An Empirical Study on Program Failures of Deep Learning Jobs

Ru Zhang
Microsoft Research
v-ruzha@microsoft.com

Wencong Xiao*
Alibaba Group
wencong.xwc@alibaba-inc.com

Hongyu Zhang
The University of Newcastle
hongyu.zhang@newcastle.edu.au

Yu Liu
Microsoft Research
v-yucli@microsoft.com

Haoxiang Lin†
Microsoft Research
haoxlin@microsoft.com

Mao Yang
Microsoft Research
maoyang@microsoft.com

ABSTRACT

Deep learning has made significant achievements in many application areas. To train and test models more efficiently, enterprise developers submit and run their deep learning programs on a shared, multi-tenant platform. However, some of the programs fail after a long execution time due to code/script defects, which reduces the development productivity and wastes expensive resources such as GPU, storage, and network I/O.

This paper presents the first comprehensive empirical study on program failures of deep learning jobs. 4960 real failures are collected from a deep learning platform in Microsoft. We manually examine their failure messages and classify them into 20 categories. In addition, we identify the common root causes and bug-fix solutions on a sample of 400 failures. To better understand the current testing and debugging practices for deep learning, we also conduct developer interviews. Our major findings include: (1) 48.0% of the failures occur 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; (2) Deep learning specific failures (13.5%) are mainly caused by inappropriate model parameters/structures and framework API misunderstanding; (3) Current debugging practices are not efficient for fault localization in many cases, and developers need more deep learning specific tools. Based on our findings, we further suggest possible research topics and tooling support that could facilitate future deep learning development.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

Deep learning jobs, Program failures, Empirical study

*Most of this author's work is done as an intern at Microsoft Research.
†Corresponding author.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380362

ACM Reference Format:
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 *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380362

1 INTRODUCTION

In recent years, deep learning (DL) has rapidly emerged as one of the most successful machine learning techniques. With the massive growth in computing power and data volume, deep learning has been widely applied in various areas (such as speech and image recognition, natural language processing, gaming with reinforcement learning, etc.), leading to many significant and exciting results that may previously have seemed out of reach.

To help data scientists train and test their deep learning models, enterprises build dedicated platforms such as Microsoft Azure Machine Learning [3], Amazon SageMaker [1], and Google Cloud AI [2] that allow multiple developers to submit and execute their deep learning programs. These platforms are shared, multi-tenant, and equipped with a large number of CPUs, GPUs or new AI accelerators like TPUs, providing support for a variety of deep learning frameworks such as TensorFlow (TF) [7], PyTorch [30], MXNet [10], and CNTK [37].

Philly is a similar deep learning platform in Microsoft, built with widely used open-source technologies and typical computing hardware. Every day, thousands of deep learning jobs are submitted to Philly by tens of research and product teams. However, we found that a noticeable percentage of these jobs threw runtime exceptions due to code or script defects and failed to complete. Such failures, especially those happened after a long time of execution, led to an expensive waste of shared resources such as GPU, storage, and network I/O. For example, a job had been running for 40 hours on 4 NVIDIA Tesla P100 GPUs and then triggered a DirectoryNotFound exception due to a wrong path configuration of test images. Furthermore, the stochastic nature of deep learning training process could also cause sudden, unexpected failures. Therefore, understanding the categories and root causes of job failures is very important for improving program quality and saving precious resources. Moreover, it can also provide guidance for preventing, detecting, debugging, and fixing of program defects associated with deep learning jobs.

Although there have been a number of empirical studies on defects of machine learning/deep learning programs [22, 38, 40, 49], there is little work on program failures of deep learning jobs running on a remote, shared platform. The most related research is reported

by Zhang *et al.* [49] and Islam *et al.* [22] on deep learning program bugs. However, their subjects are collected from GitHub issues and Stack Overflow questions, which are different in many aspects from the industrial jobs on Philly. Another related work by Jeon *et al.* [23] studies the behavior of deep learning jobs within Microsoft. However, their purpose is to understand cluster GPU utilization rather than reducing job failures.

In this paper, we conduct the first comprehensive empirical study on program failures and fixes of deep learning jobs. We study 4960 failed jobs submitted by hundreds of developers in Microsoft. Those failures are caused by developers' code/script defects, and we exclude the underlying hardware or system failures. Programs may have been tested locally before job submission, hence some failures that could be resolved by local testing may not be present in our study. The purpose of this study is to provide a systematic and generalized understanding on deep learning job failures, which could facilitate failure reduction and resource saving in a shared platform. Specifically, our study intends to address the following research questions:

- **RQ1:** What types of deep learning job failures are more frequent? To answer this question, we manually examine the failure messages of 4960 failed jobs and classify them into 20 categories. We obtain many findings. For example, 48.0% of the failures occur 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. The detailed results will be reported in Section 4.
- **RQ2:** What are the common root causes of failed deep learning jobs? To answer this question, we select a sample of 400 failed jobs and manually identify their root causes by examining the source code and associated job scripts. For some intricate ones, we contact the developers for clarification. We also investigate how developers fixed the faults. We obtain many findings. For example, deep learning specific failures (13.5%) are mainly caused by inappropriate model parameters/structures and API misunderstanding. The detailed results will be reported in Section 5.
- **RQ3:** What are the current testing and debugging practices in deep learning programming? To answer this question, we conduct in-depth face-to-face interviews with 6 representative developers from Microsoft. We find that current testing and debugging practices are inefficient in many cases. The detailed results will be reported in Section 6.

Based on our empirical study, we also summarize the lessons learned and suggest future tooling support for deep learning testing and debugging. For example, we suggest possible extensions for Philly and DL frameworks.

In summary, this paper makes the following contributions:

- We perform the first comprehensive study on program failures of deep learning jobs. We classify 4960 failures and manually analyze the root causes of 400 among them.
- We point out implications of our findings and suggest possible improvements for the development of deep learning platforms and frameworks.

The rest of the paper is organized as follows. In Section 2, we give an overview of the new deep learning programming paradigm and the platform Philly. Section 3 describes the study methodology. Section 4 and Section 5 present the failure classification, root causes and fixes. Section 6 presents our user study on current testing and debugging practices in the production environment. Section 7 discusses the generality of our study and future research work for failure reduction. We survey related work in Section 8 and conclude this paper in Section 9.

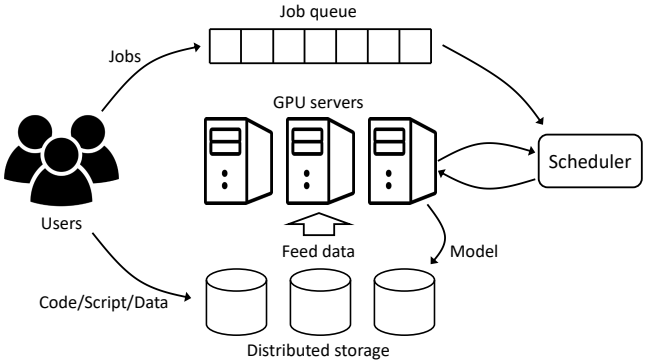

Figure 1: Overview of the infrastructure and job life cycle.

