Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers

Shijian Li
Worcester Polytechnic Institute, USA
sli8@wpi.edu

Robert J. Walls
Worcester Polytechnic Institute, USA
rjwalls@wpi.edu

Tian Guo
Worcester Polytechnic Institute, USA
tian@wpi.edu

***Abstract*—Cloud GPU servers have become the *de facto* way for deep learning practitioners to train complex models on large-scale datasets. However, it is challenging to determine the appropriate cluster configuration—e.g., server type and number—for different training workloads while balancing the trade-offs in training time, cost, and model accuracy. Adding to the complexity is the potential to reduce the monetary cost by using cheaper, but revocable, transient GPU servers.**

In this work, we analyze distributed training performance under diverse cluster configurations using CM-DARE, a cloud-based measurement and training framework. Our empirical datasets include measurements from three GPU types, six geographic regions, twenty convolutional neural networks, and thousands of Google Cloud servers. We also demonstrate the feasibility of predicting training speed and overhead using regression-based models. Finally, we discuss potential use cases of our performance modeling such as detecting and mitigating performance bottlenecks.

***Index Terms*—distributed training, measurement, modeling**

I. Introduction

The process of training deep neural networks (DNNs) has evolved from using single-GPU servers [1] to distributed GPU clusters [2, 3] that can support larger and more complex DNNs. Cloud computing, providing on-demand access to these critical yet expensive GPU resources, has become a popular option for practitioners. Today's cloud provides its customers abundant options to configure the training clusters, presenting opportunities for tailoring resource acquisition to the specific training workload. When using cloud-based GPU servers to train deep learning models, one can choose the server's CPU and memory, specify the GPU type, decide the number of servers, as well as pick the desired datacenter location. However, this configuration flexibility also imposes additional complexity upon deep learning practitioners.

Concurrently, to lower the monetary cost of training, one could also consider using a special type of cloud servers, referred to as *transient servers*, that have lower unit costs with the caveat that the server can be revoked at any time [4, 5]. Revoked GPU servers often mean significant loss of work and require manual effort by the practitioner to request new servers, to reconfigure the training cluster, and even to diagnose potential performance bottlenecks. Concretely, when a GPU server is revoked, all its local training progress will disappear and in the worst case, the revocation will also impede the functionality of saving the trained model [6, 7].

In this work, we set out to characterize and predict the impact of cluster configuration on distributed training, in the context of transient and traditional *on-demand* cloud servers. We measured and characterized several key factors that impact distributed training on transient servers and evaluated regression-based models for predicting training throughput and fault-tolerance overhead.

To streamline measurement and data collection on distributed training, we designed and built a framework called CM-DARE. It allows us to measure, monitor, and collect metrics such as training speed and revocation time, which supports our performance characterization and modeling and enables use cases such as performance bottleneck detection. We built CM-DARE on top of an existing distributed training framework (*TensorFlow* [6]) and library (*Tensor2Tensor* [8]), with transient-specific optimizations that mitigate the impact of revocation and improve fault-tolerance. Though we exclusively used *TensorFlow* and Google Cloud in this work, we argue that our measurement methodology (e.g., the use of custom convolutional neural networks) can be extended to other deep learning frameworks and cloud providers.

Our work differs from prior work in distributed training performance modeling in three key aspects. *First,* it consists of large-scale, cloud-based measurement and data-driven performance modeling rather than theoretical modeling and on-premise measurement [1, 9, 10]. *Second,* we identified use cases that benefit from having access to the raw measurement data, performance models, and CM-DARE measurement infrastructure. *Finally,* we are the first to characterize and model performance of distributed training with *transient servers*. In short, we make the following contributions.

- We conducted a large-scale measurement study that includes twenty convolutional neural networks on three types of Google Cloud GPU servers. We observe, for example, that the training speed of heterogenous clusters—i.e., clusters consisting of different GPU hardware—is approximately the sum of individual server speeds. **Note for the PC:** We will make our dataset publicly available upon publication.
- We built and evaluated performance models that predict the training speed and fault-tolerance overhead of GPU clusters with as low as 3.4% mean absolute percentage error. Such models serve as the building blocks for predicting heterogeneous cluster training performance. More impor-

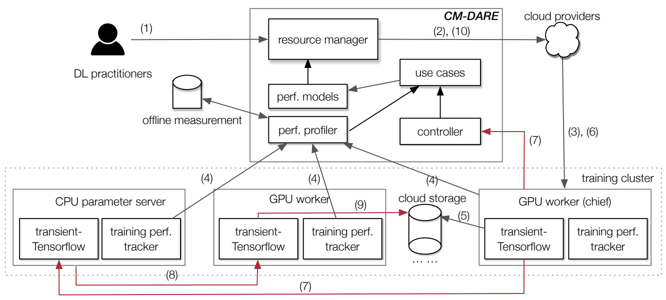


Fig. 1: CM-DARE ***architecture and workflow.*** *The key workflow for performing transient-aware distributed training is labeled numerically, with the revocation-triggered interactions highlighted in red.*

tantly, we identified appropriate deployment scenarios for each performance model.

- We identified use cases, such as detecting and mitigating distributed training performance bottlenecks, that would benefit from our prediction models.
- We designed and implemented a measurement and training framework called CM-DARE, which simplifies distributed training on transient servers and improves the robustness of existing fault-tolerance mechanisms.

II. Overview of the CM-DARE Framework

CM-DARE is a measurement framework we built and used to characterize and predict the performance of training convolutional neural networks (CNNs) on clusters of cloud-based GPU servers, i.e., *distributed training*.

Specifically, we focus on *asynchronous training with parameter servers*, a popular distributed training architecture implemented by Google's TensorFlow [6] and commonly used for models that can fit into the memory of a discrete GPU. In this architecture, servers are separated logically into two categories: *parameter servers* and *workers*. Parameter servers update the deep learning model parameters after each GPU server (i.e., worker) generates the gradients. Each worker holds its own copy of the entire deep learning model and works on subsets of the training dataset. The training is asynchronous because each worker communicates with the parameter servers at its own pace. One worker is designated as the *chief worker* and is given additional responsibilities, including periodically saving model parameters to cloud storage, i.e., *checkpointing*.

The asynchronous nature of this architecture offers two key benefits for transient distributed training. First, it is resilient to transient revocations because the cluster can continue training even if a worker is revoked. Second, it reduces the impact of hardware differences in heterogeneous clusters because slower workers do not impede others.

At the core of CM-DARE, depicted in Figure 1, is the *transient-aware performance models* which is powered by *performance profiler* that continuously monitors training performance and transient server revocations. In addition, CM-DARE includes *transient-TensorFlow*, a modified version of TensorFlow, that handles worker revocations by notifying the parameter server and supporting checkpointing even when the chief worker is revoked.

