An Empirical Study on Low GPU Utilization of Deep Learning Jobs

Yanjie Gao
yanjga@microsoft.com
Microsoft Research
China

Yichen He*
hyc2026@buaa.edu.cn
Microsoft Research
China
Beihang University
China

Bo Zhao
v-bozha@microsoft.com
Microsoft Research
China
Beihang University
China

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

Haoxiang Lin†
haoxlin@microsoft.com
Microsoft Research
China

Mao Yang
maoyang@microsoft.com
Microsoft Research
China

ABSTRACT

Deep learning is playing a critical role in many intelligent software applications. For efficient model training and testing, enterprise developers submit and run deep learning jobs on shared, multi-tenant platforms. These platforms are typically equipped with a large number of graphics processing units (GPUs) to accelerate deep learning computation. However, some jobs exhibit rather low utilization of the allocated GPUs, which not only wastes a significant amount of precious computing resources but also reduces development productivity. This paper presents a comprehensive empirical study on low GPU utilization of deep learning jobs. 400 real jobs (whose overall GPU utilization $\leq 50\%$) are collected from an internal deep learning platform of Microsoft. We manually examine the job metadata, execution logs, runtime metrics, scripts, and programs to discover 852 low-utilization issues. Further, we identify the common root causes and fix solutions. Our major findings include: (1) The low GPU utilization of deep learning jobs roots in insufficient GPU computation and interruption of non-GPU operations; (2) Over half (54.81%) of the low-utilization issues are caused by data operations; (3) 37.44% of the issues are related to deep learning models and occur at the model training stage; (4) Most low-utilization issues could be fixed by a few lines of code/script changes. Based on the study results, we suggest possible research directions that could help developers better utilize GPUs in a deep learning platform.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

deep learning jobs, GPU utilization, empirical study

1 INTRODUCTION

Deep learning (DL) has made remarkable achievements in many areas (such as natural language processing, gaming, and image recognition) and is playing a critical role in many intelligent software applications. To facilitate efficient model training and testing, enterprises build shared, multi-tenant platforms (such as Amazon SageMaker [2], Microsoft Azure Machine Learning [28], and Google Cloud AI [12]) for developers to submit and run their DL jobs. These DL platforms are equipped with a large number of hardware accelerators dedicated to DL computation, among which graphics processing units (GPUs) are typically used. As deep learning is computationally intensive [50], the GPU utilization, a measurement of the actual GPU computing time as a percentage of the GPU's total service time, is an important indicator to the runtime performance of DL jobs.

Inside Microsoft, we have ITP, an internal DL platform that is built with commodity computing hardware (*e.g.*, GPU and RDMA network) and widely used open-source software (*e.g.*, Kubernetes [45] and Docker [27]). Every day, hundreds of developers from various research and product teams train and test their models on ITP for object detection, machine translation, advertisement, *etc.* At present, these DL jobs request GPUs exclusively because there lacks mature and efficient hardware/software support for fine-grained GPU sharing [13, 46]. Although we expect that DL jobs should fully utilize the computing resources, we actually notice that some of them exhibit rather low utilization of the allocated GPUs. For instance, a job with eight NVIDIA Tesla V100 GPUs used an apparently smaller batch size, which caused its GPU utilization to fluctuate mostly between 10% and 40%. Another example is that a job frequently saved model checkpoints to the user's remote storage for fault tolerance. Because of the large model size and the synchronous remote saving, the GPU had to be idle for minutes periodically, and thus the training was severely slowed down. The low GPU utilization of DL jobs not only leads to a significant waste of precious platform resources (including GPUs, CPUs, main memory, network bandwidth, and storage) but also reduces development productivity. Furthermore, such adverse impacts may be amplified in the widely adopted practice of automated machine learning (AutoML), where many trial jobs of the same experiment could present the analogous computation process and GPU utilization since their programs, neural architectures, and hyperparameter values are highly similar. Therefore, it is important to understand why the DL jobs cannot utilize GPUs well and how to fix the problems.

There has been much prior work on CPU utilization [4, 14, 19, 29, 51]; however, they could not adequately facilitate DL developers to improve the GPU utilization due to the wide differences between

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

CPUs and GPUs. Recently, researchers conduct deep learning related empirical studies [5, 17, 18, 20, 21, 48, 55–57], many of which focus on program/job/framework/compiler bugs and failures. For example, Zhang et al. [57] studied 175 TensorFlow [1] program bugs collected from GitHub and Stack Overflow. Jeon et al. [20] analyzed low GPU utilization of large-scale, multi-tenant GPU clusters for DL training. Taking the perspective of platforms, the authors pointed out that the root causes came from resource locality, gang scheduling [36], and job failures. This research considers more about the cluster's GPU utilization, instead of that of the individual jobs. A related work by Cao et al. [5] characterized 238 performance bugs in TensorFlow and Keras [8] programs. Their study subjects were collected from 225 Stack Overflow posts, being different from our industrial jobs on ITP in many aspects. Moreover, the authors did not include DL programs written with the popular PyTorch [37] framework.

In this paper, we conduct a comprehensive empirical study on low GPU utilization of deep learning jobs. We randomly select 400 real jobs from ITP as the study subjects, whose overall GPU utilization is less than or equal to 50%, which is a utilization threshold decided with the ITP product team. After having manually examined the jobs' metadata, execution logs, runtime metrics, scripts, and programs, we discover 852 low-utilization issues (*i.e.*, issues that visibly reduce the GPU utilization) in the selected jobs. We further identify their common root causes and fix solutions and classify them into 4 high-level dimensions and 14 categories. These issues are caused by the code logic of scripts and programs instead of the platform or hardware (*e.g.*, a malfunctioning GPU freezing the job) because the latter class is beyond the scope of our study. We obtain many findings and list the majors as follows:

(1) The low GPU utilization of deep learning jobs roots in insufficient GPU computation (*e.g.*, using a small batch size) and interruption of non-GPU operations (*e.g.*, saving a model checkpoint to the distributed storage) from CPUs, I/O, network, PCI-e bus, *etc.*
(2) Over half (467; 54.81%) of the low-utilization issues are caused by data operations, such as inefficient data transfer between the main memory and GPU memory (197; 23.12%) and remote data read from the distributed storage (195; 22.89%).
(3) 37.44% (319) of the total issues are related to deep learning models and occur at the model training and evaluation stages, such as improper hyperparameter values (181; 21.24%) and model checkpointing (116; 13.62%).
(4) Most (about 87%) low-utilization issues could be fixed by a few lines of code/script changes. However, solutions of the others rely on improvements in deep learning platforms, frameworks, and tool chains.

Based on the study results, we suggest possible development guidelines for deep learning jobs to better utilize GPUs. We also point out possible future research work.

