Demystifying Developers’ Issues in Distributed Training of Deep Learning Software

Diandian Gu  
Peking University  
Beijing, China  
gudiandian1998@pku.edu.cn

Zhenpeng Chen  
University College London  
London, UK  
zp.chen@ucl.ac.uk

Yuanqiang Liu  
Peking University  
Beijing, China  
yuanqiangliu@pku.edu.cn

Zili Zhang  
Peking University  
Beijing, China  
zzlcs@pku.edu.cn

Yun Ma  
Peking University  
Beijing, China  
mayun@pku.edu.cn

Xin Jin  
Peking University  
Beijing, China  
xinjinpku@pku.edu.cn

Xuanzhe Liu  
Peking University  
Beijing, China  
liuxuanzhe@pku.edu.cn

ABSTRACT
Deep learning (DL) has been pervasive in a wide spectrum of nowadays software systems and applications. The rich features of these DL based software applications (i.e., DL software) usually rely on powerful DL models. To train powerful DL models with large datasets efficiently, it has been a common practice for developers to parallelize and distribute the computation and memory over multiple devices in the training process, which is known as distributed training. However, existing efforts in the software engineering (SE) research community mainly focus on issues in the general process of training DL models. In contrast, to the best of our knowledge, issues that developers encounter in distributed training have never been well studied. Given the surging importance of distributed training in the current practice of developing DL software, this paper fills in the knowledge gap and presents the first comprehensive study on developers’ issues in distributed training. To this end, we extract and analyze 1,054 real-world developers’ issues in distributed training from Stack Overflow and GitHub, two commonly used data sources for studying software issues. We construct a fine-grained taxonomy consisting of 30 categories regarding the fault symptoms and summarize common fix patterns for different symptoms. Based on the results, we suggest actionable implications and research avenues that can potentially facilitate the future development of distributed training.

1 INTRODUCTION
Deep learning (DL) has been widely adopted in numerous software applications, ranging from supporting daily activities (e.g., speech-to-text [28]) to safety-critical tasks (e.g., autonomous vehicles [29]). The rich features of these DL based software applications (i.e., DL software) rely on powerful DL models. To increase the accuracy of DL models, on one hand, a substantial amount of training data is required; on the other hand, sophisticated DL model architectures such as BERT [35] have emerged. As data increases in size and DL models in complexity, the computational intensity and memory demands of DL increase proportionally [27]. Therefore, it has been a common practice for developers to parallelize and distribute computation and memory to multiple devices (e.g., GPUs and server machines) during the training process of DL models, which is known as distributed training [35, 38].

The increasing dependence of current software applications on DL makes DL a crucial topic in the software engineering (SE) research community. In particular, there have been many empirical studies on characterizing the faults in DL programs that make use of DL frameworks (e.g., TensorFlow) [40, 41, 62, 63]. However, faults related to distributed training are rarely covered and discussed. Among these studies, only Humbatova et al. [40] mention a fault about data parallelism in distributed training. Besides, Zhang et al. [64] characterize job failures in a cloud-based DL platform. Even though they include distributed training jobs in their dataset, they do not discuss the particularity of distributed training and provide only generic guides for all DL developers.

In fact, compared to non-distributed training, distributed training has its unique challenges, since it involves additional modules (e.g., for communication among devices) and algorithms (e.g., for cooperation among devices). As a result, developers encounter a variety of issues about it in practice and ask about these issues frequently on developers’ Q&A forums. For example, some developers find it difficult to configure communication between multiple devices involved in distributed training [4] and complain that they cannot achieve the expected training speedup [12]. Moreover, some developers report that training may be stuck due to the drop-out of involved devices [21]. Unfortunately, as mentioned before, these developers’ issues have not been comprehensively uncovered and well characterized in existing studies.

To fill in the knowledge gap, this paper presents the first comprehensive study on developers’ issues in distributed training of DL software. Given the surging importance of distributed training in the current practice of developing DL software, understanding the relevant issues that developers meet is essential, as it can provide guidance for developers to prevent, detect, and fix common faults associated with distributed training.
We focus our study on the three most popular DL frameworks (i.e., TensorFlow [25], PyTorch [53], and Keras [19]) that support distributed training and a widely used DL framework specifically designed for distributed training (i.e., Horovod [57]) to construct the dataset of our interest. Specifically, we collect a dataset of 1,054 distributed-training-related developers’ issues that occur during the use of these frameworks from Stack Overflow (SO) and GitHub, two commonly used data sources for studying software issues [30, 36, 40, 41, 63]. Based on the dataset, we aim to answer three research questions:

RQ1: topics in how-to questions. How-to questions present the distributed training knowledge that developers are inexpert at, which tends to induce future faults. By studying frequent topics in how-to questions about distributed training, we can summarize the common difficulties that developers face in training DL models in a distributed way. The detailed results are reported in Section 4.

RQ2: symptoms of faults. By constructing a comprehensive taxonomy of fault symptoms related to distributed DL, we present frequent fault symptoms neglected by previous work. The detailed results are reported in Section 5.

RQ3: fix patterns. By studying fix patterns for each symptom, we summarize their characteristics and commonalities. The detailed results are reported in Section 6.

This paper offers findings that provide insights on better distributed DL practice for developers, future research topics for researchers, and suggestions for DL framework vendors. We also provide the labeled dataset as well as the scripts used in this study1 as an additional contribution to the research community.

2 BACKGROUND

Distributed training is a subfield of DL, which parallelizes and distributes the computation and memory of training across multiple devices, e.g., GPUs, TPUs, and server machines. With the growing computation demands for DL, distributed training becomes more and more important. It involves how to split training tasks, allocate computation resources, and coordinate various functional modules among different devices to achieve the balance of training speed and accuracy. To facilitate the understanding of distributed training, we present its common workflow in Figure 1.

Workflow. A distributed training task should first be partitioned so that they can run parallelly on different devices (❶). The two

most predominant parallelization ways are data parallelism and model parallelism [51]. For data parallelism, the training data is split into non-overlapping chunks, and then these data chunks are respectively fed into different devices that load an identical copy of the DL model [45]. For model parallelism, the DL model is split, and then each device loads a different part of the DL model for training [34]. Through data/model parallelism, the training data and the DL model are distributed on different devices. Next, every device trains its own model with the data allocated to it (❷). During this process, the devices communicate with each other to transfer essential data and to synchronize the training progress on them. Finally, the trained models distributed on different devices are aggregated to obtain a new global model (❸). Note that, in the above workflow, distributed training tasks also rely on the environment, including hardware characteristics of devices, runtime environment (e.g., memory), network setting, installed dependency libraries, etc.

