Profiling Deep Learning Workloads at Scale using Amazon SageMaker

Nathalie Rauschmayr[1], Sami Kama[1], Muhyun Kim[1], Miyoung Choi[1], Krishnaram Kenthapadi[2,*]
[1]Amazon Web Services, [2]Fiddler AI

ABSTRACT

With the rise of deep learning (DL), machine learning (ML) has become compute and data intensive, typically requiring multi-node multi-GPU clusters. As state-of-the-art models grow in size in the order of trillions of parameters, their computational complexity and cost also increase rapidly. Since 2012, the cost of deep learning doubled roughly every quarter, and this trend is likely to continue [3]. ML practitioners have to cope with common challenges of efficient resource utilization when training such large models. In this paper, we propose a new profiling tool that cross-correlates relevant system utilization metrics and framework operations. The tool supports profiling DL models at scale, identifies performance bottlenecks, and provides insights with recommendations. We deployed the profiling functionality as an add-on to Amazon SageMaker Debugger, a fully-managed service that leverages an on-the-fly analysis system (called rules) to automatically identify complex issues in DL training jobs. By presenting deployment results and customer case studies, we show that it enables users to identify and fix issues caused by inefficient hardware resource usage, thereby reducing training time and cost.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → **Software notations and tools**.

KEYWORDS

Amazon SageMaker, Performance Profiling, MLOps

ACM Reference Format:
Nathalie Rauschmayr[1], Sami Kama[1], Muhyun Kim[1], Miyoung Choi[1], Krishnaram Kenthapadi[2,*] . 2022. Profiling Deep Learning Workloads at Scale using Amazon SageMaker. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3534678.3539036

1 INTRODUCTION

Training state-of-the-art models requires substantial computational resources [26] and can take hours to days or even months [34]. Training deep neural networks is complex and requires ML practitioners to have expertise in multiple domains, such as large scale data-processing, data science, GPU programming, and distributed computing. A large-scale empirical study to identify the common challenges in developing DL applications shows that DL frameworks provide only limited debugging and profiling support [33]. As new DL models start having trillions of parameters partitioned to multiple hosts, the need to address the gap of profiling and optimizing computational performance is increasing.

We present a profiling functionality specific to training of DL models which we have deployed as an add-on to Amazon SageMaker Debugger [23], a fully-managed service that automatically captures data during model training and leverages a real-time analysis system (called rules) to identify issues. ML practitioners can reduce the amount of effort to set up profiling, get insights provided by the rules to resolve performance issues, and reduce training time and cost. The tool takes advantage of the specific resource utilization patterns of DL workloads. Our goal is not to propose a general-purpose profiling tool; instead, we focus on capturing only metrics relevant for detecting the most common performance problems in DL training. By cross-correlating these metrics, our profiling tool can provide actionable insights to users ranging from beginner ML practitioners to expert data scientists.

While many sophisticated profiling and monitoring tools are commonly used in high performance computing (e.g., [6, 9, 24, 25, 27]), these tools often require users to be proficient with performance engineering and do not leverage the utilization patterns specific to DL workloads. In contrast to general-purpose profilers, our profiler abstracts lower level details and looks instead at the model and the framework frontend to make insights more understandable. For example, time spent in a convolutional layer is more insightful for a data scientist than the time spent in every device-side activity that is associated with the convolutional layer and generated in the framework backend. Due to the data- and compute-intensive characteristics of DL workloads, they exhibit specific performance patterns that distinguish them from other types of workloads. Such key characteristics include large amounts of data (in the order of terabytes) that often needs pre-processing on the fly, the use of accelerators to speed up tensor calculations, and the need for distributed training setups. We discuss common performance bottlenecks arising from these characteristics in detail in Section 2.

The remainder of the paper is organised as follows: in Sections 2 and 3, we show the common DL workflow and present the design and implementation of a lightweight profiling capability of Amazon SageMaker Debugger (hereafter referred to as Debugger), including (1) Capturing high-level and low-level performance data; (2) Support for profiling large scale training jobs; (3) Integration with Amazon SageMaker as a scalable, secure, and fully managed service. In Section 4, we introduce new profiler built-in rules to

*Work done while at Amazon AWS AI.

KDD '22, August 14–18, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9385-0/22/08.
https://doi.org/10.1145/3534678.3539036

identify most common performance problems. In Sections 5 and 6, we present the deployment results and case studies.

We note that deploying the profiling functionality as an add-on to Debugger has enabled users to automatically identify a wide range of training issues and to reduce costs by improving resource utilization and choosing an optimal scale of GPU clusters.

2 THE DL TRAINING WORKFLOW

In this section, we discuss the most common performance problems in the DL training workflow, which motivate implementation of built-in rules that will be presented in §4.1 and the list of profiling metrics discussed in §3.2.

Data loading: DL models are usually trained on large datasets that are read from a local or a remote storage component. The file format is crucial for the I/O throughput. For instance, computer vision models are commonly trained on the ImageNet dataset which consists of more than 14 million images [7]. If those were stored as JPEG- or PNG-files, then this would significantly increase I/O latencies. The "small file problem" can be mitigated by transforming the dataset into larger shards [2]. All major DL frameworks support compressed binary formats (e.g. MXNet's RecordIO, TensorFlow's TFRecord). Such file formats improve I/O throughput by reading data continuously from storage and reducing the bytes to load through compression. To avoid further I/O bottlenecks, data should be pre-fetched and loaded asynchronously.

Data Preprocessing: DL model training usually includes a complex multi-stage processing pipeline [13]. Transformations such as data augmentation are applied to avoid overfitting and improve generalization. For instance, in case of computer vision models, images are randomly rotated, cropped, and flipped so that the model receives different input data in each epoch. These operations are commonly performed on the CPU as the model training is in progress. Certain operations can be compute intensive and can lead to a CPU bottleneck wherein the GPU is waiting for the next data batch that is still being processed on the CPU. As this can limit performance and scalability, it is desirable to execute preprocessing operations asynchronously within multiple worker processes [30] or on GPU (e.g., NVIDIA's Dataloading Library, DALI [13]).

