences of DBs, which is helpful to assess impacts and provide insights for DB diagnosis and detection. Moreover, it aims to identify the lifecycle stage and dependency where the symptom is exposed, which is helpful to guide both developers and researchers to focus more effort on these bug-affecting stages and dependencies so as to achieve the most benefit for DB diagnosis and detection.

Our *root cause analysis* in **RQ2** seeks to understand the fundamental nature of DBs, which is helpful to provide insights for DB detection and localization. Further, it seeks to locate the lifecycle stage and dependency where the root cause is introduced, which is helpful to guide both developers and researchers to spend more effort on these bug-prone stages and dependencies in order to achieve the most benefit for DB avoidance, detection and localization.

Our *fix pattern analysis* in **RQ3** attempts to characterize the fixes of DBs, which is helpful to provide insights for DB fixing. Moreover, it explores the distribution of fix patterns for root causes as well as the knowledge sources that are used to fix DBs, which is helpful for both developers and researchers to achieve DB fixing in a more automated and effective fashion.

Comparison to DBs in Other Domains. Unlike general programming, DBs in deep learning exhibits a higher prevalence of low-level issues, e.g., driver configuration problems. To the best of our knowledge, there is no literature on DBs in high-performance computing or platform-specific binaries, which also encounter configuration problems that may be as prevalent as those found in deep learning. The existing literature covers a range of topics related to dependencies, including empirical studies on dependency smells [11, 29], dependency conflicts [6, 23, 44, 63–67] and dependency-related build failures [8, 36, 38, 40, 70]. During the analysis of **RQ1**, **RQ2** and **RQ3**, we compare the symptoms, root causes and fix patterns and discuss the differences from existing literature.

3.2 Data Collection

To obtain a comprehensive understanding of DBs, we collect relevant posts on StackOverflow and relevant issues on GitHub. We selected StackOverflow and GitHub because i) they are popular sites containing a wide range of problems raised by world-wide developers in real-life development activities; and ii) they have a high potential to contain problems about the dependencies in the entire DL stack due to their diversity.

3.2.1 *Collecting SO Posts.*

Our collection of SO posts has two steps.

Step 1: Dependency Tag Selection. Developers often attach several tags to a post to indicate the topics or concepts related to the question. Therefore, tags can be used to select the posts that are relevant to dependency problems in DL stack, and we need to determine a set of tags that have a high coverage of the dependencies in DL stack. To this end, we first collected all the 21,978,327 posts from Stack Exchange Data Dump on December 20, 2021. Then, for each post with an accepted answer, we iterated its tag list, and searched for tags that co-occurred with the tag "deep learning" or "neural network". In this way, we obtained an initial set of 1,576 tags. We did not directly use the tag "deep learning" or "neural network" to select posts as it may miss posts that were not tagged with "deep learning" and "neural network" but with other dependency related tags.

Next, two of the authors independently determined whether each of the 1,576 tags was related to the dependencies in DL stack by reading the excerpt provided by StackOverflow and online materials obtained by search engines. We used Cohen's Kappa coefficient to measure agreement, and it reached 0.906. A third author was involved to resolve disagreements. Finally, we obtained 57 *Library* tags, 3 *Driver* tags, 59 *Runtime* tags, 23 *OS/Container* tags and 14 *Hardware* tags.

Moreover, we conducted a comprehensive analysis of these 156 tags on significance and relevance scores, following previous work [3, 7]. Out of these tags, 106 of them have non-zero scores in terms of both significance and relevance, while the remaining 50 tags have zero scores. These tags exhibit an average significance score of 0.040 and an average relevance score of 0.018. Compared with the thresholds used in previous work [3, 7], our results suggest that our set of DL stack tags is significant and relevant.

Step 2: Dependency Post Selection. We picked dependency-related posts in two steps. First, we chose from the 21,978,327 posts the ones whose tags contained one of the 57 *Library* tags and 3 *Driver* tags, or contained the tag "deep learning" or "neural network" as well as one of the 59 *Runtime* tags, 23 *OS/Container* tags and 14 *Hardware* tags. As *Runtime*, *OS/Container* and *Hardware* tags often have a weaker correlation with DL than *Library* and *Driver* tags, here we enforced their co-occurrence with either "deep learning" or "neural network" to reduce noisy posts. This led to 66,422 posts.

Second, to focus on high-quality posts, we removed 7,301 posts that did not have an accepted answer and 35,327 posts that did not contain dependency version information. The information of dependency versions was considered as important to determine root causes and fix patterns of DBs. We used regular expression matching to check the existence of version information. This restricted our selection to 3,814 posts.

3.2.2 Collecting GitHub Issues. Our collection of GitHub issues consists of two steps.

Step 1: GitHub Repsitory Selection. To obtain dependency-related issues, we need to select a set of repositories across the DL stack. However, GitHub mainly hosts repositories at the *Application* and *Library* layer. Therefore, we first searched the 57 *Library* tags in GitHub, which linked to 30 GitHub repositories. The repository size is smaller than the tag size as i) some tags share the same repository; ii) some repositories are archived; and iii) some libraries are not hosted on GitHub. Then, we selected the top 10 repositories in the *Application* layer by querying GitHub using "deep learning".

Step 2: Dependency Issue Collection. We collected closed issues in the 40 selected repositories using GitHub API, which led to 154,299 issues. Similar to dependency post selection, we used regular expression matching to check the existence of version information in issues, which resulted in 37,795 issues. As the issue size is still large, we randomly sampled 1,763 issues with a confidence level of 99% and a margin of error of 3%.

3.2.3 DB Identification. We manually verified the 3,814 posts and 1,763 issues to reduce noise that was not about DBs in DL stack. In particular, two of the authors independently investigated each post and issue to identify DBs. The Cohen's Kappa coefficient was 0.0.909. A third author was involved to resolve disagreements. Finally, we identified 446 DBs. 326 are from posts, and 120 are from issues.

3.3 Data Labeling

To answer the three research questions, we manually labeled each of the 446 DBs with respect to eight aspects, i.e., symptom, exposing stage and dependency, root cause, introducing stage and dependency, fix pattern, and knowledge source for fixing.

In particular, two of the authors first randomly sampled 100 DBs for a pilot labeling, following an open coding procedure [47]. They separately read all contents of a post or issue (including title, question/issue description, comments, answers, commits and reference links mentioned during discussion) and relied on search engines to carefully label DBs. Basically, the symptom of a DB was determined by analyzing the question/issue description. The root cause, fix pattern and knowledge source for fixing of a DB were inferred from the question/issue description, the fixing commit or the accepted answer. The exposing stage and dependency of a DB were determined by analyzing where its symptom was exhibited, while the introducing stage and dependency of a DB were determined by analyzing where its root cause was located. A group discussion was conducted to summarize the initial taxonomies.

Then, two of the authors independently labeled all the 446 posts based on the initial taxonomies, and finally reached Cohen's Kappa coefficients of 0.967, 0.938, 0.930, 0.840, 0.870, 0.887, 0.813 and 0.858 for the eight aspects. A third author resolved disagreements in pilot and final labeling. The manual effort, involved in our data collection and labeling, required eight person-months.

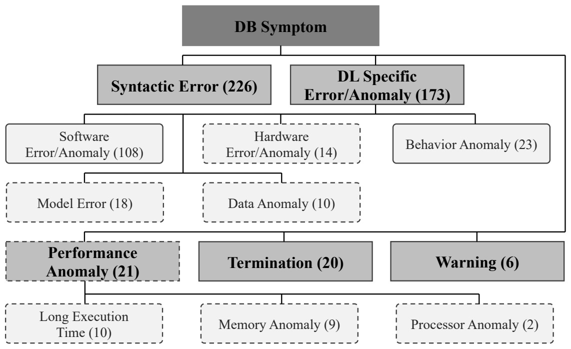


Figure 2: Taxonomy of DB Symptoms

4 RQ1: SYMPTOM ANALYSIS

We present the taxonomy of DB symptoms, and explore the stages and dependencies where symptoms are exposed.