In summary, this paper makes the following contributions:

(1) We perform a comprehensive empirical study on low GPU utilization of deep learning jobs. We discover 852 low-utilization issues in 400 industrial jobs and manually identify the root causes and fix solutions.

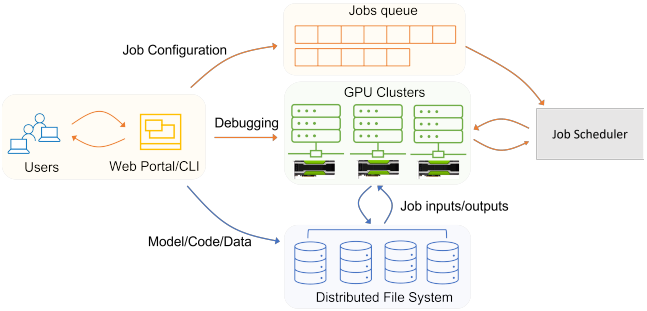

Figure 1: Workflow of ITP.

(2) We present our findings obtained from the empirical study and suggest improvements for developers and platforms to better utilize GPUs in a deep learning platform.

The rest of the paper is organized as follows. Section 2 gives an overview of deep learning jobs and ITP. Section 3 describes the methodology of the study. Section 4 presents our study results on classification, root causes, and fixes. Section 5 discusses the generality of this study and future research topics for improving the GPU utilization. We survey related work in Section 6 and finally conclude our paper in Section 7.

2 BACKGROUND: DEEP LEARNING JOBS ON ITP

As the internal deep learning platform of Microsoft, ITP is deployed on multiple physical GPU clusters and serves hundreds of developers from both research and product teams. Every day, thousands of DL jobs for various applications, such as object detection, machine translation, advertisement, and gaming, are submitted and executed on ITP.

Because ITP adopts comparable computing hardware, widely-used open-source software, and common DL programming paradigm, the job submission and execution in ITP is very similar to that in public DL platforms of Amazon SageMaker [2], Microsoft Azure Machine Learning [28], and Google Cloud AI [12]. In Figure 1, we briefly illustrate the workflow of ITP. Firstly, developers upload the input data, scripts, programs, and model checkpoints to the distributed storage via the web portal or command line tool. Their code is normally written in the Python language with DL frameworks like PyTorch [37] and TensorFlow [1]. Next, developers need to specify the job configuration, which includes the resource quota (*e.g.*, the desired GPU type and number), input/output paths, Docker [27] image, main program file, *etc.* ITP provides a standard deep learning tool chain (*e.g.*, official TensorFlow Docker images) to establish a hermetic job execution environment. The submitted jobs wait in queues initially to be scheduled. Once a job is selected, ITP allocates the requested GPUs and other resources all at once (*i.e.*, gang scheduling [36]) and instantiates Docker containers to execute the startup Shell script and the main Python program file. Model training is then performed on one or more physical GPU machines of ITP. If the job experiences an unexpected slow-down or a sudden crash during its execution, the developer can directly connect to those worker machines for remote debugging.

From a developer's perspective, the life cycle of a DL job is typically divided into the following four consecutive stages:

(1) Initialization. The job customizes the execution environment on ITP to keep the same as that on the developer's local machine because of the apparent environmental differences [55]. For example, many dependent libraries may be missing, especially when the developer specifies an official DL Docker image. Therefore, the startup Shell script must install them by either invoking the "pip" tool or cloning and building the source code.
(2) Data preprocessing. This stage is usually for cleaning and augmenting[1] the input data. Because the job continuously uses such large and remote data for a long time, it is a common practice for developers to download the data to the worker machines first.
(3) Model training. In this stage, a DL model (*i.e.*, a layered data representation [10]) is firstly constructed with mathematical operations called *operators*, including, for example, `Conv2d` (2D convolution) and `ReLU` (rectified linear unit function). Then, model training is actually to update the weight parameters of operators iteratively until the model learning performance (*e.g.*, predictive accuracy) reaches our expectation. Once every several training iterations, the job may measure current learning performance in order to tune hyperparameter values or terminate early.
(4) Model evaluation. After having finished the training, the job quantifies the final learning performance and outputs the trained model and evaluation results to the distributed storage.

ITP employs Prometheus [41] to monitor the system running status and collect the real-time metrics of each DL job at given intervals. The GPU utilization is queried using the NVIDIA Data Center GPU Manager (DCGM) [33]. Other metrics include GPU memory footprint, CPU utilization, main memory footprint, network sent/received bytes, disk read/write bytes, *etc.*

3 STUDY METHODOLOGY

3.1 Subjects

We choose real deep learning jobs on ITP as our study subjects. These jobs were submitted by developers from product and research teams and were successfully executed within a 10-day period in August 2021. Failed or terminated jobs are not considered because their life cycles are usually incomplete and thus some low-utilization issues may not be exposed. We also exclude the system jobs that were submitted for testing purpose only. Note that developers may have tested and refined their DL programs locally before job submission; therefore, some issues that could be resolved by local testing may not be present in this work.

As mentioned before, ITP collects the GPU utilization of a DL job periodically. Suppose that the job uses N GPUs, and t, t' are its start and end time, respectively. Let t_j be K time points ($1 \le j \le K$ and $t = t_0 < t_1 < \cdots < t_K = t'$) at which $u_{i,j}$, the current GPU utilization of the i-th GPU ($1 \le i \le N$), is collected. Then, we calculate the overall GPU utilization of the i-th GPU (denoted by U_i) and the job (denoted by U) as follows:

$$U_{i\in[1,N]} = \frac{\sum_{j=1}^{K} u_{i,j} \times (t_j - t_{(j-1)})}{t' - t}, \qquad U = \frac{\sum_{i=1}^{N} U_i}{N}.$$

We finally selected 400 jobs whose overall GPU utilization was less than or equal to 50%, which was a utilization threshold decided with the ITP product team. The jobs were randomly chosen across all the job submitters, teams, clusters, and application areas. For each selected job, we also collected its related information for later investigation, including, for example, the job metadata, execution logs, various runtime metrics, scripts, and programs.

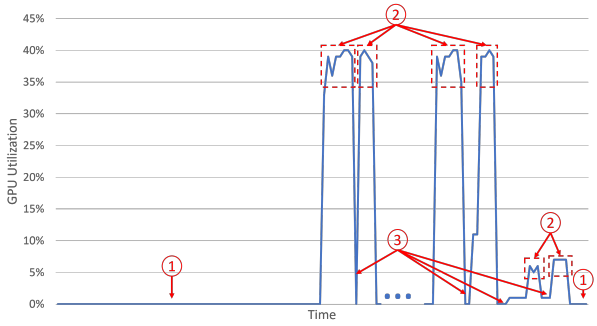

Figure 2: Simplified time series chart of GPU utilization from a single-GPU job. The X-axis denotes the time while the Y-axis expresses the GPU utilization as a percentage. ①, ②, and ③ represent three typical low-utilization patterns.