Training: A DL model defines a computational graph whose nodes are operators. Those operators usually run asynchronously on the GPUs for faster execution because they can utilize thousands of processing units in parallel. During training, inter-GPU communications and synchronizations can impact overall performance. Synchronizations are often triggered by host-device transfers such as saving model checkpoint files, computing and saving metrics, and copying new data batches. Synchronization also happens when an operator is executed on a CPU and its output is needed as an input for a GPU operator. When running jobs with distributed training, gradient accumulation and parameter updates may require a synchronization across all GPUs. If these happen too frequently, it may lead to GPU under-utilization as synchronization blocks the execution of other kernels.

GPUs run single instructions on multiple threads and data, so their utilization is directly related to the amount of data they process in parallel. Consequently, the size of the model and data batches directly impacts the GPU usage [34]. If the batch size is too small, it leads to GPU under-utilization. If the batch size too large, it might result in GPU running out of memory.

Assuming the model training is GPU-bound, inspecting the time spent on executing each kernel might help speed up training. For instance, changing data layout of tensors, operator fusion, reducing memory transfers, and/or training with mixed precision can yield further performance improvements.

3 PROFILER SYSTEM DESIGN AND ARCHITECTURE

In this section, we provide an architectural overview of the profiling functionality integrated to SageMaker Debugger.

3.1 Components Overview

SageMaker Debugger [23] is a fully-managed service and extended to automatically capture performance metrics in addition to tensor data during model training. The new Debugger profiling functionality leverages rules to identify computational performance problems and to automate training job termination by defining stopping conditions. SageMaker Debugger consists of the following two key components:

(1) **smdebug**: This is the core library that records and loads performance metrics (§3.2). The profiling tool provides the basic functionalities to read, query, and filter data by steps, time stamps, training phases, and names of performance metrics. We also extended the library to support new profiler rules (§4.1). This includes capturing performance metrics and tools to monitor and visualize the profiler data in real-time. We provide the library as an open-source library on PyPI[1] and GitHub[2]. Users can use the profiling capabilities in addition to the debugging capabilities when they run training jobs either within or outside Amazon SageMaker. Refer to [23] for an in depth description of the smdebug API.

(2) **Rules**: As part of SageMaker, users can run a set of predefined rules that run asynchronously to the training job and analyze the collected data. Debugger uses the smdebug library to record performance metrics from the training job and upload to Amazon Simple Storage Service (S3). Rules are executed in a dedicated Docker container where they fetch the performance metrics data and check for certain conditions.

3.2 Performance Metrics

Debugger's profiling functionality collects performance metrics that are gathered from different data sources. This allows users to get a comprehensive global view of the application performance without having to enable multiple generic profilers. Each metric is associated with time stamps, host, and process IDs. The profiling functionality also provides an abstraction layer that cross-correlates metrics and connects low-level metrics with high-level recommendations.

3.2.1 System metrics. Debugger's profiling functionality runs a profiler agent on every training instance, to provide a comprehensive view on the system's overall performance. It collects the utilization per CPU core to identify CPU bound processes that may

[1] https://pypi.org/project/smdebug

[2] https://github.com/awslabs/sagemaker-debugger

lead to GPU starvation. It also measures the utilization per GPU and its memory usage. Several disk related metrics are captured such as I/O operations per second that shows if the training job is hitting the maximum speed of the attached elastic block store volume (EBS[3]). The I/O wait time per core indicates the time a process was waiting for I/O requests being resolved. This metric can be used to identify I/O related bottlenecks that can lead to CPUs and GPUs being idle. Read-/write-throughput indicates how many bytes were read/written in a given time period. This can be used to identify issues related to file formats. For instance reading millions of small files would lead to a low read-throughput. The profiler agent also captures network metrics such as received/sent bytes for the training container, which can indicate if network bandwidth is saturating in distributed training setups. Users can specify different sampling intervals ranging from 100ms to 60s. The profiler agent provides a more granular view on the system than what is provided by default through Amazon CloudWatch[4], a monitoring and observability service for AWS resources and applications.

3.2.2 Framework metrics. Further, training metrics are recorded such as the start of dataloading, data preprocessing, synchronizations, forward and backward passes, and time spent in CPU- and GPU-operators. We refer to these events as framework metrics. The data is associated with meta-information such as start- and end-timestamps, process ID, and GPU ID. Debugger's profiling add-on also supports python profiling of the training script which helps users to track function calls and identify hotspots. These statistics are collected per training step so that users can inspect function calls before, during, and after each step.

3.2.3 Profiling overhead. The key technical challenge is to find a trade-off between the amount of data collected and the profiling overhead. A large profiling overhead implies that long-running training jobs incur non-negligible additional cost. Furthermore, performance profiling might impact in a way that existing bottlenecks might disappear or new bottlenecks might appear; this consequently hides the actual root causes. To address this challenge, we provide different levels of profiling which can be updated any time while training is in progress:

(1) By default, the system metrics are collected at 500ms interval. This causes a negligible overhead of less than 2%.
(2) Detailed profiling includes GPU-kernel information. This data is collected through NVIDIA CUPTI and is usually only collected for a few steps (§6.3).
(3) Users can enable higher-level framework metrics such as time spent on data loading, preprocessing, and GPU synchronizations. These metrics are directly captured in the framework backend.

A user can enable or disable profiling at any point in the training and update its configuration accordingly. Data is collected for every training instance, so it supports distributed training setups. Debugger provides specific rules that correlate those metrics to identify root causes of performance issues. A profiling report is auto-generated, which summarizes the key findings, and provides recommendations of next steps for mitigating the performance issues (§4.1). Users can use the smdebug library and SageMaker Studio[5] to visualize and analyze the performance metrics in real-time (§4.2).