4.1 Symptom Taxonomy

The taxonomy of DB symptoms is reported in Fig. 2. It is organized into five inner categories (i.e. *Syntactic Error*, *DL Specific Error/Anomaly*, *Performance Anomaly*, *Termination* and *Warning*) and eight leaf categories. The number in parentheses is the number of DBs exhibiting the corresponding symptom.

Syntactic Error. 226 (50.7%) of the DBs exhibit general syntactic errors that are similar to those in traditional programs. It is the most common symptom. Specifically, 114 (25.6%) of the DBs manifest *Element Not Found* errors; i.e., the used syntactic elements like module, class, function, key and attribute cannot be retrieved. Further, 36 (8.1%) of the DBs exhibit *Type Mismatch* errors; i.e., the variable type is inconsistent with the one that is expected. In addition, 25 (5.6%) and 18 (4.0%) of the DBs result in *Illegal Value* and *Illegal Argument* errors respectively, where a variable receives an illegal value, and a function call receives an illegal argument. Moreover, 13 (2.9%) of the DBs report *Undefined Variable* errors, denoting that the variable is not defined or initialized. Besides, some infrequent errors (e.g., compilation errors) are included in the *Others* category, which account for 20 (4.5%) of the DBs.

DL Specific Error/Anomaly. 173 (38.8%) of the DBs exhibit DL specific errors or anomalies. It is the second most common symptom, and is divided into five leaf categories. *Software Error/Anomaly* means errors or anomalies raised by software dependencies, accounting for 108 (24.2%) of the DBs. There are four cases. (1) 18 (4.0%) of the DBs exhibit software internal errors, indicated by an error message that contains the software name, e.g., CUDA_ERROR_UNKNOWN. (2) 59 (13.2%) of the DBs report that required software dependencies cannot be found. (3) 11 (2.5%) of the DBs manifest dependency initialization failures, indicating that required dependencies are not properly set up. (4) 20 (4.5%) of the DBs report that required software dependency versions do not match.

Moreover, *Hardware Error/Anomaly* denotes errors or anomalies raised by hardware dependencies; e.g., the GPU card is not correctly connected. It accounts for 14 (3.1%) of the DBs. Further, 23 (5.2%) of the DBs manifest *Behavior Anomaly*, e.g., abnormal accuracy metrics and unexpected return values of APIs. In addition, 18 (4.0%) of the DBs exhibit *Model Error*, which is indicated by an error message that contains model elements, e.g., computation operator missing, model save/load failure, tensor conversion error, and layer unrecognized. Besides, 10 (2.2%) of the DBs manifest *Data Anomaly*, reporting that input data has abnormal values or mismatched property (e.g., size).

Performance Anomaly. 21 (4.7%) of the DBs manifest abnormal performance with respect to execution time, memory usage and processor usage. Specifically, 10 (2.2%) of the DBs exhibit *Long Execution Time*; i.e., a program takes a long time to initialize or execute DL tasks, or even hangs in the middle of the execution. Further, 9 (2.0%) of the DBs cause *Memory Anomaly*, including abnormal memory utilization, memory leak, or even out of memory errors. Besides, two DBs result in *Processor Anomaly* (i.e., high GPU utilization).

Termination. 20 (4.5%) of the DBs caused the program directly terminated without any informative error code or error message. For example, it only reports a segmentation fault, or it simply reports that the task is killed or canceled.

Warning. 6 (1.3%) of the DBs show warning messages, including warnings about function change, version compatibility, and semantic mismatch in API arguments. For example, a version compatibility warning reveals that the installed version violates the working version requirements. These warnings forecast the potential DBs due to using versions with changed elements.

Comparison to DBs in Other Domains. Compared to previous work, distinct symptoms of the DBs in our study are highlighted in dotted rectangles in Fig. 2, which include *Hardware Error/Anomaly*, *Model Error*, *Data Anomaly* and *Performance Anomaly*. They account for 63 (14.1%) of the 446 DBs. These differences owe to the fact that previous work is focused on DBs raised in homogeneous dependencies in the *Application* and *Library* layer in traditional software applications, while DBs across heterogeneous dependencies are not studied. Our study investigates DBs across the whole DL stack to collect symptoms revealed not only in dependencies within one layer but also in dependencies across layers.

Summary. General syntactic errors and DL specific errors and anomalies are the most common symptoms, which account for 89.4% of the DBs and mostly cause crashes. Besides, 4.7% of the DBs slow executions down or consume high resources. These wide-ranging impacts motivate the importance of DBs.

4.2 Exposing Stage and Dependency

We identify the stage and dependency where the symptom of each DB is exposed, and analyze DB distribution over them.

Exposing Stage Analysis. We classify the entire lifecycle of engineering DL applications into four stages, i.e., environment setup, development, deployment, and maintenance. We report the DB distribution over the exposing stages in the right part of Fig. 3. Development is the most bug-affecting stage, where 231 (51.8%) of the DBs are exposed. This indicates that although the setup process of DL stack is presumably finished, more than half of the DBs will not occur until DL application development. Environment setup is the second most bug-affecting stage, where 168 (37.7%) of the DBs are exposed. It indicates that the setup of a feasible DL stack is not

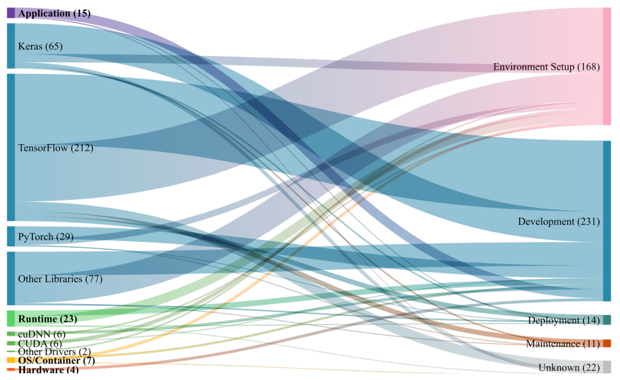


Figure 3: Exposing Dependency vs. Exposing Stage

easy. Apart from the two dominating stages, deployment exposes 14 (3.1%) and maintenance exposes 11 (2.5%) of the DBs, which are relatively smaller than in environment setup and development. The remaining 22 (4.9%) DBs have no clear indication about the exposing stage, and thus are included in the *Unknown* category.

Summary. The most bug-affecting stages are development and environment setup, exposing 51.8% and 37.7% of the DBs.

Exposing Dependency Analysis. We show the DB distribution over the exposing dependencies in the left part of Fig. 3, which is organized by the layer hierarchy in DL stack (see Sec. 2) with dominating dependencies separately highlighted. The *Library* layer is the most bug-affecting layer, where 383 (85.9%) of the DBs are exposed. Specifically, Keras, TensorFlow and PyTorch in the *Library* layer expose 65 (14.6%), 212 (47.5%) and 29 (6.5%) of the DBs respectively, which are the most bug-affecting libraries. This is reasonable as they are currently the most popular DL frameworks. The *Application* layer exposes 15 (3.4%) of the DBs, while the *Driver* layer exposes 14 (3.1%) of the DBs. CUDA and cuDNN both expose 6 (1.3%) of the DBs. Besides, there are at most 23 (5.2%) of the DBs that are exposed at the dependencies at the *Runtime*, *OS/Container* or *Hardware* layer.

The Sankey diagram in Fig. 3 illustrates where the DBs exposed in a dependency are exposed across the lifecycle stages. The width of the flow is proportional to the number of DBs. Generally, a DB can be exposed at any dependency at any layer in DL stack at any stage of the engineering lifecycle. This indicates the complexity of DBs.

