Rise of Distributed Deep Learning Training in the Big Model Era: From a Software Engineering Perspective

XUANZHE LIU and DIANDIAN GU, Peking University, China
ZHENPENG CHEN, University College London, UK
JINFENG WEN, ZILI ZHANG, and YUN MA, Peking University, China
HAOYU WANG, Huazhong University of Science and Technology, China
XIN JIN, Peking University, China

Deep learning (DL) has become a key component of modern software. In the "*big model*" era, the rich features of DL-based software (i.e., DL software) substantially rely on powerful DL models, e.g., BERT, GPT-3, and the recently emerging GPT-4, which are trained on the powerful cloud with large datasets. Hence, training effective DL models has become a vital stage in the whole software lifecycle. When training deep learning models, especially those big models, developers need to parallelize and distribute the computation and memory resources amongst multiple devices (e.g., a cluster of GPUs) in the training process, which is known as *distributed deep learning training*, or ***distributed training*** for short. However, the unique challenges that developers encounter in distributed training process have not been studied in the software engineering community. Given the increasingly heavy dependence of current DL-based software on distributed training, this paper aims to fill in the knowledge gap and presents the first comprehensive study on developers' issues in distributed training. To this end, we focus on popular DL frameworks that support distributed training (including TensorFlow, PyTorch, Keras, and Horovod) and analyze 1,131 real-world developers' issues about using these frameworks reported on Stack Overflow and GitHub. We construct a fine-grained taxonomy consisting of 30 categories regarding the fault symptoms and summarize common fix patterns for different symptoms. We find that: (1) many distributed-specific faults and non-distributed-specific faults inherently share the same fault symptoms, making it challenging to debug; (2) most of the fault symptoms have frequent fix patterns; (3) about half of the faults are related to system-level configurations. Based on the results, we suggest actionable implications on research avenues that can potentially facilitate the distributed training to develop DL-based software, such as focusing on the frequent and common fix patterns when designing testing or debugging tools, developing efficient testing and debugging techniques for communication configuration along with the synthesis of network configuration analysis, designing new multi-device checkpoint-and-replay techniques to help reproduction, and designing serverless APIs for cloud platforms.

This work was supported by the National Natural Science Foundation of China under the grant numbers 62172008 and 62102009, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and Center for Data Space Technology and System, Peking University. Zhenpeng Chen was supported by the ERC Advanced Grant under the grant number 741278 (EPIC: Evolutionary Program Improvement Collaborators).
Authors' addresses: X. Liu, D. Gu, J. Wen, Z. Zhang, Y. Ma, and X. Jin (corresponding author), Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, China, 100871; emails: {liuxuanzhe, gudiandian1998}@pku.edu.cn, jinfeng.wen@stu.pku.edu.cn, {zzlcs, mayun, xinjinpku}@pku.edu.cn; Z. Chen, Department of Computer Science, University College London (UCL), Gower Street, London WC1E 6BT, UK; email: zp.chen@ucl.ac.uk; H. Wang, Huazhong University of Science and Technology, Luoyu Road 1037, Hongshan District, Wuhan, China, 430074; email: haoyuwang@hust.edu.cn.

1049-331X/2023/09-ART156 $15.00
https://doi.org/10.1145/3597204

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → *Software creation and management*; • **Computing methodologies** → *Distributed computing methodologies;*

Additional Key Words and Phrases: Empirical study, distributed training, software engineering

ACM Reference format:
Xuanzhe Liu, Diandian Gu, Zhenpeng Chen, Jinfeng Wen, Zili Zhang, Yun Ma, Haoyu Wang, and Xin Jin. 2023. Rise of Distributed Deep Learning Training in the Big Model Era: From a Software Engineering Perspective. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 156 (September 2023), 26 pages.
https://doi.org/10.1145/3597204

1 INTRODUCTION

Deep learning (DL) has been a key component in modern software, ranging from supporting daily activities (e.g., speech-to-text [60]) to safety-critical tasks (e.g., autonomous vehicles [62]). The rich features of these DL-based software applications (i.e., DL software) increasingly rely on powerful "big" DL models. To increase the accuracy of DL models, on the one hand, a substantial volume of training data is required; on the other hand, the DL model architectures become more and more complex, e.g., BERT large [67] with 340 million parameters and GPT-3 [40] with 175 billion parameters. As data increases in volume and DL models in complexity, the computational intensity and memory demand of DL increase proportionally [59]. It is reported that the computation demand of DL training grows at a speed of 35× every 18 months [20]. As a result, developers have no other options but to parallelize and distribute computation and memory to multiple devices (e.g., GPUs and servers) during the training process of DL models, i.e., *distributed training*. Distributed training has drawn a lot of attention from the research community [67, 74, 75, 80, 82, 91, 94, 101], and gained significant considerations in popular AI frameworks like TensorFlow [49], PyTorch [46], Horovod [41], and others. It is also observed that distributed training draws a lot of attention from software developers. Figure 1 presents the cumulative number of the distributed-training-related posts on **Stack Overflow (SO)** over the years to 2021. We can see the overall trend of continuous interest from developers in this topic.

Compared to non-distributed training, distributed training has its unique features and challenges. Distributed training is performed in more complex environment settings, requiring protocols (e.g., for communication among devices) and algorithms (e.g., for collaborations among devices). As a result, developers inevitably encounter a variety of issues about distributed training in practice and these issues are frequently asked on developers' Q&A forums. For example, some developers find it difficult to configure communication between multiple devices that participate in distributed training [12] and complain that they cannot achieve the expected training speedup [29]. Moreover, some developers report that training may be stuck due to the drop-out of involved devices [47]. These issues are quite essentially significant, as they not only affect the quality of DL models, but also incur a high cost of computation resources and developers' efforts. However, characterizing faults related to distributed training is missing. To the best of our knowledge, only Humbatova et al. [77] mentioned a fault about data parallelism in distributed training. Zhang et al. [102] characterized job failures in DL platforms, but did not discuss the specific features of distributed training.

To fill in the knowledge gap, this paper presents the first comprehensive study on developers' issues in distributed training. Given the increasing dependence of current DL software on distributed training, it is important to understand the relevant issues that developers encounter, so that researchers and framework vendors could help developers prevent, detect, and fix the common issues in a targeted manner. We aim to answer three research questions:

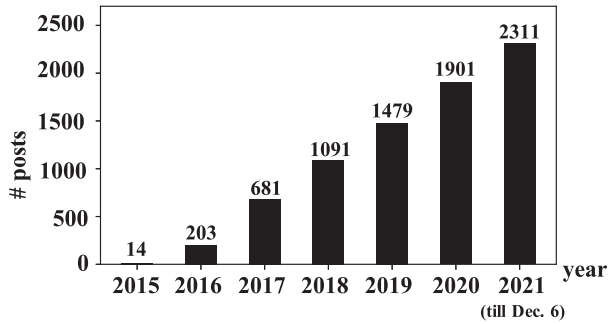

Fig. 1. The number of distributed-training-related posts on SO each year.

RQ1 (topics in how-to questions): What are the distributed training topics that developers frequently seek for help? To answer this question, we investigate the common challenges that developers encounter in distributed training by analyzing the relevant how-to-questions, which indicate the distributed training knowledge that developers are inexpert at and thus tend to induce future faults.

RQ2 (symptoms of faults): What are the fault symptoms that developers frequently encounter in distributed training? To answer this question, we summarize the frequent software faults related to distributed training, which are overlooked by previous work, via constructing a comprehensive taxonomy of the fault symptoms.

RQ3 (fix patterns): What are the common fix patterns for different fault symptoms in distribution training? To answer this question, we study each symptom's common fix patterns to provide actionable insights for automated testing and repair techniques for distributed-training-related faults.

To collect the data of our interest, we focus our study on the three most popular DL frameworks, i.e., TensorFlow [55], PyTorch [93], and Keras [44] that support distributed training and a widely-used DL framework that is specifically designed for distributed training, i.e., Horovod [97]. Specifically, we construct a dataset of 1,131 distributed-training-related developers' issues that occur during the use of these frameworks from SO and GitHub, two commonly-used data sources for studying software issues [63, 70, 77, 78, 103].

The results offer a series of findings that provide practical insights on better distributed training practice for developers, future research topics for researchers, and suggestions for DL framework vendors. We summarize the key findings and implications in Table 6. We make publicly available the code and the data in this study [54] as an additional contribution to the research community.