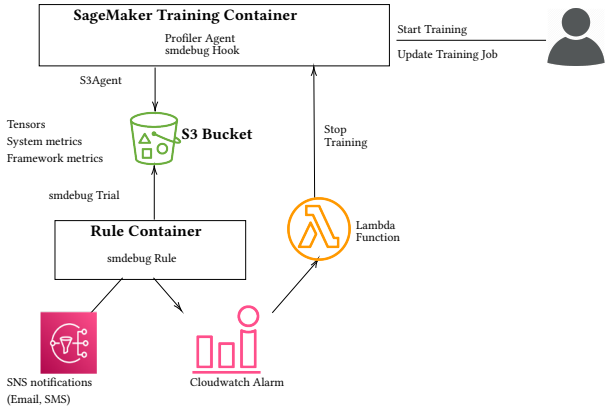

Figure 1: An architecture diagram of SageMaker Debugger workflow [23]

3.3 Integration with Amazon SageMaker

As shown in Fig. 1, a SageMaker training job runs in a training container and Debugger emits tensors and profiler data. SageMaker asynchronously uploads these data to an Amazon S3 bucket specified by the customer. Built-in rules run on separate compute instances in a rule container and do not interfere with the training itself. Rules emit metrics to Amazon CloudWatch to indicate whether an issue was found or not. Users can then set up a CloudWatch alarm and Lambda[6] function to auto-terminate training jobs after a rule has triggered. Rules also support actions such as Email or SMS notification, that is provided via Amazon Simple Notification Service (SNS)[7]. Users can also provide their own custom rules. Decoupling training and rule analysis enables the execution of an arbitrary number of rules and compute-intensive analysis at scale.

4 PROFILER DATA ANALYSIS

4.1 Built-in Rules

We introduced a new set of built-in rules for Debugger that automatically identifies performance related issues. The profiler rules have been derived from a variety of use-cases and a few of them will be discussed in §5. Next we provide an overview of these rules.

Low GPU usage: Deep neural networks are usually trained on GPUs since they offer a higher computational throughput compared to CPUs. GPUs contribute the most to the overall cloud usage bill of DL workloads and a higher GPU utilization translates into better cost efficiency. We have implemented built-in rules that monitor the usage per GPU across time and compute relevant statistics per time window. The rules trigger if there is significant under-utilization or high fluctuations. One of the most common factors leading to GPU under-utilization is too small a batch size as reported in [34]. If the batch size is too large, the GPU will run out of memory or existing CPU bottlenecks can become worse. We implemented rules that measure the 95th percentile of overall CPU-, GPU-utilization, and

[3] https://aws.amazon.com/ebs/
[4] https://aws.amazon.com/cloudwatch/
[5] https://aws.amazon.com/sagemaker/studio/
[6] https://aws.amazon.com/lambda/
[7] https://aws.amazon.com/sns/

the GPU memory footprint to evaluate if batch size can be further increased.

Workload imbalance: For distributed training, the application or the system itself can cause a workload imbalance which may lead to under-utilization [17]. Techniques such as synchronous stochastic gradient descent require synchronizations in order to gather gradients from all GPUs at each training step. For example, depending on the content of a data batch, step time can be different, so this may lead to workers finishing a training step faster and then being blocked by synchronization. As described in [22] poor coordination between the data-processing components of the DL training workflow can lead to resource contention, which may lead to a GPU waiting longer therefore resulting in poor utilization.

We introduced a rule to compute a workload histogram per GPU that measures how often the GPU has been utilized at 0%, 5%, . . ., 100%. The histogram is updated every minute and the rule then measures the distance between them to identify imbalanced workloads.

CPU Bottlenecks: As discussed in §2, the data augmentation can lead to CPU bottleneck problems. A CPU bottleneck refers to the situation where a GPU waits for the next batch, while the CPU is busy processing the data. To identify this issue, Debugger measures the utilization per CPU core and per GPU. If utilization on one or multiple CPU cores is above a pre-defined threshold and at the same time GPU-utilization is low, then it is recorded as a CPU bottleneck.

I/O Bottlenecks: Similar to CPU bottlenecks, I/O can become a limiting factor when now both GPU and CPU wait for data to arrive from disk. As described in §2, this can happen if the data loading process is not optimized or uses an inefficient file format. I/O bottlenecks can also be caused by too frequent model checkpointing, which impacts GPU utilization as data needs to be transferred to CPU from where it is written to disk. Debugger measures the I/O wait time per CPU core to identify I/O bound processes. If GPU utilization is also low, it is considered as an I/O bottleneck.

Data loading: DL models are usually trained on large datasets, and efficient data loading is essential for optimal resource utilization [30]. Debugger collects key metrics such as the start time and the end time of each data loading event, corresponding thread and process IDs, and determines whether data was loaded into page-locked memory which speeds up the host to device transfer. Creating multiple background worker processes to prefetch batches has been shown to increase performance [30]. We integrated a built-in rule that measures how many dataloader threads (processes) are active per time unit and compares them with the number of available CPU cores. Too many threads could cause a lot of context switches and hurt the overall performance. Too few threads may lead to GPU under-utilization, since data is not loaded and processed fast enough.

Step Duration: Debugger records the step duration which is the combined time required for a forward pass and a backward pass. Ideally, we expect the step durations to not vary too much and follow a normal distribution. Significant outliers may indicate an underlying performance issue: for instance in the case of Keras, a user may have configured a custom callback that periodically saves model checkpoints, which could increase the step time. A built-in rule records the step durations as training is in progress and if at any given point in time a step exceeds five times the standard deviation, a violation is recorded. The rule then correlates this timestamp with events recorded in the framework and system to identify the possible root cause. For example, in the case of model checkpoints, outliers would correlate with a higher I/O usage.

Training Initialization Time: We define initialization as the time from job start to the beginning of the training loop and we capture the time spent in initialization, training loop, and finalization. In most cases, GPUs are only used during the training loop. Long initialization time can have different root-causes and with the data obtained from Python profiling, users can identify the most expensive function calls. A common issue is the pre-processing of large datasets as part of the training job. Users can avoid this problem by decoupling data-processing from training, say, using Spark clusters.