Summary. *Library*, *Application* and *Driver* are the most bug-affecting stack layers. Keras, TensorFlow and PyTorch are the most bug-affecting libraries.

5 RQ2: ROOT CAUSE ANALYSIS

We report the taxonomy of DB root causes, and analyze the stages and dependencies where root causes are introduced.

5.1 Root Cause Taxonomy

The taxonomy of DB root causes is shown in Fig. 4. We first classify the root causes based on the criterion that whether a DB is caused by one dependency (i.e., *Intra-Dependency Cause*) or by constraints among dependencies across DL stack (i.e., *Inter-Dependency Cause*). Then, we summarize six leaf categories.

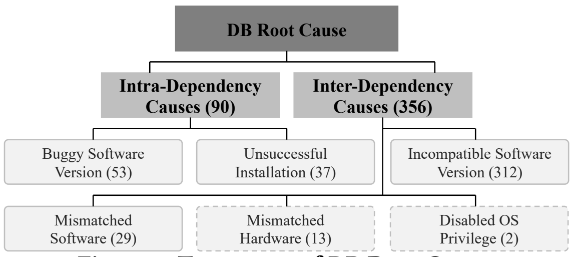

DB Root Cause
Intra-Dependency Causes (90)
Inter-Dependency Causes (356)
Buggy Software Version (53)
Unsuccessful Installation (37)
Incompatible Software Version (312)
Mismatched Software (29)
Mismatched Hardware (13)
Disabled OS Privilege (2)

Figure 4: Taxonomy of DB Root Causes

Intra-Dependency Cause. 90 (20.2%) of the DBs are caused solely by one dependency itself, and are divided into two leaf categories. Particularly, 53 (11.9%) of the DBs are caused by *Buggy Software Version*; i.e., a DB is caused by triggering bugs in software dependencies in DL stack. Moreover, 37 (8.3%) of the DBs are caused by *Unsuccessful Installation* of dependencies. There are two cases. (1) Dependency installation does not complete. For example, a developer found that there was no file named `cudnn64_6.dll`, which was caused by the missed installation of cuDNN on his/her machine [50]. (2) Dependency installation completes, but lacks proper path configuration (e.g., missing path configuration or configuring incorrect path).

Inter-Dependency Cause. 356 (79.8%) of the DBs are caused by constraints among software and hardware dependencies across DL stack; i.e., multiple dependencies have to be considered together to have a feasible DL stack, otherwise, DBs might be introduced. It is divided into four leaf categories. Specifically, 312 (70.0%) of the DBs are caused by *Incompatible Software Version*, which is the most common root cause. An incompatible software version is introduced if it violates the version constraint that has to be satisfied for it to work with other dependencies. For 58 of the 312 DBs, detailed API-level incompatibility information is provided in the post/issue, and we classify the incompatibility based on API changes [9], i.e., API removal, API addition, API replacement, API movement, API parameter list change, API renaming, and API behavior change. API addition, API behavior change and API removal are the most common root causes of API incompatibility, which respectively account for at least 20 (4.5%), 13 (2.9%) and 11 (2.5%) of the DBs. API replacement, API parameter list change, API movement and API renaming respectively cause at least 4, 4, 3 and 3 DBs. For the 177 of the 311 DBs, we can only distinguish whether they are caused by backward incompatibility (for 101 (22.6%) of the DBs) or forward incompatibility (for 76 (17.0%) of the DBs). For the remaining 77 of the 311 DBs, we can only determine they are caused by incompatibility due to the limited information in the posts/issues.

29 (6.5%) of the DBs are caused by *Mismatched Software*; i.e., while different software can provide similar functionalities, only some of them can work with the other dependencies in DL stack, but others are regarded as mismatched. Specifically, 15 of the DBs are caused by selecting wrong software as dependency. For example, the DL framework Keras and the `tf.keras` module introduced in TensorFlow 1.10 provide similar APIs, but Keras does not support TensorFlow 2.0. In that sense, if TensorFlow 2.0 is used in DL stack, Keras would be mismatched and thus cannot be used [53]. Further, 11 of the DBs are caused by choosing wrong software distribution. For example, the official pre-built TensorFlow 2.0 requires CUDA Toolkit 10.0. Developers have to re-build TensorFlow 2.0 with CUDA Toolkit 10.1 to work with CUDA Toolkit 10.1. Thus, using pre-built TensorFlow 2.0 with CUDA Toolkit 10.1 could cause a DB [55]. Further, 3 of the DBs are caused by selecting multiple conflicting software. For example, loading both TensorFlow and TensorFlow-gpu [49] would cause a DB.

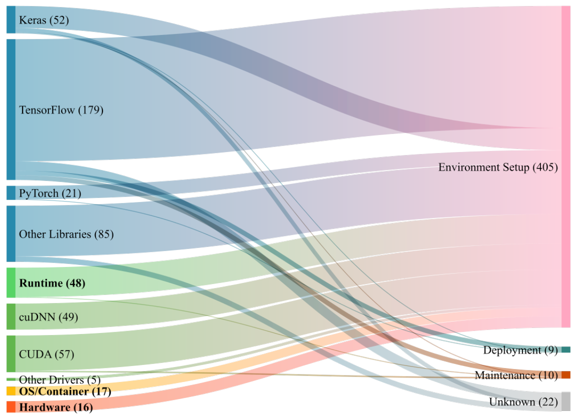


Figure 5: Introducing Dependency vs. Introducing Stage

13 (2.9%) of the DBs are caused by *Mismatched Hardware*; i.e., the hardware does not meet requirements of dependencies in upper stack layers. For example, TensorFlow 1.6 used AVX feature of CPUs, which is supported by Sandy Bridge or newer CPU architectures. Hence, using TensorFlow with non-AVX CPUs would cause a DB [48].

2 (0.4%) of the DBs are caused by *Disabled OS Privilege*; i.e., permissions required by software dependencies are not allowed from the OS or container. For example, System Integrity Protection (SIP) is enabled on MacOS 10.11 to prevent unauthorized code execution, but SIP prevents a path variable from being overridden, causing dependencies not found [54].

Comparison to DBs in Other Domains. Compared to previous work, distinct root causes of the DBs in our study are highlighted in dotted rectangles in Fig. 4, which include *Mismatched Hardware* and *Disabled OS Privilege*. They account for 15 (3.4%) of the 446 DBs. It is worth mentioning that although most root causes are shared with previous work, the dependencies that cause DBs can be different (see Sec. 5.2) as our study further considers the *Runtime*, *Driver*, *OS/Container* and *Hardware* layers.

Summary. Violation of constraints among dependencies in DL stack causes 79.8% of the DBs, where incompatible software version is the major root cause. Moreover, bugs in software dependencies cause 11.9% of the DBs.

5.2 Introducing Stage and Dependency

We locate the stage and dependency where the root cause of each DB is introduced, and analyze DB distribution over them.

Introducing Stage Analysis. The taxonomy of stages is the same to the one in Sec. 4.2. We show the DB distribution over the introducing stages in the right part of Fig. 5. Environment setup is the most bug-prone stage, where 405 (90.8%) of the DBs are introduced, while no DB is introduced in development because the DL stack is already determined in environment setup. It indicates that the setup of a feasible DL stack is important but challenging. Besides, deployment and maintenance introduce 9 (2.0%) and 10 (3.2%) of the DBs. The remaining 22 (4.9%) DBs have no clear indication about the introducing stage, and thus are put in the *Unknown* category.

Summary. The most bug-prone stage is environment setup, which introduces 90.8% of the DBs.