3.2 Root Cause and Fix Identification

Because the life cycle of a DL job is typically divided into four stages, we adopt a divide-and-conquer strategy to identify the root causes and fixes of low GPU utilization at each stage independently. We examined the job's scripts and programs to confirm the logical boundaries of the stages. The actual boundaries from the time dimension were determined by searching the execution logs with keywords that appeared in the logging statements, such as "init", "install", "download", "copy", "load", "training", "weights", "checkpoint", "validating", "evaluating", and "save". Model training is an iterative process; therefore, we further tried our best to determine the temporal interval of each training iteration.

Next, we studied the sequence of GPU utilization collected by ITP at each stage. To better illustrate, we show a simplified time series chart of GPU utilization from a single-GPU job in Figure 2, which includes only a few training and evaluation iterations for brevity. The X-axis denotes the time while the Y-axis expresses the GPU utilization as a percentage. We indeed observed three typical low-utilization patterns that appeared alternately from all the 400 jobs. The first pattern is that a GPU is idle at all; that is to say, the GPU utilization is always zero for a time, which is demonstrated by ① in Figure 2. The second is that although a GPU is in normal computation, but the GPU utilization generally remains at a low level, which is demonstrated by ②. The last one is that the GPU utilization suddenly drops severely (even to zero) for a while, which is demonstrated by ③. To determine all the temporal intervals

[1]Data augmentation is a technique to obtain more training data by transforming the existing data (*e.g.*, randomly cropping or rotating the input images).

Table 1: Classification of the 852 low-utilization issues found in 400 DL jobs by the root causes.

Dimension	Category	Description	No.	Ratio
Job	Interactive Job	The execution of a job needs interaction with the developer.	15	1.76%
	GPU Oversubscription	A job requests more GPUs than it actually uses.	6	0.70%
	Unreleased Job	A job is not closed in time after its computation has completed.	9	1.06%
	Non-DL Job	A job that is unrelated to deep learning and does not use GPUs at all is submitted and executed on the deep learning platform. For example, the job performs data analysis.	4	0.47%
	Subtotal		34	3.99%
Model	Improper Hyperparameter	Improper hyperparameter values (*e.g.*, a small batch size) are used, which reduces the amount of GPU computation.	181	21.24%
	GPU Memory-Bound	The GPU memory is insufficient to support more GPU computation.	22	2.58%
	Model Checkpointing	A job frequently saves the model checkpoint to the distributed storage.	116	13.62%
	Subtotal		319	37.44%
Data	Inefficient Host/GPU Data Transfer	The data transfer between the main memory and GPU memory is not efficient.	197	23.12%
	Remote Data Read	A job directly opens the input file and reads it from the distributed storage.	195	22.89%
	Data Preprocessing	The raw input data is preprocessed before model training.	11	1.29%
	Intermediate Result Uploading	The intermediate training data is frequently saved to the distributed storage.	14	1.64%
	Data Exchange	The GPUs of a distributed job continually exchange data among themselves, such as gradients and output tensors.	50	5.87%
	Subtotal		467	54.81%
Library	Long Library Installation	The library installation takes too much time ($\geq$ 10 minutes).	16	1.88%
	API Misuse	API usage violates the assumptions.	16	1.88%
	Subtotal		32	3.76%
Total			852	100%

matching the above mentioned patterns, we used the three-sigma-based anomaly detection [6, 40] plus manual inspection.

For each of such temporal intervals, we carefully located and examined the Python/Shell statements that were executed within it to discover the low-utilization issues and reason about their root causes. The identification also referred to other related information such as execution logs and runtime metrics. For example, the confirmation of an improper-batch-size issue additionally relied on the GPU memory footprint. In the end, we discovered 852 low-utilization issues caused by the code logic of the job. Those rooted in the platform or hardware (*e.g.*, a malfunctioning GPU freezing the job) were excluded because they were beyond the scope of this study. We devised the fix solutions with our domain knowledge and conducted some experiments for confirmation. If an issue was out of our understanding, we directly contacted the job owner for clarification.

3.3 Threats To Validity

3.3.1 Threats To Internal Validity. Our work, unlike the prior studies [5, 18, 21, 48, 56, 57], lacks existing human labels, discussions, and conclusions for reference (*e.g.*, those in Stack Overflow questions and GitHub issues). Because of the inherent complexity of deep learning and a mass of manual effort, subjectivity inevitably exists in the analysis of low-utilization issues, root causes, and fix solutions. To reduce such a threat, each DL job was analyzed by two of the authors independently. If there was disagreement, we strived to reach a group consensus before making decisions. For complicated jobs, we also directly contacted their developers to seek help.

3.3.2 Threats To External Validity. We conduct our study on deep learning jobs that were all collected from ITP. Therefore, it is possible that certain findings may be restricted to Microsoft and may not hold in other companies and deep learning platforms, although ITP and other platforms have obvious similarities in the platform architecture, software stack, job management, and deep learning tool chain. To reduce this threat, we try not to draw specific conclusions that are only applicable to ITP and Microsoft in this work. In Section 5.1, we discuss the generality of our findings in detail.

4 STUDY RESULTS

In this section, we present the classification of low GPU utilization by the root causes and describe the fixes. In Table 1, the 852 low-utilization issues found in 400 DL jobs are classified into 14 categories. We further group these categories into four high-level dimensions: Job, Model, Data, and Library.

```
#!/bin/bash
...
bash train.sh
~~sleep 10m~~
```

Figure 3: An example of Unreleased Job issues. The job is forced to sleep for 10 minutes after training. The fix is to remove the extra sleeping code.

4.1 Dimension 1: Job

This dimension includes 34 (3.99%) low-utilization issues, which are caused by inappropriate job types or configurations. These issues usually lead to a long idle time or nonuse of one or more GPUs. There are four categories in the dimension.

Interactive Job (15; 1.76%) means that the execution of a DL job needs interaction with the developer. For example, a job with four NVIDIA V100 GPUs ran for about 251.8 minutes. However, we notice that the overall GPU utilization of the job was less than 1% because it regularly used the CPUs for interaction and only used the GPUs for a very short period. In most cases, developers remotely connected to ITP via SSH. They may develop and test their models like what they usually do on the local development machines. They may even temporarily debug and optimize their jobs. In other few cases, developers launched Jupyter [23] notebooks to execute the programs line by line. Since the interaction is nondeterministic, it is impossible to know when and how long the GPUs remain idle in advance. A common fix to these issues is to eliminate interactive jobs from a DL platform. Further, the platform could use preemptible resources to build a dedicated service (*e.g.*, JupyterHub) for interactive computing.