Scope. In this paper, we focus our study on the training process of DL models on multiple devices. First, we analyze the frequent topics of general how-to questions about distributed training (RQ1). Then, we analyze the faults that occur during distributed training. Specifically, we analyze the fault symptoms (RQ2) and distill common fix patterns for different symptoms (RQ3). Note that there are two kinds of faults during distributed training: distributed-specific faults, which are caused by distributed-specific reasons (e.g., communication failure and invalid data partition), and non-distributed-specific faults, which are caused by non-distributed-specific reasons (e.g., wrong type of input data). Some of them share common symptoms (e.g., out of memory) although they are caused by different reasons. To show the whole picture of fault symptoms in distributed training, in RQ2, we construct our taxonomy based on both kinds of faults. However, the fix patterns for non-distributed-specific faults have been comprehensively studied in previous work [40–42, 62, 63]. Therefore, in RQ3, we focus on only the fix patterns of distributed-specific ones.

3 METHODOLOGY

To characterize developers’ issues in distributed training, we collect and analyze relevant SO questions and GitHub issues. The overview of the methodology is illustrated in Figure 2.

3.1 Data collection

Since distributed training is mainly supported by state-of-the-art frameworks, we collect developers’ issues that occur during the use of relevant frameworks to construct the dataset of our interest. Specifically, we focus our study on Horovod, which is the most popular framework specifically designed for distributed training and has been widely adopted in both academia [7, 8, 64] and industry [5–9]. In addition, we also consider the three most popular DL

Figure 2: Overview of the methodology

1They will be made public later.
frameworks (i.e., TensorFlow, PyTorch, and Keras) [11, 14, 23, 24], since all of them support distributed training.

3.1.1 Mining SO. SO is one of the most popular Q&A websites where developers ask for help on unresolved technical issues [30]. It has been a commonly used data source for studying developers’ software issues [30, 36, 40–42, 61, 63]. Moreover, SO users range from novices to experts [63], increasing the diversity of collected issues.

First, we download the entire SO dataset from the official Stack Exchange Data Dump [22] on June 5, 2021. The dataset (denoted as set $A$) contains SO questions generated from July 31, 2008 to June 4, 2021. Each question has one to five tags indicating its topics. From $A$, we extract 97,016 questions tagged with at least one of the four selected frameworks and denote these questions as set $B$.

Since SO questions in $B$ may contain noise that is not related to distributed training, we need to further extract the distributed-training-related questions from $B$. To this end, we randomly extract 1,000 questions from $B$ and discuss these questions carefully to identify a set of keywords that are highly related to distributed training. Next, we evaluate the recall level of these keywords, i.e., the percentage of the distributed-training-related posts that can be identified by these keywords. Specifically, we randomly select another 500 posts to perform the evaluation and also identify new keywords from them. We add the new keywords into the keyword set after the evaluation. We repeat the above evaluation process four times until the keyword set can achieve a recall of 90%. Note that here we do not consider the precision level of these keywords since any misidentified post can be filtered out during the refining process in Section 3.1.3 and will not threaten the validity of our results. As a result, we have the following keywords: ”distributed”, ”distribute”, ”parallel”, ”parallelized”, ”parallelism”, ”multi-server”, ”multi_gpus”, ”multi-gpus”, ”multi-gpu”, ”data-parallel”, ”model-parallel”, ”data-parallel”, ”workers”, ”multi-machine”, ”multiple gpus”, ”multiple gpus”, ”multiple machines”, ”multiple machine”, ”multiple server”, ”multiple servers”). We perform a case-insensitive search within the title and body (excluding code snippets) of each question in $B$ and identify 2,476 questions (denoted as set $C$) that contain at least one of these keywords. Finally, we follow previous studies [31, 40, 49] to exclude questions that do not have an accepted answer, ensuring that we consider only questions with a confirmed answer. As a result, we obtain a total of 706 questions in set $C$.

3.1.2 Mining GitHub. GitHub is another commonly used data source for studying software issues [31, 36, 40, 63]. In line with previous work [31, 36], we mine issues posted in the official GitHub repositories (”GitHub issues” for short) of the selected frameworks to identify developers’ issues that occur during their use. Compared to commits, GitHub issues contain more information such as original reports and developers’ discussions [36]. This characteristic makes GitHub issues more suitable for studying the difficulties and faults encountered by developers. In addition, on GitHub, framework vendors employ repository-specific keywords to label different types of GitHub issues, such as bug reports, feature requests, users’ questions, etc. Following previous work [31, 36], we leverage these labels of GitHub issues to help us identify relevant developers’ issues. We use the GitHub search API [17] to extract these GitHub issues from the framework repositories on June 5, 2021. The detailed process is as follows.

For each framework, the first two authors jointly examine the labels in its GitHub repository to determine which labels are related to distributed training and then extract GitHub issues marked with these relevant labels. Since Horovod is specifically designed for distributed training, we take all of the GitHub issues in its repository into consideration no matter which labels they are marked. As for Keras, we cannot find any labels related to distributed training, so we use the keywords identified in Section 3.1.1 to extract relevant issues. Then, for each repository, we follow previous work [31] to use labels to exclude GitHub issues about new feature requests, bugs in the framework itself, problematic documents, and feature requests. Additionally, to ensure that we consider only closed issues with a confirmed solution, GitHub issues without answers or responses are excluded with the help of labels. The remaining issues are denoted as set $D$. Table 1 shows the used labels and the number of identified GitHub issues in each repository, respectively.

3.1.3 Refining dataset. The first two authors further manually examine all the extracted posts (i.e., SO questions in $C$ and GitHub issues in $D$) to refine the final dataset. Specifically, we jointly read each post and exclude posts that (1) do not have clear descriptions or solutions, (2) fix a bug in the framework itself rather than in distributed training program, or (3) are not related to distributed training. During this process, any disagreement is discussed and resolved with the involvement of an arbitrator, who has more than four years of experience in distributed training and has published several related papers in top-tier conferences. The final dataset for our study consists of 998 posts, including 500 posts about Horovod, 319 posts about TensorFlow, 129 posts about PyTorch, and 55 posts about Keras. The scale of our dataset is comparable and even larger than those used in existing fault-related empirical studies with manual labeling [30, 31, 36, 63].

3.2 Manual labeling. To distill how-to topics, symptoms, and fix patterns, we label every post in the refined dataset manually, following the procedure below.