Introducing Dependency Analysis. For the DBs caused by inter-dependency causes, their root causes can be introduced by any of the involved dependencies. For example, a DB is caused by version constraint violation between TensorFlow and CUDA, and then both TensorFlow and CUDA can be the introducing dependency of this DB. If there is no clear indication about the introducing dependency in the posts/issues, we consider all involved dependencies as the introducing dependencies; otherwise, we use the introducing dependency that is decided in the posts/issues. It is worth mentioning that the introducing dependency of 114 DBs are only mentioned in the answers of the posts/issues, indicating that questioners are not aware of the introducing dependency. In 258 of the DBs, questioners only mention the dependency list to provide more detail, but there is no clue indicating that they are aware of the introducing dependency. In 74 of the DBs, questioners indicate assumptions on the introducing dependency. Specifically, of the 356 DBs that are caused by inter-dependency causes, 336 DBs have their introducing dependencies clearly indicated in the posts/issues.

We show the DB distribution over the introducing dependencies in the left part of Fig. 5, which is organized in the same way in Fig. 3. No DB is introduced in the *Application* layer as it is the client of dependencies. The *Library* and *Driver* layers are the most bug-prone layers, respectively introducing 301 (67.5%) and 94 (21.1%) of the DBs. The number is larger than the summation of DBs in all libraries or drivers because a DB can have multiple introducing dependencies. Specifically, Keras, TensorFlow and PyTorch introduce the most bugs in the *Library* layer, introducing 52 (11.7%), 179 (40.1%) and 21 (4.7%) of the DBs. CUDA and cuDNN introduce the most bugs in the *Driver* layer, introducing 57 (12.8%) and 49 (11.0%) of the DBs. There are at most 48 (10.8) of the DBs introduced at the dependencies at the *Runtime*, *OS/Container* or *Hardware* layer.

Besides, the Sankey diagram in Fig. 5 shows where the DBs introduced in a dependency are introduced across the lifecycle stages. Generally, a DB can be introduced at any dependency at any layer (except for *Application*) at any lifecycle stage (except for development). It reveals the complexity of DB localization.

Summary. *Library* and *Driver* are the most bug-prone stack layers, which introduce 301 (67.5%) and 94 (21.1%) of the DBs. Keras, TensorFlow and PyTorch introduce the most bugs in the *Library* layer, while CUDA and cuDNN introduce the most bugs in the *Driver* layer.

5.3 Introducing and Exposing Dependency

We further analyze where the DBs introduced in a dependency are exposed across the dependencies in DL stack. Overall, 227 (50.9%) of the DBs are not introduced and exposed in the same dependency. For example, 33 (7.4%) of the DBs introduced in TensorFlow are exposed in Keras, and 60 (13.5%) of the DBs introduced in CUDA and cuDNN are exposed in TensorFlow. At the stack layer level, 162 (36.3%) of the DBs are not introduced and exposed at the same stack layer. For example, 12 (2.7%) of the DBs introduced at the *Hardware* layer are exposed at the *Library* layer. These results indicate that DB localization need systematic knowledge of the entire DL stack.

Summary. 227 (50.9%) of the DBs are not introduced and exposed in the same dependency, and 162 (36.3%) of the DBs are not introduced and exposed at the same layer.

6 RQ3: FIX PATTERN ANALYSIS

We present the taxonomy of DB fix patterns, and report their distribution for root causes and the knowledge source of fixing.

6.1 Fix Pattern Taxonomy

The taxonomy of DB fix patterns is listed in Fig. 6. It is grouped into four inner categories (i.e., *Change Application Code*, *Change Dependency*, *Change DL Stack* and *Change Environment*) and 15 leaf categories. A DB can be fixed by applying multiple fix patterns. Hence, the summation of the number of DBs in Fig. 6 is larger than 446.

Change Application Code. 62 (13.9%) of the DBs are fixed via changing the application code although their root causes are not introduced by the application. Specifically, *Fixing API Usage* is used to fix 43 (9.6%) of the DBs; i.e., the library API usage has to be changed with the incompatible library version evolution. Moreover, *Adding Missing Code Logic* is utilized to fix 8 (1.8%) of the DBs. In such cases, some library APIs are removed or the behavior of some library APIs is changed, and hence developers have to implement the code logic of these library APIs by themselves at the application code level. Further, *Reformatting Data* is used to fix 7 (1.6%) of the DBs for making the data format compatible with the changed library APIs. Besides, *Changing Hyper-Parameter* (e.g., batch size and learning rate) is used to fix 4 (0.9%) of the DBs, because the constraints on hyper-parameters are changed with library version evolution.

Change Dependency. This is the most common fix pattern, which is leveraged to fix 407 (91.3%) of the DBs. In particular, *Changing Dependency Version* is used to fix 312 (70.0%) of the DBs, indicating that it is the most common pattern to fix DBs. Of these 312 DBs, upgrading dependency version is used in the fix of 188 DBs, and downgrading dependency version is used in the fix of 122 DBs. In 22 of the DBs, dependency version is changed but there is no clear indication in the posts/issues to determine upgrade or downgrade. Further, *Adding Dependency* is used to fix 53 (11.9%) of the DBs where some required dependencies are missing or not successfully installed. Moreover, *Re-building Dependency* is used to fix 30 (6.7%) of the DBs. In such cases, the source code of dependencies is re-built with other required dependencies to properly work with them, or the source code of dependencies is first changed (e.g., to fix bugs or to remove incompatibilities) and then re-built, potentially because of the huge maintenance effort in changing dependency versions. In addition, *Changing Dependency Configuration* is leveraged to fix 9 (2.0%) of the DBs, e.g., disabling SIP in MacOS. Besides, *Removing Dependency* is applied to fix 3 (0.7%) of the DBs in order to remove conflicted dependencies.

Change DL Stack. 30 (6.7%) of the DBs are fixed by changing the DL stack; i.e., some dependencies are switched to alternatives, and the DL stack becomes fundamentally different. It is divided into three leaf categories, i.e., *Switching Software* (libraries, drivers and runtimes), *Switching Hardware* and *Switching OS*, accounting for

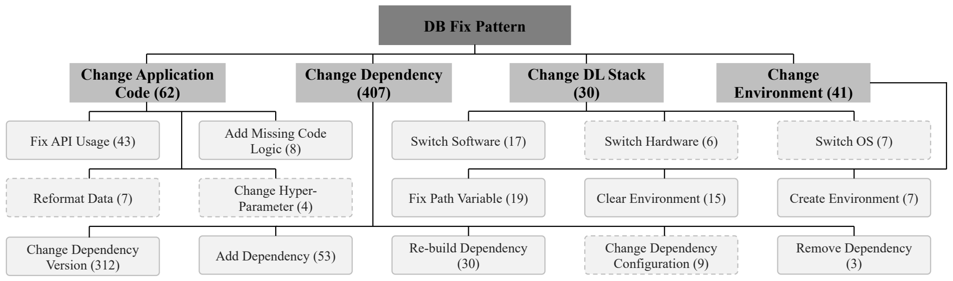


Figure 6: Taxonomy of DB Fix Patterns

17 (3.8%), 6 (1.8%) and 7 (1.6%) of the DBs. For example, a developer switched OS to Ubuntu to support distributed TensorFlow [56].

Change Environment. 41 (9.2%) of the DBs are fixed by changing the environment where libraries, drivers and runtimes can be found. Specifically, *Fixing Path Variable* is used to fix 19 (4.3%) of the DBs; i.e., the path variable is fixed to point to the correct directory that contains the required dependencies. Besides, *Clearing Environment* and *Creating Environment* are used to respectively fix 15 (3.4%) and 7 (1.6%) of the DBs. In these cases, the virtual environment (i.e., a directory that contains a specific collection of installed packages) of package managers (e.g., pip and conda) is cleared or created.

Notice that 389 (87.2%) of the DBs can be fixed by applying one fix pattern, while 73 (16.4%), 20 (4.5%) and 3 (0.7%) of the DBs can be fixed by combining two, three and four fix patterns at the same time. The summation here is larger than 446 as 37 DBs can be fixed by different combinations of fix patterns.