GPU Oversubscription (6; 0.70%): sometimes, developers request more GPUs than they actually use. A distributed job needs to know the total GPU number, which is typically passed by the developer as a parameter, to spread the workload among all the allocated GPUs. However, it is possible that the job receives a smaller number by mistake. For example, half of the eight GPUs of a PyTorch job have good utilization (about 83.24%) but the others are not used at all. By examining the scripts and programs, we found that the developer wrongly set the value of "--nproc_per_node"[2] to 4 instead of 8 (all GPUs are from a single machine); therefore, four GPUs remained idle throughout the job's life cycle. To fix this type of issue, developers need to consider carefully how many GPUs they really need and check whether or not their jobs receive the correct number. Also, the DL platform could provide a common interface for developers to query the job configuration easily (*e.g.*, via environment variables).

Unreleased Job (9; 1.06%) means that a DL job is not closed in time after its computation has completed. Figure 3 shows an example: a developer forced a job to sleep for an extra ten minutes at the end of the startup Shell script. We contacted the developer for the reason. He told us that he intended to upload the final result to the distributed storage after training. The developer had thought that the remote uploading was asynchronous, so he added the sleeping to wait for the uploading to complete. On the contrary, the access to the distributed storage was indeed synchronous; therefore, his operation was unnecessary and led to a significant waste of GPU time as the job had eight NVIDIA V100 GPUs. The fix to this issue is to remove the extra sleeping code. In other cases, the developers seemed to manage the life cycles of their jobs manually, but they failed to track the precise completion of the DL computation so that the GPUs were not released in time. In general, developers should rely on the platform to manage the life cycles of DL jobs, instead of using their own.

[2] https://pytorch.org/docs/1.10.0/distributed.html

```
import torch

train_batch_size = ~~128~~ 256
test_batch_size  = ~~128~~ 512
train_loader = DataLoader(dataset = train_data,
  ↪ batch_size = train_batch_size, shuffle = True)
test_loader  = DataLoader(dataset = test_data,
  ↪ batch_size = test_batch_size, shuffle = True)
```

Figure 4: An example of Improper Hyperparameter issues, which is caused by a small batch size. "train_batch_size" and "test_batch_size" are arguments specified in a configuration file for model training and model evaluation (*i.e.*, testing), respectively. The fix is to increase their values separately (lines 3–4).

Although ITP is a platform dedicated to deep learning, we still notice that some data analysis jobs and traditional machine learning jobs (*e.g.*, classification with Scikit-learn [38]) have been submitted, which raises *Non-DL Job* issues (4; 0.47%). These jobs used CPUs only, although the developers requested and were allocated GPUs. The fix is to prevent non-DL jobs from ITP and submit them to other dedicated platforms. For example, a data analysis job should be submitted to a big-data analytics platform.

4.2 Dimension 2: Model

There are 319 (37.44%) low-utilization issues in this dimension, which are related to DL models and occur at the model training stage.

The first of the three categories is *Improper Hyperparameter*, which includes 181 (21.24%) low-utilization issues. A DL model usually has many hyperparameters that participate in the computation of operators to control the model training process, such as the batch size, learning rate, and kernel size. Improper values of the hyperparameters could significantly reduce the amount of DL computation (denoted by the total number of floating-point operations or FLOPs) and thus the GPU utilization. Developers typically increase the batch size (*i.e.*, the number of input data items in one training iteration) to improve both the model learning performance (*e.g.*, predictive accuracy) and the GPU utilization. Note that larger batch sizes also increase the GPU memory consumption. Figure 4 shows an example in which a small batch size is used. The identification of an Improper Hyperparameter issue requires that the overall GPU utilization and the maximum GPU memory utilization at the model training or model evaluation stage are less than or equal to 50%. We add the condition on GPU memory utilization to ensure that the DL computation is not indeed limited by the GPU memory size (see the next paragraph. There are many low-utilization issues in the Improper Hyperparameter category because it is difficult for

Table 2: Overall GPU utilization and peak GPU memory utilization at the model training stage of a BERT job with different batch sizes. The original batch size is 32.

Model	Batch Size	GPU Utilization	GPU Memory Utilization
BERT [9] (large-uncased)	32	44.64%	30%
	64	54.17%	34%
	128	61.55%	44%
	256	70.31%	68%
	512	N/A (OOM)	N/A (OOM)

developers to decide the optimal hyperparameter values that fully utilize the target GPUs before job submission, which motivates the development of tools to estimate or predict the GPU utilization of a DL model. Another reason is that developers often use the same hyperparameter values for both model training and model evaluation without being aware that the amounts of GPU computation of the two stages are quite different (59 low-utilization issues). Since the latter is less computation-intensive (lacking backpropagation and weight update) than the former, the overall GPU utilization at the model evaluation stage could be very low. A simple yet common fix to this type of issue is to use different, larger batch sizes that do not raise out-of-GPU-memory exceptions for the model training and model evaluation stages separately. To understand the effect of the fix, we conducted an experiment, reran a BERT [9] job with the same setting (*e.g.*, 8 NVIDIA V100 GPUs), and measured the overall utilization of both GPU and GPU memory at the model training stage. The original batch size was 32, and we doubled the value in turn until the job exhausted the GPU memory and crashed with a batch size of 512. The experimental results in Table 2 indicate that larger batch sizes indeed bring promising growth in GPU utilization from 46.69% to 70.31%.

The second category is *GPU Memory-Bound* (22; 2.58%), which means that the GPU utilization of a DL job is limited by the GPU memory size. Compared to the main memory, the physical memory of a GPU is much smaller. If a job requires and consumes too much GPU memory (*e.g.*, for storing input/output tensors and model weights), it is hard to increase the amount of DL computation and GPU utilization (*e.g.*, enlarging the batch size) because of the potential out-of-GPU-memory exception. For instance, a job was training GPT2 [42], a huge language model, on one NVIDIA V100 GPU. Although the overall GPU utilization was only 40%, we could not simply increase the batch size from "2" to a larger value because the GPU memory was running out (the GPU memory utilization generally remained at a level of 83%). The identification of a GPU Memory-Bound issue requires two conditions. Firstly, the overall GPU utilization at the model training or model evaluation stage is less than or equal to 50%. Secondly, at the same time, the maximum GPU memory utilization is greater than 80%. To fix this type of issue, developers need to utilize the GPU memory more efficiently by careful data placement and timely swap-out of the cold data. Further, developers may try the automatic GPU memory management provided by certain DL frameworks and optimization libraries. For example, the recurrent neural network of

```
1  import torch
2 +import asyncio
3
4  async def main():
5    ...
6    for epoch in range(start_epoch, args.num_epochs):
7      if step > num_training_steps:
8        break
9      ...
10     for idx, batch in enumerate(tqdm(train_loader)):
11       loss = model(batch)
12       ...
13       optimizer.zero_grad()
14       loss.backward()
15       +if f1 > max_f1:
16         +max_f1 = f1
17         +await task
18       optimizer.step()
19       ...
20       f1, pred = evaluator.evaluate(dev_loader,
         ↪  model, step)
21       model.train()
22       ...
23       if f1 > max_f1:
24         max_f1 = f1
25         torch.save(model, ...)
26         +async def checkpoint():
27           +torch.save(model, ...)
28         +task = asyncio.create_task(checkpoint())
29
30 if __name__ == '__main__':
31   asyncio.run(main())
```

Figure 5: An example of Model Checkpointing issues. The fix is to use Python coroutines (lines 2, 4, 26–27, and 31) and save the model checkpoint asynchronously (lines 15–17 and 28).