3.2.1 Pilot labeling. First, we randomly sample 50% of our dataset for pilot labeling. The first two authors follow an open coding procedure [56] to summarize categories for how-to topics, symptoms, and fix patterns by jointly analyzing the sampled posts. Specifically, they read all the posts carefully to understand their context and assign each post with a set of labels describing (1) the how-to topic, which describes the how-to question briefly, (2) whether the fault is specific to distributed training, (3) the fault symptom, which shows what the fault looks like, and (4) the fix pattern, which tells how a fault is resolved. These labels are optional for each post. If a post is raising a how-to question (e.g., asking how to implement a specific distributed training task or inquiring conceptual knowledge about distributed training), it is labeled with only the how-to topic. Otherwise, a post with a clear fault description is labeled with whether it is distributed-specific, the fault symptom, and the fix pattern. Then, they construct taxonomies for how-to topics, symptoms, and fix patterns by grouping similar labels together into categories. The

Note that an SO post may be tagged with multiple framework tags.
we answer RQ1 in Section 4; based on the 579 real-world faults, we determine whether they are distributed-specific, symptoms, and fix patterns of the issues asked by developers cover a wide spectrum of 9 high-level categories.

Communication is the most frequently asked topic (28.21%) by developers. More specifically, 7.37% of questions are related to communication algorithms about how devices transfer data to achieve the purpose of multi-device cooperative learning, such as ring all-reduce [2] and parameter server [47]. 5.89% of questions ask about data and model aggregation. 4.21% of questions are on the communication configuration of distributed training. 4.00% of the questions are about detailed collective communication operators (e.g., send, receive, broadcast, etc.). 3.79% are concerned about backend communication libraries like NCCL [10] and gloo [18]. The remaining questions about communication ask about synchronous or asynchronous training, such as the timing to update model parameters (i.e., weights and biases) [1].

The second most frequently asked (16.21%) topic is parallelization, which describes how DL workflows are parallelized and run cooperatively. Most questions on parallelization are about the concept, support, or details of data parallelism (13.26%), which splits training data into different chunks and model replicas on different devices train with different data. The rest are about model parallelism and other novel parallelization methods.

14.74% of how-to questions are about device usage. As multiple devices are involved in distributed training, configuring devices can be difficult; 8.00% of questions are on device configuration. Developers also ask about the supported device usage of DL frameworks (e.g., whether Horovod supports training on multiple servers [3]).

### 4 HOW-TO TOPICS (RQ1)

Figure 3 shows the hierarchical how-to topics in distributed training issues with corresponding percentages. We observe that the taxonomies are adjusted continuously in the construction process. A post is assigned to all related categories if it contains multiple how-to questions or faults. During the pilot labeling process, any disagreement is resolved by the arbitrator mentioned before. All labels, categories, and taxonomies are approved by all participants.

#### 3.2.2 Reliability analysis

For reliability analysis, the first two authors independently label the remaining posts with how-to topics, whether they are distributed-specific, symptoms, and fix patterns based on the classification criteria generated in the pilot labeling. The posts that cannot be classified into the current taxonomies are labeled with a new category named Pending. Specifically, the process of reliability analysis involves five rounds, each with 20% of the remaining posts. In each round, we measure the inter-rater agreement of the independent labeling using Cohen’s Kappa (κ) [33], which is a widely-adopted metric in SE literature [31, 42]. After each round, with the help of the arbitrator, all the authors jointly solve the conflicts of labeling results and discuss the posts in Pending category to determine whether new categories need to be added. Then all the posts in Pending are assigned to the adjusted taxonomies.

Table 2 reports the κ values for the five rounds in reliability analysis. We also report the number of new categories added for how-to-topics, fault symptoms, and fix patterns in each round. In the final round, no new category is added, indicating saturation for all categories; the κ value is 0.88, indicating almost perfect agreement [46].

In summary, among the 998 posts in pilot labeling and reliability analysis, we identify a total of 1,054 developers’ issues, including 475 how-to issues and 579 faults. We merge the categories with fewer developers’ issues (less than 1% in how-to questions or less than 1% in the faults of corresponding stage in distributed training) together as “others” category. Based on the 475 how-to questions, we answer RQ1 in Section 4; based on the 579 real-world faults, we answer RQ2 and RQ3 in Sections 5 and 6, respectively.

#### Table 1: Rules to identify distributed-training-related issues of on Github.

| Framework | Labels to identify distributed-training issues | Labels to exclude issues | Filter by keywords | # extracted GitHub issues |
|-----------|-----------------------------------------------|--------------------------|-------------------|-------------------------|
| Horovod   | N.A.                                          | bug, enhancement, update docs, wontfix, awaiting response | X                 | 742                     |
| TensorFlow | comp-dist-strat                              | type-feature, type:bug, type:docs-bug, stalled, stat:awaiting response, type:docs-feature | X                 | 133                     |
| PyTorch   | oncall: distributed, module: ddp, module: multi-gpu, module: data parallel, pt_distributed_rampup | enhancement, feature, function request | X                 | 565                     |
| Keras     | N.A.                                          | type:bug/performance, type:feature, type:docs, stale, Enhancement, stat:awaiting response | ✓                 | 251                     |

#### Table 2: The process of reliability analysis. The third column shows the number of newly added categories for how-to-topics, symptoms, and fix patterns in each round, respectively.

| Round | # analyzed issues | # new categories | κ    |
|-------|------------------|------------------|------|
| 1     | 100              | 3/3/3            | 0.37 |
| 2     | 100              | 1/4/6            | 0.65 |
| 3     | 100              | 0/2/5            | 0.74 |
| 4     | 100              | 0/2/2            | 0.80 |
| 5     | 99               | 0/0/0            | 0.88 |
| Total | 499              |                  | 0.88 |

Figure 3: Topics in how-to-questions of distributed training
The rest are questions on memory usage, device utilization, device information and monitoring, etc.

Another topic that developers are concerned about is the performance of distributed training, accounting for 14.53% of all how-to questions. Among them, 10.74% are about the efficiency and accuracy of distributed training compared to non-distributed methods. Developers also ask about how to profile performance.

Overall, the how-to questions vary from naive concepts (e.g., basic knowledge) to very advanced algorithms (e.g., synchronization and aggregation), from general questions (e.g., training efficiency) to particular details (e.g., network configuration). This diversity may owe to the huge differences of novices’ and experts’ posts on SO and GitHub, both of which reveal the difficulties and vulnerabilities in distributed training. Besides communication, which is asked most, parallelization, device usage, and performance are also asked frequently, indicating that different aspects of distributed training are all noteworthy.

Finding 1: Developers ask a wide range (9 high-level categories) of how-to topics on distributed training. Communication (28.21%), parallelization (16.21%), device usage (14.74%), and performance (14.53%) are asked most frequently.

5 SYMPTOMS (RQ2)

Figure 4 presents the hierarchical taxonomy of fault symptoms in distributed training. The root node consists of four children nodes, which are linked to four stages of distributed training. Each leaf node represents a category. We use a number in the top right corner to represent the number of faults assigned to such a category and another number in brackets to represent the distributed-specific faults in this category. There are 579 faults in total and 522 of them are distributed-specific.