2 BACKGROUND

2.1 Deep Learning Programs

Deep learning (DL) is a subfield of machine learning, which learns layered data representations called neural networks or models. Developers write DL programs using frameworks such as TensorFlow, PyTorch, MXNet or CNTK, plus toolkit libraries like NumPy [28], DLTK [31], Detectron [16], and Fairseq [15]. Python is the most popular programming language, while C++ or Java are also used in some cases.

Typical DL code is composed in three logically consecutive stages: data pre-processing, model training & validation, and model evaluation. The first stage is usually for input data cleaning and augmentation (*e.g.*, randomly cropping input images for more training data). Next, developers construct the model using computational primitives such as basic neural network layers (e.g., CNN), activation functions (e.g., ReLU), loss function optimizers (e.g., Adam[25]), and variable initializers. The model consists of *weight* parameters of layers and allows tensors (multi-dimensional arrays of numerical values) to flow across. Training is actually to find the best weight parameters by updating the model iteratively until its learning performance (*e.g.*, loss and accuracy) meets the requirements. Validation is executed once every several training iterations to provide timely feedback for decision of hyperparameter (*e.g.*, learning rate, number of hidden units) tuning or early stop. In the last evaluation (*i.e.*, testing) stage, developers quantify the final model performance.

2.2 Deep Learning Jobs on Philly

Philly is a DL platform in Microsoft, deployed on multiple physical clusters equipped with different generations of GPUs. Every day, thousands of jobs from both research and product teams are executed on Philly, including machine translation (*e.g.*, Transformer [43]), reading comprehension (*e.g.*, BERT [13]), object detection (*e.g.*, Mask

R-CNN [21]), gaming (*e.g.*, DQN [27]), advertisement (*e.g.*, Wide & Deep [11]), etc.

The workflow of job submission and execution in Philly is similar to that of Microsoft Azure Machine Learning, Amazon SageMaker, and Google Cloud AI. Figure 1 gives an overview of the job life cycle in Philly. Developers first upload their code/scripts and input data to the distributed storage. Then, they specify the job configuration such as the numbers of desired processes/GPUs, Docker [26] image, input/output folders, and the main Python code file. Philly offers Docker images of standard deep learning toolchain to establish a hermetic job execution environment. Nevertheless, custom Docker images are also allowed to suit additional software requirements (*e.g.*, to install dependent Python libraries). Multiple jobs with the same program but different hyperparameters could be submitted together in either manual or automated way [17, 35] to search for the best model. Jobs initially wait in queues for scheduling. Once a job is chosen, Philly instantiates Docker containers to execute the launching Shell script and later run the main Python code file. The job may fail if a Python exception or Shell error code is thrown. Philly could automatically re-run it for a certain number of times [23] to recover from non-deterministic system failures for fault-tolerance. Each run instance is called a *retry*. In the end, the job will enter one of the three terminal states: SUCCEEDED, KILLED (proactive termination by developers), and FAILED.

Note that although Philly is developed by Microsoft, it is actually similar in principle to other commonly used industrial DL platforms. The hardware, system architecture, and job submission/execution mechanism of Philly are widely adopted [9, 23, 47]. Furthermore, most DL programs executed on Philly also use the programming paradigm (such as Python and TensorFlow) that is common in DL development.

3 METHODOLOGY

3.1 Subjects

We took failed deep learning jobs on Philly as our study subjects. These jobs were submitted by research and product teams in Microsoft and had the final status "FAILED". For each failed job, we collected all related information including input data, source code, job scripts, execution logs, and runtime statistics for analysis.

Failures in our study manifested as unexpected runtime errors that led to job termination. We did not study semantic errors where a job finished successfully but its results were different from the expectation as we had no test oracles. We also excluded failed jobs caused by hardware malfunction or system issues because they were beyond the scope of this study. Since developers may have tested their programs locally before job submission, some failures that could be resolved by local testing may not be present in the subjects.

To investigate failure classification, we chose 4960 failed jobs caused by developers' code/script defects within a three-week period in October 2018. We called this set the *Full Sample Set*. Due to the retry mechanism of Philly, a failed job may have several instances, and we only considered the last one. To further understand the root causes and fixes, we randomly selected 400 out of the total failed jobs and performed detailed analysis of them. We called this set the *Small Sample Set*.

3.2 Failure Classification

Job failures in *Full Sample Set* were manually classified based on their runtime error types. For each failed job, we first located the failure messages by searching the execution logs with keywords such as *assert*, *wrong*, *error*, *exception*, *fault*, *fail*, *crash*, *unsuccessful*, etc. Then, we manually inspected all the related log messages around the failures, understood the context, filtered out the false positives, located the actual failure that led to job termination, and categorized it. For example, a failure with the message *"cuda runtime error (2) : out of memory"* is categorized into *GPU Out of Memory*, instead of *CPU Out of Memory*. We also grouped all failure categories into 4 major dimensions from the execution point of view: Deep Learning Specific, Execution Environment, General Code Error, and Data. We discussed the category of each failure in our group meetings until a consensus was reached. If there was a discrepancy, we contacted the job submitter for help.

3.3 Root Cause and Fix Identification

We manually studied the source code and scripts of each failed job in *Small Sample Set*. For data related failures, additional inspection on the corresponding input data was carried out. For each failure, we then analyzed its occurrence stage and root cause. We also identified a later "fixed" version of the job by matching both job and submitter names, and examined the bug-fixing changes by comparing the failed and fixed versions. If a complicated fix was beyond our understanding, we contacted the job submitter for clarification. To calculate a failure's runtime cost, we simply counted its total GPU service time (*i.e.*, GPU number × job execution time).

3.4 Threats To Validity

Threats To Internal Validity. Due to the complex nature of the work and a large amount of manual effort, subjectivity may exist in failure classification and root cause analysis. To reduce this threat, we strived to achieve group consensus before making decisions. For unclear or difficult job failures, we also directly communicated with the developers to seek clarification.

Threats To External Validity. Our study subjects are all collected from Philly. Although there are apparent similarities in the deep learning software stack, platform architecture, and job management between Philly and other deep learning platforms, it is possible that some findings might not hold in other platforms or in other companies. To mitigate this threat, in this study, we try not to draw too specific conclusions that only pertain to Philly and Microsoft. In Section 7, we discuss findings that could be generalized to other platforms similar to Philly.