TensorFlow (*i.e.*, `tf.compat.v1.nn.dynamic_rnn`) is able to swap tensors transparently via a `swap_memory` Boolean parameter. For another example, Microsoft DeepSpeed [44] uses a novel Zero Redundancy Optimizer (ZeRO) [43] to save significant GPU memory.

The third and last category is *Model Checkpointing* (116; 13.62%). A DL job may perform frequent model checkpointing, which interrupts the normal GPU computation regularly. Model checkpointing is necessary to recover from job failures; however, it is also time-consuming because all the model weights must be copied back to the main memory and then saved to the distributed storage. As the model gets larger, the GPUs have to be idle for a longer time until the checkpointing ends. For instance, a job with one NVIDIA V100 GPU saved a new checkpoint every epoch (*i.e.*, a full training on the entire input data). Each model checkpointing took about 96 seconds on average, which was more than half the time of an epoch (168 seconds on average). As a consequence, the overall GPU utilization dropped significantly to 39.57%. A common fix is to use asynchronous model checkpointing to keep both GPUs and I/O

```
1  import torch
2
3  train_loader = torch.utils.data.DataLoader(trainset,
   ↪  ..., num_workers=8 , pin_memory=True)
4  test_loader = torch.utils.data.DataLoader(testset,
   ↪  ..., num_workers=8 , pin_memory=True)
5
6  for epoch in range(num_epoch):
7    ...
8    for _, data in enumerate(train_loader, 0):
9      # get the inputs
10     inputs, labels = data
11     inputs = inputs.to(device , non_blocking=True)
12     labels = labels.to(device , non_blocking=True)
```

Figure 6: An example of Inefficient Host/GPU Data Transfer issues. The fix is to set both `pin_memory` and `non_blocking` parameters to `True` (lines 3–4 and 11–12), which enables locked memory pages and asynchronous data transfer.

working in parallel. Figure 5 shows an example and its fix. More advanced fixes are discussed in Section 5.2.

Model	Save Frequency	Method	GPU Utilization	Time(s)
BERT [9] (large-uncased)	per epoch	no save	49.12%	6519.8
		sync	44.64%	6867.7
		async	47.65%	6570.6
BERT [9] (large-uncased)	4 times per epoch	sync	37.69%	6873.5
		async	47.41%	6572.1

Table 3: Comparison of asynchronous and synchronous DL model checkpointing.

4.3 Dimension 3: Data

This is the largest among the four dimensions, consisting of 467 (54.81%) low-utilization issues that are caused by data operations. There are five categories, four of which are related to the input data and the last one is related to the data exchange among multiple GPUs.

Model training requires that the input tensor has been loaded into the GPU memory. PyTorch developers invoke the `torch.tensor.to` API to copy an input tensor from host (*i.e.*, the main memory) to device (*i.e.*, the GPU memory) since the input data initially resides in the main memory. However, the default use of this API increases the data transfer latency, causes continual fluctuations in GPU utilization, and thus raises *Inefficient Host/GPU Data Transfer* issues. This category has 197 (23.12%) issues and is the largest one among all the 14 categories. Note that these issues are specific to PyTorch jobs at present because TensorFlow transparently performs the host/device data transfer, so developers do not have any control. Figure 6 shows an example. A simple fix has two steps. PyTorch developers usually use the `torch.utils.data.DataLoader` class for loading and managing the input data; therefore, the first step is to set its Boolean parameter `pin_memory` to `True` (lines 3–4), which loads the input data into the locked main memory pages and fastens

```
1 +from nvidia.dali import *
2
3 +@pipeline_def(batch_size=..., num_threads=...,
  ↪  device_id= ...)
4 +def data_pipeline(image_path):
5   +jpegs, labels =
    ↪  fn.readers.file(file_root=image_path)
6   +images = fn.decoders.image(jpegs, ...)
7   +flipped = fn.flip(images, ...))
8   +return flipped, labels
9
10 for image_path in files:
11   ...
12   cur_out_file = os.path.join(out_path, cur_file)
13   tf.reset_default_graph()
14   ~~feat_extract.main(image_path = image_path, ...)~~
15   +pipe = data_pipeline(image_path = image_path)
16   +pipe.build()
17   ...
```

Figure 7: An example of Data Preprocessing issues. The fix is to use the NVIDIA Data Loading Library (DALI) to accelerate the data preprocessing by using GPUs.

the later host/device data transfer. The second step is to set the Boolean parameter `non_blocking` of the `torch.tensor.to` API to `True` (line 11–12) such that PyTorch tries to transfer the data asynchronously. Besides, developers could use bulk transfer (*i.e.*, transferring multiple input tensors at once) and tensor prefetching to improve the transfer efficiency further.

Remote Data Read (195; 22.89%) means that a DL job directly opens the input file and reads it from the distributed storage, which is the second largest category. Since the network transfer has obvious latency, the GPUs have to wait for the input data periodically. Furthermore, if the job does not hold all the input data in the main memory (due to, for example, an excessive amount of data or memory swapping), the remote file will be read repeatedly. A common practice adopted by developers is to directly copy the input file to the local storage of the worker machines. A better solution is to pipeline the data copying and model training; so, the latter can work on the input data that has just been copied.

Developers usually need to preprocess the raw input data. For example, there exist some dirty data that should be eliminated. Another example is data augmentation, in which developers flip, crop, or rotate the input images in a computer vision task to increase the amount of training data for better learning performance. However, the data preprocessing normally uses CPUs instead of GPUs, which raises *Data Preprocessing* issues (11; 1.29%). A common fix is to preprocess the raw input data in advance by a data analysis job. If the data are images, audios, or videos, developers can use the NVIDIA Data Loading Library (DALI) [34] to accelerate the preprocessing by GPUs. Figure 7 shows an example whose fix uses DALI.