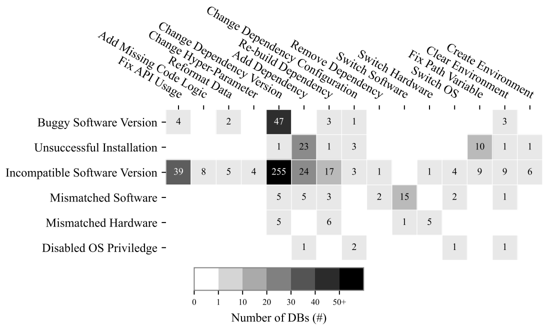
Comparison to DBs in Other Domains. Compared to previous work, distinct fix patterns of the DBs in our study are highlighted in dotted rectangles in Fig. 6, which include *Reformat Data*, *Change Hyper-Parameter*, *Switch Hardware*, *Switch OS*, and *Change Dependency Configuration*. They are used to fix 32 (7.2%) of the 446 DBs. Moreover, multiple fix patterns need to be combined to fix some DBs in DL stack, which is not the case in fixing dependency conflicts [6, 23, 44, 63–67] where only one fix pattern is needed.

Summary. The most common fix pattern is to change dependency versions, which is used to fix 70.0% of the DBs. Adding dependency is the second most common pattern, which is leveraged to fix 11.9% of the DBs. 21.5% of the DBs can be fixed by combining multiple fix patterns.

6.2 Distribution of Fix Patterns for Root Causes

We report the distribution of fix patterns for root causes in Fig. 7, where each cell denotes the number of DBs that are caused by a particular root cause and fixed by a particular fix pattern. Specifically, except for *Switching Software*, all fix patterns are utilized in fixing DBs that are caused by *Incompatible Software Version* for at least once. While *Fixing API Usage* and *Changing Dependency Version* are the two major fix patterns, there exist diverse ways to fix the most common root cause *Incompatible Software Version*. The challenge is to decide which fix pattern to use given a DB context.

Further, *Changing Dependency Version* is used in mitigating five root causes, and is involved in the fix for 312 (70.0%) of the DBs. It has a strong correlation to the root causes *Buggy Software Version*



	Fix API Usage	Add Missing Code Logic	Reformat Data	Change Hyper-Parameter	Change Dependency Version	Add Dependency	Re-build Dependency	Change Dependency Configuration	Remove Dependency	Switch Software	Switch Hardware	Switch OS	Fix Path Variable	Clear Environment	Create Environment
Buggy Software Version	4		2		47		3	1						3	
Unsuccessful Installation					1	23	1	3					10	1	1
Incompatible Software Version	39	8	5	4	255	24	17	3	1		1	4	9	9	6
Mismatched Software					5	5	3		2	15		2		1	
Mismatched Hardware					5		6			1	5				
Disabled OS Priviledge						1		2				1		1	

Figure 7: Distribution of Fix Patterns for Root Causes

and *Incompatible Software Version*. While the fix pattern itself is very simple, the key challenge is to determine which dependency version to use for addressing a DB. Besides, *Adding Dependency*, *Rebuilding Dependency* and *Clearing Environment* are the other three fix patterns spanning at least four root causes. Notice that *Adding Dependency* is the accompanied fix pattern for fixing DBs caused by *Incompatible Software Version*. For example, upgrading dependency version solves an *Incompatible Software Version*, but this upgraded dependency version may further depend on a new dependency.

Summary. *Incompatible Software Version* can be fixed by diverse patterns, whereas *Fixing API Usage* and *Changing Dependency Version* are the two major fix patterns. *Changing Dependency Version* is also the major fix pattern for *Buggy Software Version*.

6.3 Knowledge Source of DB Fixing

To fix a DB, developers usually rely on knowledge about DL stack, e.g., dependency version constraints and dependency bugs. To characterize how fixes of DBs are derived, we investigate the knowledge sources that are used to fix DBs. We identify five knowledge sources. Multiple knowledge sources can be used in fixing one DB, and hence the summation of the number of DBs below is larger than 446.

Library Source Code. 52 (11.7%) of the DBs are fixed after digging into the source code of libraries. The source code of libraries is a good knowledge source to know library version evolution, e.g., how a library API is renamed, and how a library API's code logic is changed.

Dependency Documentation. 76 (17.0%) of the DBs are fixed after looking into dependency documentation. Documentation of libraries, drivers and hardware often provide informative knowledge about dependency's installation requirements and version constraints. For example, TensorFlow documentation lists both the hardware requirements and software requirements [60].

Issue Tracker. 23 (5.2%) of the DBs are fixed after being aware of the dependency bugs. Such bugs are tracked on issue trackers with their symptoms and affected versions described.

Other Online Resource. 22 (4.9%) of the DBs are fixed by referencing other online resources, e.g., mailing lists, StackOverflow posts and technical blogs.

Unknown. For 309 (69.3%) of the DBs, there is no clear indication about the used knowledge source in the posts/issues, and thus we include them into the *Unknown* category. However, such posts/issues themselves become a knowledge source.

Summary. Library source code, dependency documentation, issue tracker, and other online resource are important knowledge sources that are directly leveraged to fix DBs.

7 IMPLICATION, APPLICATION AND THREAT

We discuss the implications of our study, demonstrate an application, and analyze the threats to our study.

7.1 Implication to Developers and Researchers

Application Developers. Our study uncovers the common DB symptoms that developers should be aware of when engineering DL applications for detecting potential DBs as early as possible. Our study also identifies the common root causes and fix patterns of DBs that could be useful for application developers to diagnose, localize and fix DBs. Our study also shows the most bug-introducing and bug-affecting dependencies where application developers should pay more attention when installing, using or maintaining them so that most DBs could be avoided or detected at the first place. Moreover, our findings provide some engineering suggestions. Application developers should be trained to have a comprehensive understanding of the DL stack, as our study reports that a DB could be introduced or exposed across the entire DL stack and engineering lifecycle. In this way, application developers are equipped with the sufficient knowledge to deal with DBs. Appilcation developers should carefully look into dependency documentation to learn version constraints, and be aware of the bugs and API changes in library version evolution. In this way, DBs caused by the most common root causes (i.e., *Buggy Software Version* and *Incompatible Software Version*) might be effectively reduced.

Library Developers. Our study reveals that around half of the DBs are not introduced and exposed in the same dependency. This requires library developers to write informative error messages in exposing dependencies to help indicate the root causes in introducing dependencies. In addition, our study identifies *Incompatible Software Version* as the common root cause and *Change Dependency Version* as the common fix pattern for DBs. This highlights the importance of providing precise version constraints by library developers to allow application developers to follow and thus prevent DB occurrences. Furthermore, if library developers integrate certain version constraint checking in dependency installation scripts and provide potential version constraint violation hints for application developers, it would eliminate DBs at the first place.

Researchers. Our findings provide future research implications in four directions. First, *a dependency knowledge graph for the entire DL stack is needed to provide fundamental knowledge for the ease of dependency management.* As uncovered by our root cause analysis, a diversity of dependency knowledge is involved in DBs, e.g., version constraints among software and hardware dependencies, bugs in dependencies, and API changes in version evolution. However, such knowledge is scattered across different sources, e.g., documentation, issue tracker and source code, as revealed by our investigation of the knowledge source of DB fixing. Online resources like StackOverflow posts and GitHub issues also provide practical solutions to fix DBs. Hence, the main challenges to construct the knowledge graph are that i) designing a high-level schema to fuse various knowledge, ii) leveraging various techniques like natural language processing and program analysis to automatically extract knowledge from different sources; and iii) developing graph analysis techniques for various dependency management tasks. This knowledge graph serves as the foundation of the following three research directions. Along this direction, Ye et al. [72] and Cheng et al. [15] construct a knowledge graph for the *Library* and *Runtime* layers for Python programs, but fail to support lower layers in the DL stack.