4 FAILURE CLASSIFICATION

In this section, we present the classification of 4960 job failures in *Full Sample Set*. Table 1 summarizes the 20 failure categories, and Figure 2 illustrates failure distribution across the 4 dimensions.

4.1 Dimension 1: Deep Learning Specific

13.5% of the total failures are deep learning specific and are divided into 5 categories. *Tensor Mismatch* means that the shape or name of a tensor does not match its expectation. There are usually three cases: (1) Interconnected model layers have incompatible tensor

Dimension	Category	Failure Description	No.	Ratio
Deep Learning Specific	GPU Out of Memory	Insufficient GPU memory to continue the DL computation	434	8.8%
	CPU Out of Memory	Insufficient main memory	15	0.3%
	Framework API Misuse	API usage violates framework assumptions	138	2.8%
	Tensor Mismatch	Tensor shape or name does not match the expectation	57	1.1%
	Loss NaN	Loss value is not a number	24	0.5%
	Subtotal		668	13.5%
Execution Environment	Path Not Found	File or directory cannot be found	1954	39.4%
	Library Not Found	Python modules or dependent DLLs cannot be found on the search path	309	6.2%
	Permission Denied	Insufficient permission to perform actions (*e.g.*, installing Python packages into system folders)	116	2.3%
	Subtotal		2379	48.0%
General Code Error	Illegal Argument	Argument does not satisfy program or function requirement	722	14.6%
	Type Mismatch	Applying an operation or function to an object of inappropriate type	566	11.4%
	Key Not Found	Accessing collection items with a non-existent key	154	3.1%
	Null Reference	Dereference on null value objects	92	1.9%
	Attribute Not Found	Referencing a non-existent Python class field, function, etc.	81	1.6%
	Syntax Error	Violation of the grammatical rules	64	1.3%
	Illegal Index	Accessing array elements with an out-of-range or non-integer index	64	1.3%
	Undefined Variable	Referencing a variable before its definition	59	1.2%
	Not Implemented	Functionality is not implemented yet	7	0.1%
	Division by Zero	Dividing a decimal value by zero	3	0.1%
	Subtotal		1812	36.5%
Data	Corrupt Data	Exceptional schema or contents in data	90	1.8%
	Unexpected Encoding	Data cannot be correctly encoded or decoded	11	0.2%
	Subtotal		101	2.0%
Total			4960	100.0%

Table 1: Classification of 4960 job failures.

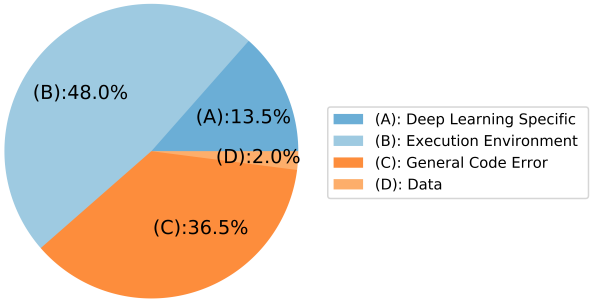

Figure 2: Distribution of 4960 job failures.

requirements; (2) Input data with unmatched shape is fed to a model; (3) A mismatched model file is restored to the constructed model. *GPU Out of Memory* failures occur when the working set exceeds GPUs' available physical memory. It is the top category in this dimension. Since GPU memory is relatively limited, developers need to size the model very carefully. *CPU Out of Memory* failures occur when a job runs out of the main memory (*e.g.*, processing large amounts of data). *Loss NaN* indicates that the calculated loss value becomes undefined or unrepresentable (*e.g.*, $0 \times \log 0$).

The last category is *Framework API Misuse*, which means that a framework API call violates the inherent assumptions. For example, the TensorFlow API "session.run()" assumes that all the variables should have been initialized properly. Otherwise, it throws an exception with the message *"FailedPreconditionError: Attempting to use uninitialized value"*.

Finding 1: 13.5% of the total failures are deep learning specific. Among the 5 categories, *GPU Out of Memory* is the largest one, accounting for 65.0% of the failures in this dimension.

4.2 Dimension 2: Execution Environment

Execution environment related failures occur in the interaction with the platform rather than in the execution of code logic, accounting for nearly half (48.0%) of the total failures. It is apparent that the job execution environment provided by a platform is rather different

to that for local development. For example, the required Python modules (*e.g.*, Fairseq [15]) or dependent DLLs (*e.g.*, libcudnn.so) may not have been pre-installed in the Docker container, then a *Library Not Found* failure happens. Developers are allowed to make in-place modifications to the job execution environment. However, if they do not have sufficient permissions, such operations fail and trigger *Permission Denied* failures. For example, a developer executes "pip install fairseq" in the launching script for the required Fairseq library. This command cannot succeed since root permission is required. She needs to specify the "--user" option so as to install the library into her home directory. The dominating category in this dimension is *Path Not Found* (39.4%), which means that a non-existent file or directory is accessed. For example, developers forget to change the local file path in code before job submission.

Finding 2: Nearly half (48.0%) of the total failures occur in the interaction with the platform rather than in the execution of code logic. Among the 3 categories, *Path Not Found* is dominating, which accounts for 82.1% of the failures in this dimension.

4.3 Dimension 3: General Code Error

We find that 36.5% of the total failures are common and similar to those in other computer programs. 14.6% are *Illegal Argument* (argument not satisfying the requirement), and nearly half of them occur while feeding arguments to Python code in the launching script. There are also other categories such as *Key Not Found* (accessing collection items with a non-existent key), *Attribute Not Found* (referencing a non-exist Python class field or function), *Null Reference*, and *Syntax Error* (*e.g.*, incorrect Python indentation). Because Python is a dynamic programming language, *Type Mismatch* failures are rather common (11.4%). For example, concatenation of a "PosixPath" object and a string using the "os.path.join" function throws an exception with the message *"TypeError: join() argument must be str or bytes, not 'PosixPath'"*.

Finding 3: Failures in the General Code Error dimension account for a large percentage (36.5%) of the total failures. *Illegal Argument* and *Type Mismatch* are the top two categories.

4.4 Dimension 4: Data

Failures in this dimension are related to unexpected data that cannot be processed, possibly because of unsuccessful uploading or misunderstanding of data properties. One of the two categories is *Corrupt Data*, indicating that the data integrity is compromised. For example, a JSON file which may be accidentally truncated loses the value part in an attribute-value pair. The other is *Unexpected Encoding*, which means that some data fields cannot be correctly encoded or decoded. We notice a real program that uses the default ASCII encoding to decode a string with some non-ASCII characters.

Finding 4: A number of deep learning jobs (2.0%) fail to handle data errors such as corrupt data and unexpected data encoding.