Finding 2: We construct a taxonomy of 30 fault symptom categories related to four stages in distributed training, indicating the diversity of faults that developers meet in distributed training. 90.16% of faults in our dataset are distributed-specific.

Package build & import (A). To write distributed training programs, developers import certain modules (e.g., `torch.distributed`) or build and install distributed training frameworks (e.g., Horovod) in their codes. Faults that appear in this stage are included in the package build & import category, accounting for 13.30% of the distributed training faults. Only one fault in this stage is not specific to distributed training, 63.64% of faults in this stage happen when installing and building frameworks from source (i.e., installation & build failure (A.1)). Many developers reported that the error messages in this stage are difficult to understand [16]. This makes it difficult for developers to resolve such faults and makes it difficult for us to further classify this category. Developers might also fail to import framework packages or certain package modules even though they have already installed frameworks successfully (i.e., import error (A.2)).

Communication setup (B). Communication setup is an essential step in distributed training when devices build up a topology to prepare for the communication in training. 22.45% of faults related to distributed training show symptoms in this stage. As there is no need to set up communication in non-distributed training, all of the faults in this stage are specific to distributed training. Unexpected protocol (B.1) happens when devices do not communicate through the network protocol that developers set to. Setup failure (B.2), which covers up to 63.08% of faults in this stage, is triggered when the devices cannot access each other correctly. Besides, there are cases when the whole program is stuck at the communication setup stage or crashes because of timeout. We classify these issues into hang & timeout (B.3) category.

Data and model preparation (C). Before training, developers load or download training datasets; they also load or construct DL models to be trained. Then, as described in Figure 1, developers split the dataset and models, and then distribute them to multiple devices for distributed training. Faults that appear in the above steps are included in the data and model preparation category. We observe only a few related cases (8.81%) in the entire dataset; 19.61% of faults in this stage can also happen in non-distributed training. We use communication error (C.1) to refer to program crashes because of unsuccessful data transfer in this stage. Model loading failure (C.2) occurs when developers cannot load pre-trained models into memory. Device error (C.3) refers to program crashes because of invalid device assignment. Distributed training might also hang or crash because of timeout (i.e., `hang & timeout` (C.4)) in this stage. Developers sometimes encounter problems using APIs related to dataset (i.e., `data loading failure` (C.4), partition error (C.6), and iteration failure (C.7)).

Finding 3: Although only a small portion of faults (8.81%) occur in data and model preparation, these faults cover up to eight different symptom categories. 19.61% of faults in this stage are not specific to distributed training.

Training & Evaluation (D). Training & evaluation (D) is the most important stage of DL. It is also the largest category in our taxonomy, including a wide range of issues related to all facets of training and evaluation. We observe 321 faults that occur during this stage, accounting for 55.44% of all the identified faults and covering 17 symptom categories. 46 of them (14.33%) are not distributed-specific. As there are too many symptoms in this stage, we classify these symptoms into two sub-categories: program failures (D.1) and unexpected performance (D.2). Program failures (D.1) refer to faults that lead to program crashes. Unexpected performance (D.2) refers to cases when there are no crashes but the programs do not behave as developers expect.

Finding 4: Most (i.e., 55.44%) of faults in distributed training occur in the training & evaluation, covering a variety of symptoms (i.e., 17 categories). Up to 14.33% of the faults in training & evaluation are not distributed-specific.

There are various program failures (D.1) symptoms in this stage. 3.12% of faults are triggered due to reference to non-exist variables or functions (i.e., `attribute not found` (D.1.1)). As common symptoms in both training & evaluation and data and model preparation, communication error (D.1.2) and device error (D.1.3) account for 9.97%
and 8.41% of faults in training & evaluation, respectively. Not Implemented Error (D.1.4) happens when developers use functions or methods not implemented by frameworks. Checkpoint nonfunctioning (D.1.5) is triggered when developers fail to save DL models. 8.72% of the faults occur when there is illegal memory access or no additional memory can be assigned for use (i.e., memory issue (D.1.6)). Path error (D.1.7) refers to crashes because of unfounded path references. A few faults are caused by the shape or type of a tensor not matching its expectation (i.e., tensor mismatch (D.1.8)). Graph execution error (D.1.9) occurs when a program crashes because of improper computational graph that represents the structure of the DL model, even though no symptom was shown before this stage. Sometimes, programs try to read or write an illegal memory location and trigger segmentation fault (D.1.10). Even though some of these symptoms occur frequently in non-distributed training as well, the root causes and fix patterns of them might be specific to distributed settings. We will discuss the details of root causes and fix patterns in Section 6.

Some faults do not trigger program failures explicitly, but generate problematic outputs or behave unexpectedly in training & evaluation stage. Low efficiency (D.2.1) indicates distributed training does not achieve the speed-up as expected. Such cases account for 9.03% of faults identified in the stage. Sometimes the distributed training workflow does not parallelize on devices as expected. We classify these issues into unexpected parallelization & device (D.2.2). Developers also encounter faults when the model does not converge as expected (i.e., non-convergence (D.2.3)) or programs give problematic outputs (i.e., unexpected intermediate result (D.2.4)). Hang (D.2.5) means the program is stuck. As the most common symptom in this stage, 12.77% of faults belong to hang (D.2.5).

Finding 5: Hang (D.2.5) is the most common symptom in training & evaluation, accounting for 12.77% of faults in the stage.

### 6 FIX PATTERNS (RQ3)

To capture how developers fix the observed distributed training faults, we summarize fix patterns for each symptom category. Since existing studies have already shown prevalent fix patterns for generic DL faults, here, we only focus on the fix patterns of the faults caused by distributed-specific mistakes. For the four stages in distributed training, we show the frequency of different fix patterns on their leaf categories in Figure 5, 6, 7, and 8. Due to space limit, patterns with low frequency (i.e., #faults < 7 for training & evaluation and #faults < 5 for other stages) are not shown. In each figure, X axis represents leaf categories with letter identifiers consistent with Figure 4; Y axis shows fix patterns following with their frequencies on their leaf categories in Figure 5, 6, 7, and 8. Due to space limit, patterns with low frequency (i.e., #faults < 7 for training & evaluation and #faults < 5 for other stages) are not shown. In each figure, X axis represents leaf categories with letter identifiers consistent with Figure 4; Y axis shows fix patterns following with their frequencies in the corresponding stage. We next elaborate on the prevalent fix patterns and demonstrate some real-world examples of faults and fixes. Except for the fix patterns that are already described, we present fix patterns for each stage in frequency order.

