Performance and Consistency Analysis for Distributed Deep Learning Applications

Danlin Jia*, Manoj Pravakar Saha†, Janki Bhimani†, and Ningfang Mi*
* Northeastern University, Boston, USA
† Florida International University, Miami, USA

2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC) | 978-1-7281-9829-3/20/$31.00 ©2020 IEEE | DOI: 10.1109/IPCCC50635.2020.9391566

***Abstract*—Accelerating the training of Deep Neural Network (DNN) models is very important for successfully using deep learning techniques in fields like computer vision and speech recognition. Distributed frameworks help to speed up the training process for large DNN models and datasets. Plenty of works have been done to improve model accuracy and training efficiency, based on mathematical analysis of computations in the Convolutional Neural Networks (CNN). However, to run distributed deep learning applications in the real world, users and developers need to consider the impacts of system resource distribution. In this work, we deploy a real distributed deep learning cluster with multiple virtual machines. We conduct an in-depth analysis to understand the impacts of system configurations, distribution typologies, and application parameters, on the latency and correctness of the distributed deep learning applications. We analyze the performance diversity under different model consistency and data parallelism by profiling run-time system utilization and tracking application activities. Based on our observations and analysis, we develop design guidelines for accelerating distributed deep-learning training on virtualized environments.**

I. Introduction

Deep learning is currently used to solve many unsolved problems in a plethora of disciplines. Due to the proliferation of data and the boost of model complexity (i.e., over millions of parameters), training time for a single DNN model can range from days to months [1]. Since long training time significantly slows down development and deployment of deep learning models, an array of distributed deep learning frameworks has been introduced to accelerate the DNN model training [2], [3]. Parameter server architecture [4] is one of the most common deep learning frameworks that distribute workloads to a set of workers and save model parameters in an array of parameter servers (PSs). Parameter server architecture supports two different model consistency policies, i.e., synchronous and asynchronous stochastic gradient descent (SGD) [5]. Synchronous SGD updates model parameters once, with all calculated gradients received from workers. Whereas, asynchronous SGD updates model parameters immediately when local gradients are received.

The issues of scalability and model consistency (i.e., how the updated weights are synchronized between PSs and workers) in distributed deep learning frameworks are evident. The common problem is to decide the number of workers and PSs, given limited computing resources. On top of that, users need to consider how workers and PSs communicate with each other to transfer data. Besides, how to configure the hyper-parameters of the deep learning application itself needs attention as well. There are many existing works [6]–[8] that have been done to solve these problems. However, the proposed solutions either affect the performance of deep learning applications or interfere with each other. For example, allocating more workers, which can increase throughput, may also slow down model convergence (e.g., accuracy improvements). Therefore, the problem of accelerating training without sacrificing prediction correctness turns out to be a complicated combinatorial optimization problem. Therefore, in this paper, we strive to investigate and understand the impacts of manipulations of system configurations, distribution topologies, and application parameters on performance and correctness of distributed DNN models. Particularly, we deploy a distributed deep learning cluster with multiple virtual machines and construct various experiments to observe the performance under various scenarios. We analyze how changes in one factor affect others and discuss the possible solutions to achieve high throughput and accuracy.

The major contributions of our work are as follows:

1) **Analyzing run-time system resource consumption**: We collect run-time system utilization and communication traces to characterize model consistency policies. We deliver insights about how synchronous and asynchronous SGD utilize system resources with respect to scheduling, data caching, IO activities, and network communications.

2) **Studying the impact of parallelism on training procedure**: We conduct experiments with different levels of parallelism varying the number of parameter servers and workers. We compare training time and model accuracy under different model consistency policies. We discuss essential concerns in deploying distributed deep learning clusters.

3) **Investigating the distribution of workloads**: We study the impact of the batch size on training time and accuracy. We explore the relationship between workloads on workers and parameter servers. We investigate the unavoidable CPU idleness on individual workers due to the workload distribution and network latency.

In the rest of this paper, we discuss the background and motivation in Sec. II. We present the details of experiment environments and system architecture in Sec. III. The experimental results and analysis of observations are shown in Sec. IV. We present the related work in Sec. V. The conclusion and our future work are discussed in Sec. VI.

II. Background and Motivation

In this section, we introduce the workflow of deep learning training procedure. We investigate how the parameter server architecture distributes DNN models and workloads, and benefits from parallel Stochastic Gradient Descent (SGD). We study and explore the challenges different types of model consistency methods, i.e., synchronous and asynchronous.

[1]This work was partially supported by the National Science Foundation Career Award CNS-1452751 and National Science Foundation Awards CNS-2008324 and CNS-2008072.