To collect measurements with CM-DARE, (1) we provide a training script with information such as cluster configuration, which (2) the *resource manager* uses for setting up the cloud training cluster. (3) All training servers, including on-demand parameter servers and transient GPU workers, will run *transient-TensorFlow* which establishes RPC connections between parameter servers and workers and (4) the *performance tracker* that sends training performance to the *performance profiler*. (5) After the specified checkpoint interval, the chief worker saves the current model parameters to cloud storage. (6) In the case that the chief worker is revoked, (7) the chief will notify the parameter server, as well as the *controller*, about its revocation. (8) The parameter server will then select one GPU worker to take over checkpointing, (9) and the worker will save the checkpoint to the same cloud storage at the specified interval. (10) The *resource manager* fulfills cluster configuration changes that are determined by the *controller* based on the specified use cases, performance models, and online measurement. Finally, we obtain the trained models and measurement data once the training is completed.

Currently, CM-DARE runs on Google Cloud. We chose Google Cloud because it allows customization of GPU servers, which provides better control and flexibility for training deep learning models with different resource requirements. Further, Google's transient servers, called preemptible VMs in the Google Cloud argot, have a maximum lifetime of 24 hours and are offered at fixed prices that are significantly lower than their on-demand counterparts.

We characterize and predict distributed training performance in the context of training speed in Section III, fault-tolerance overhead in Section IV, and revocation overhead in Section V. Finally, in Section VI, we explore two potential use cases that could benefit from our study: predicting training speed of heterogeneous clusters and detecting training bottlenecks.

III. Understanding and Predicting Training speed

Understanding how training speed varies based on key factors such as *GPU server type* and *model characteristics*, is the first step toward predicting distributed training performance. In this section, we quantify such relationships with CM-DARE-enabled empirical measurements. In summary, we find that regression-based prediction is a promising approach due to the strong correlation between training speed, GPU computational capacity, and model complexity. Further, the limited selection of available cloud GPUs make it feasible to build predictive models for individual GPU types and thus achieve higher prediction accuracy. Moreover, the training speed of an entire cluster is approximately the sum of individual worker speeds until a parameter-server-based bottleneck is reached. Finally, compared to prior approaches that do not consider transient server revocations and assume stable training environment [11–13], our data-driven approach achieves low prediction error of 9%.

TABLE I: ***Training speed (in steps per second) for the simplest cluster configuration.*** *We measured a cluster consists of one GPU worker and one parameter server in the same data center. The GFLOPs of CNN models are calculated based on the* CIFAR-10 *dataset.*

GPU (teraflops)	CNN model (GFLOPs) ResNet-15 (0.59)	ResNet-32 (1.54)	Shake Shake small (2.41)	Shake Shake Large (21.3)
K80 (4.11)	9.46 ± 0.19	4.56 ± 0.08	2.58 ± 0.02	0.70 ± 0.002
P100 (9.53)	21.16 ± 0.47	12.19 ± 0.41	6.99 ± 0.35	1.98 ± 0.03
V100 (14.13)	27.38 ± 0.88	15.61 ± 0.38	8.80 ± 0.24	2.18 ± 0.04

A. Measurement Methodology

Training Dataset. We chose *CIFAR-10*, one of the most widely used datasets in deep learning research, as the training dataset [14]. *CIFAR-10* contains a total of 60K images with dimensions of 32 × 32 pixels. The training workload is provided by practitioners in the form of *number of steps* which each step goes through a mini-batch of images. Larger-scale datasets, such as *ImageNet*, that are commonly used for improving the real-world model accuracy, were unnecessary as our measurements focus on training speed.

Models. We used two *ResNet* [15] and two *Shake Shake* [16] implementations from the *Tensor2Tensor* framework. These four CNN models are popular for image classification and have different characteristics such as model complexity that are useful for our study. *Model complexity* is defined as the number of floating point operations (FLOPs) required by the CNN model to train on one image. We further generated an additional 16 variants of CNN models by varying the number of hidden layers and the size of each hidden layer; these custom models allowed us to better observe how model complexity impacts training time. We used the built-in TensorFlow profiler tool to calculate the FLOPs for each model.

GPU Types. We used three GPU types offered by Google Cloud: Nvidia Tesla *K80*, *P100*, and *V100*. These GPUs used PCIe and had 12GB, 16GB, and 16GB of memory, respectively. They had computational capacity of 4.11, 9.53, and 14.13 teraflops. We chose these GPU types because they are the only three offered by Google Cloud that are commonly used for training. We refer to a server with access to a GPU as a *GPU server*. Each GPU server was configured with 4 vCPUs and 52GB of main memory. During our experiments, neither the CPU nor the main memory were saturated.

Cluster Configuration. For measuring the impact of model and GPU type, we used a simple cluster consisting of one GPU server and one parameter server with both servers residing in the same data center. We ran the parameter server on a non-revocable server with 4vCPUs, 16GB of main memory, and Ubuntu 18 LTS. GPUs were not needed for the parameter server as its primary tasks, aggregating gradients and updating parameters, are less computation-intensive and are often bound by network communication [17].

Measuring Training Speed. We utilized built-in TensorFlow functionality to log training speed of the entire cluster. Training speed is defined as *steps per second* where each step involves the generation of gradients based on the new model parameters using a batch of images. Unless otherwise specified, we averaged the training speed every 100 steps. For each cluster, we trained and recorded for 4000 steps. We used the same training workload for all clusters and set the checkpoint interval to be larger than our measurement duration to avoid measuring checkpoint overhead, which we consider in Section IV. To measure the training speed of individual workers, without incurring logging overhead caused by invoking hook functions, we used the TensorFlow *TFProf* tool.

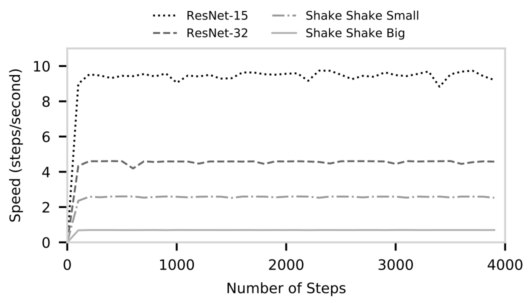


Fig. 2: ***Training speed for the simplest cluster configuration (K80).*** *We plotted the training speed of all four representative CNN models, and observed that training speed is rather stable after warmup.*

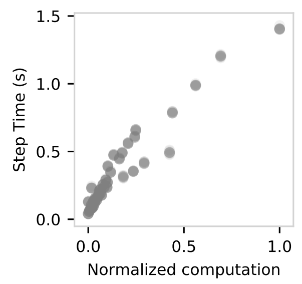
B. Impact of Model and GPU Type

Table I shows the average training speed (and standard deviation) for different combinations of model and GPU type. To avoid including noisy data, we discarded the measurements associated with the *first 100* steps. As expected, the higher the computational capacity of the GPU, the faster the training speed. For example, the *V100* server has the highest training speed for all four CNN models. Further, the training speed drops as model complexity increases. For instance, training a *ResNet-32* of 1.54 GFLOPs is almost 2X slower than training a *ResNet-15* of 0.59 GLOPs using the same *K80* GPU server.

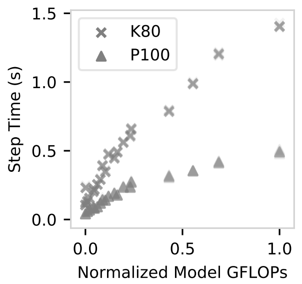
Another important observation, visualized in Figure 2 for a *K80* server, is that training speed was stable after the warm-up period, with a maximum coefficient of variation of 0.02. We observed similar behavior for the other two GPU servers. This training speed consistency has several important implications, namely the feasibility of predicting the speed using historical data and the possibility to quickly detect (and address) under-performing workers.