5 ROOT CAUSES AND FIXES

We conducted root cause and fix analysis on *Small Sample Set* (400 failures) since it required a large amount of manual efforts to study the source code/scripts. We also contacted the developers to seek clarification when necessary. We have identified 5 common root causes. Figure 3 illustrates the distribution of failure categories and root causes across the 400 failures. In the following subsections, we will describe the root causes in detail, and demonstrate some real-world examples of failed programs and corresponding fixes.

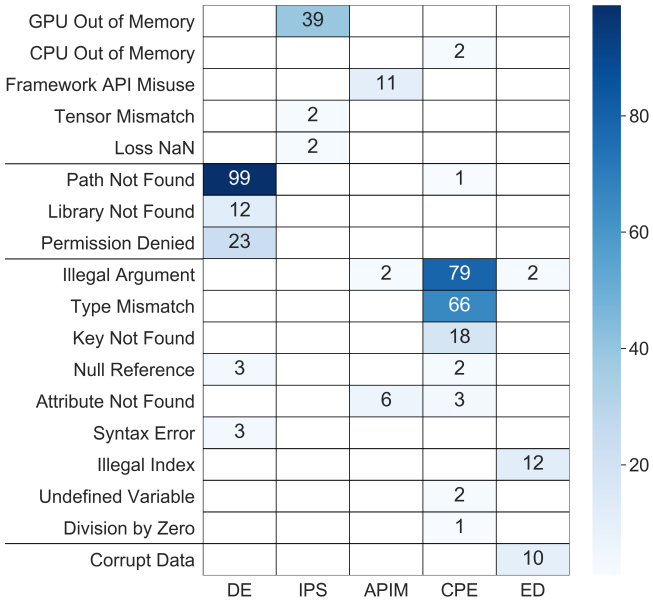

Figure 3: Distribution of failure categories (vertical) and root causes (horizontal) across 400 job failures. The full name of each root cause abbreviation can be found in the titles of subsections in Section 5.

5.1 Differences in Environment (DE)

Although Philly establishes a hermetic job execution environment via Docker images of standard DL toolchain, 140 (35%) failures in *Small Sample Set* are caused by the discrepancies between local and platform execution environments, and account for 14.8% GPU service time. Major differences include environment variables, input/output paths, dependent software, versions of Python, frameworks and toolkit libraries, etc. Another significant difference is that processes of a DL job run in Docker containers under a new credential, therefore the granted permissions are very likely different from those used on the local development machine.

Most of these failures are in the Execution Environment dimension. The fix of *Path Not Found* category is to parameterize file/folder paths and obtain them from external arguments or configuration files. In addition, developers should verify these paths as early as possible because later failure occurrences (*e.g.*, those in the model evaluation stage) will waste a lot of resources. *Library Not Found* and part of *Permission Denied* (installing Python packages into system folders) failures can be solved by a custom Docker image with all desired software pre-installed. Sampled *Syntax Error* failures are due to the misuse of an early version of Python and can be also solved in this way. In case that the dependent software is manually installed during job initialization, developers should specify the "--user" option to the *pip* command and install Python packages into their home folders. The remaining *Permission Denied* failures

are due to the insufficient permission in operating external files and folders. The fix is to set up the correct access modes in advance (*e.g.*, executing "chmod" in the launching Shell script) and validate them early in code.

Finding 5: Most failures in the Execution Environment dimension are caused by the environmental discrepancies between local and platform. The many discrepancies (in installed software, storage paths, granted permissions, etc.) make DL programs error prone.
Implication: Developers are encouraged to use custom Docker images with all desired software pre-installed, modify code to be more environment-adaptive, and verify paths/permissions as early as possible.

5.2 Inappropriate Model Parameters/Structures (IPS)

All of the 43 (10.75%) Deep Learning Specific failures in *Small Sample Set* are caused by inappropriate model parameters (*e.g.*, learning rate and batch size) or model structures (*e.g.*, numbers of layers and hidden units).

5.2.1 GPU Out of Memory. Most of the 39 failures are due to overlarge batch size and/or overcomplicated model. Figure 4 shows an example. Larger batch size and more sophisticated model may improve the model learning performance; however, they significantly increase GPU memory consumption. At present, developers largely rely on their domain knowledge to choose the optimal model configuration, due to the lack of necessary tooling support. For example, developers must use the memory-efficient 1×1 kernel convolution [39] under certain circumstances.

Some on-going works are proposed to address *Out of Memory* failures. For example, GPU memory consumption is statically estimated for certain computer vision tasks [34]. Another promising technique is GPU memory virtualization where data can be automatically swapped in/out between host and GPU when necessary [36, 44]. TensorFlow also has basic GPU memory swapping mechanism (*i.e.*, the "swap_memory" argument), which is enabled in certain model layers (*e.g.*, tf.nn.dynamic_rnn [4]). Swapping should be implemented efficiently, otherwise data latency will slow down the computation.

Finding 6: *GPU Out of Memory* failures are mostly caused by overlarge batch size and/or overcomplicated model.
Implication: Developers should proactively choose the optimal model parameters and structures, taking into consideration both available GPU memory and expected learning performance. Estimation and virtualization of GPU memory are two promising techniques.

5.2.2 Tensor Mismatch & Loss NaN. Although there are only 4 such failures in *Small Sample Set*, they are very DL specific and deserve more investigation.

One of the two *Tensor Mismatch* failures is caused by variable name mismatch between the saved model file and the constructed model. After communicating with the developer, its root cause was identified to be the different variable naming rules between single-GPU and multi-GPU in PyTorch. More specifically, the saved model

```
1  from keras.models import Sequential
2  from keras.layers import LSTM, Dense, Dropout
3
4  batch_size = ~~32~~ 24
5  max_context_len = ~~400~~ 200
6
7  # read dataset by batch using iterator
8  input_data = batch_iterator(batch_size, dataset, ...)
9
10 # model construction
11 model = Sequential()
12 model.add(LSTM(hidden_unit, input_length=max_context_len,...))
13 model.add(Dense(1, activation='relu'))
14 model.add(Dropout(0.5))
15 ...
16 train(model, input_data)
```

Figure 4: An example of *GPU Out of Memory* failure caused by overlarge batches and overlong input sequences. "batch_size" and "max_context_len" are arguments specified in the configuration file. The fix is to reduce both of them yet keep the model learning performance acceptable (lines 4-5).

```
1  import torch
2  import torch.nn as nn
3
4  input_channels = 3
5  out_channel = 64
6
7  class Discriminator(nn.Module):
8    def __init__(self, ...):
9      super(Discriminator, self).__init__()
10     self.main = nn.Sequential(
11       # input shape should be [batch_size, channel, height, width]
12       nn.Conv2d(input_channels, out_channel, ...),
13       ...
14       )
15
16   def forward(self, input):
17     return self.main(input)
18
19 # data is in [batch_size, height, width, channel] format
20 ~~-indata = data.to(device)~~
21 +indata = data.permute([0, 3, 1, 2]).to(device)
22 netD = Discriminator(...).to(device)
23 result = netD(indata)
```

Figure 5: An example of *Tensor Mismatch* failure caused by the wrong shape (line 19) of the initial input data. The fix (line 21) is to permute data to the correct shape (line 11).

file is from a multi-GPU training job while the constructed model is for single-GPU inference. The model restoration fails to match these variable names. The fix is using matched variable names in the constructed model or switching to multi-GPU inference. The other failure is demonstrated in Figure 5 whose root cause comes from shape mismatch between the initial input data and constructed model. The fix is to permute data to the correct shape.