A. Back-propagation

A DNN model is a set of stacked layers to learn a complicated function F from data so that it can be applied to new input (data) for predicting the output. Back-propagation algorithm is used to train a DNN model , which consists of three steps: 1) **forward path** that computes the associated model parameters of the layers of network sequentially, 2) **backward path** that performs gradient computations conversely with the chain rule, and 3) **weights update** that updates model parameter weights for each layer with the manipulated gradients [9], [10].

Back-propagation repeats the above three steps iteratively until reaching the desired accuracy. We denote a cycle of three steps as an *iteration*. Each iteration is responsible for processing a portion of training data that is called a *batch*. An *epoch* contains a set of iterations, which goes through the whole training data set. Thus, the number of iterations equals to the training data set size divided by the batch size. The total training time is the summation of the duration of all epochs, where the number of epochs is predefined or determined based on the desired accuracy. Additionally, in an iteration, each worker gets a portion of batch data to learn and compute their local gradients. We call such a portion of batch data as *mini-batch*, and the number of mini-batches is equal to the number of workers in the cluster.

B. Distributed Deep Neural Networks

Millions of parameters define a typical DNN model. Training such DNN models from a large amount of data is computationally expensive and may take weeks to train on a single machine. If the model or the training data is too large, then they may not fit in the memory of a single machine. Thus, one solution for addressing such situations is to spread the model and/or data across multiple machines.

Model Parallelism: Model parallelism refers to the distributed training method that trains different layers of the model across different machines. In a typical scenario, each layer of the model can be fitted into the memory of a single machine and is trained thereof. The forward and backward paths then needs to be proceed sequentially among these machines. Hence, model parallelism does not provide any speed-up benefit. It should only be used when the model is too large to fit into a single machine [11].

Data Parallelism: In this method, data is distributed across multiple machines. Each machine trains the whole model only on a subset of the data that is local. In every training iteration, the weights of model parameters are aggregated and updated in a centralized or decentralized fashion. In contrast to model parallelism, data parallelism can help training process converge faster by learning the subset of data in parallel [11] on multiple machines.

C. Synchronous and Asynchronous PS Architecture

Stochastic gradient descent (SGD) is an algorithm that uses randomly selected data samples as input to train the DNN model in each epoch. SGD enables distributed DNN which provides considerable training speed acceleration, as it takes advantage of parallel data processing. However, how to share model parameters and maintain model consistency across different nodes is an issue. Parameter server architecture was thus proposed to provide an efficient mechanism for aggregating and synchronizing model parameters among multiple nodes [4]. As shown in Fig. 1, the parameter server framework consists of **parameter servers (PSs)** and **workers**. The parameter server framework supports parallel data processing by dividing training data into partitions (or mini-batches) and distributing mini-batches across workers. Each worker has a copy of the DNN model, where the back-propagation is executed. The calculated gradients are sent from each worker to the PSs via a *push* call. After updating the weights, the PSs issue a *pull* request to transfer the latest updated parameter weights to workers.

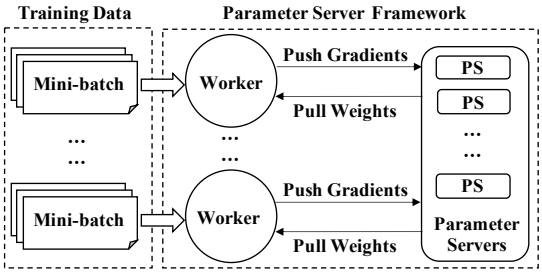


Fig. 1: Parameter server architecture.

Two policies that are commonly used for model consistency in the parameter server framework are Synchronous SGD and Asynchronous SGD.

Synchronous SGD (Sync-SGD): Figure 2 shows the events during synchronous SGD. In a data-parallel setup with Sync-SGD, each worker (e.g., Worker 1, ..., Worker N) fetches the latest weights from PS(s), computes and pushes its local gradients (e.g., $\Delta w_i^{(1)}$) to PS(s). In case of multiple PSs, each PS is responsible for updating a subset of model parameters. Upon receiving the local gradients (e.g., $\Delta w_i^{(1)}$, $\Delta w_i^{(N)}$) from every worker, PS(s) average the gradients and use the corresponding gradient subset to calculate the new parameter weights (shaded regions in Figure 2). All the workers are updated together with the new parameter weights (e.g., w_{i+1}) in a synchronized fashion by a call of *pull*. Under this Sync-SGD policy, parameter server(s) compute the new parameter weights only after they receive local gradients from *all* workers, and synchronize the new weights after computation. Thus, all workers have to start next iteration synchronously with the same new parameter weights.