Second, *dependency recommendation techniques are needed.* Our introducing stage analysis reveals that environment setup is the most bug-prone stage which introduces 90.8% of the DBs. Therefore, developers often face difficulties in setting up a feasible DL stack. Further, our root cause analysis shows that 70.0% of the DBs are caused by *Incompatible Software Version*, although dependency documentation provides prerequisite information about setting up dependencies and their version constraints. Therefore, developers might not always refer to the documentation. In that sense, dependency recommendation techniques become useful for developers to ease the setup of a feasible DL stack; i.e., given some dependencies installed, they recommend other dependencies to form a complete DL stack. For example, given the available hardware and OS, they suggest required dependencies in *Driver*, *Runtime* and *Library* layers.

Third, *DB detection, localization and fixing techniques are needed.* Our study indicates that 90.8% of the DBs are introduced in environment setup, while only 37.7% of the DBs are exposed in environment setup. Thus, it may indicate that many DBs stay undetected until later lifecycle stages. To detect or localize DBs as early as possible, one possible remedy is to identify the dependencies currently adopted in the DL stack, and then check against our dependency knowledge graph to detect potential dependency constraint violations. Here the challenge is to automatically identify all heterogeneous dependencies as well as their versions across the entire DL stack. Along this direction, Tan et al. [58] proposed a technique to identify homogeneous dependencies at the *Application* and *Library* layer. Moreover, as many DBs are caused by software bugs or API incompatibilities, fine-grained call graph analysis is needed to accurately detect and localize DBs, i.e., to decide whether such bugs or incompatible APIs are in the execution path and thus can be triggered. Besides, our study indicates that questioners are often unaware of the introducing dependencies of the DBs, which calls for automated DB localization techniques. Once a DB is localized, automated fixing techniques can use the fix patterns derived from our study to fix it. However, the challenge is to decide which fix pattern or combination of fix patterns is applicable and how a fix pattern is instantiated. A way is to use search-based approach by applying fix patterns to generate potential fixes and using the dependency knowledge graph to decide the fix fitness.

Fourth, *dependency upgrading and migration techniques are needed.* Our introducing stage analysis uncovers that some DBs are introduced in deployment and maintenance. More specifically, DL stack in deployment environment can be different from the one in development environment. Hence, dependency migration techniques are needed to check whether dependencies in development environment can be replaced with the ones in deployment environment. Besides, dependency versions can be upgraded for the benefit of fixed bugs and improved features. However, it may also introduce incompatibilities. Therefore, dependency upgrading techniques are needed to analyze API changes and assess the risk in terms of potential DBs and the effort in terms of potential code adaptation.

7.2 Application for Usefulness Demonstration

To demonstrate the usefulness of our implications, we design a prototype to automatically detect and fix DBs.

Prototype Design. Our prototype has one knowledge base, i.e., *dependency constraint knowledge*, and two components, i.e., *DB detection* and *DB fixing*. To collect dependency constraint knowledge, we target at the documentations of TensorFlow, Pytorch and Keras as i) they expose and introduce the most DBs at the *Library* layer; and ii) their documentations often list requirements for dependencies at lower layers, e.g., Python at the *Runtime* layer, CUDA and cuDNN at the *Driver* layer, Linux at the *OS/Container* layer. We then manually extract dependency constraints from their online documentations via reading installation guides of each version, where they are either described in natural language or illustrated with a table. Each dependency constraint is denoted as a tuple $\langle dep_a, dep_b, v_a, v_{b1}, v_{b_2}\rangle$ where version v_a of dependency dep_a depends on dep_b under the condition that the version of dep_b is within the range of v_{b1} and v_{b_2}. v_{b_2} can be null to represent an opening scope. Overall, we collect 588 dependency constraints in 3 days.

Our prototype takes a Docker image as an input, and detects whether it contains a DB. If yes, it also tries to fix it. To detect DBs, we first need to identify dependencies used in the DL stack. To this end, we support two package managers, i.e., PyPI and Conda, at the *Library* layer. We obtain the virtual environment location of PyPI virtualenv by *"find / | grep bin/activate"*, or the environment names of Conda by *"conda env list"*. Then, we activate the corresponding environments by *"source <path>/bin/activate"* or *"conda activate <envname>"*, where we retrieve the whole list of dependency versions under each environment. For the *Runtime* layer, we identify Python runtime in PATH (*"echo $PATH"*), where there exists an executable named *"python"* or *"python3"*. For the *Driver* layer, we use *"nvcc –version"* to identify installed version of CUDA and *"which nvcc"* to identify installed path of CUDA. Under *"<cuda_path>/lib64"*, we find the cuDNN version by checking if there exists a dynamic linking library of cuDNN (i.e., *cudnn.so.<version>*). For the *OS/Container* layer, we use *"uname -a"*, *"cat /etc/centos-release"*, *"lsb_release -a"*, etc. to identify hosting OS. We do not identify hardware versions as the Docker image does not contain hardware information.

Then, we search the identified dependency versions of each environment for a combination of dependencies $\langle dep_a, dep_b, v_a, v_b\rangle$ that violates a dependency constraint $\langle dep_a, dep_b, v_a, v_{b1}, v_{b2}\rangle$, which is regarded as a DB. We first anchor dep_a at version v_a and change dep_b's version indicated in version constraint from v_{b1} to v_{b2}. Meanwhile, as dep_b's version changes, we also change dep_c's version to satisfy version constraint in $\langle dep_b, dep_c, v_b, v_{c1}, v_{c2}\rangle$ if needed. The process is conducted recursively. If there is no satisfied version combinations, we anchor dep_a's version into a newer version of v_a using our knowledge base and repeat the above process. Once a satisfied version is found, we change the dependency version by running uninstall and install script to fix the DB.

Comparison with Related Tools. We find and review three closely related tools. DockerizeMe [22] is a tool to infer the dependencies needed to execute a Python code snippet without import error. The inference is based on a knowledge base which contains packages, their versions and resources, and the relationships between them. The knowledge base is built by applying static and dynamic analysis to top ten thousand Python packages on PyPI and applying association rule mining to public GitHub Python projects. PyEGo [72] extends the knowledge base of DockerizeMe by further including Python interpreters and system libraries, and achieves a better accuracy on inferring compatible dependencies. DockerizeMe and PyEGo mainly support packages installed by commands of *pip* and *apt*. Different from DockerizeMe and PyEGo, our prototype extracts dependencies and version constraints knowledge from official documentation, and supports package installation commands beyond *pip* and *apt*. While further work is needed to automate the knowledge extraction, our approach can offer more generalizability across different types of dependencies at different DL stack layers.

Different from the knowledge-based inference in DockerizeMe and PyEGo, PyDFix [40] takes a trial and error approach, i.e., it first identifies dependency errors and possible dependencies causing the errors from build log, and then iteratively re-runs the build with intermediate patches until the error disappears. Differently, our prototype does not rely on error logs since not all DBs indicate explicit error logs or reveal dependency names in their error logs.

Effectiveness Evaluation. To evaluate our prototype, we reproduce DBs from our study and export them as Docker images. We randomly sample 80 DBs from our study, and successfully reproduce 18 DBs. The reasons of unsuccessful reproduction are twofold. First, the exposing or introducing dependency of DBs locates in *Hardware* or *OS/Container* which does not match with our machines. Second, only part of the DL stack is revealed in the posts or issues, and hence we fail to derive the full DL stack to reproduce DBs.