Intermediate Result Uploading (14; 1.64%) means that a DL job frequently uploads the intermediate training result (*e.g.*, normal validation output or trial statistics of an AutoML experiment) to the distributed storage. These issues are similar to those in the Model Checkpointing category because all of them involve remote file

```
1  # Launch script:
2  # CUDA_VISIBILE_DEVICES=0,1 python -u train.py ...
3  import torch
4
5  # Main trainer:
6  class Trainer:
7    def __init__(self, model, train_dataset,
     ↪  test_dataset, config):
8      self.model = model
9      self.train_dataset = train_dataset
10     self.test_dataset = test_dataset
11     self.config = config
12
13     # take over whatever gpus are on the system
14     self.device = 'cpu'
15     if torch.cuda.is_available():
16       self.device = torch.cuda.current_device()
17      +if torch.cuda.device_count() > 1:
18       +self.model =
         ↪  torch.nn.DataParallel(self.model)
19       self.model = self.model.to(self.device)
```

Figure 8: An example of API Misuse issues, in which a data-parallel job uses only one of the two allocated GPUs. The fix is to contain the model with the `torch.nn.DataParallel` class (lines 17–18) for utilizing all the GPUs.

uploading. Therefore, the fix is also similar: developers create an asynchronous task to upload the intermediate result.

The GPUs of a distributed job need to continually exchange data, such as gradients and output tensors, among themselves. Therefore, the data exchange through the network or PCI-e bus frequently interrupts the allocated GPUs and causes the GPU utilization to drop to zero suddenly for a while. There are 50 (5.87%) issues in this *Data Exchange* category. Since DL frameworks do not provide explicit APIs to control the data exchange, we suggest that developers try optimization libraries. For example, the 3D (data, model, and pipeline) parallelism of Microsoft DeepSpeed has greatly improved communication efficiency. Besides, Horovod [47] users could increase the `backward_passes_per_step` configuration to reduce the frequency of data exchange.

4.4 Dimension 4: Library

This dimension includes 32 (3.76%) low-utilization issues, which are caused by problems of dependent libraries (including DL frameworks). These issues usually lead to a long idle time or nonuse of one or more GPUs. There are two categories in the dimension.

Long Library Installation (16; 1.88%) means that a DL job spends too much time (≥ 10 minutes) to install dependent libraries at the initialization stage. For example, it costs a job about 16 minutes to copy a huge code folder from the distributed storage to the worker machine. During such a period, the allocated NVIDIA V100 GPU has nothing to do but remains idle. Platforms provide official DL Docker images to establish a hermetic job execution environment. They also support custom Docker images in which developers can pre-install their dependent libraries. However, we notice that most jobs still manually manage the libraries by invoking "pip install" for standard Python packages, cloning the source code from the Internet, or simply copying a folder from the distributed storage. The reason is that the fast evolution of the deep learning tool chain makes a custom Docker image difficult to maintain. We still suggest that developers pre-build custom Docker images with all the dependent libraries installed, which not only reduces the waiting time but also avoids many library installation failures [55].

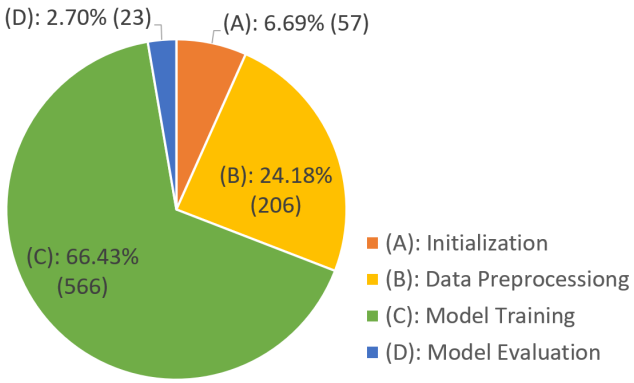

Figure 9: Distribution of 852 low-utilization issues at different execution stages.

API Misuse issues (16; 1.88%) are caused by mistaken usage of the complex framework/library APIs. Several jobs set a wrong value to the environment variable `CUDA_VISIBLE_DEVICES` that controls which GPUs can be seen by a CUDA application. For instance, a job uses an empty string to set `CUDA_VISIBLE_DEVICES`. As a consequence, all the allocated GPUs are hidden from the framework so that the job has to perform CPU training for about 481 minutes. The developers may have copied their startup Shell scripts from somewhere else (*e.g.*, their local development machines) but forgot to modify the value of `CUDA_VISIBLE_DEVICES` before job submission. Another API Misuse example is shown in Figure 8 in which a data-parallel job uses only one of the two allocated GPUs. The code at line 18 invokes the `torch.nn.Module.to()` API to load the model into the GPU memory; however, when there exist multiple GPUs, the `torch.nn.DataParallel` class[3] should be used first to contain the model (lines 17–18) for utilizing all the GPUs.

4.5 Execution Stage Analysis

In this section, we manually identify the stage at which each low-utilization issue occurred. If an issue is raised by the Python code, we locate the corresponding programs for identification. Otherwise, the issue is caused by a problem in the job configuration (*e.g.*, inappropriate job types) or the startup Shell script (*e.g.*, wrong script arguments), we ascribe its occurring stage to "Initialization". Figure 9 reports the distribution of the 852 low-utilization issues at different execution stages. 66.43% of the total issues occurred at the "Model Training" stage. The issues that occurred at the "Data Preprocessing" stage account for 24.18% of all the issues. As a consequence, developers should pay more attention to such two stages to improve the GPU utilization of DL jobs. Although only 6.69%

[3] https://pytorch.org/docs/1.10.0/generated/torch.nn.DataParallel.html

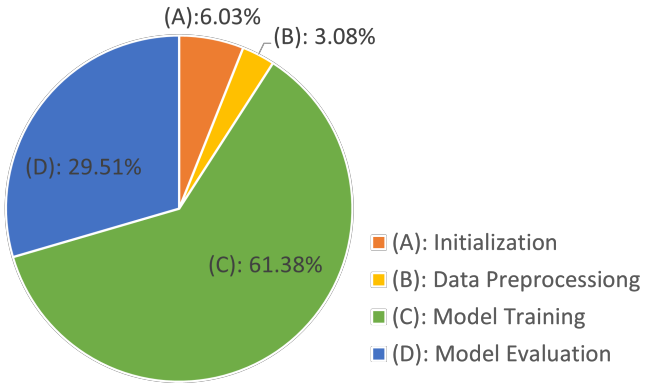

Figure 10: Average execution time of each stage of 400 jobs.

of the low-utilization issues were introduced at the "Initialization" stage, they could affect the jobs for a long time or throughout the whole life cycle. For example, setting an empty string to the environment variable `CUDA_VISIBLE_DEVICES` disables all the GPUs. Developers should proactively localize and prevent them as many as possible. As DL jobs usually perform simple and short tasks at the final "Model Evaluation" stage, the number of low-utilization issues that occurred at this stage is the least (2.70%).

Figure 10 reports the distribution of the average execution time of GPUs in each stage. Model Checkpointing, Improper Hyperparameter, GPU Memory-Bound, Inefficient Host/GPU Data Transfer, Remote Data Read, Intermediate Result Uploading and Data Exchange mainly occur in the model training phase, which spend more than 60% of the total time and may cause more waste of gpu resources if using improper configuration.