Loss NaN failures root in the stochastic nature of deep learning algorithms. One of them occurs long after the training & validation stage starts. We contacted the developer for help as it was rather hard to discover the root cause. The developer told us that the job was to fine-tune a pre-trained model. However, if such pre-trained model has certain problematic parameters, the gradient

may lead to a bad training situation and cause the loss to be *NaN*. A quick fix is to try a new pre-trained model from the candidate set. Another possible root cause is the overlarge learning rate, whose fix is simply decreasing its value. The developer also suggested that *Loss NaN* was sometimes due to exceptional data in the initial input or the intermediate results. Filtering them out should solve the problem. *Loss NaN* failures are non-deterministic and very difficult to reproduce; therefore, the fix largely depends on the domain knowledge.

5.3 API Misunderstanding (APIM)

19 (4.75%) failures are caused by misunderstanding of the complex assumptions made by framework/library APIs. Figure 6 shows a *Framework API Misuse* failure triggered by an incorrect API argument value. The error message is *"ValueError: Variable attweights/kernel_weights_conv already exists, disallowed. Did you mean to set reuse=True or reuse= tf.AUTO_REUSE in VarScope?"*. Since the TensorFlow variable "kernel_weights_conv" (line 25) is shared among several GPUs (line 16, 19, 20), its parent scope (line 24) should be constructed with the parameter "reuse" explicitly set to "tf.AUTO_REUSE". However, the developer was unaware of this assumption and forgot to set the "reuse" parameter. Therefore, a default value was used, and TensorFlow runtime failed to create the variable at the second time.

A few failures are caused by the evolution of internal APIs. The interfaces between DL-related components are altered after software upgrade, therefore breaking the internal compatibility. Developers may think that their programs will still work since those changes are invisible to them. Figure 7 demonstrates a failed Keras [12] job with the error message *"TypeError: softmax() got an unexpected keyword argument 'axis' while using layers.Dense"*. Execution logs indicate that Keras 2.2.2 is installed in a TensorFlow 1.3 Docker container. The developer may have thought that this latest version of Keras would be compatible with TensorFlow 1.3. However, Keras 2.2.2 upgrades its Softmax implementation and requires TensorFlow 1.5 or higher. The fix is to downgrade Keras to a compatible version with TensorFlow 1.3. Also, to manage complex dependencies among the packages, DL platforms could provide a global package manager and a dependency checker.

Finding 7: Developers may not fully understand the complex assumptions made by framework/library APIs due to the rapid evolution of DL-related software, which results in failures related to framework API misuse, illegal argument, etc. Such internal compatibility issues are often invisible to the programs. **Implication:** Developers need deeper understanding of framework/library APIs. A custom Docker image could help mitigate part of these failures.

5.4 Exceptional Data (ED)

In *Small Sample Set*, 24 (6%) failures are caused by the exceptional data. Figure 8 illustrates such an example. The developer wants to get the area under the ROC (receiver operating characteristic) curve (*i.e.*, AUC) in the model evaluation phase [14]. However, there is only a single category existed in the "labels" variable (line 12), which fails to satisfy the multiple-category requirement of calculating AUC. Proactively writing exceptional-data-handling code could help avoid this type of failures. For example, the fault in Figure 8 can be fixed with "try-catch" to capture the thrown exception, thus avoiding the calculation of AUC in erroneous scenarios.

DL developers could learn from the distributed data-parallel programming practices: manual dataset cleaning before submission, defensive programming for exceptional data handling, and input data sampling for local testing [25]. More sampling tools are needed, which could help developers sample out representative data to not only ensure the distribution, but also take some corner

```
config.encode_layer = 3
model = CustModel(config, ...)

class CustModel(BaseModel):
  def __init__(self, config, **kwargs):
    self.encode_layer = config.encode_layer
    self.build_graph()

  def build_graph(self):
    with tf.device('/device:GPU:0'):
      # define some variables on the first GPU
    with tf.variable_scope('encoder'):
        self.model_encoder()

  def model_encoder(self):
    gpu_id = 0
    for i in range(self.encode_layer):
      if i > 1:
        gpu_id = 1
      with tf.device('/device:GPU:%d'%(gpu_id)):
        output = encoder_block(...)

  def encoder_block(...):
+   with tf.variable_scope('attweights', reuse=tf.AUTO_REUSE):
      var1 = tf.get_variable('kernel_weights_conv', shape, dtype, ...)
```

Figure 6: An example of *Framework API Misuse* failure caused by an incorrect API argument value. The fix is to set the parameter "reuse" to "tf.AUTO_REUSE" explicitly (line 24).

```
x = layers.Dense(classes, activation='softmax')(x) # in user code

def softmax(x, axis=-1): # Softmax implementation in Keras
  ...
  return tf.nn.softmax(x, axis=axis)
```

(a) Softmax in Keras 2.2.2 requires TensorFlow 1.5 or higher.

```
def softmax(logits, axis=None, name=None, dim=None):
```

(b) TensorFlow 1.5

```
def softmax(logits, dim=-1, name=None):
```

(c) TensorFlow 1.3 has different parameters.

Figure 7: An example of *Framework API Misuse* failure caused by framework evolution. Keras 2.2.2 is mistakenly installed with TensorFlow 1.3. The fix is to downgrade Keras to a compatible version.

```
1  from sklearn.metrics import roc_auc_score
2
3  def creat_model():
4    wide = create_wide(...)
5    deep = create_wide(...)
6    model = combine(wide, deep)
7    return model
8
9  model = creat_model()
10 for i in range(args.stpes)
11   train(model)
12   # AUC for 2-category model
13   x_test, labels = read_from_file(...)
14   scores = model.predict(x_test)
15 + try:
16     auc = roc_auc_score(labels, scores)
17     logging.info('test ROC AUC score:' + str(auc))
18 + except ValueError:
19 +   logging.info('fail to calculate ROC AUC score, may due to the
   ↪  same category in labels')
```