Profiler report: We added a rule to Debugger - profiler report rule - that runs all aforementioned rules and generates an output report with further insights, visualizations, and recommendations of next steps.

4.2 Interactive Visualizations and Analysis

An interactive analysis of the profiler data helps discover more complex patterns. The open-source smdebug library and SageMaker Studio are extended with new real-time visualization tools that allow users to navigate between inter-correlated performance metrics. Different types of visualizations are provided to address the needs of different users. For instance, less experienced users may find the profiling report to be most useful, as it provides a high level summary of detected performance bugs as well as recommendations. More advanced users may prefer to dive deep in the performance data by inspecting timeline-charts. As long-running and large scale training jobs can easily generate millions of datapoints, a key challenge is how to efficiently handle the massive data while keeping visualizations interactive and responsive. To address this, data is down-sampled by default and fine-grained metrics are displayed only when users zoom in. Debugger's profiling feature provides the following types of visualizations:

(1) It provides system utilization statistics aggregated per worker node and indicates whether resources are optimally utilized, under-utilized, or over-utilized.
(2) Users can obtain a granular view by inspecting utilization per CPU core and GPU. It provides time series charts (Fig. 3), step timeline charts (Fig. 4) and heatmap visualizations (Fig. 7). Users can select a target range of system metrics and correlate these with framework metrics.
(3) The merged timeline view provides the most comprehensive view as it displays all collected metrics in a single timeline. Users can inspect single events in a thread or GPU stream and inspect system metrics in the same plot (Fig. 2).

5 DEPLOYMENT RESULTS

The profiling capabilities of SageMaker Debugger have helped in several use-cases to improve resource utilization.

Multi-GPU BERT model training. A user attempted to train a PyTorch BERT model [8] on multiple GPUs and reported slow training. By enabling the profiling capabilities of Debugger, the

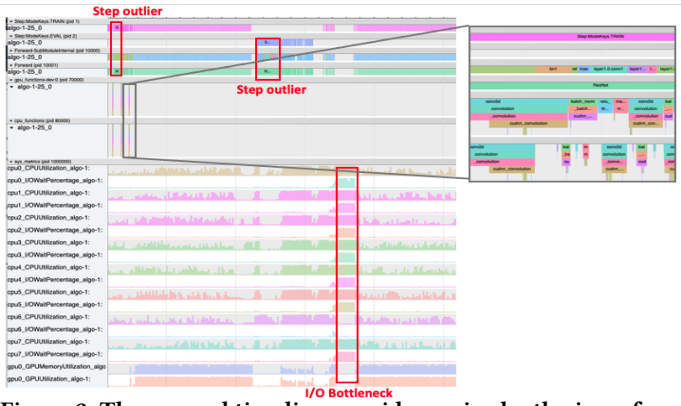

Figure 2: The merged timeline provides an in-depth view of system and framework metrics.

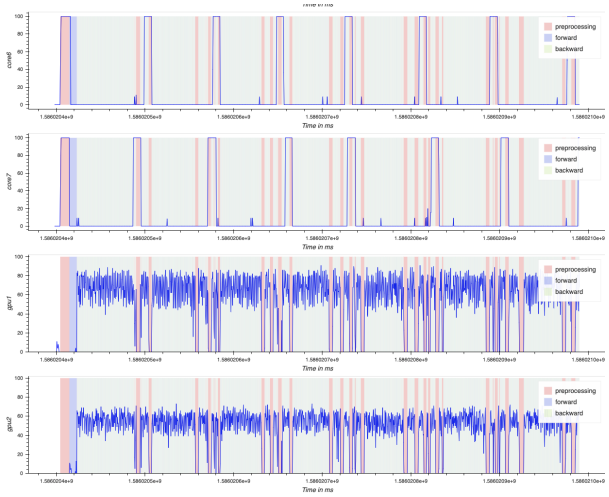

Figure 3: Correlation of CPU and GPU utilization rates (blue lines) and framework metrics (vertical bars) indicate CPU-bottlenecks caused by a data pipeline issue.

user was able to detect two major performance issues. First, the training job suffered from a workload imbalance where the primary GPU was consistently utilized at about 80% while all others (worker GPUs) only reached a peak utilization of 60%. The root cause was the distributed training strategy. The user had initially used PyTorch's DataParallel strategy instead of DistributedDataParallel [18]. The key difference is that PyTorch's DataParallel strategy collects the outputs to one GPU to calculate the loss instead of computing loss independently in each GPU. This led to an imbalance causing a higher usage and memory utilization on the primary GPU. The second issue was that GPU utilization frequently suffered from CPU bottlenecks. Overlaying system metrics with framework metrics uncovered a data pipeline issue; the CPU bottlenecks are correlated with data pre-processing phases. Fig. 3 shows the timeseries charts for the profiled training job. For simplicity, it only shows utilization for CPU cores 6-7 and GPU 1-2 (blue lines). GPU 2 has, on average, a lower utilization than GPU 1 and both suffer from CPU bottlenecks caused by data preprocessing issues on CPU side.

Increasing the number of dataloaders and switching from DataParallel to DistributedDataParallel improved the overall utilization rate by 30%, approximately.

Stalled training. In another use-case, a DL model was trained on multiple nodes. One of the dataloaders crashed when reading a corrupted file. The training still continued, but would fail during finalization as the main process was not able to join the sub-processes which resulted in a deadlock: the instance went idle and could not be shutdown. In this scenario, multiple profiler rules helped spot this issue. After the dataloader crashed, the average step time is increased and triggered the *step outlier* rule. The *GPU utilization* rule triggered once the utilization dropped to 0% causing both 5th and 95th percentiles to fall below the predefined thresholds.