5 DISCUSSION

5.1 Generality of Our Study

Although our study is conducted exclusively within Microsoft, we believe that the study results on low GPU utilization are prevalent and can be generalized to other deep learning platforms, such as Amazon SageMaker, Microsoft Azure Machine Learning, and Google Cloud AI. The key reason is that our DL jobs on ITP share obvious similarities with those on other platforms from two aspects. Firstly, the programs of our jobs adopt the common deep learning programming paradigm. For example, they use the same Python language, DL algorithms, frameworks, and dependent libraries. Also, they target common DL applications, such as natural language processing and image recognition. Secondly, the system architecture, computing hardware, and software stack of ITP are widely adopted [3, 20, 53]. Therefore, our job management mechanism (including submission, scheduling, and execution) is very similar to that of other platforms.

As an example, some low-utilization issues root in the characteristics of GPUs and DL frameworks, so they are general to other platforms. The physical memory of a GPU is much smaller than the main memory; therefore, it is impossible to hold too much data in the GPU memory. Since GPUs and frameworks do not provide a transparent and efficient mechanism for data placement and swapping at present, developers are likely to make mistakes that lead to Inefficient Host/GPU Data Transfer and GPU Memory-Bound issues. Also, it is very difficult to choose appropriate hyperparameter values (*e.g.*, the batch size) and structures that best match a GPU's computing capacity before job submission, because the actual DL computation is hidden by frameworks and the proprietary NVIDIA CUDA/cuBLAS/cuDNN APIs.

As another example, some low-utilization issues root in the environmental differences between platforms and local machines. It is common that developers run their DL programs interactively for local development. However, interactive jobs are not affordable since the platform resources are much more precious. Such environmental differences can also lead to library-related issues. For instance, developers need to install many dependent libraries since they are missing in the standard DL Docker images, but in the meantime the allocated GPUs have to remain completely idle.

We indeed notice that several similar low-utilization issues and fix suggestions have been listed in the performance and cost optimization guide [11] of Google Cloud AI, which confirms the generality of our study. For example, user-managed notebooks (a type of interactive jobs) "should be switched off or deleted unless they are really running experiments" [11]. Another suggestion is that developers preprocess the input data in advance and save it as a TFRecord file for later training. Also, developers can use TensorFlow Transform [49] to create flexible pipelines so that model training does not wait for the end of data preprocessing.

5.2 Future Research Direction

In this section, we propose the following future research works based on our study:

Platform Improvement.

GPU Sharing. The NVIDIA Multi-Instance GPU (MIG) [35] technique partitions a GPU (*e.g.*, A100) into many isolated instances, each of which has its "own high-bandwidth memory, cache, and compute cores" [35]. DL platforms can employ MIG and extend their job schedulers to pack several low-GPU-utilization jobs dynamically to different instances of a group of shared GPUs so as to improve the overall utilization of these GPUs.

Distributed Cache. Distributed training for large DL models has become popular recently. It is not affordable to simply copy the input data from the distributed storage to each worker machine; sometimes, it is even impossible when the data volume is overlarge. A distributed cache for DL workloads can be very useful to fasten the training process and reduce resource waste, especially in the AutoML scenario where an experiment launches many trial jobs consuming the same input data. Like prior big-data platforms such as Apache Spark [54] and Microsoft Cosmos [39], DL platforms should work with frameworks together to understand the data access pattern of DL jobs for better cache performance.

Heterogeneous Pipelining. We observe that many low-utilization issues involve synchronous data downloading, preprocessing, and uploading, which do not use GPUs in the vast majority of cases. However, the GPUs have been allocated from the beginning because of the gang scheduling and cannot be reclaimed at any time. It is desirable to construct a single DL job as a pipeline of heterogeneous jobs. For example, the data preprocessing stage can be executed by a CPU-only big-data job, which transfers its output data in either

the batch or the streaming mode to one or more successor GPU training jobs.

Framework Improvement.

Unified Computing. At present, DL frameworks formalize a model as a tensor-oriented *computation graph*, whose nodes denote DL operators. With such a graph, frameworks are able to know and optimize the underlying DL computation. However, they are not aware of other operations performed by CPUs (*e.g.*, learning performance calculation and host/GPU data transfer), I/O devices (*e.g.*, model checkpointing), and network cards (*e.g.*, distributed gradient aggregation), which may interrupt GPUs at any time. A research direction is to unify all the GPU and non-GPU operations into a single computation graph, so that DL frameworks could schedule their execution in parallel and keep GPUs always busy.

Efficient Model Checkpointing. Model checkpointing is very necessary for DL jobs to recover from both hardware and software failures. However, frequent checkpointing, especially for a large DL model, may idle GPUs for a time constantly. We could use the techniques such as asynchrony, delta checkpointing, compression, pre-fetching, and high-performance serialization to implement more efficient model checkpointing.

Tool Support.

Estimation and Prediction of GPU Utilization. At present, developers largely depend on their domain knowledge and job submission to understand how GPUs are utilized, which is challenging and resource-consuming. An estimator can be developed to infer the GPU utilization of a DL model by building an analytic utilization model from the actual DL computation and GPU computing capacity. Also, we could train a prediction model from the historical GPU utilization data. These tools are useful to developers for selecting the optimal hyperparameter values and model structures that fully utilize the target GPUs.

Static Checker. A static checker performs an analysis on computer software without the need to execute the software's programs. We notice that most (about 87%) low-utilization issues could be fixed by a few lines of code/script changes. Therefore, it is promising to develop a static checker that proactively detects low-utilization issues from the code, scripts, and configurations before job submission, such as those in the GPU Oversubscription, Inefficient Host/GPU Data Transfer, Remote Data Read, and Model Checkpointing categories.

6 RELATED WORK

There have been many prior works that target the modeling, estimation, and prediction of CPU utilization [4, 14, 19, 29, 51]. However, they could not adequately facilitate DL developers to improve the GPU utilization due to the wide differences between CPUs and GPUs. Also, many researches focus on performance issues [15, 16, 22, 24, 25, 31, 32]. Jin et al. [22] conducted a comprehensive study on 110 real-world performance bugs that were randomly sampled from five software suites. The authors then formulated efficiency rules from patches for future performance bug detection. Nistor et al. [32] and Nistor [31] studied how performance bugs were discovered, reported, and fixed by developers. PCatch [24] automatically and accurately predicted performance cascading bugs in representative distributed systems based on the execution under only small-scale workloads. Although these works do not directly target deep learning, they still provide general information and hints for improving the overall performance of DL jobs.