Asynchronous SGD (Async-SGD): In Async-SGD, every worker independently pushes their local gradients to parameter server(s) and pulls the latest parameter weights immediately at the end of each layer. Figure 3 shows a simplified chain of events during Async-SGD, where a PS partially updates (shaded regions under PS) parameter weights upon receiving local gradients from a worker and each worker pulls the latest weights from PS asynchronously. Consequently, a worker in Async-SGD can continue to train the model without waiting for other workers. This speeds up the entire learning process

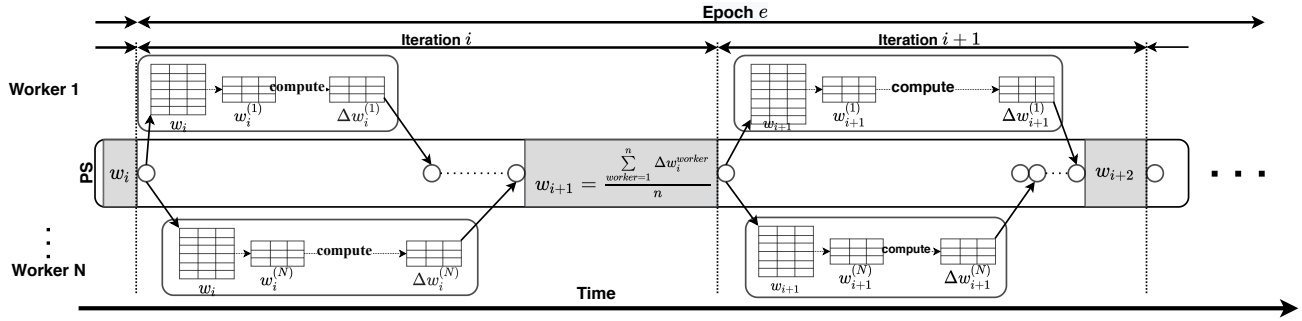


Fig. 2: Typical training sequence in Sync-SGD with data parallelism.

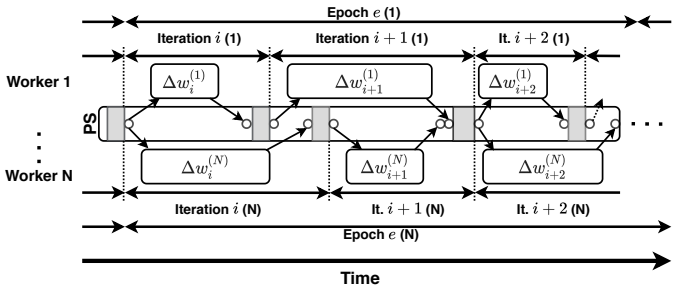


Fig. 3: Typical training sequence in Async-SGD with data parallelism.

but increases the calculation load on parameter servers as well.

D. Challenges in Model Consistency

Researchers have studied model consistency using mathematical formulations [12], [13] and practical experiments [14], [15], with assumptions that the physical computing resources are over-provisioned. These works are conducted over sufficient physical resources (CPU, memory, and network bandwidth). However, the resources are limited and shared among many applications in most of the real distributed deployment infrastructures. Thus, *it is not trivial to answer the question of how to obtain the maximum throughput without sacrificing accuracy with limited physical computing resources.* Specifically, we have identified the following main challenges that exist in real distributed deep learning clusters.

Communication Overhead: Parameter server architecture suffers from communication overhead, as each weights update needs to trigger a bi-directional data communication. Such communication overhead heavily depends on network bandwidth and latency. This problem is even more intensive in Sync-SGD because each worker needs to pull the new parameter weights synchronously from PSs at the same time which might lead to network congestion.

Staleness: Staleness happens when some of the workers compute gradients using the parameter weights that may be several gradient steps behind the latest updated parameter weights. For example, parameter weights in PSs have already been calculated at vector time N. However, PS later receives some local gradients that were computed at vector time $N-t$ (where $N > t > 0$) due to low computation power at the corresponding worker or network communication delay. We say the staleness equals to t. Updating parameter weights based on those stale gradients then induces loss of accuracy as well as computational resource dissipation. It has been found that Async-SGD has a high possibility of incurring staleness [14].