1) Predicting the Impact: The next question we explore is how to leverage the above observations to predict the training speed of an individual worker, especially when training a previously unobserved CNN model. In Section III-D, we further investigate the question of how to predict the training speed of an entire cluster.

Figures 3(a) and 3(b) show the relationship between *step time* S and normalized computation ratio C_{norm} and normalized model complexity C_m, respectively. The *computation*



(a) *Normalized computation*



(b) *Normalized model complexity*

Fig. 3: ***Step time vs. normalized computation ratio C_{norm} and model complexity C_m.*** *We observed strong positive correlation between average* step time *and normalized computation and model complexity for both* K80 *and* P100 *GPU servers.*

TABLE II: ***Comparison of step time prediction models.*** *We evaluated the listed regression models in predicting* step time*, using k-fold cross validation MAE and test dataset MAE. The Unit for MAE is seconds.*

Regression Model	Input Feature	K-fold MAE	Test MAE
Univariate, GPU-agnostic	C_{norm}	0.072 ± 0.015	0.068
Multivariate, GPU-agnostic	C_m, C_{gpu}	0.103 ± 0.026	0.093
Univariate, K80	C_m	0.065 ± 0.013	0.068
SVR Polynomial Kernel, K80	C_m	0.035 ± 0.014	0.041
SVR RBF Kernel, K80	C_m	0.026 ± 0.012	0.031
Univariate, P100	C_m	0.029 ± 0.008	0.031
SVR Polynomial Kernel, P100	C_m	0.019 ± 0.007	0.020
SVR RBF Kernel, P100	C_m	0.012 ± 0.008	0.016

ratio is defined as model complexity divided by GPU computational capacity C_{gpu}, and *step time* is the inverse of training speed. The computation ratio and model complexity were normalized using min-max normalization.[1] Each dot represents the observed step time, averaged over 1400 steps, from training a CNN model. We collected data for a set of twenty CNN models, comprising the 4 models used for the observations in the previous section and the 16 custom models mentioned in the methodology.

We make two key observations. *First,* the step time of different GPUs form a trend line when using C_{norm} and are distinctly separated when using C_m. This suggests that both normalized computation ratio and normalized model complexity are useful in predicting training speed. *Second,* the shapes of the trend lines indicate that linear functions might be fitted for predicting the step time.

Based on the observations above, we evaluated eight regression models, listed in Table II, for predicting training speed. We chose a mix of univariate, multivariate, and support vector regression (SVR) models because the former two are simple and commonly used, and the latter has been shown to work well in modeling performance in cloud environments [18]. These models can be divided into two categories: GPU-agnostic and GPU-specific. The GPU-agnostic univariate regression is modeled as $S = a \times C_{norm} + b$, while the GPU-agnostic multivariate regression is modeled as $S = a \times C_m + b \times C_{gpu} + c$, where a, b, c are learned parameters.

Training GPU-specific prediction models are feasible because cloud GPUs are often limited in selections and are usually not customizable. Specifically, we considered the following three GPU-specific regression models:

$$S_{gpu} = a \cdot C_m + b, \tag{1}$$

$$S_{gpu} = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot (C_{m_i}, C_m)^2, \tag{2}$$

$$S_{gpu} = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot \exp(-\frac{||C_{m_i} - C_m||^2}{2\sigma^2}), \tag{3}$$

where S_{gpu} denotes the *step time* of one specific GPU; α_i and α_i^* are Lagrange multipliers used in SVR to determine support vectors; and $(C_{m_i}, C_m)^2$ and $\exp(-\frac{||C_{m_i} - C_m||^2}{2\sigma^2})$ are two-degree polynomial and RBF kernel functions, respectively.

For training each regression model, we randomly split the dataset into training data and test data with 4:1 ratio. We conducted k-fold cross validation on the training data, and evaluated the performance of resulting regression models using mean absolute error (MAE) for both training and test data. We chose MAE because it provides a more natural and unambiguous measurement compared to other metrics such as root mean square error (RMSE) [19]. Further, k-fold MAE allows us to compare against different regression models and test MAE provides insight regarding the robustness of each regression model. For training SVR-based models, we used grid search cross validation to search for the optimal set of hyperparameters, i.e., penalty p and ϵ, that yield the best MAE. We followed common practice and set the range for p to be $[10, 100]$ with a step increment of 10, and ϵ to be $[0.01, 0.1]$ with a step increment of 0.01.

As shown in Table II, the GPU-specific regression models achieved higher MAE than the GPU-agnostic predictive models. For example, all six GPU-specific models had a MAE of less than 0.068 seconds on the test dataset. As the average *step time* across different CNN models is 0.48 seconds, we believe models with such MAEs can produce reasonable predictions. In comparison, the GPU-agnostic regression models had up to 0.093 seconds MAE on the test dataset. Furthermore, the SVR models with the non-linear RBF kernel function provided a better fit than those with the polynomial kernel function—providing, for example, the best MAE for both k-fold cross validation and test dataset. The mean absolute percentage error (MAPE) on test dataset for the K80-specific SVR model with RBF kernel was 9.02%, compared to 13.79% for the P100-specific SVR model with polynomial kernel.

C. Impact of Cluster Size and Heterogeneity on Worker Speed

To predict training speed for an entire cluster, we must first understand the impact of cluster size and mixing GPU types on the training speed of an individual worker. Table III shows the average step time for individual *K80*, *P100*, and *V100* workers when used as part of both homogeneous and heterogeneous

[1] We also considered *z-score standardization* for preprocessing; however as our data does not follow a Gaussian distribution, it would be less beneficial to apply this technique.

TABLE III: ***Average step time (in millisecond) of an individual worker when training ResNet-32.*** *We studied the impact of adding more GPU servers on a single GPU's training speed. Clusters are represented as* (x, y, z) *where x, y, z denote the number of* K80, P100, V100 *GPU servers respectively.*

	Baseline	Homogeneous			Heterogeneous
	(1, 0, 0)	(2, 0, 0)	(4, 0, 0)	(8, 0, 0)	(2, 1, 1)
K80	229.85 ± 3.04	232.08 ± 2.22	229.57 ± 3.15	227.46 ± 5.06	221.16 ± 2.66
P100	105.45 ± 1.99	105.27 ± 1.45	112. 73 ± 6.52	198.11 ± 18.65	107.61 ± 2.13
V100	92.38 ± 3.64	95.90 ± 4.07	106.36 ± 6.16	191.72 ± 26.38	93.52 ± 4.58

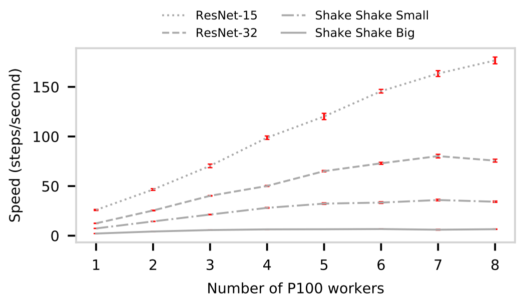


Fig. 4: ***Empirically measured cluster training speed.*** *We observed that training speed bottleneck can be reached faster for more complex CNN models and powerful GPUs such as* P100. Shake Shake *models exhibit negligible variations compared to* ResNet *models, potentially due to higher model computation complexity saturating GPU.*

clusters. The baseline column shows the average step time for a cluster consisting of a single worker.