Recently, some researchers conduct empirical studies on deep learning [5, 17, 18, 20, 21, 48, 55–57]. Many of them focus on the bugs and failures of DL programs, jobs, compilers, and frameworks. Zhang et al. [56] examined 715 Stack Overflow questions to study common challenges in DL application development, many of which asked for help on shape inconsistency bugs. Shen et al. [48] analyzed 603 bugs in three popular DL compilers and developed a fuzzer to test Apache TVM [7]. Jeon et al. [20] studied low GPU utilization of large-scale, multi-tenant GPU clusters for DL training. The authors pointed out that the root causes came from resource locality, gang scheduling, and job failures. They also suggested how to improve the future generation of cluster schedulers. Nevertheless, this research considers more about the cluster's GPU utilization, instead of that of the individual jobs. A related work by Cao et al. [5] characterized 238 performance bugs in TensorFlow and Keras programs collected from 225 Stack Overflow posts. We notice that some of the bugs (*e.g.*, those caused by improper hyperparameters) also exist in our work. However, because their study subjects are different from our industrial jobs on ITP, we find many different types of low-utilization issues arising in a cloud-based DL platform.

Some researches have also been proposed to improve the performance of deep learning programs and jobs. Nicolae et al. [30] proposed asynchronous model checkpointing to efficiently hide the serialization and I/O overhead and spread the workload among all the processes. Wu et al. [52] implemented EDL to support elastic model training, which can dynamically adjust the number of GPUs to achieve GPU efficiency. Rammer [26] is a novel DL compiler design that automatically optimizes the execution of DL jobs by "exploiting parallelism through inter- and intra- operator co-scheduling" [26]. Microsoft DeepSpeed [44] is an optimization library built atop PyTorch that achieves extreme-scale and efficient distributed model training. These works can resolve some types of the low-utilization issues in this study from specific aspects; however, based on our study results, we think that a holistic solution is desired to improve the overall GPU utilization of DL jobs.

7 CONCLUSION

In this paper, we present a comprehensive empirical study on low GPU utilization of deep learning jobs. We randomly select 400 real jobs from ITP and discover 852 low-utilization issues caused by the code logic of scripts and programs. Further, we manually identify the common root causes and fix solutions. Based on the findings, we propose possible research topics to improve the GPU utilization of jobs. We believe that our work provides valuable guidelines for the future development of deep learning programs to better utilize GPUs.

REFERENCES

[1] 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Amazon. 2022. Amazon SageMaker. https://aws.amazon.com/sagemaker.

[3] 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*.

[4] Samira Briongos, Pedro Malagón, José L. Risco, and José M. Moya. 2017. Building Accurate Models to Determine the Current CPU Utilization of a Host within a Virtual Machine Allocated on It. In *Proceedings of the Summer Simulation Multi-Conference* (Bellevue, Washington) *(SummerSim '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 33, 12 pages.

[5] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2021. Characterizing Performance Bugs in Deep Learning Systems. *CoRR* abs/2112.01771 (2021). arXiv:2112.01771 https://arxiv.org/abs/2112.01771

[6] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (jul 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] François Chollet et al. 2015. Keras. https://keras.io.

[9] 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, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[11] Google. 2022. Best Practices for Performance and Cost Optimization for Machine Learning. https://cloud.google.com/solutions/machine-learning/best-practices-for-ml-performance-cost.

[12] Google. 2022. Google Cloud AI. https://cloud.google.com/products/ai.

[13] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns. *CoRR* abs/1707.03750 (2017). arXiv:1707.03750 http://arxiv.org/abs/1707.03750

[14] Hugo Lewi Hammer, Anis yazidi, and Kyrre Begnum. 2016. Reliable Modeling of CPU Usage in an Office Worker Environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 480–483. https://doi.org/10.1145/2851613.2851872

[15] Xue Han, Daniel Carroll, and Tingting Yu. 2019. Reproducing performance bug reports in server applications: The researchers' experiences. *Journal of Systems and Software* 156 (2019), 268–282. https://doi.org/10.1016/j.jss.2019.06.100

[16] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2961111.2962602

[17] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 104, 15 pages. https://doi.org/10.1145/3458817.3476223

[18] 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[19] Deepak Janardhanan and Enda Barrett. 2017. CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. 55–60. https://doi.org/10.23919/ICITST.2017.8356346

[20] 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* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 947–960.

[21] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I* (Jeju, Korea (Republic of)). Springer-Verlag, Berlin, Heidelberg, 604–620. https://doi.org/10.1007/978-3-030-59410-7_40

[22] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[23] Jupyter. 2022. Project Jupyter. https://jupyter.org.

[24] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/3190508.3190552

[25] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE '14)*. Association for Computing Machinery, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[27] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).

[28] Microsoft. 2022. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning-service.

[29] Langston Nashold and Rayan Krishnan. 2020. Using LSTM and SARIMA Models to Forecast Cluster CPU Usage. *CoRR* abs/2007.08092 (2020). arXiv:2007.08092 https://arxiv.org/abs/2007.08092

[30] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 172–181. https://doi.org/10.1109/CCGrid49817.2020.00-76

[31] Adrian Nistor. 2014. Understanding, detecting, and repairing performance bugs.

[32] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering* (Florence, Italy) *(ICSE '15)*. IEEE Press, 902–912. https://doi.org/10.1109/ICSE.2015.100

[33] NVIDIA. 2022. NVIDIA Data Center GPU Manager (DCGM). https://developer.nvidia.com/dcgm. (2022).

[34] NVIDIA. 2022. The NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/dali.

[35] NVIDIA. 2022. NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu.

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

[37] 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*, Vol. 32. Curran Associates, Inc., 8024–8035. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[38] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

[39] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, and Amrish

Kumar. 2021. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3148–3161. https://doi.org/10.14778/3476311.3476390

[40] Friedrich Pukelsheim. 1994. The Three Sigma Rule. *The American Statistician* 48, 2 (1994), 88–91. https://doi.org/10.1080/00031305.1994.10476030 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00031305.1994.10476030

[41] Björn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.

[42] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf. (2019).

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[44] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[45] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale.* 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[46] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[48] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[49] TensorFlow. 2022. Get Started with TensorFlow Transform. https://www.tensorflow.org/tfx/transform/get_started.

[50] Neil C. Thompson, Kristjan H. Greenewald, Keeheon Lee, and Gabriel F. Manso. 2020. The Computational Limits of Deep Learning. *CoRR* abs/2007.05558 (2020). arXiv:2007.05558 https://arxiv.org/abs/2007.05558

[51] Thomas Wang, Simone Ferlin, and Marco Chiesa. 2021. Predicting CPU Usage for Proactive Autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 31–38. https://doi.org/10.1145/3437984.3458831

[52] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2022. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 144–158. https://doi.org/10.1109/TPDS.2021.3064966

[53] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*. USENIX Association, USA, 595–610.

[54] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[55] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, Republic of Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/3377811.3380362

[56] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020

[57] 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* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866