2 BACKGROUND

With the growing computation demands of DL, distributed training has been an important enabling technique for DL software. It parallelizes and distributes the computation and memory of DL training across multiple devices, e.g., GPUs, TPUs, and server machines. This process involves how to split training tasks, allocate computation resources, and coordinate various functional modules among different devices to achieve a balance between training speed and accuracy. To facilitate the understanding of distributed training, we present its common workflow in Figure 2.

2.1 Workflow

A distributed training job first needs to be partitioned into multiple tasks that can run in parallel on different devices (❶). The most common parallelization ways are data parallelism and model

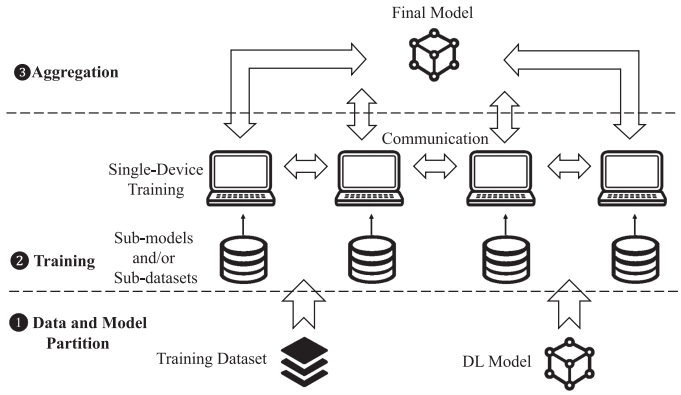

Fig. 2. Workflow of distributed training.

parallelism [90]. For data parallelism, the training data is split into non-overlapping chunks, and then these data chunks are fed into different devices; each device loads an identical copy of the DL model [84]; For model parallelism, the DL model is split, and then each device loads a different part of the DL model for training [66]. Through data/model parallelism, the training data and the DL model are distributed on different devices. Then, 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 synchronize the training progress on them. Finally, the trained models distributed on different devices are aggregated to obtain a new global model (❸). In the workflow, distributed training also relies on the environment, including hardware characteristics of devices, runtime environment (e.g., memory), network setting, and installed dependency libraries.

2.2 Related Work

Distributed training. With the increment of data size, model size, and computation requirement, distributed training has become a standard practice [81]. Distributed training for DL comes with many possibilities for parallelization, among which data and model parallelism are predominant. In data parallelism [84], each worker (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. In model parallelism [66], the DL model is split, and each device loads a different part of the model. Apart from data and model parallelism, there are also novel parallelization methods such as hybrid [81, 90] and pipeline parallelism [76]. With the innovation of parallelization methods, the distributed DL ecosystem has become rich and diverse [98]. DL frameworks such as TensorFlow [55] and PyTorch [93] support distributed training. Many distributed training frameworks and systems have also emerged, such as Horovod [97], BytePS [82], and PaddlePaddle [45].
Empirical study on faults. There have been a number of empirical studies that focus on faults in software systems, including traditional parallel computing systems and large-scale distributed systems [57, 71, 87]. However, distributed training is different from traditional parallel computing and traditional distributed programs in hardware devices they run on and program characteristics. First, in parallel computing, the processors can access shared memory to share information between processors, whereas memory is usually not shared in distributed training [13]. For example, a developer asked how to implement a sparse matrix in shared memory for parallel computing [1]. This kind of issue rarely happens in distributed training because in distributed training the processors share information with communication between devices. Also, compared to traditional parallel computing and distributed computing, distributed training is more likely to run on GPU/TPU

Table 1. Summary of Publication Years, Main Objectives, and Datasets in Related Work

Paper	Year	Main Objective	Dataset
[103]	2018	Symptoms and root causes of TensorFlow program bugs and challenges in TensorFlow program bugs detection and localization.	SO posts, GitHub commits
[78]	2019	Types, root causes, impacts, prone stages, commonality, and evolution of bugs in the usage of DL libraries.	SO posts, GitHub commits
[63]	2020	Popularity trend, difficulty, and taxonomy of challenges in deploying DL-based software.	SO posts
[79]	2020	Common bug fix patterns, fix pattern across bug types and libraries, risk in fix, and challenges of fixing bugs inside deep neural networks.	SO posts, GitHub issues
[77]	2020	The taxonomy of faults in DL systems.	SO posts, GitHub issues, developer interview
[102]	2020	Failure error type, common root causes, and current testing and debugging practices in DL programming.	Failed jobs in Philly, developer interview
[64]	2021	Symptoms and fix strategies of DL-based mobile applications.	SO posts, GitHub issues

devices instead of CPU devices [53]. Therefore, the GPU-related and TPU-related faults (e.g., GPU device error) do not happen in traditional parallel computing or distributed programs. Moreover, many faults in traditional distributed programs are related to data processing and state consistency [71]. These faults do not happen often in distributed training, because distributed training is compute-intensive instead of data-intensive, and many aggregation algorithms do not require strict consistency. Therefore, existing studies on parallel computing and 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. Table 1 summarizes the publication years, main objectives, and datasets of these empirical studies. Zhang et al. [103] categorized four high level symptoms and seven root causes of TensorFlow program faults. They found that TensorFlow users relied on statistical values to determine test results and non-determinism was prevalent in the training process. Compared to our paper, this paper focused on only one DL framework. In addition, none of them focused on bugs related to distributed training or analyzed the fix patterns accordingly. Islam et al. [78] studied the characteristics of DL bugs. They found that data bugs and logic bugs were the most severe bug types in DL software and the major root causes of these bugs were incorrect model parameters and structural inefficiency. They characterized the bugs at a high level and did not focus on distributed training. Humbatova et al. [77] built a large taxonomy of faults in DL systems. Their taxonomy is mainly based on the root cause and include only one fault about data parallelism in distributed training. There are studies on the deployment challenges and faults of DL-based mobile applications [63, 64]. Different from them, we focus on the training process instead of the deployment process and we focus on a specific domain, i.e., distributed training. Zhang et al. [102] studied the symptoms, root causes, and fix patterns of job failures in a cloud-based DL platform. They found that 48.00% of the failures occurred in the interaction with the platform rather than in the execution of code logic, mostly due to the discrepancies between local and platform execution environments and deep learning specific failures were mainly caused by inappropriate model parameters/structures and framework API misunderstanding. Distributed training jobs were included in the dataset, but they did not consider the differences between distributed jobs and single-device jobs when summarizing the failures. Our paper builds a fine-grained taxonomy of fault symptoms that includes distributed-specific symptoms and analyzes the fix patterns of distributed-specific faults.

2.3 Scope

In this paper, we focus on the technical issues that developers encounter in distributed training. First, we analyze the frequent topics of general how-to questions about distributed training (**RQ1**).

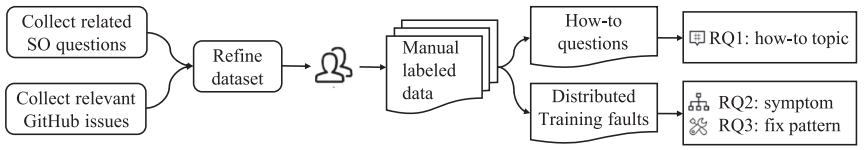

Fig. 3. Overview of the methodology.

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 [77–79, 102, 103]. 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 our methodology is illustrated in Figure 3.

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 [18, 21, 104] and industry [16–18, 21, 22]. In addition, we also consider the three most popular DL frameworks, i.e., TensorFlow, PyTorch, and Keras [32, 50–52], 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 [63]. It has been a commonly used data source for studying developers' software issues [63, 77–79, 100, 103]. Moreover, SO users range from novices to experts [103], increasing the diversity of collected issues.

First, we download the entire SO dataset from the official Stack Exchange Data Dump [48] on December 6, 2021. The dataset (denoted as set $\mathcal{A}$) contains all of the questions that were ever posted on SO, covering a time period from July 31, 2008 to December 6, 2021. Each question has one to five tags indicating its topics. From $\mathcal{A}$, we extract 103,099 questions tagged with at least one of the four selected frameworks and denote these questions as set $\mathcal{B}$.