Operator on CPU instead of GPU. Inspite of an optimized data pipeline and increased batch size, a user's training job suffered from high fluctuations in GPU usage. The detailed framework metrics and merged timeline view revealed that the issue was caused by the loss calculation running on the CPU instead of the GPU. This meant that frequent synchronizations and host-device transfers prevented the GPU from achieving an optimal utilization rate. The user was able to improve the GPU utilization by nearly 15% by replacing the operator.

Large scale distributed training. In another use-case, DL models were trained at large scale on a cluster of 128 or more GPUs. The user occasionally observed that a GPU would run significantly slower than others due to a hardware problem or slow communication links. The straggler GPU [31] slows down the entire training and the user would need to manually initiate a restart. With Debugger the user could easily automate this action; the *step outlier* and *workload imbalance* rules could detect this problem and trigger an alarm.

Overall system overhead. Image augmentation is a commonly applied technique for training deep neural networks for vision. Different APIs can be used to perform the augmentation, for instance, the deep learning framework, NumPy API, or image libraries such as OpenCV [5]. OpenCV creates multiple internal threads for fast image processing. This might cause problems in multi-GPU training jobs where separate processes are launched per GPU and each training process creates multiple dataloaders. This misconfiguration can overload the whole system because the number of threads exceeds the total number of CPU cores by several multiples. This causes frequent context switches and adds system overhead. It was apparent in Debugger's heatmap visualization showing all CPUs were 100% utilized during the entire training time, while this symptom was not identified with other framework profilers as they do not measure system metrics. By turning off multi-threaded feature of OpenCV, the system overhead is lifted and the CPU utilization dropped below 50% and improved processing and training speed.

6 CASE STUDY

The previous section showed how Debugger's new profiling capabilities helped improve computational performance through different use-cases. However, disclosing the profiler reports and detailed results from the use-cases is limited due to restrictions around privacy-sensitive datasets. In this section, we showcase an end-to-end profiling example using publicly available datasets. More

specifically, we use a training script based on a tutorial from the PyTorch website which fine-tunes a pre-trained Mask R-CNN model for instance segmentation on the Penn-Fudan dataset [1]. The goal is to show-case how different metrics, visualizations, and high level abstractions help to identify performance bugs.

6.1 Debugger Profiling Report

In the initial setup, we train the model on a single GPU ml.p3.2xlarge instance that provides one NVIDIA V100 Tensor Core GPU and 8 CPU cores. Tab. 1 shows some of the statistics provided by Debugger's profiling report. The training ran for 2478 seconds and the median GPU utilization was only 4% and 95th percentile was 100%, indicating GPU under-utilization.

Debugger also detected that the training job suffered from CPU bottlenecks 45% of the time and I/O bottlenecks 9% of the time. We ran the same training job on a ml.p3.8xlarge instance with 4 GPUs using PyTorch DistributedDataParallel. Debugger revealed that these bottlenecks are increased to 65% and 12% respectively. These bottlenecks could not be identified with other framework specific profilers as they do not measure the system level metrics.

The training time decreased from 2478 to 1396 seconds. However, using the 4-GPU instance, the training speed is only improved by less than 50%. This is due to performance bottlenecks that often limit the scaling to multiple GPUs, so before considering multi-GPU training we need to optimize the single GPU training first.

The profiling report (Tab. 1) also reveals a problem with the training loop; roughly 25% of the time is spent on running validation steps and 27% on training steps. This means that nearly 48% of the training loop is spent on the preprocessing or postprocessing of the model's forward pass or backward pass, respectively. We can use Debugger's interactive analysis tools to dive deeper into this problem (see §6.2). The framework metrics collected by Debugger

	p3.2xlarge	p3.8xlarge
Training time	2478s	1396s
GPU utilization p50 (median)	4%	0%
GPU utilization p95	100%	82%
CPU Bottlenecks	45%	65%
I/O Bottlenecks	9%	12%
EVAL Steps	25%	39%
TRAIN Steps	27%	42%

Table 1: Summary of statistics from the profiling report.

reveal that one of most expensive GPU kernel is the NMS operator. Non-maximum-suppression (NMS) is a post-processing step in object detection models to select the most appropriate bounding box for a given object. The Mask R-CNN model [14] consists of a region proposal network which creates proposals to be fed into the object detection model. Users can specify pre- and post-NMS thresholds that specify the number of proposals to keep. This can increase accuracy but also computational cost.

6.2 Interactive Analysis

6.2.1 Analyzing step durations. As described in §4.2 Debugger provides tools to analyze and visualize the profiler data. The step duration timeline chart (Fig. 4) shows the step duration on the y-axis and the steps that have been measured throughout the training on the x-axis. In this training job, the chart reveals patterns where each of them represents one training epoch. Further analysis shows that validation steps (EVAL) take twice longer than training steps (1.2 seconds versus 0.6 seconds). Fig. 4 indicates that the validation phase accounts for nearly half the time in an epoch which is undesirable, because the training loop spends as much time in evaluating the model than actually training it. Fig. 5 indicates that GPU

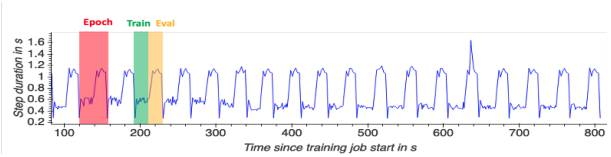

Figure 4: Step timeline chart reveals that validation phase is taking as much time as training phase.

utilization is relatively high during training steps, but consistently low during the validation phase and after the training phase ended. We use Debugger to get more detailed metrics for the validation phase and the time after training phase to better understand the GPU under-utilization. Large gaps between validation steps (see Fig. 5) indicate that a significant amount of time is spent in either the pre- or postprocessing of the model forward pass. The merged timeline reveals that CPU utilization on one of the cores is high during the validation phase. Neither framework specific profilers nor general-purpose profiling tools could easily spot this problem. The latter would only notice system metrics such as GPU utilization, but could not associate it with the step metric. On the other hand, framework profilers do not measure system specific metrics.