We make three key observations. *First,* for homogeneous clusters, the average training speed of an individual worker was roughly the same until the cluster became large enough to encounter a parameter server bottleneck. This bottleneck arises when the workers' output (i.e., computed gradients) exceeds the parameter server's capacity. Consequently, the training is bounded by how fast the parameter server can update model parameters. Notice that the *K80* workers, with the least powerful GPU, did not reach this bottleneck in our experiments and the average step time was within 1% for all tested cluster sizes. In contrast, workers with the more powerful GPUs hit this bottleneck at smaller cluster sizes (8 for *P100* and 4 for *V100*). We discuss how to mitigate the impact of parameter server bottlenecks in Section VI-B.

Second, as the cluster size increases, we observe higher variations for the average step time. For example, the coefficient of variation increases from 0.019 to 0.094 for *P100* clusters. *Third,* the use of heterogeneous clusters does not appear to impact the training speed of an individual worker. For instance, the average step time of a *V100* worker is 92.38ms in the baseline cluster and 93.52ms in the heterogeneous cluster.

D. Impact of Cluster Size on Cluster Training Speed

To understand the impact of the number of GPU servers on the cluster training speed, we trained the four Tensor2Tensor models with clusters comprised of an increasing number of *P100* GPU servers. Figure 4 shows the average training speed for each cluster.

We make three key observations. *First,* the cluster training speed increases as the cluster size grows. The upward trend is most obvious for *ResNet-15*, the least computationally-intensive model of the four. *Second,* for both *ResNet-32* and *Shake Shake Small* models, the training speed starts to plateau after more than four GPU servers in the cluster, caused by the parameter server bottleneck discussed previously. *Third*, the lack of training speed improvement for *Shake Shake Big*, the most complex of the four models, suggests that the computational capacity of the *P100* GPU was insufficient for the model. In a separate experiment, not shown, we observed a positive correlation between the training speed and cluster size for *Shake Shake Big* after switching from *P100* to the more powerful *V100* GPU.

All the observations in this subsection and the preceding subsections indicate that we can effectively predict unknown clusters' training speed by leveraging our understanding of individual worker's performance, and composing from our previously built performance models. Further, if the predicted performance deviates from the online measurement, CM-DARE can flag parameter servers as the bottleneck and start provisioning additional parameter servers.

IV. Modeling Fault-tolerance Overhead

Current deep learning frameworks, such as TensorFlow, often provide basic fault-tolerance mechanisms. For example, TensorFlow allows deep learning practitioners to periodically save the most recent model parameters to remote storage. These model files serve as an intermediate result, and allow resuming the training from the checkpoint in case of a failed training session. Fault-tolerance mechanisms are especially important when using transient servers for distributed training, as these mechanisms can reduce the amount of work loss when a worker is revoked.

In this section, we study how fault-tolerance mechanisms, specifically checkpointing CNN models, impact distributed training time. Our observations indicate that the tasks of training and checkpointing happen sequentially and that one can take into account the checkpoint overhead by directly adding to the predicted training time. Our checkpoint prediction models yield only 5.38% mean absolute percentage error and our analysis suggests the value of using different prediction models in different deployment scenarios.

A. Measurement Methodology

Checkpoint-related Files. TensorFlow generates three types of files, i.e., *data, index and meta* files, when checkpointing deep learning models. Both index file and meta file sizes are highly correlated to the number of tensors in the CNN model. We denote the size of data, meta, and index files with S_d, S_m, and S_i, respectively, and use S_c to denote the sum of three files.

Measuring Checkpoint Time. We instrumented the checkpointing function used by TensorFlow and measured the time to checkpoint all twenty CNN models described in Section III-A. In TensorFlow, the chief worker is responsible

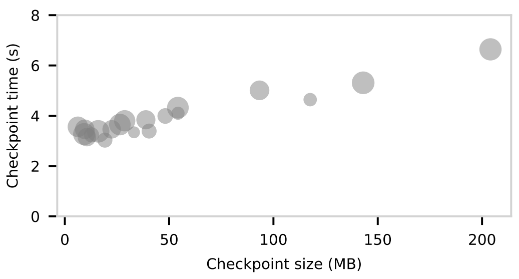


Fig. 5: ***Checkpoint duration vs. checkpoint size.*** *We measured the total checkpoint time for all twenty CNN models for five times and plotted both the average and coefficient of variation (circle size).*

for checkpointing for the entire cluster. Further, checkpointing does not run on the GPU. Consequently, we measured the checkpointing time using a cluster consisting of a parameter server and a single *K80* worker. To minimize the network impact on the measured checkpointing time, we configured the worker to save checkpoints to remote storage in the same data center as the training cluster.

B. Understanding Checkpoint Time

Figure 5 shows the checkpoint time, averaged over five checkpoints, for all twenty CNN models. We observed a low coefficient of variation for all models, ranging from 0.018 to 0.073, and a positive correlation between checkpoint size and time. By cross-examining the training speed with and without checkpointing, we confirmed that the tasks of checkpointing and training are conducted in sequence. As an example, in the case of training *ResNet-32*, it takes 25.64 versus 21.93 seconds on average to finish 100 steps with and without checkpointing, respectively. The difference of 3.71 seconds is consistent with the measured *ResNet-32* checkpoint time of $3.84{\pm}0.25$ seconds. This indicates that we can directly add the checkpoint overhead to the distributed training time modeled without checkpointing. Finally, recall that only *one* worker performs the checkpointing. As such, we just need to account for one interrupted worker when predicting the overhead.

C. Predicting Checkpoint Time

We considered the four regression models listed in Table IV for predicting checkpoint time. Further, given that *index* and *meta* file sizes are both correlated to the number of tensors, we use principal component analysis (PCA) to preprocess the input features to automatically reduce the variable dimensions to two components. Similar to predicting training speed (Section III-B), we considered the following models: *(i)* $T_c = a \cdot S_c + b$, *(ii)* $T_c = a \cdot S_d + b \cdot S_m + c$, *(iii)* $T_c = (a, b) \cdot PCA(S_d, S_m, S_i) + c$, *(iv)* $T_c = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*) \cdot \exp(-\frac{||S_c^i - S_c||^2}{2\sigma^2})$, where T_c denotes checkpointing time; α_i and α_i^* are Lagrange multipliers used in SVR to determine support vectors; and $\exp(-\frac{||S_c^i - S_c||^2}{2\sigma^2})$ is RBF kernel function, respectively.