SO questions in $\mathcal{B}$ are tagged with DL frameworks, but may contain noise that is not related to distributed training. For example, there are posts about traditional single-device DL [4, 27]. Therefore, we need to further extract the distributed training-related questions from $\mathcal{B}$. To this end, we randomly extract 1,000 questions from $\mathcal{B}$ and two authors discuss these questions carefully to manually 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 questions to perform the evaluation and also identify new keywords from them. We add the new keywords to the keyword set after the evaluation. We repeat the above evaluation process four times

Table 2. Rules to Identify Distributed-training-related Issues 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	✘	762
TensorFlow	comp:dist-strat	type:feature, type:bug, type:docs-bug, stalled, stat:awaiting response, type:docs-feature	✘	135
PyTorch	oncall: distributed, module: ddp, module: multi-gpu, module: data parallel, pt_distributed_rampup	enhancement, feature, function request	✘	726
Keras	N.A.	type:bug/performance, type:feature, type:docs, stale, Enhancement, stat:awaiting response	✔	305

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*", "*paralleled*", "*parallelism*", "*data-parallel*", "*dataparallel*", "*model-parallel*", "*modelparallel*", "*workers*", "*multi-server*", "*multiple server*", "*multiple servers*", "*multi_gpu*", "*multi-gpus*", "*multi-gpu*", "*multiple gpus*", "*multiple gpu*", "*multi-machine*", "*multiple machines*", "*multiple machine*"}. We perform a case-insensitive search within the title and body (excluding code snippets) of each question in $\mathcal{B}$ and identify 2,311 questions (denoted as set C) that contain at least one of these keywords. The cumulative numbers of the questions in set C per year are shown in Figure 1. Finally, we follow previous studies [64, 77, 89] 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 724 questions from set C and denote them as set $\mathcal{D}$.

3.1.2 Mining GitHub. GitHub is another commonly used data source for studying software issues [64, 70, 77, 103]. In line with previous work [64, 70], 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 [70]. 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, and so on. Following previous work [64, 70], we leverage these labels of GitHub issues to help us identify relevant developers' issues. We use the GitHub search API [38] to extract these GitHub issues from the framework repositories on December 6, 2021. The detailed process is as follows.

For each framework, 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 [64] to use labels to exclude GitHub issues about new feature requests, bugs in the framework itself, and problematic documents.

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 $\mathcal{E}$. Table 2 shows the used labels and the number of identified GitHub issues in each repository, respectively.

3.1.3 Refining Dataset. Two authors further manually examine all the extracted posts (i.e., SO questions in C and GitHub issues in $\mathcal{E}$) 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. The disagreements are all resolved with the involvement of an arbitrator, who has more than five years of experience in distributed training and has published many related papers in top-tier conferences. For example, an author labelled a post that mentioned multiple GPU devices as an eligible post to be included in our study, while another author excluded this post because it did not mention model training [10]. After analyzing the code snippets provided in the post and discussing it with the arbitrator, they reach an agreement that this post is about partitioning data to multiple GPUs and distributed training with the CIFAR10 dataset, which fits in the scope of our paper. Therefore, this post is included in our final dataset. We measure the agreement during the data refining process with Cohen's Kappa (κ) [65], which is commonly adopted for inter-rater agreement measurement [64, 79]. The κ value is 0.95, indicating almost perfect agreement [85]. The final dataset for our study consists of 1,075 posts, including 511 posts about Horovod, 329 posts about TensorFlow, 157 posts about PyTorch, and 83 posts about Keras.[1] The scale of our dataset is comparable and even larger than those used in existing fault-related empirical studies with manual labeling [63, 64, 70, 103].

3.2 Manual Labeling

To distill how-to topics, symptoms, and fix patterns, we label every post in the refined dataset manually. To get an overview of the entire dataset, we first conduct a pilot labeling procedure with 50% of the dataset. We choose a 50% dataset because it is sufficient for the authors to be familiar with the posts and the 50% dataset left is also sufficient for validating the taxonomy for multiple rounds. During pilot labeling, we build a pilot taxonomy. Then, we validate the taxonomy built by pilot labeling with the rest of the dataset for five rounds and continuously refine the taxonomy. Specifically, we follow the procedure below.

3.2.1 Pilot Labeling. First, we randomly sample 50% of our dataset for pilot labeling. Two authors follow an open coding procedure [96] 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 and finally establish hierarchical taxonomies in a bottom-up way. 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, two authors independently label the remaining posts with how-to topics, whether they are distributed-specific, symptoms, and fix patterns

[1]Note that an SO post may be tagged with multiple framework tags.

Table 3. The Process of Reliability Analysis

Round	# analyzed posts	# new categories	κ
1	108	3/4/3	0.39
2	108	0/5/8	0.66
3	108	0/3/4	0.73
4	108	0/1/2	0.78
5	106	0/0/0	0.88
Total	538	3/13/17	-

The third column shows the number of newly added categories for how-to topics, symptoms, and fix patterns in each round, respectively.

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) [65], which is suitable for the scenarios where two raters examine the same set of data and assign the data to a set of categories. In addition, we use fixed marginal kappa because we have a fixed set of categories that have been determined during the pilot study and fixed marginal kappa is suitable for such cases [61, 95]. It is also a widely-adopted metric in SE literature [64, 79]. 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 3 reports the κ values for the five rounds in reliability analysis. We also report the number of new categories added in each round. In the final round, no new category is added, indicating saturation for all categories; the κ value is 0.88, indicating an almost perfect agreement [85].

In summary, among the 1,075 posts in pilot labeling and reliability analysis, we identify a total of 1,131 developers' issues, including 494 how-to questions and 637 faults. We merge the categories with few developers' issues (less than 1% in how-to questions or less than 1% in the faults of the corresponding stage in distributed training) together as the "others" category. Based on the 494 how-to questions, we answer RQ1 in Section 4; based on the 637 real-world faults, we answer RQ2 and RQ3 in Sections 5 and 6, respectively.

4 HOW-TO TOPICS (RQ1)

Figure 4 shows the hierarchical distribution of how-to topics in distributed training issues. We observe that the topics asked by developers cover a wide spectrum of nine high-level categories. There is no overlap between the topics in Figure 4, where communication, parallelization, device usage, model, and evaluation can be mapped to different steps in distributed training. The remaining three categories (i.e., performance, function support, and API usage) include general questions about distributed training and these questions are not about specific step(s) in distributed training. Questions about performance ask about the efficiency or accuracy of distributed training. Questions about function support are general ones about the current support of distributed training (e.g., whether distributed training frameworks support multi-GPU [37]). Questions about API usage ask about APIs that are related to distributed training but not specific to a step of distributed training. For example, a developer asked what changes to neural networks were made by the `strategy.scope()` API [34]. This API covers most steps in distributed training, including data and model partition, data and model aggregation, and communication between devices, so it is not specific to any stage or component. Next, we will elaborate on the most frequent how-to topics.

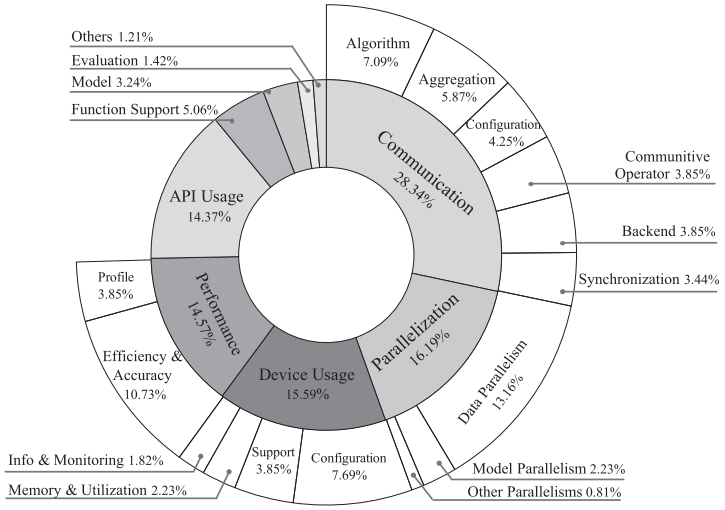

Fig. 4. Topics in how-to questions of distributed training.

Communication is the most frequently asked topic (28.34%). More specifically, 7.09% of questions are related to algorithms about how to achieve the purpose of multi-device cooperative learning, such as ring all-reduce [5] and parameter server [86]. 5.87% of the questions ask about data and model aggregation. 4.25% of the questions are on the communication configuration. 3.85% of the questions are about collective communication operators (e.g., send, receive, broadcast, etc.). 3.85% are concerned about backend communication libraries such as NCCL [24] and Gloo [39]. The remaining questions about communication ask about synchronous or asynchronous training, such as the timing to update model parameters (i.e., weights and biases) [3].

The second most frequently asked (16.19%) 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 (13.16%). The rest are about model parallelism and other novel parallelization methods.