Figure 8: An example of *Corrupt Data* failure caused by the unexpected data distribution of input data for AUC calculation (*i.e.*, all values in "labels" are the same). The fix is to catch the thrown exception (lines 15, 18-19).

```
1  beam_width = 0 # original code reads it from a program argument
2
3  # generate nantural language output
4  def gen_text(bs_infer, ...):
5    n = bs_infer.shape[0]
6    hyps = dict((e, []) for e in range(beam_width + 1))
7    ...
8    # gereate text for beamsearch
9    for i in range(n):
10     for ii in range(beam_width):
11       hyps[ii + 1].append(get_natural(bs_infer[i, :, ii]))
12   return hyps
13
14 def infer_evaluate_save(data, ...):
15   for batch in data.batch_q_iter:
16     feed_dict = {inputs: batch}
17     results = model.infer(sess, feed_dict)
18     bs_infer = results['output']['bs_infer']
19     batch_hyps = gen_text(bs_infer, ...)
20 +   if beam_width > 0:
21       bs_turn = dict(question=question, answer=batch_hyps[1][i])
```

Figure 9: An example of *Key Not Found* failure caused by incorrect "beam_width" value. The fix is to add a check (line 20).

cases into account for better testing locally. Ideally, the DL platform should also provide data schema checker to ensure the data correctness automatically before job execution, hence improving user experience.

> **Finding 8:** The exceptional data handling problem is important for deep learning programming. DL developers could learn from the distributed data-parallel programming practices to avoid this problem.
> **Implication:** The exceptional-data-handling code could be improved. DL frameworks could provide dataset APIs to handle this problem. A data validation procedure or tool could be used to sample and check the validity of the data before a full-scale training is conducted.

5.5 Common Programming Errors (CPE)

Nearly half (174; 43.5%) of the failures in *Small Sample Set* are caused by common programming errors. Most of them (*e.g.*, the path concatenation example in Section 4.3) are easy to understand and have quick fixes by referring to the error messages and failure sites in code. However, a few failures are fairly complex and need thorough examination of the code logic. Figure 9 presents a *Key Not Found* failure of an NLP job. The original program is much simplified to ease presentation. The exception is thrown at line 21 in the model evaluation stage; however, no clues are found here or in the same function. After reading the complete source code, we notice two hints at lines 1 and 6, which are far away from the original failure site. It turns out that the developer mistakenly disables beam search [42] at program entrance, while the failed code needs beam search. The fix is to add a check for the "beam_width" variable.

According to our observation, DL developers often use many parameters/arguments in their code and scripts to control the experiments. However, sometimes missing or inappropriate arguments can violate the implicit assumptions in the code and lead to incompatible configurations. The failure symptoms include *Illegal Argument* and *Type Mismatch*, which dominate the CPE according to our statistics in Figure 3. A possible solution is to perform formal code review or advanced static code analysis to detect the programming errors earlier.

6 USER STUDY ON CURRENT TESTING AND DEBUGGING PRACTICES

6.1 Study Design

To find out why failures of DL jobs happen and how they are resolved, we need to better understand the current practices of testing and debugging DL jobs. More specifically:

- Testing practices. We would like to know whether developers test their programs locally in the same environment as Philly. If not, what their current testing environment is. We would also like to know the challenges developers face when testing DL programs.
- Debugging practices. We would like to know how developers currently debug their failed jobs, how they detect non-deterministic bugs resulted from the stochastic nature of DL, what support they want from the debugging tools, and whether a "Capture and Replay" tool is useful.

To answer the above questions, we conducted in-depth face-to-face interviews with 6 representative developers of Microsoft. All of the interviewees are algorithm engineers or researchers with 0.5 to 5 years of DL experience. They work on different products and tasks that are related to gaming, computer vision, natural language processing, speech, graphics, and advertisement. Table 2 shows the interviewee demographics and their experience in DL. Note that although only 6 developers are chosen, the cost of this study is significant as each interview is one-on-one and lasts 1-3 hours.

Id	Field	DL Exp. (year)
U1	NLP	5
U2	RL for Gaming	3
U3	Object Detection	0.5
U4	Speech Generation	4
U5	Graphics	5
U6	Ad. Click-Through Rate	1.5

Table 2: Interviewee demographics and their experience.

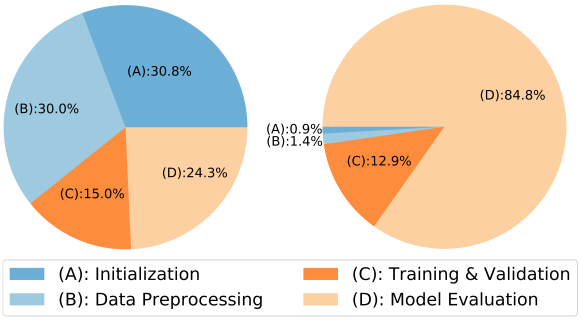

Figure 10: The number (left) and GPU service time (right) distribution of 400 job failures at different execution phases.

6.2 Testing Practices

Since Philly does not offer a local simulator, one challenge is to obtain a testing environment comparable to Philly (with the same hardware and DL-related software). We first study how developers prepare DL testing from three aspects: hardware, development environment, and input data. Four interviewees (U1, U2, U5, U6) had a few GPUs on hand. However, such GPUs were much smaller in memory and much slower in performance than those on Philly. Considering the hardware limitation, all interviewees admitted reducing the batch size, model complexity, and GPU count in local testing. Regarding to the development environment, U3 said that he took Philly as the testbed due to reasons such as lacking powerful GPUs. This may explain the existence of some simple bugs that could be solved before job submission. The others had their own local workstations. They manually set up a native Python DL environment by referring to the Docker images of standard DL toolchain provided by Philly. In addition, they simulated environment variables of Philly such as the input/output directories. About input data examination, three people (U1, U4, U6) would use a sampled smaller dataset for testing since the volume of original input data was large. Except U3, they always placed data files on their local machines.

The interviewees also told us a challenge about the test space in DL testing. There are a large number of hyperparameter combinations to test due to the stochastic nature of DL. For example, U2 used to submit a bunch of AutoML jobs to Philly [35]. However, they could only afford testing a very small set of candidates.

Furthermore, the interviewees described a challenge about testing at different DL phases. They usually focus more on the correctness of training pipeline (*e.g.*, data reading, model construction and serialization). Four interviewees (U2-U5) did not pay enough attention to the model validation and evaluation phases. They may stop testing after the first model checkpoint was saved successfully. To further investigate failures in different DL phases, we divided the execution of DL jobs into 4 phases and analyzed the failure occurrences of *Small Sample Set*, as shown in Figure 10. The number of failures occurred at the "Initialization" phase accounts for 30.8% of total failures, and they only consume 0.9% of total GPU service time. The number of failures occurred at the "Model Evaluation" phase accounts for 24% of total failures, but they consume 81% of total GPU service time. These results show that many failures actually happen in the model validation and evaluation phases, which could have been thoroughly tested.