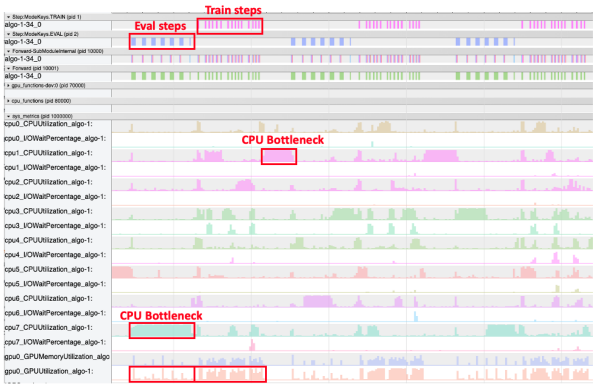

Figure 5: Merged timeline indicates CPU bottlenecks.

6.2.2 Analyzing function hotspots. To identify the root cause, we leverage Debugger's Python profiling capabilities to investigate the time spent per function call during the validation phase. Debugger records stack-traces per step and before/after each step. Fig. 6 shows that the function *evaluate* takes roughly 1.2 seconds out of which most time is spent in the function *encode* and *Tensor.to*. The function *encode* takes as input the predicted image segmentation mask and computes its run-length encoding. Those encoded masks are then compared with the target segmentation masks. The function *Tensor.to* is called when the predicted image masks are copied from GPU to CPU, where they are passed into the evaluation function.

This host-to-device transfer triggers synchronization causing GPU utilization to drop. Both functions present a performance bottleneck which is causing high CPU usage and delaying the execution of the next validation step, which explains the large gaps between validation steps seen in Fig. 5.

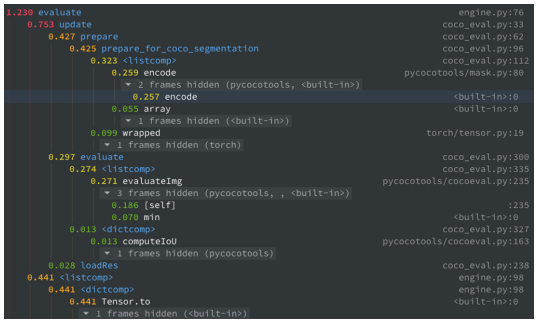

Figure 6: Time-consuming functions during validation phase.

We perform the same analysis for the training phase to identify potential bottlenecks. The stack-traces indicate that certain steps spent a large amount of time in the function *FloatStorage.write_file* which is called whenever tensors are saved to disk. It turns out that the user had misconfigured the interval at which model checkpoints were saved which resulted in a performance issue. These performance bottlenecks are also apparent in the heatmap representation (Fig. 7): the y-axis of the heatmap shows the different system metrics and the x-axis the utilization across time, where yellow indicates 100% utilization and purple 0%. We can see in Fig. 7 that I/O wait time is frequently reaching 100% during initialization: this is expected as the entire training dataset is first downloaded from Amazon S3 and then read into CPU RAM. The heatmap reveals that I/O wait time often reaches 100% during the training loop which is caused by frequent model checkpointing.

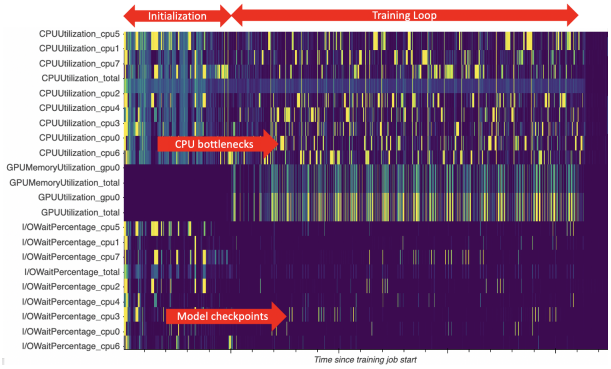

Figure 7: Heatmap reveals I/O- and CPU-bottlenecks (yellow indicates 100% utilization and purple 0% utilization).

6.2.3 Analyzing GPU kernels. To understand why validation steps take twice longer, we look more closely at the merged timeline (Fig. 8) which shows the detailed profiling metrics specifically for train and validation steps. The green horizontal bars in Fig. 8 indicate the time spent in data transformation operations: it appears to take significantly more time for validation steps compared to training steps. Further during validation phase, the model forward pass is followed by a post-transform function and data copy. The post-transform resizes the predicted image segmentation masks to the size of the original input image. This is done to compute the evaluation metrics. Since each image can have a different size, it is difficult to process them in a batch. As a result, batches are unstacked and each image is processed independently on the GPU. This explains the lower GPU utilization during validation steps. At the end of the step, data is copied from GPU to CPU, where the data is passed into the *evaluate* function.

Figure 8: Merged timeline of train steps (left) and eval steps (right).

With the profiling insights obtained from Debugger, we now optimize the training script: we reduce the frequency of model checkpoints. Further we execute the run-length encoding of validation images in parallel on the CPU. These are just a few minor modifications, but they decrease the training time by 20% for the single GPU training job.

6.3 Profiling at Scale

We demonstrated the value of Debugger's profiling capabilities with the example of a single-GPU training job in §6.2. As highlighted in the introduction, these profiling capabilities are intended for DL workloads that run at large scale in the cloud where performance is critical for cost reduction. To evaluate the performance impact, we run the Mask R-CNN model training with SageMaker's Distributed Data Parallel Library, which is a distributed data parallel training framework for PyTorch and TensorFlow [28]. The training runs for 3000 steps and with the following two configurations: (1) system metrics sampled at 500ms; and (2) system metrics sampled at 500ms and framework metrics captured for 2 training steps.

The model is trained on the COCO dataset [19], which is loaded using Lustre [4], a high-performance file-system enabling fast data access. Tab. 2 shows that the overhead introduced by collecting system metrics is negligible compared to the baseline where no metrics are collected. Detailed profiling for framework metrics causes a slowdown between 2.4% and 7.4% in our benchmark.