Batch and Mini-batch Size: Deep learning frameworks iterate over input data (such as images) in batches. The batch size has an impact on throughput and accuracy, as it decides the extent to aggregate gradients [16]. However, in distributed deep learning, a batch is evenly distributed across workers as a set of mini-batches. If the worker number is fixed, then a smaller batch size leads to more iterations in an epoch, which can increase the frequency for weight updates in Sync-SGD and thus add more computation load on PSs. On the other hand, if the batch size is fixed, then increasing the number of workers increases data parallelism but also decreases the mini-batch size. Consequently, for Async-SGD, the frequency for updating weights in an iteration is increased, which increases the computation load on PSs as well. Therefore, we claim that developers need to take into account the tradeoff between data parallelism and computation load on PS as well as the model consistency methods when deciding the sizes of batch or mini-batch.

Idleness: There exists unavoidable idleness of computing resources, as the latter iteration requires the results from the former iteration. Although communication and computation can be overlapped in the hidden layer level [6], the forward path of the latter iteration needs an input of the parameter weights updated at the last layer from the former iteration. Figure 4 shows an example of a model with three layers. The operations of push, update, and pull of layer 3 can execute simultaneously with the backward path of layer 2 (L2_bw). However, such simultaneous execution works for all layers except layer 1, which introduces an idle period after the backward path of layer 1 (L1_bw). This is because the forward path of next iteration (e.g., iteration N+1 in Figure 4) cannot start without receiving the updated parameter weights from iteration N. We thus claim that the solid dependency between two iterations causes idleness and limits the throughput, which actually exhibits under both Sync- and Async-SGD policies. We further notice that different system configurations and hyper-parameters have various impacts on the idleness.

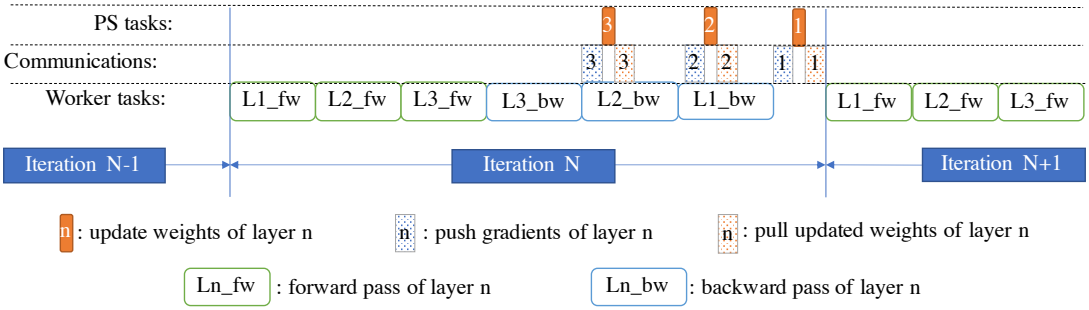


Fig. 4: Example of idleness of computing resources on workers.

III. Architecture

In this section, we describe our system architecture and experiment environment, including cluster configurations and workload specifications.

A. System Architecture

In this work, we develop a parameter server architecture on a virtualized CPU cluster. Fig. 5 shows our implementation of a real distributed deep learning cluster. On top of our CPU-based infrastructure, the virtual machine (VM) hypervisor resides on the host OS, which monitors five homogeneous VMs. The guest OS preserves computing resources to VMs. Our cluster specifications are summarized in Tab. I. Specifically, we

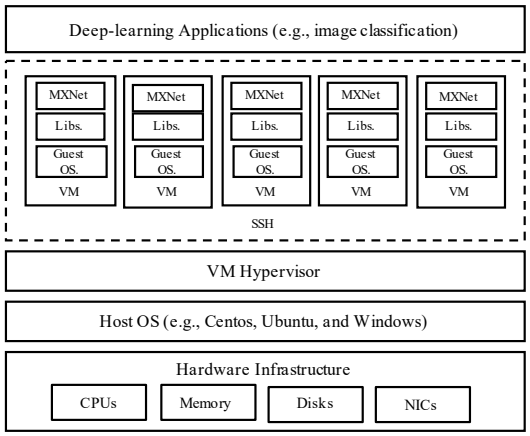


Fig. 5: System architecture.

install a distributed deep learning framework, named Apache MXNet [17], to support distributed learning in our virtualized cluster. The reason we choose this particular framework is that previous works [17], [18] have claimed that MXNet promises the same accuracy with shorter training time, compared with other frameworks, e.g., TensorFlow [19] and Caffe [20]. MXNet supports both CPU and GPU deep learning. To compile MXNet with CPU nodes in our cluster, we add the BLAS (Basic Linear Algebra Subprograms) library and the LAPACK (Linear Algebra Package) library in each VM for numerical computations and other operations. We also construct the deep learning cluster through cluster managers (e.g., Yarn and Kubernetes) that help manage resource allocation among VMs. In our cluster, VMs are communicated with each other via Secure Shell (SSH).

TABLE I: System specifications.