15.59% of how-to questions are about device usage. As multiple devices are involved in distributed training, configuring devices can be difficult; 7.69% 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 [7]). The rest are questions on memory usage, device utilization, device information, and monitoring.

Developers are also concerned about the performance of distributed training (14.57%). 10.73% 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 setting). This diversity may owe to the huge differences between novices' and experts' posts on SO and GitHub, both of which reveal the difficulties and vulnerabilities in distributed training. We conduct Chi-Square test [72] to compare the similarity of the how-to topic distribution for each of the two frameworks at a 95% confidence level. The Chi-Square test is suitable for comparing the distribution of categorical data [72] and suits our purpose well. Since we carry out multiple tests, we adopt the Benjamini-Yekutieli method to adjust the p-values [68]. We hypothesize that there is no significant difference between the observed

Table 4. Adjusted p-values of the Distributions of How-to topics between the Frameworks

	Horovod	TensorFlow	PyTorch	Keras
Horovod	1.000	0.897	1.000	0.897
TensorFlow	0.897	1.000	1.000	0.897
PyTorch	1.000	1.000	1.000	1.000
Keras	0.897	0.897	1.000	1.000

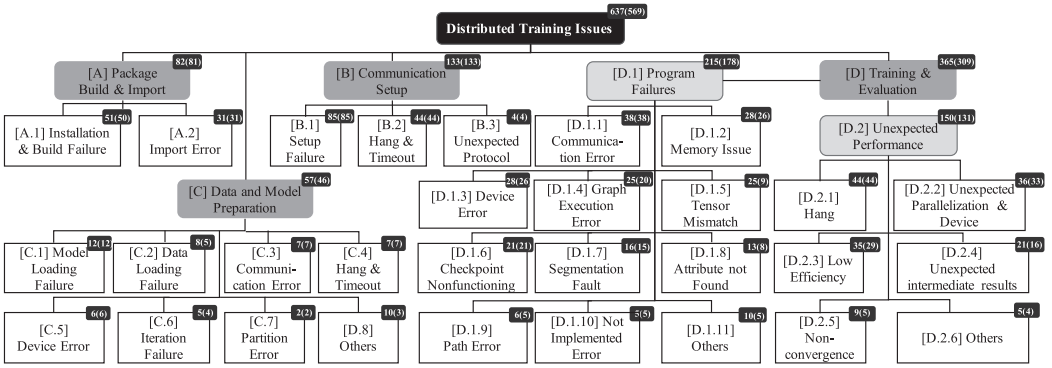

Fig. 5. Taxonomy of symptoms in distributed training. Each category has the number of corresponding faults in its top right corner with the number of distributed-specific faults in the bracket.

frequencies of how-to topics across frameworks:

$$H_0 : p_i^a = p_i^b, \forall\ i\ \leq\ k, \tag{1}$$

where a and b are two frameworks and k is the number of how-to topics. Table 4 shows the adjusted p-values found from this test across the frameworks. We find that the adjusted p-values for all framework pairs are more than 0.05. This implies that the distributions of how-to topics in different frameworks are similar.

For RQ1, see Findings F.1 and Implications I.1 in Table 6.

5 SYMPTOMS (RQ2)

Some steps of distributed training have been integrated in the APIs of DL frameworks. For example, communication between devices and data/model aggregation are integrated in the `torch.nn.parallel.DistributedDataParallel` API in PyTorch. Therefore, we cannot tell the exact stage of distributed training that a fault belongs to from the symptom. Fortunately, we can accurately know whether a fault is caused by the used packages and whether a fault happens before training starts. Therefore, we construct the hierarchical taxonomy of fault symptoms in distributed training according to the programming steps of developers, which is shown in Figure 5. 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.

Package build & import (*A*). To write distributed training programs, developers import certain modules in their code (e.g., *torch.distributed*) or build and install distributed training frameworks (e.g., Horovod). Faults that appear in this stage are included in the *package build & import* category, accounting for 12.87% of the distributed training faults. 62.20% 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 [35]. 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 for the communication in training. 20.88% 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. 63.91% of faults in this stage are triggered when the devices cannot access each other correctly (*setup failure (B.1)*). 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.2)*. *Unexpected protocol (B.3)* happens when devices do not communicate through the network protocol that developers set to.

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 2, developers split the datasets 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 (8.95%) in the entire dataset; 19.30% of faults in this stage can also happen in non-distributed training. *Model loading failure (C.1)* occurs when developers cannot load pre-trained models into memory. We use *communication error (C.3)* to refer to program crashes because of unsuccessful communication between devices in this stage. The program might also hang or crash because of timeout (i.e., *hang & timeout (C.4)*) in this stage. *Device error (C.5)* refers to program crashes because of invalid device assignments. Developers sometimes encounter problems with datasets (i.e., *data loading failure (C.2)*, *iteration failure (C.6)*, and *partition error (C.7)*).

Training & Evaluation (D). *Training & evaluation (D)* is the most important stage of DL. It is also the largest category (57.30% of identified faults) in our taxonomy, including a wide range of issues (17 symptom categories) related to all facets of training and evaluation. We classify these symptoms into two sub-categories: *program failures (D.1)* and *unexpected performance (D.2)*. *Program failures (D.1)* refers to faults that lead to program crashes. *Unexpected performance (D.2)* refers to cases when there is no crash but the programs do not behave as developers expect.

There are various *program failures (D.1)* symptoms in this stage. As common symptoms in both *training & evaluation* and *data and model preparation*, *communication error (D.1.1)* and *device error (D.1.3)* account for 10.41% and 7.67% of faults in *training & evaluation*, respectively. 7.67% of the faults occur when there is illegal memory access or the memory is not enough for use (i.e., *memory issue (D.1.2)*). *Graph execution error (D.1.4)* occurs because of the improper computational graph that represents the architecture of the DL model, even though no symptom was shown before this stage. A few faults are caused by the shape or type of a tensor not matching its expectation (i.e., *tensor mismatch (D.1.5)*), which is a common symptom in both distributed and non-distributed training. *Checkpoint nonfunctioning (D.1.6)* is triggered when developers fail to save DL models. Sometimes, programs try to read or write an illegal memory location and trigger *segmentation fault (D.1.7)*. Some faults are triggered due to reference to non-exist variables or functions (i.e., *attribute not found (D.1.8)*). *Path error (D.1.9)* refers to crashes because of unfound path references. *Not Implemented Error (D.1.10)* happens when developers use functions or methods not implemented by frameworks. Even though some of these symptoms occur 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 failures explicitly, but generate problematic outputs or behave unexpectedly. As the most common symptom in this stage, 12.05% of faults belong to *hang (D.2.1)*, which means the program is stuck. Sometimes the distributed training workflow does not parallelize on devices as expected. We classify these issues into *unexpected parallelization & device (D.2.2)*. *Low efficiency (D.2.3)* indicates distributed training does not achieve the expected speedup. Such cases account for 9.59% of faults identified in the stage. Developers also encounter faults when programs give problematic outputs (i.e., *unexpected intermediate result (D.2.4)*) or the model does not converge (i.e., *non-convergence (D.2.5)*).

Compared to traditional single-device training, *setup failure (B.1)*, *unexpected protocol (B.3)*, *partition error (C.7)*, and *communication error (C.3, D.1.1)* are uniquely specific to distributed training and not been reported by previous studies [77–79, 102, 103]. These symptoms are related to data partition or communication between workers, which are the specific steps of distributed training as shown in Figure 2.

For the rest of the symptoms, it is difficult to tell whether a fault is caused by distributed factors (e.g., data partition) or non-distributed factors (e.g., model architecture) from only the symptom, which poses a big challenge to debugging. Developers can identify which component is responsible for a fault by troubleshooting, i.e., checking whether the distributed-specific modules work properly. Moreover, unit testing (i.e., testing one or more modules together) for distributed-specific modules can potentially help identify whether the fault is caused by the distributed-specific modules. For example, *memory issue (D.1.2)* can be caused by setting a too-large batch size in the data loader, assigning data to the wrong devices, and so on. To identify which component leads to the fault, developers can adopt unit testing on the relevant components.

For RQ2, see Findings F.2 and F.3, as well as Implications I.2 and I.3 in Table 6.

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 6, 7, 8, and 9. Due to space limit, patterns with low frequency (i.e., #faults < 10 for *training & evaluation* and #faults < 5 for other stages) are not shown. In each figure, the X-axis represents leaf categories with letter identifiers consistent with Figure 5; the 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 six prevalent fix patterns for faults in *package build & import* and illustrate the distribution of these patterns on leaf categories in Figure 6.