Our prototype successfully detects and fixes 8 of the 18 DBs. Three DBs caused by *Mismatched Software*, two DBs caused by *Buggy Software Version* and one DB caused by *Unsuccessful Installation* are not detected as our prototype is focused on violated version constraints. Of the twelve DBs caused by *Incompatible Software Version*, five DBs are detected and fixed using version constraint between TensorFlow and CUDA, two are detected and fixed using version constraint between CUDA and cuDNN, and one is detected and fixed using version constraint between TensorFlow and cuDNN. The other four DBs are not detected because the root cause dependencies are not in our scope of dependency constraint knowledge acquisition. These results demonstrate the potential of our prototype.

Moreover, we try to apply PyEGo and PyDFix to fix the 18 DBs. Notice that DockerizeMe is not selected because PyEGo has achieved better performance than it. We successfully run PyEGo against the

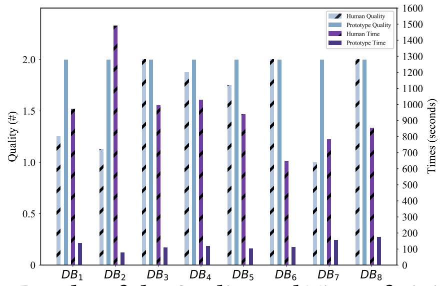


Figure 8: Results of the Quality and Time of Fixing 8 DBs

18 DBs. It successfully detects and fixes only one DB. It successfully detects 11 DBs, but generates wrong version recommendation on all of them. Besides, it fails to detect the rest 6 DBs. Unfortunately, we fail to launch PyDFix due to the limited setup documentation. However, PyDFix relies on analyzing error logs to fix DBs. Consequently, we can conclude that PyDFix are unable to fix at least 13 of the DBs since these DBs produce normal outputs instead of error logs. These results indicate the potential of our prototype.

Human Study. We observe from our effectiveness evaluation that our prototype takes 2 seconds for the DBs that are not successfully fixed, and these DBs are even not detected by our prototype. In other words, it takes negligible time for our prototype to determine whether a given DB is out of the scope of our prototype. Therefore, we are interested to investigate how much effort can be saved for developers for the DBs that are in the scope of our prototype.

To this end, we conduct a human study with 8 participants to manually fix the 8 DBs that can be automatically fixed by our prototype. The participants are recruited voluntarily at our college who are familiar with Linux shell and packages, and has sufficient background in deep learning. Four participants have worked on at least one or two research projects that employ DL techniques, and the other four participants have hands-on experience with open source DL projects. The tasks are 8 reproduced DBs in a Docker environment where the error trace of each DB could be invoked via a command (i.e., *python script.py*). The participants are told that the error is caused by a DB and they are required to locate and fix the DBs with their expertise and any online resources. The order of the tasks are randomized for each participant to avoid bias.

We use two indicators to compare participants' manual fixes and our automated fixes. The first indicator is the quality of the fix in each task. We use 2 to indicate a successful and perfect fix, 1 to indicate a successful but imperfect fix, and 0 to indicate an unsuccessful fix. The success of the fix is judged by the dismissing of the DB's errors when re-launching scripts. The perfection and imperfection of the fix is judged by two of the authors on whether the fix steps would have any side effect. After the discussion and mutual agreement from two of the authors, a final quality is resolved. The second indicator is the consumed time on finishing each task.

Fig. 8 shows the result of fix quality and time. In terms of quality, all 8 participants obtain full score in 3 DBs. The rest 5 DBs are not fixed successfully and perfectly by all. 19 participant-DB pairs are not fully scored. Specifically, we assign 1 to 17 participant-DB pairs. Of these 17 participant-DB pairs, 2 participant-DB pairs fix a DB by using soft links to redirect the incorrect dependency into a correct dependency and 3 participant-DB pairs fix a DB by replacing dynamic linked libraries (i.e., change a correct dependency's file name into the original one using *mv*). They are imperfect because such tricks are unstable and confuse other users. The rest 12 participant-DB pairs freshly reinstall TensorFlow using an up-to-dated version. We assign them to 1 because setting up a new environment carries the risk of disrupting the initial environment, making it impractical when there are multiple users and applications. We also assign 0 to 2 participant-DB pairs. They fail to fix as it still has the reported error. In terms of time, none of the manual fix from 64 participant-DB pairs surpasses our prototype. The manual fix takes averagely 8.8 times longer than our prototype. Generally, our prototype achieves a higher quality of 2 against the human group with a score of 1.4, and costs averagely 109.2 seconds against the human group with averagely 963.0 seconds. Therefore, our prototype can be useful for developers to provide high quality fix and greatly saving fixing time.

8 RELATED WORK

Dependency Bugs. Dependency bugs have been explored for different ecosystems, e.g., Debian and Red Hat [6], JavaScript [44], Java [23, 38, 65–67], Python [40, 64], C/C++ [31] and Go [63]. To the best of our knowledge, our work is the first to systematically investigate dependency bugs in DL ecosystem.

Deep Learning Bugs. Empirical studies have been conducted to characterize bugs in DL systems. Some are focused on a general scope of bugs [25, 27, 28, 42, 76], and others are focused on a specific type of bugs [10, 14, 62, 73, 75]. These studies uncover partial characteristics of dependency bugs in DL stack. There lacks a comprehensive study to characterize dependency bugs in DL stack, and our work fills this gap. Several advances have also been made to detect DL bugs, e.g., numerical bugs [68, 71, 77] and shape bugs [32, 33, 61, 69]. However, little attention has been received to detecting dependency bugs in DL stack, and our work sheds light on it.

Empirical Studies about DL. Many studies have empirically investigated various aspects in developing, deploying and maintaining DL systems [4, 5, 13, 16–18, 37, 39, 41, 45, 46, 59, 74] and DL frameworks [19, 20, 34, 35, 58, 78]. These studies motivate the importance of dependency management. For example, incompatible dependency installation or environment setup is recognized as a common challenge [4, 13, 74]. However, they lack an in-depth analysis of the characteristics. Our work is inspired by them to systematically characterize dependency bugs across the DL stack.

9 CONCLUSIONS

We have conducted the first comprehensive study to characterize DBs across the entire DL stack. We provide useful findings to raise the awareness of DBs in DL stack in the DL community, and provide actionable implications for developers and researches.

10 DATA AVAILABILITY

The data of our study is available at https://dl-dep.github.io.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2021ZD0112903) and the China Postdoctoral Science Foundation (2022M720768).

REFERENCES

[1] 2022. *Docker GPU.* Retrieved March,2022 from https://developer.nvidia.com/blog/nvidia-docker-gpu-server-application-deployment-made-easy/

[2] 2022. *Josh - A Practical Guide for Data Scientists Using GPUs with TensorFlow.* Retrieved March,2022 from http://www.pattersonconsultingtn.com/blog/datascience_guide_tensorflow_gpus.html

[3] Syed Ahmed and Mehdi Bagherzadeh. 2018. What do concurrency developers ask about? a large-scale study using stack overflow. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement.* 1–10. https://doi.org/10.1145/3239235.3239524

[4] Moayad Alshangiti, Hitesh Sapkota, Pradeep K Murukannaiah, Xumin Liu, and Qi Yu. 2019. Why is developing machine learning applications challenging? a study on stack overflow posts. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–11. https://doi.org/10.1109/ESEM.2019.8870187

[5] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice.* 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[6] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why do software packages conflict?. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories.* 141–150. https://doi.org/10.1109/MSR.2012.6224274

[7] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going big: a large-scale study on what big data developers ask. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering.* 432–442. https://doi.org/10.1145/3338906.3338939

[8] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering* 22, 6 (2017), 3117–3148. https://doi.org/10.1007/s10664-017-9510-8

[9] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering.* 507–511. https://doi.org/10.1109/SANER.2018.8330249

[10] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* https://doi.org/10.1145/3540250.3549123

[11] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look At Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. https://doi.org/10.1109/TSE.2022.3191353