As shown in Table IV, the SVR model with RBF kernel yielded the best MAEs for both k-fold cross validation and on the test dataset. The mean absolute error percentage of the SVR model with RBF kernel on the test dataset is 5.38%. The other three models have up to 1.74X higher k-fold MAEs and around 1.45X higher test MAEs. All four models would have reasonable utility in predicting total training time. For example, in the case of *ResNet-32* that trains to $64K$ steps with $4K$ checkpoint interval, with linear regression model the actual and predicted checkpoint time are 3.83 and 3.96 seconds—a difference of 3.4%. Even though the prediction error is accumulative, it has minimal impact on the final training time that is in the order of magnitude of hours.

TABLE IV: ***Comparison of checkpoint prediction models.*** *We evaluated the listed regression models in predicting checkpointing time, using k-fold cross validation MAE and test dataset MAE.* S_c, S_d, S_m, *and* S_i *denote the file sizes for total checkpoint, data, meta, and index files, respectively. The Unit for MAE is seconds.*

Regression Model	Input Feature	K-fold MAE	Test MAE
Univariate	S_c	0.345 (± 0.099)	0.356
Multivariate	S_d, S_m	0.291(± 0.139)	0.353
Multivariate, Two Components PCA	S_d, S_m, S_i	0.286 (± 0.142)	0.354
SVR RBF kernel	S_c	0.198 (± 0.135)	0.245

Finally, practitioners might decide to choose a prediction model based on factors other than the prediction accuracy, such as the time to retrain the model. For instance, if the practitioner is monitoring a running cluster and observes variable performance, then the prediction model needs to be retrained with new measurement data. It might be better to choose models that can be retrained faster, e.g., multivariable models instead of SVR models, as the latter requires hyperparameter tuning.

V. Characterizing Revocation Overhead

One of the key challenges of using transient servers for distributed training is that they can be revoked at any time. Even the revocation of a single worker can lead to significant performance degradation [7]. In this section, we characterize the revocation patterns of Google Cloud's transient servers. In summary, we observed that cloud region, GPU type, and time-of-the-day are important factors for understanding revocation patterns. Further, we found that immediately requesting a replacement worker after a revocation is a valid strategy as the time to request transient GPU servers is not impacted by revocations. Lastly, the workload of a transient server does not appear to impact its likelihood of revocation.

A. Measurement Methodology

CM-DARE Measurement Infrastructure. To measure the revocation of Google transient servers, we implemented a hook function in TensorFlow in conjunction with startup and shutdown scripts provided by Google Cloud. Each GPU worker in the training cluster connected to the CM-DARE *controller* running in the parameter server via RPC. *Transient-TensorFlow*, running on the GPU workers, monitored the triggering of each script and forwarded the corresponding timestamped signals to the controller.

Measuring Transient Startup Time. *Transient server startup time* is defined as the time between when the cloud customer

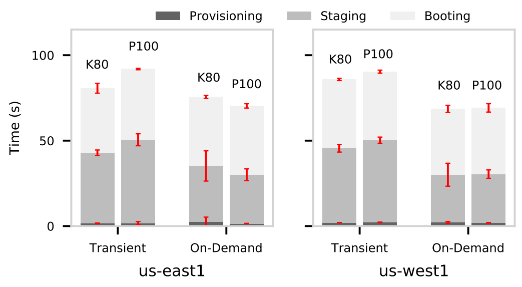


Fig. 6: *Startup time breakdown of requesting new servers without observing revocation.* *We requested* K80 *and* P100 *GPUs in* us-east1 *and* us-west1 *regions, with both transient and on-demand servers.*

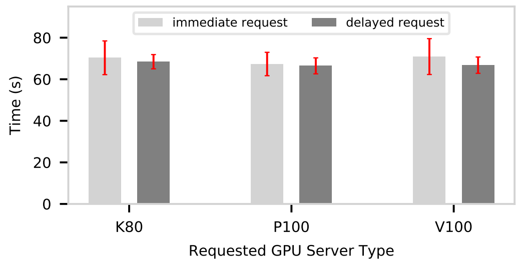


Fig. 7: *Startup time comparison.* *We studied whether revocation events impact server startup time with immediate and delayed acquisition requests.*

requested the transient server and when the transient server became available in the training cluster. For each transient server, we measured the time for three consecutive stages [20]. First, resources are allocated for the server during the *provisioning* stage. Second, after resource acquisition, the instance is prepared for booting in the *staging* phase. Third, once the server boots up, it enters the *running* stage. We used the Google Cloud API in conjunction with the startup script to request servers and measured the duration for each stage by periodically querying the cloud-returned state information. For each GPU-region combination, we requested transient and the equivalent on-demand servers for comparison. To quantify availability-related startup overheads, we measured the time to start different transient GPU servers after a predefined time window upon a revocation event through CM-DARE.

Measuring Revocations. We requested transient GPU servers in batches. For each batch we requested the maximum number of servers allowed for our account. We let these servers run for their maximum lifetime of 24 hours and recorded any revocations that occurred prior to the 24-hour cutoff. We repeated this process for a total of twelve non-consecutive days. We divided the transient servers into two equally-sized groups: the first group contained idle servers and the second group consisted of servers that were stressed in CPU, memory, and GPU resources. For stressing CPU and memory resources, we used a popular benchmark [21] and for stressing the GPU, we used built-in TensorFlow tasks that performed operations

TABLE V: *Transient GPU servers revocations by regions.* *We observed that revocations vary based on GPU types and regions, e.g., us-west1 region has the highest overall revocation percentage. Further, idle and stressed servers exhibit similar revocation rates.*

Regions	K80	P100	V100
us-east1	30 (46.67%)	30 (70%)	N/A
us-central1	48 (56.25%)	30 (53.33%)	30 (66.67%)
us-west1	48 (22.92%)	30 (66.67%)	30 (73.33%)
europe-west1	30 (66.67%)	30 (26.67%)	N/A
europe-west4	N/A	N/A	30 (43%)
asia-east1	N/A	N/A	30 (47%)
total	156 (46.15%)	120 (54.17%)	120 (57.5%)

similar to distributed training workloads. We repeated the above measurements for the three GPU types described previously in six geo-graphically distributed regions. We chose three US-based regions, two Europe-based regions, and one Asian-based region to study the impact of the region and time-of-day on revocations.

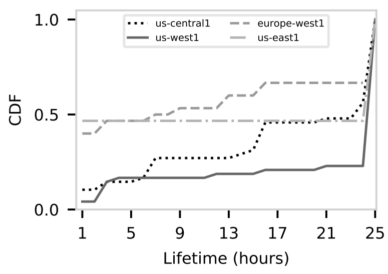
Measuring Worker Replacement Overhead. *Worker replacement overhead* denotes the time component of configuring the environment for distributed training after a worker replacement. This includes starting the deep learning framework, joining the existing training session, downloading the training dataset that the revoked server held, and recomputing from the last checkpoint if needed. We measured the *cold start* and *warm start* worker replacement overhead with a cluster compromised of one *K80* GPU worker and one parameter server. Cold start refers to the overhead when using a newly requested GPU server while warm start uses an existing GPU server.

Measuring Recomputation Overhead. We trained *ResNet-15* with 2-worker clusters and configured the checkpoint interval to be 4K steps. We manually revoked the chief worker at 1K steps since the last checkpoint, and added a new worker to the training session at a specified interval. In particular, *recomputation overhead* denotes the time difference between adding a replacement worker with the chief's old IP address and adding a replacement worker with a new IP address.