Finding 9: The current DL testing practices are often insufficient due to the characteristics of deep learning. There are three major challenges: (1) incomparable testing environment; (2) large test space; (3) necessity of testing at different DL phases. **Implication:** Developers are encouraged to test more cases and all the DL phases. The local simulator of the platform, estimation of GPU memory consumption, and test data generator could be useful for DL testing.

6.3 Debugging Practices

We are also interested in how developers resolve failures. The six interviewees use conventional code editors (*e.g.*, Visual Studio Code) and debugging methods (*e.g.*, logging, breakpoints, and single stepping) for debugging DL programs. All of them adopt similar debugging practices for failed jobs on Philly: they first examine the log files, understand the problem context, and locate the fault in source code/script. For evident bugs, they simply apply fix to code (*e.g.*, correcting the object type), data (*e.g.*, replacing with a cleansed dataset), and experiment configurations. For the complex bugs, they turn to more debugging methods (*e.g.*, adding code to filter out the exceptional data) and resolve them by leveraging domain knowledge (*e.g.*, reducing learning rate for *Loss NaN*) or framework documents (*e.g.*, the example described in Figure 6). There are also some DL specific tools to help debugging, such as TensorFlow debugger [6] and TensorBoard [5, 45]. Developers could use such tools to track the Inf/NaN value and locate the operator that generates it. There are still some failures that are difficult to reproduce locally, therefore developers had to debug them through a trial-and-error process — submitting the modified programs to Philly repeatedly and observing the results.

For *GPU OOM*, reducing the batch size is often developers' first solution to this problem, then followed by reducing the network structure complexity. They also have some domain specific parameters such as the max sequence length in NLP and the image resolution in graphics, which can all be used to reduce the model complexity.

For *Loss NaN*, developers usually try to reproduce it by printing the intermediate results and checking the calculations. This process is usually without any debugging tool support and involves a lot of time-consuming manual work.

U1, U2, U3, and U4 mentioned that they wanted a tool to visualize the change of variable values, which they thought would be helpful to debug and improve the model learning performance. All interviewees think that it could be good to provide a replay mechanism, which can capture some useful data in the computation process and replay them when there is a failure. This would help locate bugs that are related to the stochastic nature of DL and exceptional data.

Finding 10: The current DL debugging practices are not efficient for fault localization in many cases. Existing debugging tools may not work well due to the discrepancies between local and platform execution environments, and the stochastic nature of DL.
Implication: Developers need more DL specific debugging tools. A good replay mechanism for saving the intermediate results and restoring them at later re-run could be helpful for debugging. A memory usage estimation tool could be complementary to current debugging methods.

7 DISCUSSION

7.1 Generality of Our Study

Although our study is conducted exclusively in Microsoft, we believe that our failure categories are prevalent and most of our results can be generalized to other DL platforms. The main reason is that most deep learning programs executed on Philly use the common programming paradigm (*e.g.*, Python language, stochastic algorithms, frameworks, and toolkit libraries) and target at common applications (*e.g.*, image/video/speech recognition and NLP). Besides, the hardware, system architecture, and job submission/execution mechanism of Philly are widely adopted [9, 23, 47].

As an example, Findings 1 and 6 reveal *GPU Out of Memory* failures. It is well-known that DL models with sophisticated network structures and large batch sizes may improve the model learning performance but also significantly increase memory consumption [13, 18]. Therefore, we believe Findings 1 and 6 are general to other platforms offering GPUs or even other AI accelerators like TPUs.

As another example, Findings 2 and 5 are about environment-related failures, which are not unusual to Philly jobs, even when Philly establishes a hermetic job execution environment via Docker images of standard DL toolchain. The underlying reason is that such hermetic environment could not always be identical to the local environment developers use. We believe these findings apply to other platforms too as developers are also likely to make the same mistakes. For instance, the troubleshooting webpage of Google Cloud AI lists several similar environmental failures[1].

7.2 Future Research Direction

Based on our study, we propose the following future research work:
Platform Improvement.

Avoiding Unnecessary Retries. Automatic retry is a common design in various platforms, mainly for recovery from non-deterministic system failures. Developers can also leverage this mechanism for non-deterministic deep learning specific failures (*e.g.*, *Loss NaN*). However, if a failure roots in the DL program deterministically (*e.g.*, accessing a non-existent data folder), re-running the job multiple times is apparently unnecessary and would simply waste resources. The platform could design more-refined heuristics to infer whether a failed job deserves retry and avoid the unnecessary ones.

Local Simulator. From Findings 2 and 5, we can see that unawareness of the environmental differences results in nearly half of the failures. Therefore, it is desirable to provide a platform simulator for local development. The local simulator can behave like a single-node implementation of the platform, which exports the same environment variables, accesses the distributed storage, and emulates one or more GPU devices.

Tool Support.

Estimation of GPU Memory Consumption. It is rather challenging to debug and fix *GPU Out of Memory* failures. To choose good model parameters/structures that satisfy memory consumption requirements, developers largely depend on their domain knowledge or adopt a trial-and-error strategy (*i.e.*, submitting several jobs with different model configurations). An estimator can be developed to infer the upper bound of GPU memory consumption based on features such as current data distribution, model structure, and batch size. A prediction model can be built by analyzing source code and historical *GPU Out of Memory* failures. Such tools could help developers better estimate memory usage of their DL programs and reduce *Out of Memory* failures.

Static Program Analysis. Static program analysis is a powerful technique to detect code defects without actually running the programs. It will have great potential to proactively handle many kinds of job failures including *Framework API Misuse*, *Tensor Mismatch* and those in the General Code Error dimension etc. Performing whole-program analysis across function calls can reduce the false alarms.

Data Synthesis. Motivated by Finding 8, it is useful to develop a data synthesis tool for testing the robustness of data processing logic by extending symbolic execution techniques [24]. This tool could generate special datasets conforming to the initial input distribution to trigger potential bugs.

Framework Improvement.

Automatic GPU Memory Management. Although DL frameworks hide most complexity of GPUs, developers still need manual GPU memory management for data placement and consumption control. Frameworks could provide an automatic GPU memory management mechanism like what OSes have done to the main memory. For example, cold data on GPU can be automatically swapped out [36, 44] to avoid GPU memory exhaustion. Another example is that the mechanism could prefetch the input data and model partitions used in the upcoming computations.

Replay. Debugging *Loss NaN* and other non-deterministic failures is challenging since it is hard to reproduce the same failing execution. Frameworks could use existing replay techniques [20] to record the precise execution sequence and interaction with the environment. Hence when a developer wants to track down a failure, she can deterministically replay to the faulty state, which can greatly facilitate fault diagnosis.