Fix dependency installation/version & install missing dependency. 43.21% of distributed-specific faults in *package build & import* are solved by re-installing DL framework dependencies, switching dependencies to a different version, or installing missing dependencies. These strategies are frequently adopted in both *installation & build failure (A.1)* and *import error (A.2)*. The installation of DL frameworks usually relies on compilers and third-party libraries (such as communication libraries [24, 31, 39] and device-specific computing tools [92]). Horovod also relies on other DL frameworks (TensorFlow, PyTorch, etc.). Wrong installation or version of any dependency leads

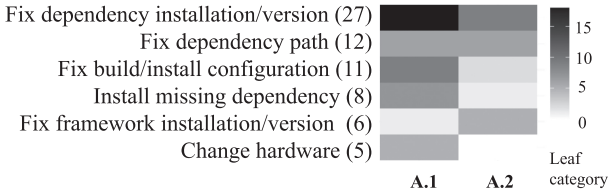

Fig. 6. Distribution of fix patterns for leaf categories in package build & import issues.

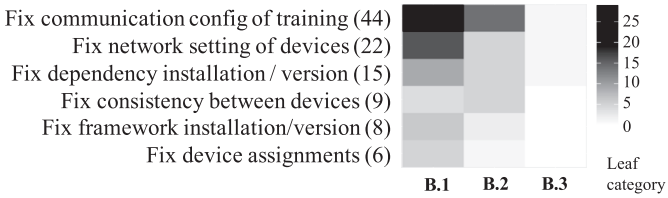

Fig. 7. Distribution of fix patterns for leaf categories in communication setup issues.

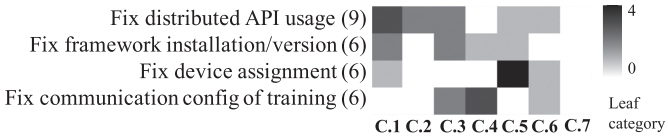

Fig. 8. Distribution of fix patterns for leaf categories in data and model preparation issues.

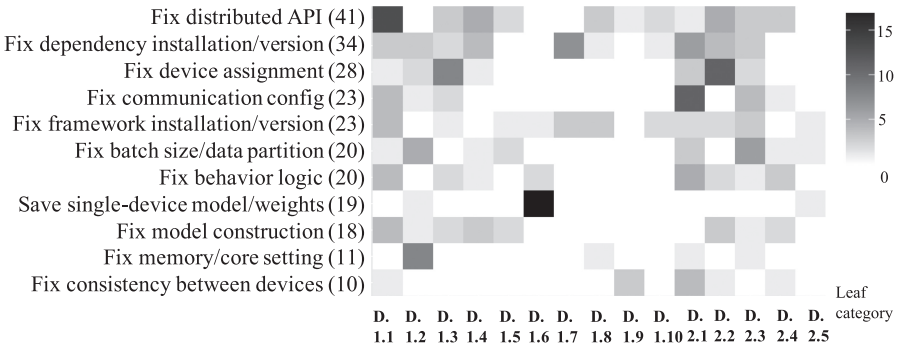

Fig. 9. Distribution of fix patterns for leaf categories in training & evaluation issues.

Question Description:
`Horovod pypi release doesn't have horovod.spark package`
Symptom: Import Error (A.2)
Fix Pattern: fix framework version
Fix: update Horovod to v0.15.3.

Example (a) – Horovod GitHub issue # 818

to failure in installation, build, or import. For example, a developer solved an installation failure by fixing CUDA version and NumPy installation [14]).

Fix dependency path. This pattern fixes 14.81% 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 elsewhere, they should explicate the paths to dependencies in environmental variables. For example, a developer tried to install Horovod, but the build was

unsuccessful because certain header files were not found [30]. She resolved the problem by adding header files of dependencies to the "CPLUS_INCLUDE_PATH" environmental variable.

Fix build/install configuration. 13.58% of distributed-specific faults in *package build & import* are resolved by fixing the build or install configuration of DL frameworks, including fixing dependency library reference, fixingcompilation options, and so on. This fix pattern mainly resolves *installation & build failure (A.1)*.

Fix framework installation/version. On one hand, some failures in building or installing frameworks are caused by broken framework packages or environment misconfiguration. On the other hand, sometimes framework vendors fix bugs that lead to unsuccessful installation or build in the updated versions of frameworks. Therefore, re-installing DL frameworks or switching the frameworks to a different version fixes certain faults in *package build & import*. For example, in Example (a), a developer encountered an import error because the old versions of Horovod did not support the package she wanted to use [23]. The fault was fixed by updating Horovod to a new version.

Change hardware. Sometimes developers' hardware devices do not support the certain instruction set to build frameworks. In this case, the only approach to solving *installation & build failure (A.1)* is using devices with required supports. For example, a developer resolved an installation failure after switching to a server with CPUs that support AVX [36]).

Most fixes in this stage are related to software dependencies, including framework installation, framework version, dependency installation, dependency version, dependency path, and others. Although many DL frameworks leverage dependency management tools such as Pip in installation, developers still need to fix the installation problems because the dependencies are too diverse and complex for these tools.

Please see Finding F.4 and Implication I.4 in Table 6 for fix patterns in *package build & import*.

6.2 Faults in Communication Setup

We iden tify six frequent fix patterns for faults in *communication setup* and present the distribution of these patterns in Figure 7.

Fix communication configuration of training. Developers can configure the world size (i.e., the number of processes participating in communication), ranks (i.e., unique IDs of processes), and other configurations in the distributed settings. Correctly configuring them mainly fixes *setup failure (B.1)* and *hang & timeout (B.2)*, accounting for 33.08% of distributed-specific faults in the stage. In Example (b), the developer encountered a fault when initializing RPC communication [43]. The fix was to modify the configuration options.

Fix network setting of devices. 16.54% of faults in *communication setup* can be resolved by fixing network settings such as IP address, 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 could not build up the connection between two nodes because of "permission denied" [19]. The 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. For example, a developer had the problem of being unable to build up the communication connection between two nodes [11]. 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

Question Description:
`init_rpc: TENSOR_PIPE backend sigaborts when CUDA is not available`

Symptom: Setup Failure (B.1)

Fix Pattern: fix communication configuration

Code Fix:

```
  import tempfile
  import torch.distributed.rpc as rpc
  init_file = tempfile.mkstemp()[1]
  options = rpc.TensorPipeRpcBackendOptions(
-.init_method="file://" + init_file, _transports=["uv"])
+.init_method="file://" + init_file, _channels=["uv"])
  rpc.init_rpc("worker0", rank=0, world_size=1,
    backend=rpc.BackendType.TENSORPIPE,
    rpc_backend_options=options)
```

Example (b) – PyTorch GitHub issue # 54266

error "connection refused" [2]. The reason was she did not successfully start training on the same number of devices as her topology configuration, leading to inconsistent device states.

Fix device assignments. In *communication setup*, each process should specify the device they are responsible for correctly, especially for the backends that rely on GPU-GPU communication (e.g., NCCL [24]).

The remaining fix patterns have already been described in Section 6.1. They are 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 [24] and Gloo [39]. The faults mainly belong to the symptom *setup failure (B.1)* and *hang & timeout (B.2)*. For instance, a developer fixed the hang in Horovod when setting up communication by changing Open MPI version [15].

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

In *communication setup*, wrong communication configuration and wrong device network setting lead to most of the communication problems. The options of configurations and settings are diverse and scattered. This indicates that the communication configuration and network setting are too tedious to be set correctly.

Please see Finding F.5 and Implication I.5 in Table 6 for fix patterns in *communication setup*.

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 8.

Fix distributed API usage. DL frameworks provide APIs for distributed training, such as `torch.nn.DataParallel` and `torch.nn.parallel.DistributedDataParallel` in PyTorch and `tf.distribute.Strategy` in TensorFlow. Developers follow certain steps required by frameworks and write distributed training programs with these APIs. However, the complicated arguments of APIs and excessive procedures are difficult for developers to follow. For example, a developer could not initialize the communication topology because she forgot to call a certain API [28]. The API is indispensable in distributed training with PyTorch.

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.5)*. Developers need to be careful with the device assignment (on type of device to use and the device id) of data and model to avoid *device error (C.5)*.

The remaining fix patterns have been described in Sections 6.1 and 6.2.