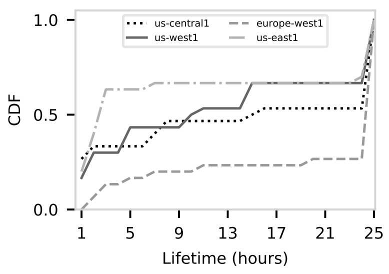
B. Breaking Down Transient Startup Time

Intuitively, transient startup time impacts transient distributed training because it is the amount of time the training cluster has to run with *fewer* GPU workers after a revocation. In this subsection, we focus on quantifying the transient startup time under different scenarios, such as immediately after server revocations. Our understandings can help deep learning practitioners make informed decisions about provisioning transient GPU servers.

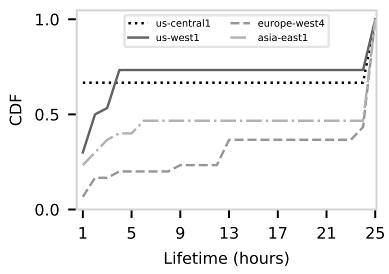
Figure 6 shows the average startup time of transient and on-demand GPU servers in two cloud regions. Our *first* observation is that it takes less than 100 seconds to startup transient GPU servers. This short startup time makes it feasible for practitioners to quickly react to a training slowdown by requesting and adding transient servers to the ongoing training session. *Second,* it is on average 8.7% slower to startup the more powerful transient *P100* GPU servers than *K80* GPU servers, with the staging time contributing most to the



(a) *K80 GPU server.*



(b) *P100 GPU server.*



(c) *V100 GPU server.*

Fig. 8: ***Lifetime analysis for different regions and GPU servers.*** *We launched three types of transient GPU servers in six data centers throughout 12 non-consecutive days. We observed that up to 47.98% lived up to, and even beyond, the specified maximum of 24 hours.*

difference. The longer and more variable staging time for transient *K80* might be an indication of higher demand and lower availability of *K80* GPUs. *Third,* compared to their on-demand counterparts, transient startup time was only 11.14 seconds slower on average for *K80* and 21.38 seconds for *P100* servers. Such slowdown is negligible for distributed training workloads which often last hours if not days [17, 22].

Figure 7 shows the impact of recent revocations on transient startup time. In particular, we studied *immediate requests* and *delayed requests*. For the former, we immediately requested a *K80*, a *P100*, and a *V100* GPU server after one of our *K80* servers was revoked. Delayed requests are the same as immediate requests except that we waited for at least an hour before requesting.

We observed little impact, up to 4 seconds in the case of *V100* GPU servers, of revocation events on transient startup time. These results are counter-intuitive as one of the potential reasons for revocation is high demand for a given resource [23]. These results suggest that deep learning practitioners do not need to consider the revocation overhead associated with low availability. Further, the average startup time for immediate requests for both *P100* and *V100* are within 3 seconds to that of *K80*. This suggests the possibility to request any GPU type as replacement for the revoked server. The average startup time for immediate and delayed requests are within 4 seconds for all GPU types. However, for immediate requests, we observed a 4X higher coefficient of variance (12% compared to 3%)—startup time is more variable immediately after a server revocation.

C. Understanding Transient Revocations

Next, we look at the different factors that impact the revocation frequency. Table V summarizes the 206 revocations for *396* transient GPU servers launched throughout twelve non-consecutive days, in six different data centers. Our *first* observation is that the workload of transient servers does not seem to impact the revocation frequency; roughly half of all observed revocations were for unstressed servers, i.e., idle servers. Our *second* observation is that different regions can lead to different revocation frequencies. For example, *europe-west1* has the lowest revocation frequency for *P100* while *us-west1* region has the highest revocation frequency for *P100* and *V100* GPU servers. As a simple strategy, deep learning practitioners can avoid high revocation regions to mitigate the impact on distributed training. *Third,* more expensive GPU servers, i.e., *V100*, are more likely to be revoked compared to cheaper GPU servers. This suggests the need to balance the computation needs and revocations when choosing GPU servers.

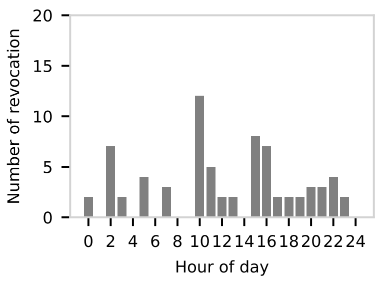
Figure 8 shows that different GPU servers in different regions tend to have distinct lifetime characteristics. For example, more than 50% of *K80* servers from *europe-west-1* were revoked in the first two hours compared to less than 5% from *us-west-1*. The mean time to revocation for *K80* ranges from 10.6 hours to 19.8 hours. This suggests the benefits of launching training clusters in regions such as *us-central1* when using *K80*. In addition, more powerful GPU servers tend to have a shorter mean time to revocation, e.g., *V100* servers in *us-central1* had a mean time to revocation of 7.7 hours. Combined, these observations also indicate the challenge of selecting the initial cluster configuration—a region that provides more stable *K80* servers might have volatile *V100* servers.

Figure 9 illustrates the hour of the day when revocations occurred, represented in each region's local time. Each GPU type exhibited different revocation patterns. For example, *K80* servers had the highest number of revocations at 10AM, perhaps caused by a surge of demand. Further, for *V100* servers, we did not observe any revocations between 4PM and 8PM.

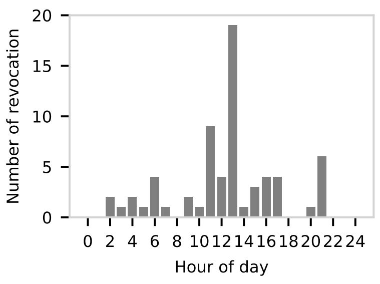
Finally, our observations suggest an avenue for future work: investigating how strategically launching transient clusters at different times of day and different data center locations can help mitigate revocation impacts.

D. Worker Replacement Overhead

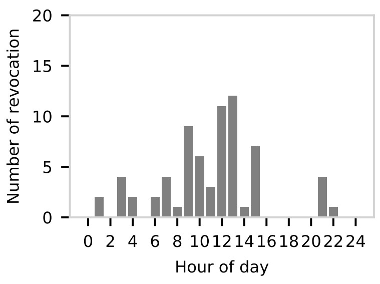
Figure 10 compares the cold start and warm start worker replacement time. We make two key observations. *First,* requesting new workers after revocations (cold start) are much more costly than scenarios where only restarting the training framework (warm start) is needed. For example, in the case of *ResNet-15*, it took about 75.6 seconds compared to 14.8 seconds. *Second,* both the cold and warm start time increase with model size and complexity. For instance, the worker



(a) *K80 GPU server.*



(b) *P100 GPU server.*



(c) *V100 GPU server.*

Fig. 9: ***Time-of-day impact on revocations.*** *We analyzed the revocation events for three types of GPU servers that were launched throughout twelve non-consecutive days. We found that revocations are both server and time-of-day dependent.*

replacement overhead for *Shake Shake Big* was 15 seconds longer than *ResNet-15*, with most of the overhead coming from the training computation graph setup. We expect to observe similar overheads for *P100* and *V100* clusters given that such overheads are not GPU-dependent.