[1] https://cloud.google.com/ml-engine/docs/troubleshooting

8 RELATED WORK

Machine learning / deep learning bugs. Thung *et al.* [40] and Sun *et al.* [38] performed empirical analysis of bugs in machine learning systems. Such system defects can also lead to job failures; however, our study focuses on program/script bugs, which are written by DL application developers. Zhang *et al.* [49] conducted a research on GitHub and StackOverflow to study the TensorFlow program bugs. Islam *et al.* [22] extended the work to cover more DL frameworks. Some of the bugs they found (*e.g.*, bugs due to API evolution) are consistent with our observation. Our study analyzes real-world industrial DL jobs with multiple DL frameworks (*e.g.*, TensorFlow, PyTorch, and CNTK). We found many different types of bugs occurred in a cloud-based DL platform, which could not be easily discovered by the developers through local testing.

Empirical study. There are some empirical studies on the faults of Big Data platforms and software systems [25, 46, 48, 50]. For example, Xiao *et al.* [46] studied non-deterministic bugs in SQL-like MapReduce programs. Li *et al.* [25] analyzed code/data defects in distributed data-parallel programs. Zhou *et al.* [50] performed an empirical study on service quality issues of a production Big Data computing platform, which has a variety types of issues including hardware failures and user errors. Zhang [48] analyzed the distribution of faults in large-scale software systems. Our study concentrates more on program failures of production deep learning jobs.

Cluster infrastructure. Much research [19, 23, 33, 47] aims to improve the GPU cluster utilization for deep learning by studying the unique job characteristics. Jeon *et al.* [23] pioneered in the understanding of low GPU utilization of a multi-tenant cluster. Such low utilization comes from gang scheduling [29], resource locality, and job failures. Those job failures distribute across infrastructure, DL framework, and user code. Apart from classifying the failure symptoms, our work studies DL program failures and presents a deep analysis of their root causes.

Testing. Recently, there is a large amount of work on testing DL models. For example, DeepXplore [32] proposes a new metric to measure the test coverage of a deep neural network. DeepTest [41] further introduces a domain-specific automated testing tool to maximize the neuron coverage through realistic transformations over the image data. TFX [8] builds a machine learning pipeline that eases user efforts in deployment, which includes a set of system components for data analysis, transformation, and validation. Therefore, the platform can significantly reduce training failures and improve model quality. Our work provides a systematic analysis of program failures of DL jobs, which can help the community design better, more practical testing and debugging toolkits for DL programming.

9 CONCLUSION

Like other computer programs, deep learning programs could also fail to execute. In this paper, we describe the first empirical study on program failures of deep learning jobs. We manually examine the failure messages collected from 4960 failed jobs in Microsoft and classify them into 20 categories. In addition, we identify the common root causes and bug-fix solutions on a sample of 400 failures. To better understand the current testing and debugging practices for deep learning programs, we also conduct developer interviews. Based on our findings, we suggest possible research topics and tool support that could facilitate deep learning training and testing. We believe our work could help understand the quality issues of deep learning programs and provide valuable guidelines for future deep learning development.

ACKNOWLEDGMENT

We would like to thank all interviewees participated in our user study. Hongyu Zhang would like to acknowledge the support of ARC DP200102940.

REFERENCES

[1] 2019. Amazon SageMaker. https://aws.amazon.com/sagemaker.
[2] 2019. Google Cloud AI. https://cloud.google.com/products/ai.
[3] 2019. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning-service.
[4] 2019. The "swap_memory" argument in TensorFlow operator "tf.nn.dynamic_rnns" example. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/nn/dynamic_rnn.
[5] 2019. TensorBoard: TensorFlow's Visualization Toolkit. https://github.com/tensorflow/tensorboard.
[6] 2019. TensorFlow Debugger. https://www.tensorflow.org/guide/debugger.
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, USA, 265–283.
[8] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, and et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1387–1395. https://doi.org/10.1145/3097983.3098021
[9] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, K JAYARAM, Michael Kalantar, Vinod Muthusamy, Priya NAG-PURKAR, and Florian Rosenberg. 2017. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS*.
[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274
[11] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, and et al. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems (DLRS 2016)*. Association for Computing Machinery, New York, NY, USA, 7–10. https://doi.org/10.1145/2988450.2988454
[12] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras.
[13] 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, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. 4171–4186.
[14] Tom Fawcett. 2006. An Introduction to ROC Analysis. *Pattern Recogn. Lett.* 27, 8 (June 2006), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010
[15] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. *CoRR* abs/1705.03122 (2017). arXiv:1705.03122 http://arxiv.org/abs/1705.03122
[16] Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár, and Kaiming He. 2018. Detectron. https://github.com/facebookresearch/detectron.
[17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1487–1495. https://doi.org/10.1145/3097983.3098043
[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017). arXiv:1706.02677 http://arxiv.org/abs/1706.02677
[19] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proceedings of the 16th USENIX*

Conference on Networked Systems Design and Implementation (NSDI '19). USENIX Association, USA, 485–500.

[20] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, USA, 193–208.

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2980–2988. https://doi.org/10.1109/ICCV.2017.322

[22] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[23] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 947–960.

[24] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[25] 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 the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 963–972.

[26] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014, 239, Article 2 (March 2014).

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[28] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[29] J.K. Ousterhout. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 22–30.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, 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 *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035.

[31] Nick Pawlowski, Sofia Ira Ktena, Matthew C. H. Lee, Bernhard Kainz, Daniel Rueckert, Ben Glocker, and Martin Rajchl. 2017. DLTK: State of the Art Reference Implementations for Deep Learning on Medical Images. *CoRR* abs/1711.06853 (2017). arXiv:1711.06853 http://arxiv.org/abs/1711.06853

[32] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[33] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3190508.3190517

[34] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the International Conference on Learning Representations*.

[35] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3135974.3135994

[36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Article 18, 13 pages.

[37] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 2135. https://doi.org/10.1145/2939672.2945397

[38] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An Empirical Study on Real Bugs for Machine Learning Programs. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*. 348–357. https://doi.org/10.1109/APSEC.2017.41

[39] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[40] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12)*. IEEE Computer Society, USA, 271–280. https://doi.org/10.1109/ISSRE.2012.22

[41] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[42] Christoph Tillmann and Hermann Ney. 2003. Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation. *Comput. Linguist.* 29, 1 (March 2003), 97–133. https://doi.org/10.1162/089120103321337458

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS '17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[44] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[45] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mané, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. 2018. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics* 24, 1 (2018), 1–12.

[46] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. 2014. Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-Commutative Aggregators in MapReduce Programs. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 44–53. https://doi.org/10.1145/2591062.2591177

[47] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, and et al. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, USA, 595–610.

[48] Hongyu Zhang. 2008. On the Distribution of Software Faults. *IEEE Transactions on Software Engineering* 34, 2 (March 2008), 301–302.

[49] 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)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[50] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An Empirical Study on Quality Issues of Production Big Data Platform. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, 17–26.