Question Description:
Script freezes with no output when using DistributedDataParallel
Symptom: Hang (D.2.1)
Fix Pattern: fix behavior logic
Code Fix:

```
for trial in range(maxtrials):
        # code for training or inference
        finish = time.time()
-.      if finish - start >= mintime and trial >= mintrials:
+.      if trial >= mintrials:
                break
```

Example (c) – PyTorch GitHub issue # 22834

Fix framework installation/version. Reinstalling framework or switching framework versions can also avoid faults in this stage, such as *communication error (C.3)*, *model loading failure (C.1)*, 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.3)* and *hang & timeout (C.4)*.

From the above fix patterns, we find that fix patterns such as fixing communication configuration and fixing framework installation/version are frequent in different stages of executing distributed training software. We also observe that there can be multiple fix patterns for one symptom, indicating that many fault symptoms in distributed training are attributed to diverse factors.

Please see Finding F.6 and Implication I.6 in Table 6 for fix patterns in *data and model preparation*.

6.4 Faults in Training & Evaluation

We identify 11 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 9.

Fix batch size/data partition. This fix pattern mainly solves faults in *memory issue (D.1.2)*, *hang (D.2.1)*, and *low efficiency (D.2.3)*. 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 [9]. The solution was to make the number of samples divisible by $batch\_size \times N$, where N is the number of GPU devices to use.

Fix behavior logic. Behavior logic refers to the logical relationship between the behaviors of different devices, such as profile writing and communication operations. Developers make mistakes in behavior logic when they are confused with the complicated logic or unfamiliar with distributed-related APIs. Inappropriate behavior logic leads to conflicts in distributed training or unexpected training performance. Example (c) shows a program hang problem [25]. This is because the training speeds on different devices are not exactly the same, leading to one process exiting before the other one. To fix this fault, the developer needs to delete the 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.1.6)*). In 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 or saving model weights only is an effective workaround.

Question Description:
`RuntimeError: arguments are located on different GPUs.`
Symptom: Device Error (D.1.3)
Fix Pattern: fix model construction
Code Fix:

```
 class custom_model(torch.nn.Module):
        def __init__(self): super(custom_model, self).__init__()
              self.layer = torch.nn.Linear(10,5)
-.            self.weight = torch.ones(5,1, device='cuda:0')
+.            self.weight = torch.nn.Parameter(torch.ones(5,1))
+.            self.weight.requires_grad = False
       def forward(self, x):
              return self.layer(x) @ self.weight
```

Example (d) – SO post # 60799655

Fix model construction. Fixing how the model is constructed resolves faults in 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 (d) is *device error (D.1.3)* which throws "*RuntimeError: arguments are located on different GPUs*" [33]. 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 being 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 memory/core setting. This group mainly resolves *memory issue (D.1.2)* problems. By increasing the execution memory and cores in use, more resources will be allocated, which can resolve out of memory errors. Besides, modifying the configuration of how DL frameworks allocate memory is also effective [6].

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

Fix distributed API usage. This group fixes incorrect distributed API usage of developers or fixes hyperparameter configuration in these APIs. Since distributed APIs of frameworks control the whole distributed training procedure including data and model aggregation, synchronization, and so on, 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.1.7)* and *hang (D.2.1)*.

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 [8]. The fixing strategy is to modify the 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.1.1)* and *hang (D.2.1)*. Some developers that encountered these faults also reported that they could not reproduce their faults [42]. This is because multi-process communication can easily cause nondeterministic behaviors, which makes fault reproduction difficult.

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.1.7)* 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

Table 5. Adjusted p-values of the Distributions of Fix Patterns between the Frameworks

	Horovod	TensorFlow	PyTorch	Keras
Horovod	1.000	0.052	0.022	7e-4
TensorFlow	0.052	1.000	1.000	0.002
PyTorch	0.022	1.000	1.000	1e-4
Keras	7e-4	0.002	1e-4	1.000

such faults by changing the DL framework to a proper version. For example, a developer reported that she received "*RuntimeError: ProcessGroupNCCL does not support barrier*" [26]. The corresponding fix is to upgrade PyTorch to v1.0.1 or a later version because "barrier" is not supported by the 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 the code and datasets on all servers are in the exact same directory avoids *path error (D.1.9)*.

From all of the fix patterns of the 30 fault symptoms, we find that fixes related to communication are the most frequent, resolving faults in three out of the four stages. Fixing dependency/framework installation/version, fixing distributed API usage, and fixing device assignment are also frequent; in total, they resolve up to 37.93% of distributed-specific faults in distributed training, covering 25 frequent symptoms. Most of the faults in 20 out of the 30 symptoms can be fixed with no more than three fix patterns, indicating that there are frequent fix patterns for these symptoms. We also found that about 47.25% of the fixes are system-level (including setting hardware devices, configuring environment, etc.) instead of training algorithm programming, among which 93.69% are distributed-specific. This indicates that compared to single-device training, the system-level configuration is challenging in distributed training.

Although some symptoms also happen in traditional single-device training, the fix patterns of these faults can be different from the ones in single-device training. Many fix patterns of distributed-specific faults are to fix the modules that are specific to distributed training. These modules, such as process groups of communication and device assignment, are not required in traditional DL performed on a single device. For example, for *low efficiency (D.2.3)*, fixing batch size is a frequent fix pattern in single-device training, while fixing communication configuration and fixing data partition are frequent in distributed training.

Please see Finding F.7∼9 and Implication I.6∼8 in Table 6 for fix patterns in *training & evaluation* and the overall distributed training process.

6.5 Fix Patterns Across Frameworks

To explore whether similar fix patterns occur on different frameworks, we study the distribution of fix patterns across frameworks. Similar to RQ1, we conduct Chi-Square test [72] to compare the similarity of the fix pattern distribution for each of the two frameworks at a 95% confidence level. Since we carry out multiple tests, we adopt the Benjamini/Yekutieli method to adjust the p-values [68]. We hypothesize that there is no significant difference between the observed frequencies of fix patterns across different frameworks:

$$H_0 : p_j^a = p_j^b, \forall\, j \;\leq\; q, \tag{2}$$

where a and b are two frameworks and q is the number of fix patterns. The adjusted p-value of framework pairs are shown in Table 5. Horovod-TensorFlow (0.052) and TensorFlow-PyTorch