#### 6.1 Faults in package build & import

We identify five prevalent fix patterns for faults in package build & import and illustrate the distribution of these patterns on leaf categories in Figure 5.

Fix dependency installation/version & install missing dependency (26) Fix dependency path (11) Fix build/install configuration (11) Install missing dependency (8) Change hardware (5)
We identify six frequent fix patterns for faults in communication setup (such as communication libraries [10, 13, 18] and device-specific computing tools [52]). Horovod also relies on other DL frameworks (TensorFlow, PyTorch, etc.). Wrong installation or version of any dependency leads to failure in installation, build, or import. For example, a developer solved an installation failure by fixing CUDA version and numply installation (Horovod GitHub issue #161).

**Fix dependency path.** This pattern fixes 14.47% of both installation & build failure (A.1) and import error (A.2). DL frameworks set default values for the paths where dependencies are installed. If developers install dependencies in different paths, they should explain the paths to dependencies in environmental variables. For example, a developer reported that when she tried to install Horovod, the build was unsuccessful because certain header files were not founded (Horovod GitHub issue #1910). She resolved the problem by adding header files of dependencies to “CPLUS_INCLUDE_PATH” environmental variable.

**Fix build/install configuration.** 14.47% of distributed-specific faults in package build & import are resolved by fixing build or install configuration of DL frameworks, including fixing dependency library reference, fixing compilation options, etc. This fix pattern mainly resolves installation & build failure (A.1).

**Change hardware.** Sometimes developers’ hardware devices do not support the correct instruction set to build frameworks. In this case, the only approach to solving installation & build failure (A.1) is to use devices with required supports. For example, a developer resolved an installation failure after switching to a server with CPUs that support AVX (Horovod GitHub issue #1798).

**Finding 6:** We identify five frequent fix patterns for faults in package build & import. Among them, fix patterns related to dependencies resolve 59.21% of the faults.

### 6.2 Faults in communication setup

We identify six frequent fix patterns for faults in communication setup and present the distribution of these patterns in Figure 6.

**Fix communication configuration of training.** Developers can configure the world size (i.e., number of processes participating in communication), ranks (i.e., unique IDs of processes), and other configurations in distributed setting. Correctly configuring them mainly fixes setup failure (B.2) and hang & timeout (B.3), accounting for 32.31% of distributed-specific faults in the stage.

**Fix network setting of devices.** 16.92% of faults in communication setup can be resolved by fixing network settings such as ip, port, firewall, access permission, and so on. Wrong network setting is the main reason why devices cannot communicate with each other. The pattern can be adapted to all symptoms in this stage. For example, a developer reported that she could not build up the connection between two nodes because of “permission denied” (Horovod GitHub issue #467). The corresponding solution is to fix the public key setting for ssh.

**Fix consistency between devices.** The inconsistency between different devices may lead to unsuccessful communication connections. Even though the installation of dependencies or frameworks on all servers can meet the requirements of distributed training, inconsistent installation between servers might cause communication nonfunctioning. For example, a developer had the problem of being unable to build up communication connection between two nodes (Horovod GitHub issue #133). She found out the reason was that the installation configurations of Open MPI on the two nodes were different. The final solution was to reinstall Open MPI with the same configuration on the nodes. Sometimes, if some devices are ready to train DL models whereas others are not, the inconsistency of device states attributes to unsuccessful communication setup as well. A developer reported that she got an error “connection refused” (SO post #38937984). The reason for this fault was she did not successfully start training on the same number of devices as her topology configuration, leading to inconsistent device states.

**Fix framework installation/version.** This group of fixes re-install DL frameworks or switch the frameworks to a different version. As distributed training frameworks and the distributed-related modules in DL frameworks are still in development, framework vendors fix bugs inside frameworks and update frameworks versions frequently. In addition, sometimes developers should change the framework version to make it compatible with certain dependencies. Therefore, this strategy solves certain faults in communication setup.

**Fix device assignments.** In communication setup, each process should specify the device they are responsible for correctly, especially for backends that rely on GPU-GPU communication such as NCCL. Fixing device assignments mainly solves setup failure (B.2).

The remaining fix pattern has been described in Section 6.1. It is also applicable to faults in communication setup.

**Fix dependency installation/version.** Communication in distributed training depends largely on third-party communication libraries such as NCCL [10] and gloo [18]. Fixing dependency installation/version resolves 11.54% of faults that occur in communication setup. These faults mainly belong to the symptom setup failure (B.2) and hang & timeout (B.3). For instance, a developer fixed the hang in Horovod when setting up communication by changing MPI version (Horovod GitHub issue #638).

**Finding 7:** We identify six frequent fix patterns for faults in communication setup. The most common pattern is to fix communication configuration of training, resolving 32.31% of faults in this stage.

6.3 Faults in data and model preparation

The solutions for distributed-specific faults in this stage are very diverse. Only four fix patterns are frequent. These fix patterns are illustrated in Figure 7.

**Fix device assignment.** In data and model preparation, if data or model cannot be correctly assigned to corresponding devices,
there will be a *device error (C.3).* For example, as shown in Example (a) (SO post #60750288), an *AssertionError* was thrown, reporting that the program received an invalid device id. The cause is that not all of the devices she wished to use were visible to CUDA. She solved the problem by specifying the IDs of available devices.

**Fix distributed API usage.** DL frameworks provide APIs for distributed training, such as `torch.nn.parallel.DistributedDataParallel` and `tf.distribute.Strategy` in TensorFlow. Developers follow certain steps required by frameworks and write distributed training programs with these APIs. However, the complicated hyperparameters of APIs and excessive procedures are difficult for developers to follow. For example, a developer couldn’t initialize the communication topology because she forgot to call a certain API (PyTorch GitHub issue #38300). The API is indispensable in distributed training with PyTorch.

The remaining two fix patterns have already been described in Section 6.1 and Section 6.2.

**Fix framework installation/version.** Except for faults in package build & import and communication setup, reinstalling framework or switching framework versions can also avoid faults in this stage, such as *communication error (C.1), model loading failure (C.2),* and so on. This is also because only certain framework versions provide mature support for some functionalities in distributed training.

**Fix communication configuration of training.** Communication is mandatory when assigning data and models to different devices. Fixing communication configuration helps avoid symptoms such as *communication error (C.1) and hang & timeout (C.4).*

### Finding 8: Fixes in data and model preparation are diverse and scattered. Few of them show frequency.

### 6.4 Faults in training & evaluation

We identify 12 fix patterns for faults in training & evaluation stage, which includes the most symptoms and real-world faults. The distribution of these patterns is shown in Figure 8.