E. TensorFlow-specific Recomputation Overhead

In unmodified TensorFlow, we observed the following phenomenon: when the chief worker is revoked and a replacement worker is assigned the chief's previous IP address, the cluster will recompute from the last checkpoint. In other words, the cluster will discard any progress made since the last checkpoint. By design, the IP address is bound to the role of chief. Therefore, the replacement worker effectively becomes the new chief. As the chief worker is responsible for saving the checkpoint, the recomputation overhead can be high. Note, CM-DARE's *transient-tensorflow* avoids such overhead; consequently,we do not consider this overhead in modeling distributed transient training.

Figure 11 shows the recomputation overhead of training *ResNet-15* using a two-*K80* GPU cluster. We configured the checkpoint interval to be $4K$ and manually revoked the chief worker $1K$ steps after the last checkpoint. We evaluated the impact of the *replacement timing*, i.e., when the replacement worker is added and starts training. For each replacement timing, we measured the total time to reach the next designated checkpoint, with and without reusing the chief worker's IP and calculated the time difference (i.e., recomputation overhead). When using CM-DARE, an existing worker in the training session will be assigned the responsibility of checkpoint and therefore the recomputation overhead is bounded by the checkpoint interval. In Figure 11, such overhead is up to 224 seconds with a $4K$ steps checkpoint interval.

VI. Use cases of performance modeling

Finally we discuss how deep learning practitioners might leverage the findings and insights of our work for *(i)* predicting cluster training speed and *(ii)* detecting training bottlenecks. These use cases represent promising extensions to the measurement study presented in this work. Below we describe our preliminary evaluations but leave a comprehensive analysis for future work.

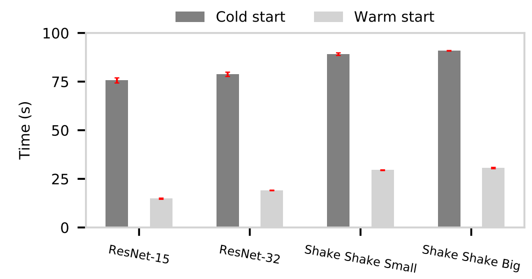


Fig. 10: ***Worker Replacement Overhead.*** *Cold start refers to the overhead when using a newly requested GPU server while warm start uses an existing GPU server.*

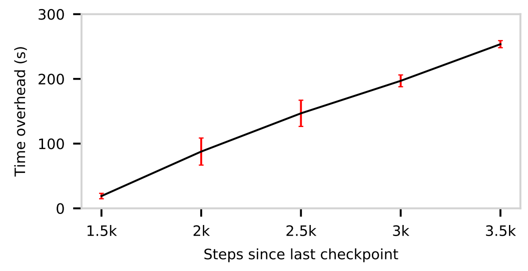
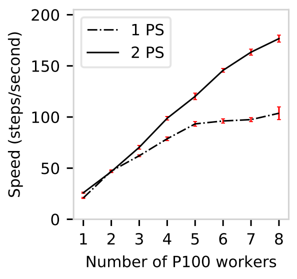


Fig. 11: ***TensorFlow-specific recomputation overhead.*** *We plotted the time savings of assigning a new IP address to the replacement worker, compared to reusing the old chief worker's IP address.*

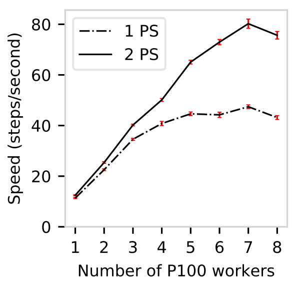
A. Heterogeneous Training Prediction

To predict the speed of *heterogeneous clusters*, i.e., clusters that consist of different types of GPU servers, we can leverage the GPU worker and parameter server performance models described in Section III. These models can be built offline using historical measurement data and retrained with continuous monitoring.

In Section III, we observe that individual server speed can be predicted using CNN model complexity and the computational capacity of the server's GPU. Further, we observe that adding GPU servers of different types to an asynchronous training session will not impact existing GPU workers' training speed. Therefore, we can predict the cluster training speed



(a) *ResNet-15.*



(b) *ResNet-32.*

Fig. 12: ***Parameter-server based bottleneck detection and mitigation.*** *We observed plateaued training speed for larger clusters with one parameter server. By adding a second parameter server, the training speed was improved by up to 70.6%. Similar trend was observed for* Shake Shake *models as well.*

as $sp = \sum_i^n sp_i$ for a cluster of n GPU servers, where sp_i denotes the training speed of GPU server i. The predicted training time T for N_w amount of training work, measured in *number of training steps*, is then:

$$T = \frac{N_w}{sp} + \lceil \frac{N_w}{I_c} \rceil \times T_c + N_r \times (T_p + T_s), \tag{4}$$

$$N_r = \sum_i^n Pr(R_i), \tag{5}$$

where I_c, T_c, T_p, and T_s denote the checkpoint interval (number of steps), checkpoint time, time to provision a new GPU server, and time to start the deep learning framework. We assume that N_w and I_c are user-specified values, sp and T_c are predicted for CNN models given their FLOPs, and T_p and T_s are running averages based on historical measurements.

The expected number of revocations N_r is calculated as the sum of the probabilities $Pr(R_i)$ that each worker i will be revoked during the training. We obtain these probabilities by querying the empirical CDFs shown in Section V-C. For simplicity, we do not consider the impact on the number of expected revocations by adding in more GPU transient servers. However, we have additional empirical data for supporting other more complicated modeling scenarios.

When using Equations (4) and (5), we observe a 0.8% prediction error for *ResNet-32* with $N_w = 64K$ and $I_c = 4K$ using our measurements. In summary, this work and CM-DARE provides the foundation for understanding and modeling transient distributed training performance.

B. Detecting Training Bottlenecks

Troubleshooting distributed training performance is challenging as bottlenecks can be caused by a plethora of factors such as network variations between parameter servers and GPU workers, and cloud server performance fluctuations. We illustrate detecting one such bottleneck caused by overloaded parameter servers. However, we believe that our method and CM-DARE are extendable to detect and resolve other bottlenecks.

Figure 12 compares the training speed for clusters with one parameter server and ones with two parameter servers. When training *ResNet* models using one parameter server, we observed that larger clusters, e.g., with six *P100* servers, do not yield reasonable speedup compared to smaller clusters. Although sublinear scalability in distributed training is not a myth [24, 25], CM-DARE allows one to detect when such bottlenecks arise during training by estimating the theoretical steady-state training speed as described in Section VI-A and comparing to the measured training speed. If the theoretical and measured speeds differ by a configurable threshold, CM-DARE flags the bottleneck. Currently, we use a warmup period of 30 seconds and a threshold of 6.7% based on empirical observation. Similar approaches can be used to detect slower GPU workers as well.