Table 6. Summary of Findings and Implications

Findings about how-to questions	Implications
F.1 Developers ask a wide range (9 high-level categories) of topics on distributed training. Communication, parallelization, device usage, and performance are most frequently asked, indicating that these topics are hot, challenging, and even confusing to developers.	**I.1** The frequency of performance-related questions indicates that current DL frameworks' implementations of distributed training are still far away from the expected performance in practice. For example, the communication overhead has been evidenced to be a main cause of performance issues [82]. Therefore, we suggest DL framework vendors and researchers should carefully design useful communication optimization techniques. In practice, some techniques can be potentially promising. For example, the pipeline parallelism can better utilize all of the devices in distributed training [91] and the gradient compression can reduce the data to be transferred in communication [58].
Findings about faults symptoms	**Implications**
F.2 We construct a taxonomy of 30 fault symptoms for distributed training. Among them, there is no dominant symptom. The most common symptoms are hang, setup failure, and communication error, accounting for 35.32% of the faults.	**I.2** The diverse and non-dominated symptoms suggest the challenge of designing automated tools for detecting and fixing distributed-training faults, which need to cover a broad spectrum of faults. The three most common symptoms are all related to communication. Researchers can pay more attention to these categories that developers find difficult to fix.
F.3 We find that 80.88% of the non-distributed-specific faults show the same symptom as distributed-specific ones.	**I.3** It is difficult to tell whether a fault is caused by distributed factors (e.g., data partition) or non-distributed factors (e.g., model architecture) from only the symptom, which poses a big challenge to debugging. Developers can identify which component is responsible for a fault by troubleshooting, i.e., checking whether the distributed-specific modules work adequately. Moreover, unit testing, i.e., testing one or more modules together, can potentially help identify whether the fault is caused by the distributed-specific modules. For example, *memory issue (D.1.2)* can be caused by setting a too-large batch size in the data loader, assigning data to the wrong devices, etc. To identify which component leads to the fault, developers can adopt unit testing on the relevant components.
Findings about fix patterns	**Implications**
F.4 Most fixes in *package build & import* are related to the installation or version of frameworks and dependencies. This indicates that the currently-used dependency management tools such as Pip do not well support the complex and diverse dependencies of distributed training.	**I.4** Framework vendors can design dependency management and version management techniques that aim at the distributed environments to mitigate these problems (e.g., multi-device dependency check and automated version check). Developers can be more careful with package version requirements and can use virtual environments or dockers for package management.
F.5 In *communication setup*, wrong communication configurations and wrong device network settings lead to most of the communication problems. The configuration and setting options are diverse and scattered. Not only are there many parameters in communication-related APIs, but also many environmental variables to be set.	**I.5** This finding indicates that the communication configurations and network settings are too tedious, error-prone, and time-consuming to be adequately set by developers, which heavily increases the development cost. To help developers avoid and fix communication-related faults, the synthesis of software engineering and network systems can be worth exploring. For example, researchers can develop efficient testing and debugging techniques for communication configuration, along with the synthesis of network configuration analysis [69, 105].
F.6 Many faults in distributed training are attributed to diverse factors, indicating challenges in fault localization. For example, communication errors can be caused by misuse of distributed-training APIs, wrong dependency version, wrong model construction, invalid network setting, etc.	**I.6** Framework vendors are encouraged to provide deeper hints for faults to assist developers' resolution. For SE researchers, we suggest that they build runtime monitoring frameworks to collect traces for reproduction or adopt dynamic-analysis-based repair techniques. Existing fault reproduction methods such as checkpoint-and-replay may not be directly applied to distributed training because of the high runtime overhead or recovery overhead [99]. Researchers can design new multi-device checkpoint-and-replay techniques to help developers reproduce their faults efficiently.
F.7 Distributed training is usually multi-processing and can easily cause nondeterministic behaviors [99]. Sometimes developers cannot reproduce faults by running the same code again because of these characteristics of distributed training [42]. This also makes fault localization challenging.	
F.8 For 20 out of the 30 symptoms, most of the issues in these categories can be fixed with no more than three fix patterns.	**I.7** Our results of the frequent symptoms and their corresponding fix patterns also provide implications for testing of distributed training. When designing testing or debugging tools, researchers can focus on the frequent and common fix patterns of these symptoms. For example, for *communication error (C.3, D.1.1)*, developers can test the device assignments, device ranks, the IP address, the port, the world size, etc.; for *tensor mismatch (D.1.5)*, the batch size, parameters of data partition, and parameters of model layers can be tested to locate the fault. For symptoms without frequent fix patterns, there is still a lot of space and challenges to integrate more existing patterns and explore more fix strategies.
F.9 About 52.75% of the issues can be resolved through programming. The fix patterns of the remaining 47.25% are related to hardware devices, environment, and configurations, among which 93.69% are distributed-specific.	**I.8** The system-level configurations are challenging for developers who do not have expertise knowledge of underlying systems. To alleviate low-level device management and environment configurations for developers, one solution is to alleviate these efforts in a serverless way, i.e., cloud providers provide efficient APIs by abstracting the low-level and tedious system configurations and allocate resources on demand according to the training jobs [73, 83]. This has been demonstrated to be effective for reducing developers' programming efforts [88]
F.10 Horovod, Tensorflow, and PyTorch show similar fix pattern distribution.	**I.9** Similar automatic bug fix tools may be reused for these frameworks after being converted into a common intermediate representation.

(1.000) are over 0.05, which implies that the fix pattern distributions in these frameworks are statistically the same. This suggests that similar bug-fix patterns can be applied to different frameworks after being converted into a common intermediate representation.

For fix patterns across frameworks, see Findings F.10 and Implications I.9 in Table 6.

7 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 the 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 the relevant issues can be identified. As for the label-matching identification, to reduce manual efforts, we first follow previous work [64, 70] to automatically filter the collected dataset by a set of rules, which may result in a potential threat to the validity. For example, we directly exclude new feature requests and reports about bugs in frameworks from GitHub. However, there could be data in them related to facilitating distributed training.

Selection of data sources. Given the rapid evolution of distributed training, it is possible that new challenges and bugs can emerge in the future. Following the same methodology presented in this paper, we can reproduce the study with new industry and academia efforts of distributed training, in order to keep our results updated. Also, 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, the two most widely-used data sources in empirical studies in the SE community [56, 77–79, 103], 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 an agreement through discussions. We also use Cohen's Kappa to measure the inter-rater agreement of independent labeling. The high kappa values indicate almost perfect inter-rater agreement.

8 CONCLUSION

In this paper, we presented an empirical study on issues in distributed training of DL software by manually inspecting 1,131 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 summarized fix patterns for different fault symptoms. Our findings are helpful to developers of distributed training software and the framework vendors of distributed training platforms. In the future, researchers can develop debugging, testing, and auto-configuration tools based on the frequent combinations of fault symptoms and fix patterns, our findings, and insights.

REFERENCES

[1] 2012. Parallel array or array of structures [closed]. Retrieved on December 21, 2022 https://stackoverflow.com/questions/13239607. (2012).

[2] 2016. Distributed tensorflow on localhosts failed by "socket error, connection refused". Retrieved on March 16, 2022 https://stackoverflow.com/questions/38937984. (2016).

[3] 2016. Synchronous vs asynchronous computation in Tensorflow. Retrieved on March 16, 2022 https://stackoverflow.com/questions/34349316/synchronous-vs-asynchronous-computation-in-tensorflow. (2016).

[4] 2016. Why neural network tends to output “mean value”? Retrieved on December 21, 2022 https://stackoverflow.com/questions/39863606. (2016).
[5] 2017. Baidu-Allreduce. Retrieved on March 16, 2022 https://github.com/baidu-research/baidu-allreduce. (2017).
[6] 2017. CUDA_ERROR_OUT_OF_MEMORY: How to activate multiple GPUs from Keras in Tensorflow. Retrieved on March 16, 2022 https://stackoverflow.com/questions/45546737. (2017).
[7] 2017. Horovod’s Work Pattern? Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/117. (2017).
[8] 2017. How to use multiple GPUs effectively when training deep networks? Retrieved on March 16, 2022 https://stackoverflow.com/questions/43236349. (2017).
[9] 2017. Keras predict not working for multiple GPU’s. Retrieved on March 16, 2022 https://stackoverflow.com/questions/43620478. (2017).
[10] 2017. Memory management when using GPU in TensorFlow? Retrieved on December 21, 2022 https://stackoverflow.com/questions/42307975. (2017).
[11] 2017. Run distributed : ERROR: ORTE_ERROR_LOG: Data unpack would read past end of buffer. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/133. (2017).
[12] 2017. TensorFlow: Is There a Rule to Set the Port of Worker/PS When Creating ClusterSpec? Retrieved on March 16, 2022 https://stackoverflow.com/questions/41649708/tensorflow-is-there-a-rule-to-set-the-port-of-worker-ps-when-creating-clustersp. (2017).
[13] 2018. Difference Between Parallel and Distributed Retrieved on April 18, 2023 https://www.differencebetween.com/difference-between-parallel-and-vs-distributed-computing/. (2018).
[14] 2018. Horovod dosn’t work with CUDA 9.1. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/161. (2018).
[15] 2018. Horovod hangs with multi gpus on one machine. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/638. (2018).
[16] 2018. Introducing HorovodRunner for Distributed Deep Learning Training. Retrieved on March 16, 2022 https://databricks.com/blog/2018/11/19/introducing-horovodrunner-for-distributed-deep-learning-training.html. (2018).
[17] 2018. NVIDIA: Accelerating Deep Learning with Uber’s Horovod. Retrieved on March 16, 2022 https://eng.uber.com/nvidia-horovod-deep-learning/. (2018).
[18] 2018. Open Source at Uber: Meet Alex Sergeev, Horovod Project Lead. Retrieved on March 16, 2022 https://eng.uber.com/alex-sergeev-horovod/. (2018).
[19] 2018. Permission denied (publickey,password) when I run on muti node. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/467. (2018).
[20] 2019. AI and Compute. Retrieved on March 16, 2022 https://openai.com/blog/ai-and-compute/. (2019).
[21] 2019. Distributed Deep Learning with Horovod. Retrieved on March 16, 2022 https://developer.download.nvidia.cn/video/gputechconf/gtc/2019/presentation/s9321-distributed-deep-learning-with-horovod.pdf. (2019).
[22] 2019. Fabric for Deep Learning (FfDL). Retrieved on March 16, 2022 https://github.com/IBM/FfDL. (2019).
[23] 2019. Horovod pypi release doesn’t have horovod.spark package. Retrieved on December 19, 2022 https://github.com/horovod/horovod/issues/818. (2019).
[24] 2019. NCCL. Retrieved on March 16, 2022 https://developer.nvidia.com/nccl. (2019).
[25] 2019. Script freezes with no output when using DistributedDataParallel. Retrieved on March 16, 2022 https://github.com/pytorch/pytorch/issues/22834. (2019).
[26] 2019. torch.distributed.launch receives RuntimeError: ProcessGroupNCCL does not support barrier. Retrieved on March 16, 2022 https://github.com/pytorch/pytorch/issues/17848. (2019).
[27] 2019. Where does the documentation point to a list of values for the loss property of the compile function? Retrieved on December 21, 2022 https://stackoverflow.com/questions/57244733. (2019).
[28] 2020. AssertionError: Default process group is not initialized. Retrieved on March 16, 2022 https://github.com/pytorch/pytorch/issues/38300. (2020).
[29] 2020. CIFAR Scaling Efficiency. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/2103. (2020).
[30] 2020. Installation issue with MXNet built from source - No such file or dictionary <dmlc/base.h>. Retrieved on March 16, 2022. https://github.com/horovod/horovod/issues/1910. (2020).
[31] 2020. Open MPI: Open Source High Performance Computing. Retrieved on March 16, 2022 https://www.open-mpi.org. (2020).
[32] 2020. Popular Deep Learning Frameworks: An Overview. Retrieved on March 16, 2022 https://analyticsindiamag.com/deep-learning-frameworks/. (2020).
[33] 2020. Pytorch DataParallel doesn’t work when the model contain tensor operation. Retrieved on March 16, 2022 https://stackoverflow.com/questions/60799655. (2020).