Instances	GPUs	system metrics	system + framework metrics
1	8	0.4%	6.9%
2	16	0.3%	7.4%
4	32	0%	4.0%
8	64	0.1%	4.7%
16	128	1.9%	2.4%

Table 2: Performance impact of the profiling functionality.

We now use the data collected from a distributed training job that ran on 8 nodes with 8 GPUs each. Metrics can be aggregated by their

type and node ID which helps to find common problems in large scale distributed training jobs. For instance, the straggler problem occurs when one node slows down the entire training cluster [31]. With a heatmap visualization, users can then identify the specific timestamps at which the overall utilization on the training cluster changed. Fig. 9 shows the overall utilization per worker node, and after half the training loop overall usage significantly changed for a certain period of time.

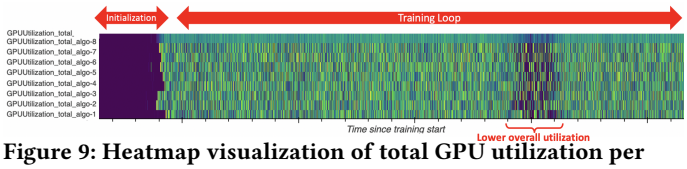

Figure 9: Heatmap visualization of total GPU utilization per worker node (yellow 100%, purple 0% utilization).

We now use a more detailed view to identify the utilization per GPU during this time period. Fig. 10 reveals that some GPUs suddenly run at 100% utilization while at the same time there is at least one GPU per node whose utilization drops to 0. This may indicate a data pipeline issue where fast events drain the pipeline and slow events are staggering.

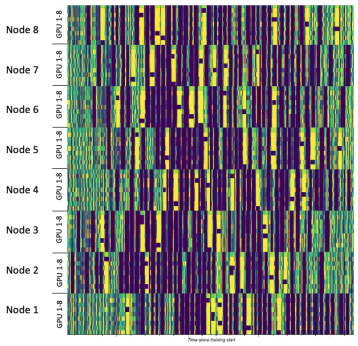

Figure 10: Heatmap visualization of utilization per GPU (yellow 100%, purple 0% utilization).

Thus, we further showed the value of Debugger's profiling functionality in the case of a single GPU as well as a large scale distributed training job. With the associated meta-data, users can easily aggregate and slice the data or obtain a more granular view on specific events and time ranges.

7 RELATED WORK

Most DL frameworks provide a concise high-level API that allows users to easily define model and training loop. This code is then translated by the backend into optimized code that is executed asynchronously by concurrent GPU operations. The gap between frontend and underlying execution can lead to the problem that performance issues may not always be obvious. To address this problem, several tools have been proposed and implemented by industrial leaders in the field as well as by researchers.

Authors of [32] showcase Skyline, an interactive tool for deep neural network training that provides performance profiling, visualization, and debugging. In particular, they show that the performance of an iteration is predictable with respect to its batch size. Skyline uses metrics such as training throughput, operation run times, and memory allocated for the model, to predict an optimal batch size. DeepProf [12] is a tool that summarizes GPU traces from TensorFlow programs and creates a performance analysis graph to pinpoint potential performance issues. While these tools provide very valuable insights, they focus on only a few metrics such as training throughput or GPU traces. In contrast, Debugger captures substantially more system metrics which can provide a more comprehensive view of the system.

TensorFlow profiler [11] is a tool to track the performance of TensorFlow models. It captures operation execution time and their dependencies. It is integrated in TensorBoard, where it provides a rich set of interactive visualizations and recommendations. PyTorch profiler [29] offers similar functionalities: users can collect execution times, input shapes, and stack traces. They can export the performance data as timeline file to view it in Chrome trace viewer [10] or retrieve aggregated statistics. Both tools provide valuable insights during model training, but they collect information only on the processes they are activated in and are unaware of other processes running in the host. Certain use cases require a broader view of the system, for instance, to identify I/O and network bottlenecks.

NVIDIA offers a wide range of profiling tools. NVProf is a visual profiler that collects device-side statistics such as memory transfers, kernel launches, and other function calls [21]. It creates a unified CPU and GPU timeline, and also monitors GPU power, thermal, and clock values. NVIDIA's Nsight Systems is a system-wide performance analysis tool that monitors CPU threads, GPU traces, and memory bandwidth [15]. It also leverages MPI traces, library API injections, program counter sampling, NVLink statistics, and a unified memory profiling. It can significantly slowdown the training. Both NVProf and Nsight require ownership of the process and are not suitable for multi-host distributed profiling. The NVIDIA Tools Extension library (NVTX) is is a platform agnostic API that is used to annotate source code, events, and code ranges [16]. CUDA Profiling Tools Interface (CUPTI) is a library used to create profiling and tracing tools [20], and is used by TensorFlow and PyTorch profilers.

The aforementioned tools give deep insights into either the system or the framework. Framework profilers such as TensorFlow and PyTorch profilers provide a macro-level overview of the performance such as time spent in each framework operation or function. However they do not provide a detailed view of the system or lower level information. Thus, framework profilers have less detailed information about the global picture, and cannot identify system bottlenecks as well as bottlenecks caused by third party libraries.

In contrast to stand-alone profilers, Debugger's profiling feature is offered as a fully managed service as part of Amazon SageMaker, wherein several system tasks are handled automatically without the need for the user to intervene. For instance, profiling often generates a large amount of data and puts a training job at the risk of running out of disk space. This problem is taken care of by Debugger's profiling feature which automatically applies file rotation and only keeps the most recent data on local disk. Further Debugger automatically annotates the training script, enables/disables profiling, uploads data, and performs the correlation and analysis in real-time. Debugger also provides tools for data collection, analysis, and visualization. By allowing users to update profiling configurations as training is in progress, the overhead can be kept minimal

because data will only be collected if requested. Further Debugger provides the concept of rules that allows users to automatically stop training jobs if resources are not optimally utilized.