[12] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision.* 2722–2730. https://doi.org/10.1109/ICCV.2015.312

[13] 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. https://doi.org/10.1145/3368089.3409759

[14] 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 IEEE/ACM 43rd International Conference on Software Engineering.* 674–685. https://doi.org/10.1109/ICSE43902.2021.00068

[15] Wei Cheng, Xiangrong Zhu, and Wei Hu. 2022. Conflict-Aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph. In *Proceedings of the 44th International Conference on Software Engineering.* 451–461. https://doi.org/10.1145/3510003.3510078

[16] Alex Cummaudo, Rajesh Vasa, Scott Barnett, John Grundy, and Mohamed Abdelrazek. 2020. Interpreting Cloud Computer Vision Pain-Points: A Mining Study of Stack Overflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* 1584–1596. https://doi.org/10.1145/3377811.3380404

[17] Görkem Giray. 2021. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software* 180 (2021), 111031. https://doi.org/10.1016/j.jss.2021.111031

[18] 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.* 810–822. https://doi.org/10.1109/ASE.2019.00080

[19] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. 2020. An empirical study of the dependency networks of deep learning libraries. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution.* 868–878. https://doi.org/10.1109/ICSME46990.2020.00116

[20] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering* 25, 4 (2020), 2694–2747. https://doi.org/10.1007/s10664-020-09819-6

[21] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* 223–238. https://doi.org/10.1145/2694344.2694347

[22] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering.* 328–338. https://doi.org/10.1109/ICSE.2019.00047

[23] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 518–529. https://doi.org/10.1145/3368089.3409689

[24] Huawei. 2020. *Overview of the Ascend AI Software Stack.* https://support.huaweicloud.com/intl/en-us/accessg-atlas200dkappc32/atlaspd_19_0001.html#atlaspd_19_0001__fig10978124614814

[25] 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 ACM/IEEE 42nd International Conference on Software Engineering.* 1110–1121. https://doi.org/10.1145/3377811.3380395

[26] Intel. 2020. *Deep Learning Reference Stack.* https://intel.github.io/stacks/dlrs/index.html

[27] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 510–520. https://doi.org/10.1016/j.infsof.2022.107004

[28] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering.* 1135–1146. https://doi.org/10.1145/3377811.3380378

[29] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency smells in Javascript projects. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3790–3807. https://doi.org/10.1109/TSE.2021.3106247

[30] Sébastien Jean, KyungHyun Cho, Roland Memisevic, and Yoshua Bengio. 2015. On Using Very Large Target Vocabulary for Neural Machine Translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing.* 1–10. https://doi.org/10.3115/v1/p15-1001

[31] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering.* 86–98. https://doi.org/10.1109/ICSE43902.2021.00021

[32] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static analysis of shape in TensorFlow programs. In *Proceedings of the 34th European Conference on Object-Oriented Programming.* 1–29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[33] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. 2021. Detecting TensorFlow Program Bugs in Real-World Industrial Environment. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering.* 55–66. https://doi.org/10.1109/ASE51524.2021.9678891

[34] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society.* 1–10. https://doi.org/10.1145/3377815.3381377

[35] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2021. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. *Empirical Software Engineering* 26, 2 (2021), 1–36. https://doi.org/10.1007/s10664-020-09917-5

[36] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 617–628. https://doi.org/10.1145/3368089.3409760

[37] 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 World Wide Web Conference*. 1234–1244. https://doi.org/10.1145/3308558.3313639

[38] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 106–117. https://doi.org/10.1109/SANER.2018.8330201

[39] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software Engineering for AI-Based Systems: A Survey. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 1–59. https://doi.org/10.1145/3487043

[40] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451. https://doi.org/10.1145/3460319.3464797

[41] Amin Nikanjam and Foutse Khomh. 2021. Design Smells in Deep Learning Programs: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 332–342. https://doi.org/10.1109/ICSME52107.2021.00036

[42] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. 2022. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. *Automated Software Engineering* 29, 1 (2022), 1–32. https://doi.org/10.1007/s10515-021-00313-x

[43] Ziad Obermeyer and Ezekiel J Emanuel. 2016. Predicting the future—big data, machine learning, and clinical medicine. *The New England journal of medicine* 375, 13 (2016), 1216.

[44] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751. https://doi.org/10.1145/3180155.3180184

[45] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. 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*. 771–783. https://doi.org/10.1145/3324884.3416545

[46] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*. 2503–2511.

[47] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[48] StackOverflow. 2022. *AVX*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/51599488/

[49] StackOverflow. 2022. *Conflict GPU*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/42473052

[50] StackOverflow. 2022. *CuDNN Not Installed*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/43721690/

[51] StackOverflow. 2022. *Disable Secure Boot*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/67045622/

[52] StackOverflow. 2022. *GPU Device is not Found*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/44269047/

[53] StackOverflow. 2022. *Keras and tf.keras*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/56851895/

[54] StackOverflow. 2022. *MacOS SIP*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/44354694/

[55] StackOverflow. 2022. *Prebuilt Won't Work*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/58610020/

[56] StackOverflow. 2022. *Switch OS for Distributed TensorFlow*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/52041077/

[57] StackOverflow. 2022. *Tensorflow Bug*. Retrieved August 29, 2022 from https://stackoverflow.com/questions/48713335/

[58] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An Exploratory Study of Deep Learning Supply Chain. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*. https://doi.org/10.1145/3510003.3510199

[59] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An empirical study of refactorings and technical debt in Machine Learning systems. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 238–250. https://doi.org/10.1109/ICSE43902.2021.00033

[60] TensorFlow. 2022. *Install TensorFlow*. Retrieved August 29, 2022 from https://www.tensorflow.org/install/gpu

[61] Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. *CoRR* abs/2011.13452 (2020).

[62] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are Machine Learning Cloud APIs Used Correctly?. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 125–137. https://doi.org/10.1109/ICSE43902.2021.00024

[63] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. HERO: On the Chaos When PATH Meets Modules. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 99–111. https://doi.org/10.1109/ICSE43902.2021.00022

[64] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring dependency conflicts for python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 125–135. https://doi.org/10.1145/3377811.3380426

[65] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 319–330. https://doi.org/10.1145/3236024.3236056

[66] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could i have a stack trace to examine the dependency conflict issue?. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. 572–583. https://doi.org/10.1109/ICSE.2019.00068

[67] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. 2021. Will Dependency Conflicts Affect My Program's Semantics. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3057767

[68] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[69] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2021. Tensfa: Detecting and Repairing Tensor Shape Faults in Deep Learning Systems. In *Proceedings of the IEEE 32nd International Symposium on Software Reliability Engineering*. 11–21. https://doi.org/10.1109/ISSRE52982.2021.00014

[70] Rongxin Wu, Minglei Chen, Chengpeng Wang, Gang Fan, Jiguang Qiu, and Charles Zhang. 2022. Accelerating Build Dependency Error Detection via Virtual Build. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. https://doi.org/10.1145/3551349.3556930

[71] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. 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*. 627–638. https://doi.org/10.1145/3468264.3468612

[72] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. In *Proceedings of the 44th International Conference on Software Engineering*. 1245–1256. https://doi.org/10.1145/3510003.3510127

[73] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering*. 1159–1170. https://doi.org/10.1145/3377811.3380362

[74] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020

[75] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 359–371. https://doi.org/10.1109/ICSE43902.2021.00043

[76] 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*. 129–140. https://doi.org/10.1145/3213846.3213866

[77] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. 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*. 826–837. https://doi.org/10.1145/3368089.3409720

[78] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the mystery of api evolution in deep learning frameworks a case study of tensorflow 2. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. 238–247. https://doi.org/10.1109/ICSE-SEIP52600.2021.00033

Received 2023-02-02; accepted 2023-07-27