**Fix model construction.** Fixing how the model is constructed resolves 6.18% of distributed-specific faults in this stage, involving eight different symptoms. On one hand, model parallelism requires appropriate model partition. On the other hand, properly defining model layers and parameters (i.e., weights and biases) is essential for distributed training. For example, the symptom of the fault in Example (b) is *device error (D.1.3)* which throws "*RuntimeError: arguments are located on different GPUs*" (SO post #60799655). This is because the developer did not define a certain tensor as an instance of `torch.nn.Parameter` in her model. This results in the tensor not assigned to certain GPU devices in graph replication. The corresponding solution is fixing the definition of this tensor in model construction. Although her model is not correctly constructed, such fault does not happen in non-distributed training as there is no need for graph replication.

**Fix batch size/data partition.** This fix pattern mainly solves faults in *memory issue (D.1.6), low efficiency (D.2.1),* and *hang (D.2.5).* Batch size and data partition influence memory usage and distributed training efficiency. As distributed training introduces communication overheads and additional memory usage, only a proper batch size can make sure of high efficiency without out of memory faults. Besides, DL frameworks such as Horovod and Keras implement data parallelism naively. They require the dataset to be partitioned equally over devices. Otherwise, there might be a tensor shape mismatch problem or synchronization problem, because the number of data samples on different devices does not match up. For example, a developer encountered a tensor shape mismatch problem in distributed training (SO post #43620478). The solution was to make the number of samples divisible by `batch_size * N`, where `N` is the number of GPU devices to use.

**Fix behavior logic.** Behavior logic refers to the logical relationship between the behavior of different devices, such as profile writing, communication operations, etc. Developers make mistakes in behavior logic when they are confused with the complicated logic or unfamiliar with distributed-related API. Inappropriate behavior logic leads to conflicts in distributed training or unexpected training performance. Example (c) shows an program hang problem PyTorch GitHub issue #22834. This is because the training speeds on different devices are not exactly the same, leading to one process exits before the other one. To fix this fault, the developer needs to delete timing code so that the training or inference on each device executes exactly the same number of steps.

**Save single-device model/weights only.** This pattern applies to only model saving problems (i.e., *checkpoint nonfunctioning (D.1.5)*). In frameworks such as PyTorch and Keras, the “single-device models” and “distributed-training models” belong to different classes. In the case of unsuccessful model saving, saving the “single-device model” instead of saving model weights only is an effective workaround.

**Fix memory/core setting.** This group mainly resolves *memory issue (D.1.6)* problems. By increasing the execution memory and cores in use, more resources will be allocated, which can resolve out of memory error. Besides, modifying the configuration of how DL frameworks allocate memory is also an effective fix (SO posts #45546737).

The remaining fix patterns have been described in Section 6.1, Section 6.2, and Section 6.3. They are also applicable to faults in training & evaluation.

**Fix distributed API usage.** This group fixes incorrect distributed APIs that developers use or fixes hyperparameter configuration in these APIs, resolving 13.09% of distributed-specific faults in this stage. Since distributed APIs of frameworks control the whole distributed training procedure including data and model aggregation,
Symptom: Fix pattern: Code Fix

Script freezes with no output when using

Question Description:

for trial in range(maxtrials):
    # code for training or inference
    if finish < start >> mintime and trial >= mintrials:
        if trial >= maxtrials:
            break

Example (b) – SO post # 60799655

Figure 8: Distribution of fix patterns for leaf categories in training & evaluation issues.

Example (c) – PyTorch GitHub issue # 22834

synchronization, and so on, fixing distributed API usage resolves faults with almost every symptom in this stage.

Fix distributed API. Fixing distributed API usage resolves faults with almost every symptom in this stage.

Fix dependency installation/version. Fixing dependency installation or version is the most frequent fix pattern in this stage. This strategy resolves 12.00% of the distributed-specific faults with symptoms such as segmentation fault (D.1.10) and hang (D.2.5).

Fix device assignment. Fixing device assignment of model or data mainly fixes device error (D.1.3) and unexpected parallelization & device (D.2.2). For instance, a developer encountered such an unexpected parallelization behavior that TensorFlow allocates only one GPU device for computation (SO post #4326349). The fixing strategy is modifying device allocation code and assign the model to every GPU device that is expected to be in use.

Fix communication configuration of training. Wrong communication configurations lead to communication problems in training & evaluation. Therefore, fixing communication configuration mainly resolves communication error (D.1.2) and hang (D.2.5).

Fix framework installation/version. This strategy also applies for training & evaluation stage. On one hand, bugs in outdated frameworks may lead to segmentation fault (D.1.10) symptoms. On the other hand, developers sometimes misuse APIs in a way unsupported by the current framework version, since APIs frequently evolve with DL frameworks. Therefore, developers should resolve such faults by changing the DL framework to a proper version. For example, a developer reported that she received "RuntimeError: ProcessGroupNCL does not support barrier" (PyTorch GitHub issue #17848). The corresponding fix is upgrading PyTorch to v1.0.1 or a later version because "barrier" is not supported by 1.0.x version.

Fix consistency between devices. As we have described in Section 6.2, consistent installation configurations and device states are essential for communication. These faults might not show until this stage. Besides, making sure that codes and datasets on all servers are in the exact same directory avoids path error (D.1.7).

Change hardware. Changing hardware devices mainly solves low efficiency (D.2.1). For example, when network bandwidth becomes the bottleneck of distributed training, developers can change the hardware devices they use for a larger bandwidth.

Finding 9: Fixing dependency installation/version, fixing framework installation/version setting, and fixing distributed API usage are common fix patterns for symptoms in most stages. In total, they resolve up to 37.93% of distributed-specific faults in distributed training, covering 25 frequent symptoms.

7 IMPLICATIONS

In this section, we discuss our implications for DL-based application developers, researchers, and DL framework vendors.