[34] 2020. What does “with strategy.scope():” or “with tf.distribute.experimental.TPUStrategy(tpu).scope():” do to the creation of a NN? Retrieved on January 16, 2023 https://stackoverflow.com/questions/65358676. (2020).
[35] 2020. When Build Docker Container with Ubuntu16.04 Install Horovod Failed with Error Code -4. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/1798. (2020).
[36] 2020. When build docker container with ubuntu16.04 install horovod failed with error code -4. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/1798. (2020).
[37] 2021. does it automatically use multiple gpu, if availabe? Retrieved on Feburary 11, 2023 https://github.com/keras-team/keras/issues/106. (2021).
[38] 2021. Github Search API. Retrieved on March 16, 2022 https://developer.github.com/v3/search/. (2021).
[39] 2021. Gloo. Retrieved on March 16, 2022 https://github.com/facebookincubator/gloo. (2021).
[40] 2021. GPT-3 Powers the Next Generation of Apps. Retrieved on March 16, 2022 https://openai.com/blog/gpt-3-apps/. (2021).
[41] 2021. Horovod. Retrieved on March 16, 2022 https://github.com/horovod/horovod. (2021).
[42] 2021. I meet deadlock problem when use horovod. Retrieved on March 16, 2022 https://github.com/horovod/horovod/issues/2506. (2021).
[43] 2021. init_rpc: TENSOR_PIPE backend sigaborts when CUDA is not available. Retrieved on December 19, 2022 https://github.com/pytorch/pytorch/issues/54266. (2021).
[44] 2021. Keras: Deep Learning for Python. Retrieved on March 16, 2022 https://github.com/keras-team/keras. (2021).
[45] 2021. PaddlePaddle. Retrieved on March 16, 2022 https://github.com/PaddlePaddle/Paddle. (2021).
[46] 2021. PyTorch. Retrieved on March 16, 2022 https://github.com/pytorch/pytorch. (2021).
[47] 2021. Running a Basic Distributed MNIST Solver in TensorFlow. Retrieved on March 16, 2022 https://stackoverflow.com/questions/49984317/running-a-basic-distributed-mnist-solver-in-tensorflow. (2021).
[48] 2021. Stack Exchange Data Dump. Retrieved on December 6, 2021 https://archive.org/details/stackexchange. (2021).
[49] 2021. TensorFlow. Retrieved on March 16, 2022 https://github.com/tensorflow/tensorflow. (2021).
[50] 2021. *Top 5 Deep Learning Frameworks You Should Try in 2021*. Retrieved on March 16, 2022 https://nexart.tech/blog/top-10-deep-learning-frameworks-you-should-try-it-in-2021/
[51] 2021. *Top 5 Deep Learning Frameworks in 2021*. Retrieved on March 16, 2022 https://makeinbusiness.com/top-5-deep-learning-frameworks/
[52] 2022. *Top 10 Deep Learning Frameworks in 2022 You Can’t Ignore*. Retrieved on March 16, 2022 https://www.upgrad.com/blog/top-deep-learning-frameworks/
[53] 2023. Distributed training of deep learning models on Azure. Retrieved on April 18, 2023 https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/ai/training-deep-learning. (2023).
[54] 2023. Supplemental Materials. Retrieved on April 22, 2023 https://github.com/gudiandian/TOSEM23-DistributedTraining. (2023).
[55] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*. 265–283.
[56] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 1199–1210.
[57] Sara Abbaspour Asadollah. 2018. *Concurrency Bugs: Characterization, Debugging and Runtime Verification*. Ph.D. Dissertation. Mälardalen University College, Västerås, Eskilstuna, Sweden.
[58] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel DNN training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SIGOPS 2021*. 359–375.
[59] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.* 52, 4 (2019), 65:1–65:43.
[60] Alexandre Berard, Olivier Pietquin, Christophe Servan, and Laurent Besacier. 2016. Listen and translate: A proof of concept for end-to-end speech-to-text translation. *CoRR* abs/1612.01744 (2016).
[61] Robert L. Brennan and Dale J. Prediger. 1981. Coefficient Kappa: Some uses, misuses, and alternatives. *Educational and Psychological Measurement* 41, 3 (1981), 687–699.
[62] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of 2015 IEEE International Conference on Computer Vision, ICCV 2015*. 2722–2730.

[63] 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 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020*. 750–762.

[64] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*. 674–685.

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

[66] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012*. 1232–1240.

[67] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. 4171–4186.

[68] J. A. Ferreira and A. H. Zwinderman. 2006. On The Benjamini–Hochberg method. *The Annals of Statistics* 34, 4 (2006), 1827–1849.

[69] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. A general approach to network configuration analysis. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*. 469–483.

[70] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*. 509–519.

[71] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*. 539–550.

[72] Priscilla E. Greenwood and Michael S. Nikulin. 1996. *A Guide to Chi-Aquared Testing*, Vol. 280.

[73] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 266–280.

[74] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture, HPCA 2018*. 620–629.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. 770–778.

[76] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Proceedings of 32nd Annual Conference on Neural Information Processing Systems, NeurIPS 2019*. 103–112.

[77] 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 42nd International Conference on Software Engineering, ICSE 2020*. 1110–1121.

[78] 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/SIGSOFT FSE 2019*. 510–520.

[79] 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.

[80] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proceedings of 2019 USENIX Annual Technical Conference, USENIX ATC 2019*. 947–960.

[81] 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*.

[82] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. 463–479.

[83] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[84] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012*. 1106–1114.

[85] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[86] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*. 583–598.

[87] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of 35th International Conference on Software Engineering, ICSE 2013*. 963–972.

[88] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: A mashup-based framework for service composition. *Sci. China Inf. Sci.* 57, 1 (2014), 1–20.

[89] 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 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020*. 617–628.

[90] 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.

[91] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*. 1–15.

[92] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *ACM Queue* 6, 2 (2008), 40–53.

[93] 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 Annual Conference on Neural Information Processing Systems, NeurIPS 2019*. 8024–8035.

[94] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*. 16–29.

[95] Justus J. Randolph. 2005. Free-marginal multirater Kappa (multirater κfree): An alternative to Fleiss' Fixed-Marginal Multirater Kappa. *Online Submission* (2005).

[96] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572.

[97] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018).

[98] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A survey on distributed machine learning. *ACM Comput. Surv.* 53, 2 (2020), 30:1–30:33.

[99] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: Fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*. 338–352.

[100] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. 416–428.

[101] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *Proceedings of 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*.

[102] 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 42nd International Conference on Software Engineering, ICSE 2020*. 1159–1170.

[103] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*. 129–140.

[104] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is network the bottleneck of distributed training?. In *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2020*. 8–13.

[105] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: Scalable network testing for programmable data planes. In *Proceedings of ACM SIGCOMM 2022 Conference*. 350–364.

Received 23 June 2022; revised 14 February 2023; accepted 17 April 2023