8 CONCLUSION

Motivated by the challenges faced by ML practitioners when developing DL applications and the lack of sufficient profiling support, we presented a profiling tool specific to the DL workflow. We described the design and architecture of the profiler system, and how it integrates with Amazon SageMaker as a lightweight fully-managed service. We discussed the profiler algorithms to detect most common performance issues in DL model training in real-time. Our deployment results and customer case studies demonstrate that the profiler functionality is helpful for users to identify and fix performance bugs and increase resource utilization for model training, thereby saving training time and cost. Profiling DL applications is an active research area that has become increasingly important as DL models grow in size and complexity. While our work focused on automatically identifying performance bugs, future work could focus on anticipating performance issues and taking preemptive actions for recovery from these issues.

ACKNOWLEDGMENTS

The profiling functionality of SageMaker Debugger and the respective analysis tools have been developed and designed with the help of several members from the SageMaker team, most notably Vikas Kumar, Andrea Olgiati, Ishaaq Chandy, Satadal Bhattacharjee, Zhihan Li, Sifei Li, Jiacheng Gu, Connor Goggins, Yang Shi, Nihal Harish, Vandana Kannan, Amol Lele, Anirudh Acharya, Neelesh Dodda, Lakshmi Ramakrishnan, Kohen Chia, Balajee Nagarajan, Lu Huang, Tyler Hill, and Kevin Haas. We thank Isaac Privitera and Corey Barrett for insightful feedback and discussions. We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2022. Penn-Fudan Database for Pedestrian Detection and Segmentation. https://www.cis.upenn.edu/ jshi/ped_html/.
[2] Alex Aizman, Gavin Maltby, and Thomas Breuel. 2019. High Performance I/O For Large Scale Deep Learning. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 5965–5967.
[3] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. 2018. AI and Compute. https://openai.com/blog/ai-and-compute.
[4] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).
[5] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. 2012. A brief introduction to OpenCV. In *2012 proceedings of the 35th international convention MIPRO*. IEEE, 1725–1730.
[6] Arnaldo Carvalho de Melo. 2009. Performance counters on Linux. In *Linux Plumbers Conference*, Vol. 118.
[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[9] Kaniz Fatema, Vincent C Emeakaroha, Philip D Healy, John P Morrison, and Theo Lynn. 2014. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *J. Parallel and Distrib. Comput.* 74, 10 (2014), 2918–2933.
[10] Google. 2021. Chrome Frame Viewer Overview and Getting Started. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/frame-viewer.
[11] Google AI. 2021. Optimize TensorFlow performance using the Profiler. https://www.tensorflow.org/guide/profiler.
[12] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns. *arXiv preprint arXiv:1707.03750* (2017).
[13] Joaquin Anton Guirao, Krzysztof Lecki, Janusz Lisiecki, Serge Panev, Michał Szołucha, Albert Wolant, and Michał Zientkiewicz. 2019. Fast AI Data Preprocessing with NVIDIA DALI. https://developer.nvidia.com/blog/fast-ai-data-preprocessing-with-nvidia-dali/.
[14] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*. 2961–2969.
[15] Daniel Horowitz. 2018. Nsight Systems Exposes New GPU Optimization Opportunities. https://developer.nvidia.com/blog/nsight-systems-exposes-gpu-optimization/.
[16] Jiri Kraus. 2013. CUDA Pro Tip: Generate Custom Application Profile Timelines with NVTX. https://developer.nvidia.com/blog/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/.
[17] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 45–61.
[18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch Distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).
[19] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft COCO: Common objects in context. In *European conference on computer vision*. Springer, 740–755.
[20] NVIDIA. 2020. CUPTI. https://docs.nvidia.com/cupti/Cupti/index.html.
[21] NVIDIA. 2021. nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.htmlnvprof-overview.
[22] Sarunya Pumma, Daniele Buono, Fabio Checconi, Xinyu Que, and Wu-chun Feng. 2020. Alleviating load imbalance in data processing for large-scale deep learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 262–271.
[23] Nathalie Rauschmayr, Vikas Kumar, Rahul Huilgol, Andrea Olgiati, Satadal Bhattacharjee, Nihal Harish, Vandana Kannan, Amol Lele, Anirudh Acharya, Jared Nielsen, Lakshmi Ramakrishnan, Ishaaq Chandy, Ishan Bhatt, Zhihan Li, Kohen Chia, Neelesh Doddaand, Jiacheng Gu, Miyoung Choi, Balajee Nagarajan, Jeffrey Geevarghes, Denis Davydenko, Sifei Li, Lu Huang, Edward Kim, Tyler Hill, and Krishnaram Kenthapadi. 2021. Amazon SageMaker Debugger: A System for Real-time Insights into Machine Learning Model Training. In *MLSys*.
[24] James Reinders. 2005. VTune performance analyzer essentials. *Intel Press* (2005).
[25] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.
[26] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 13693–13696.
[27] Nathan Tallent, John Mellor-Crummey, Laksono Adhianto, Michael Fagan, and Mark Krentel. 2008. HPCToolkit: Performance tools for scientific computing. In *Journal of Physics: Conference Series*, Vol. 125. IOP Publishing.
[28] Indu Thangakrishnan, Derya Cavdar, Can Karakus, Piyush Ghai, Yauheni Selivonchyk, and Cory Pruce. 2020. Herring: Rethinking the parameter server at scale for the cloud. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA*. 1–13.
[29] The PyTorch team. 2021. Automatic differentiation package - torch.autograd. https://pytorch.org/docs/stable/autograd.html.
[30] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating data loading in deep neural network training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 235–245.
[31] Donglin Yang, Wei Rang, and Dazhao Cheng. 2020. Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters. In *Proceedings of the 21st International Middleware Conference*. 386–399.
[32] Geoffrey X Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 126–139.
[33] 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)*. IEEE, 104–115.
[34] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and analyzing deep neural network training. In *IEEE International Symposium on Workload Characterization (IISWC)*. 88–100.