For Developers. (1) Resolving faults with less trial and error. We summarize the frequent fix patterns for faults in different stages of distributed training. These results can act as guidance for developers to resolve distributed-training-related faults with less trial and error. Moreover, compared to directly asking for help on platforms such as SO, such a guidance can help reduce the time of debugging, especially considering that distributed training is so challenging that only 28.5% of identified related questions have an accepted answer on SO. (2) Taking extensive factors into consideration. From Findings 2 and 4, we observe that many faults in distributed training are caused by non-distributed factors (such as model construction and dataset), especially in data and model preparation and training & evaluation stages. Therefore, we suggest developers consider both distributed-specific factors and non-distributed factors when debugging distributed training issues.
**For Researchers.** (1) *Distributed DL testing.* Existing DL testing work focuses on testing training data [32], DL frameworks [55], DL compilers [58], and DL models (such as their correctness, robustness, and fairness) [50, 54], etc. To the best of our knowledge, there is little work that detects faults (e.g., misuse of multiple devices and misconfiguration of communication) in distributed training of DL software. In fact, detecting distributed-training-related faults poses new challenges to the testing practice. For example, since many distributed-training-related faults rely on the environment and devices (Findings 6 and 9), it is difficult to detect these faults via traditional static code analysis. Due to the increasing dependence of current DL software on distributed training, we encourage researchers to propose specific and effective testing techniques (such as multi-device simulation) for it. (2) *Fault localization in distributed training.* In this study, we observe that many fault symptoms in distributed training can be attributed to diverse factors. For example, communication error (D.1.2) in distributed training can be caused by misuse of distributed-training-related APIs, wrong dependency version, wrong model construction, invalid network setting, etc. These cases make it difficult to manually identify the real causes of the faults. In addition, distributed training is usually multi-processing and can easily cause nondeterministic behaviors [60]. We find that sometimes developers cannot reproduce their faults by running the same code again (e.g., Horovod GitHub issue #2506). Although these characteristics make fault localization in distributed training rather difficult, to the best of our knowledge, little work in SE focuses on developing automated tools for this task. Given the importance of distributed training, we encourage SE researchers to tackle the diverse faults in this direction. (3) *Automated communication configuration.* Many frequent fix patterns in Section 6 are related to communication configuration (e.g., fix network setting, fix backend configuration, etc.) and they resolve faults in almost every stage. In Section 6, we can find that communication configuration (e.g., network setting, backend configuration, etc.) is related to faults in almost every stage. This observation motivates researchers to propose automated communication configuration techniques to simplify the configuration process and help developers, especially novice developers, avoid communication-related faults.

**For DL Framework Vendors.** (1) *More support for distributed training.* In Section 4, we observe that developers ask a lot about what functionalities and device settings are supported by frameworks. It implies that current DL frameworks are not friendly enough for developers to address these issues. Indeed, it is impractical for framework vendors to support all kinds of algorithms, functionalities, and device settings in advance. We suggest framework vendors mine platforms such as SO and GitHub to collect related issues reported by developers, and then first meet the most urgent requirements, e.g., better support for elastic training [15]. (2) *Simpler APIs.* With the development of distributed DL, DL frameworks vendors are implementing more and more distributed DL functionalities in their frameworks. However, we observe that a large proportion of faults are related to APIs (Finding 9). Many developers encounter issues because the distributed-training APIs of frameworks are too complicated. Therefore, framework vendors should improve the APIs in terms of usability and simplicity, especially for the APIs that developers encounter problems frequently.

### 8 THREATS TO VALIDITY

**Selection of frameworks, keywords, and labels.** First of all, the selection of frameworks may lead to possible selection bias in this study. To mitigate this threat, we focus on three most commonly-used DL frameworks and Horovod, which is widely-adopted for distributed training. In addition, the keyword- and label-matching identification may result in false positive posts and loss of relevant posts. The false positives are all discarded during the refining process in Section 3.1.3. Moreover, as mentioned in Section 3.1.1, our keywords have a high level of recall (i.e., 90%), ensuring that most of relevant issues can be identified. As for the label-matching identification, we follow previous work [31, 36] to carefully select effective issue labels of GitHub repositories.

**Selection of data sources.** It is impossible to collect all the issues about distributed training of DL software in the world, which may lead to a threat to the external validity of our study. To mitigate this threat, we select SO and GitHub, two most widely-used data sources in empirical studies in SE [26, 40–42, 63], as data sources to collect representative real-world issues reported by developers.

**Subjectivity of researchers.** The subjectivity in manual labeling presents a possible internal threat to the validity of our results. To minimize this threat, we follow the widely-adopted open coding procedure, in which two authors are involved in inspecting cases and another experienced arbitrator helps to reach agreement through discussions. We also use Cohen’s Kappa to measure the inter-rater agreement of independent labeling. The high kappa value indicates that the perfect level of inter-rater agreement.

### 9 RELATED WORK

In this section, we summarize the related work to well position our study within the literature.

**Empirical study on faults.** There have been a number of empirical studies that focus on faults in different types of software systems, including faults in distributed systems. Gao et al. [37] conducted an empirical study on recovery faults in large-scale distributed systems. Li et al. [48] studied failures of production distributed data-parallel programs. However, distributed training is different from most traditional distributed programs in hardware devices they run on and program characteristics. Therefore, existing studies on distributed program faults are not applicable to distributed training faults. With the rapid development of DL technologies, empirical studies on faults in software applications that make use of DL frameworks have emerged. Zhang et al. [63] categorized the symptoms and root causes of TensorFlow program faults. Islam et al. [41] and Humbatova et al. [40] studied the characteristics or fix patterns of DL faults. Zhang et al. [62] studied the symptoms, root causes, and fix patterns of job failures in a cloud-based DL platform. Chen et al. [31] studied deployment faults of DL-based mobile applications. Different from previous studies, we focus on a specific domain, i.e., distributed training.

**Distributed training.** With the increment of data size and model size, distributed training has become a standard practice [43]. Distributed training for DL comes with many possibilities for parallelization. There are mainly two predominant parallelization methods, namely data and model parallelism. In data parallelism[45], each process on several devices (e.g., machines and GPUs) loads...
an identical copy of the DL model. Training data is split into non-overlapping chunks and fed into the model replicas of the workers for training. In model parallelism\cite{34}, the DL model is split, and each device loads a different part of the model for training. Apart from data and model parallelism, there are also novel parallelization methods such as pipeline parallelism\cite{39} and hybrid parallelism\cite{43,51}. With the innovation of parallelization methods, the distributed DL ecosystem has become rich and diverse\cite{59}. DL frameworks such as TensorFlow\cite{25} and PyTorch\cite{53} implement some of these parallelization methods. Many distributed training frameworks and systems have also emerged, such as Horovod\cite{57}, BytePS\cite{44}, and PaddlePaddle\cite{20}.

10 CONCLUSION

In this paper, we have presented an empirical study on issues in distributed training of DL software by manually inspecting 1,054 related issues from Stack Overflow and GitHub. We distilled frequent topics in developers’ how-to questions. We also constructed a fine-granularity taxonomy of 30 fault symptom categories and summarize fix patterns for different fault symptoms. Frequent combinations of fault symptoms and fix patterns discovered by our study can be adopted to facilitate fault fix in distributed training of DL software. Finally, we discussed implications for developers, researchers, and framework vendors based on our findings.

REFERENCES

[1] 2016. Synchronous vs asynchronous computation in Tensorflow. https://stackoverflow.com/questions/3434916/synchronous-vs-asynchronous-computation-in-tensorflow. Retrieved on September 3, 2021.