One potential way to resolve the this parameter-server-based bottleneck is to increase the number of parameter servers to two. This improved the training speed of all clusters by up to 70.6%. However, currently deep learning frameworks such as TensorFlow do not support dynamically adding parameter servers while training is ongoing—one has to restart the training session which incurs an overhead of about 10 seconds. We leave overhead-aware bottleneck mitigation as future work.

VII. Related work

Distributed Training. Cloud computing has become the *de facto* platform for hosting a plethora of modern applications, deep learning as an emerging workload is no exception [26]. Popular deep learning frameworks [6, 27–29] provide distributed SGD-based algorithms [30, 31] to train increasingly bigger models on larger datasets. Existing works towards understanding distributed training workloads can be broadly categorized into performance modeling [1, 9, 10] and empirical studies [1, 7, 22, 32, 33]. In contrast to prior model-driven performance modeling studies [10, 12, 13], where a static end-to-end training time prediction is the main focus, our work leverages data-driven modeling that is powered by a large-scale empirical measurement in a popular cloud platform. The insights provided from both theoretical and empirical characterizations of distributed training lead to numerous system-level optimizations. For example, prior work [34–37] explicitly considered the heterogeneity and designed distributed training systems that handle shifting bottleneck problems [36] or leverage the mathematical properties of training, e.g., identifying sufficient vectors that are required for training [37]. Our work adds unique knowledge of transient distributed training and includes transient-aware framework modifications that is valuable for optimizing transient-aware distributed training systems and resource managers.

Performance Optimization for Transient Servers. To utilize the economic advantage transient servers bring, researchers have extensively studied how to effectively running specific applications on cloud transient servers without making many modifications to the applications [38]. The targeted application includes web services [38, 39], big data analytics [40–43] and other memory-intensive applications [44]. These studies proposed system-level techniques such as dynamic checkpointing to achieve fault-tolerance, and devised transient-aware resource managers [45, 46]. Our work provides different perspective of transient-aware modeling for distributed training, an increasingly important cloud workload.

VIII. Summary

We explored the characteristics of and key factors impacting *distributed training on transient servers*. We envision that our measurements will lead to efficient GPU resource usage for deep learning practitioners by, for example, providing the foundations for predicting heterogeneous cluster training speed and detecting performance bottlenecks. We chose three commonly-used GPU servers, representative and custom CNN models for measuring and modeling performance. We found that simple regression models have adequate prediction accuracy, even for clusters with different GPU servers and when training CNN models with diverse characteristics. Additionally, we demonstrated that the overhead of commonly used fault-tolerance mechanisms (i.e., model checkpointing) can be predicted with high accuracy and the associated impact can be directly added to the predicted training speed. Lastly, we explored potential use cases of our performance modeling including detecting and mitigating performance bottlenecks.

References

[1] S. Shi *et al.*, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th DASC/PiCom/DataCom/CyberSciTech*, 2018.

[2] F. N. Iandola *et al.*, "Firecaffe: Near-linear acceleration of deep neural network training on compute clusters," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*.

[3] H. Cui *et al.*, "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server," in *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*.

[4] Amazon, "EC2 Spot Instances," https://aws.amazon.com/ec2/spot/.

[5] Google, "Preemptible VM Instances," https://cloud.google.com/compute/docs/instances/preemptible.

[6] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[7] S. Li *et al.*, "Speeding up Deep Learning with Transient Servers," in *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC'19)*.

[8] A. Vaswani *et al.*, "Tensor2tensor for neural machine translation," *arXiv:1803.07416*, 2018.

[9] F. Yan *et al.*, "Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15)*.

[10] H. Qi *et al.*, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

[11] Y. Peng *et al.*, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth European Conference on Computer Systems (Eurosys'18)*.

[12] S.-H. Lin *et al.*, "A model-based approach to streamlining distributed training for asynchronous sgd," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018.

[13] H. Zheng *et al.*, "Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training," in *Proceedings of the 48th International Conference on Parallel Processing*. ACM, 2019.

[14] A. Krizhevsky *et al.*, "Cifar-10," http://www.cs.toronto.edu/~kriz/cifar.html, 2017.

[15] K. He *et al.*, "Deep residual learning for image recognition," *arXiv:1512.03385*, 2015.

[16] X. Gastaldi, "Shake-shake regularization," *arXiv:1705.07485*, 2017.

[17] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012.

[18] S. Kundu *et al.*, "Modeling virtualized applications using machine learning techniques," in *ACM Sigplan Notices*, vol. 47. ACM, 2012.

[19] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, 2005.

[20] Google, "Instance life cycle," https://cloud.google.com/compute/docs/instances/instance-life-cycle, 2019.

[21] C. King, "stress-ng - a tool to load and stress a computer system," https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2017.

[22] C. Coleman *et al.*, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, vol. 100, 2017.

[23] X. Ouyang *et al.*, "Spotlight: An information service for the cloud," in *IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)*. IEEE.

[24] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv:1802.05799*, 2018.

[25] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv:1802.09941*, 2018.

[26] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[27] F. Research, "Caffe2," https://caffe2.ai, 2019.

[28] M. Research, "Microsoft cognitive toolkit," https://github.com/Microsoft/CNTK, 2019.

[29] A. Foundation, "Apache mxnet," https://mxnet.incubator.apache.org, 2019, accessed in 2019.

[30] S. Zhang *et al.*, "Deep learning with elastic averaging sgd," in *Advances in Neural Information Processing Systems 28*, 2015.

[31] J. Chen *et al.*, "Revisiting distributed synchronous sgd," *arXiv:1604.00981*, 2016.

[32] S.-X. Zou *et al.*, "Distributed training large-scale deep architectures," in *International Conference on Advanced Data Mining and Applications*. Springer, 2017.

[33] M. Jeon *et al.*, "Analysis of Large-scale Multi-tenant GPU Clusters for DNN Training Workloads," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (ATC'19)*.

[34] J. Jiang *et al.*, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017.

[35] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference (ATC'17)*.

[36] L. Luo *et al.*, "Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*.

[37] P. Xie *et al.*, "Orpheus: Efficient Distributed Machine Learning via System and Algorithm Co-design," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*, 2018.

[38] P. Sharma *et al.*, "Spotcheck: Designing a derivative iaas cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[39] A. Harlap *et al.*, "Tributary: spot-dancing for elastic services with latency slos," in *USENIX Annual Technical Conference (ATC'18)*.

[40] N. Chohan *et al.*, "See spot run: Using spot instances for mapreduce workflows," *HotCloud*, 2010.

[41] P. Sharma *et al.*, "Flint: batch-interactive data-intensive processing on transient servers," in *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*.

[42] S. Subramanya *et al.*, "Spoton: a batch computing service for the spot market," in *Proceedings of the sixth ACM symposium on cloud computing (SoCC'15)*.

[43] P. Ambati and D. Irwin, "Optimizing the cost of executing mixed interactive and batch workloads on transient vms," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, 2019.

[44] C. Wang *et al.*, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *Proceedings of the Twelfth European Conference on Computer Systems (Eurosys'17)*, 2017.

[45] P. Sharma *et al.*, "Portfolio-driven resource management for transient cloud servers," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2017.

[46] A. Harlap *et al.*, "Proteus: agile ml elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the Twelfth European Conference on Computer Systems (Eurosys'17)*.