[2] 2017. Baidu-Allreduce. https://github.com/baidu-research/baidu-allreduce. Retrieved on September 3, 2021.

[3] 2017. Horovod’s Work Pattern? https://github.com/horovod/horovod/issues/117. Retrieved on September 3, 2021.

[4] 2017. Tensorflow: Is There a Rule to Set the Port of Worker/PS When Creating ClusterSpec? https://stackoverflow.com/questions/41649708/tensorflow-is-there-a-rule-to-set-the-port-of-worker-ps-when-creating-cluster-spec. Retrieved on September 3, 2018.

[5] 2018. Introducing HorovodRunner for Distributed Deep Learning Training. https://databricks.com/blog/2018/11/19/introducing-horovodrunner-for-distributed-deep-learning-training.html. Retrieved on September 3, 2018.

[6] 2018. NVIDIA. Accelerating Deep Learning with Uber’s Horovod. https://eng.uber.com/nvidia-horovod-deep-learning/. Retrieved on September 3, 2021.

[7] 2018. Open Source at Uber: Meet Alex Sergeev. Horovod Project Lead. https://eng.uber.com/alex-sergeev-horovod/. Retrieved on September 3, 2018.

[8] 2019. Distributed Deep Learning with Horovod. https://developer.download.nvidia.cn/video/gpuchronconf/gtc/2019/presentation/s9321-distributed-deep-learning-with-horovod.pdf. Retrieved on September 3, 2019.

[9] 2019. Fabric for Deep Learning (FIDL). https://github.com/IBM/FIDL. Retrieved on September 3, 2021.

[10] 2019. NCCL. https://developer.nvidia.com/nccl. Retrieved on September 3, 2021.

[11] 2019. Top 5 Deep Learning Frameworks for 2019. https://www.springboard.com/top-5-deep-learning-frameworks/. Retrieved on September 3, 2021.

[12] 2020. CIFAR Scaling Efficiency. https://github.com/horovod/horovod/issues/2103. Retrieved on September 3, 2020.

[13] 2020. Popular Deep Learning Frameworks: An Overview. https://analyticsindiamag.com/deep-learning-frameworks/. Retrieved on September 3, 2020.

[14] 2020. Possible to Add a Worker On-the-Fly? https://stackoverflow.com/questions/62005656/possible-to-add-a-worker-on-the-fly. Retrieved on February 21, 2021.

[15] 2020. When Build Docker Container with Ubuntu16.04 Install Horovod Failed with Error Code 4. https://github.com/horovod/horovod/issues/1796. Retrieved on September 3, 2020.

[16] 2021. Github Search API. https://developer.github.com/v3/search/. Retrieved on June 5, 2021.

[17] 2021. Gloo. https://github.com/facebookincubator/gloo. Retrieved on September 3, 2021.

[18] 2021. Keras: Deep Learning for Python. https://github.com/keras-team/keras. Retrieved on September 3, 2021.

[19] 2021. PaddlePaddle. https://github.com/PaddlePaddle/Paddle. Retrieved on September 3, 2021.

[20] 2021. Running a Basic Distributed MNIST Solver in TensorFlow. https://stackoverflow.com/questions/49984317/running-a-basic-distributed-mnist-solver-in-tensorflow. Retrieved on September 3, 2021.

[21] 2021. Stack Exchange Data Dump. https://archive.org/details/stackexchange. Retrieved on June 5, 2021.

[22] 2021. Top 10 Deep Learning Frameworks in 2021 You Can’t Ignore. https://www.upgrad.com/blog/top-deep-learning-frameworks/. Retrieved on September 3, 2021.

[23] 2021. Top 5 Deep Learning Frameworks in 2021. https://makeinbusiness.com/top-5-deep-learning-frameworks/. Retrieved on September 3, 2021.

[24] 2019. NCCL. https://developer.nvidia.com/nccl. Retrieved on September 3, 2019.

[25] 2019. Top 5 Deep Learning Frameworks for 2019. https://www.upsgrad.com/blog/top-deep-learning-frameworks/. Retrieved on September 3, 2019.

[26] 2019. CIFAR Scaling Efficiency. https://github.com/horovod/horovod/issues/2103. Retrieved on September 3, 2019.

[27] 2019. Fabric for Deep Learning (FIDL). https://github.com/IBM/FIDL. Retrieved on September 3, 2019.

[28] 2019. NCCL. https://developer.nvidia.com/nccl. Retrieved on September 3, 2019.

[29] 2019. Top 5 Deep Learning Frameworks for 2019. https://www.springboard.com/blog/deep-learning-frameworks/. Retrieved on September 3, 2019.

[30] 2020. CIFAR Scaling Efficiency. https://github.com/horovod/horovod/issues/2103. Retrieved on September 3, 2020.

[31] 2020. Open MPI. Open Source High Performance Computing. https://www.openmp.org. Retrieved on September 3, 2020.

[32] 2020. Popular Deep Learning Frameworks: An Overview. https://analyticsindiamag.com/deep-learning-frameworks/. Retrieved on September 3, 2020.

[33] 2020. Possible to Add a Worker On-the-Fly? https://stackoverflow.com/questions/62005656/possible-to-add-a-worker-on-the-fly. Retrieved on February 21, 2021.

[34] 2020. When Build Docker Container with Ubuntu16.04 Install Horovod Failed with Error Code 4. https://github.com/horovod/horovod/issues/1796. Retrieved on September 3, 2020.

[35] 2021. Github Search API. https://developer.github.com/v3/search/. Retrieved on June 5, 2021.

[36] 2021. Gloo. https://github.com/facebookincubator/gloo. Retrieved on September 3, 2021.
[40] Nargiz Humbatova, Gunel Jahanigarova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In Proceedings of 42nd International Conference on Software Engineering, ICSE 2020 ACM.

[41] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In Proceedings of 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019. 510–520.

[42] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In Proceedings of 42nd International Conference on Software Engineering, ICSE 2020. 1135–1146.

[43] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In Proceedings of Machine Learning and Systems 2019, MLSys 2019.

[44] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. ACM Queue 6, 2 (2008), 40–53.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of 32nd Conference on Neural Information Processing Systems, NeurIPS 2019. 8024–8035.

[46] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A characteristic study on failures of production distributed data-parallel distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput. Surv. 53, 1 (2020), 3:1–3:37.

[47] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. IEEE Trans. Software Eng. 25, 4 (1999), 557–572.

[48] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. ACM Queue 6, 2 (2008), 40–53.

[49] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In Proceedings of 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019. 510–520.

[50] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A characteristic study on failures of production distributed data-parallel distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput. Surv. 53, 1 (2020), 3:1–3:37.

[51] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput. Surv. 53, 1 (2020), 3:1–3:37.