Component	Specs
Host Server	Dell PowerEdge R420
Host Processor Speed	2.20GHz
Host Memory Data Rate	1333 MHz
Host OS	Ubuntu 18.04.2 LTS
Host Disk	NVMe SAMSUNG PM963 2TB
Guest OS	Ubuntu 16.04.5 LTS
Per VM Memory Capacity	8GB
Per VM CPU number	8
Network Protocol	Secure Shell (SSH)

B. Parameter Server Deployment

We deploy our distributed deep learning cluster on virtual machines. We use one virtual node as a cluster manager, where deep learning workloads are submitted, and the others as working nodes (or executors) to execute deep learning tasks. Fig. 6 shows our deployment architecture.

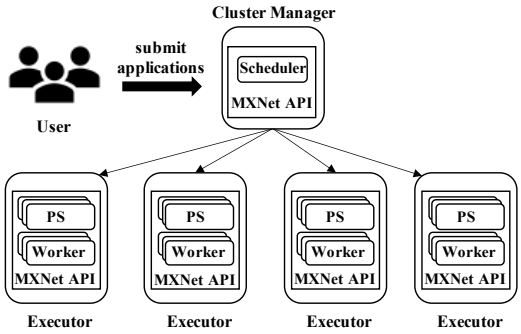


Fig. 6: Parallel deployment of MXNet instances for distributed deep learning.

The MXNet API needs to be installed on each node, including both the cluster manager and executors. Users submit applications to the cluster manager. The MXNet scheduler residing on the cluster manager then spawns a number of PSs and workers based on the user specification. MXNet instantiates PS or worker processes cyclically across all executors. As shown in Fig. 6, four executors (each runs on one VM) in our cluster have the same number of PSs and workers. For

example, we configure the MXNet scheduler to launch one PS and two workers evenly on each executor (or VM). Thus, we have a total of four PSs and eight workers in our infrastructure. We find that to avoid unbalanced workloads, it is necessary to launch PSs and workers that are multiple of four in our case.

C. Workload Specifications

In this work, we use image classification benchmarks with the Cifar10 [21] training data set that contains 60k 32x32 color images of 10 different classes of objects, such as vehicles, dogs, and cats. We conduct experiments on two classic DNN models, i.e., ResNet-18 [22] and Inception-v1 [23]. These DNN models have multiple layers to learn model parameters and can be configured with different hyper-parameters (e.g., learning rate and momentum). Particularly, ResNet-18 [22] is a convolutional neural network with 18 layers; and Inception-v1 [23] is also a convolutional neural network with 27 layers. Both of these models contain millions of parameters. We apply an adaptive learning rate based on the number of weight updates that have been performed. We set the momentum to 0.9 (i.e., a common choice to reduce noisy data by averaging gradients) and feed each worker with the same number of images in one batch. We evaluate the DNN training performance with the metric of the top-1 accuracy. The accuracy is the measure of resemblance between the predicted and actual labels. Particularly, the top-1 accuracy means that given the actual label, what is the possibility that the top 1 predicted label is the same as the actual.

IV. Experiments and Analysis

In this section, we describe our experimental methodology to systematically study the performance and consistency of distributed deep learning on a virtualized framework. Specifically, we first understand the system resource consumption of distributed deep learning applications under different updating methods. Second, we investigate the impact of data parallelism and parameter distribution on training time and accuracy. Thirdly, we study the impact of various batch sizes on training time and accuracy. Finally, we dive into the hidden layer level to investigate the impact of configurations on computing resource idleness.

A. Resource Utilization

In this experiment, we set four PSs and eight workers. We fix the batch size to 128. We summarize the profile statistics for Inception-v1 and ResNet-18 under Sync- and Async-SGD in Tab. II. The memory and CPU utilization is provided by *dstat*, and MXNet's data profiler collects the rest of the data. As DNN training is an iterative procedure, where each iteration has the same workload characteristics, we focus on both overall statistics and average statistics over iterations.

We first measure the system resource utilization, regarding CPU and memory, and calculate the average resource utilization over four VMs. As shown in Tab. II, both Sync- and Async-SGD are resource-intensive, and their resource utilization is similar. We track the I/O activities regarding reads and writes and observe that *more write requests are triggered than read ones*. We anticipate the reason is that in distributed deep learning, we persist checkpoints after each epoch that introduces many write I/Os. The relatively low number of reads is because the dataset (i.e., Cifar10) is small enough to be cached in the memory of each node in our cluster.

TABLE II: Profiling statistics (Tra. = Training, util. = utilization, R/W num. = Read/Write number, WU = Weights update, Incp. = Inception-v1, Res. = ResNet-18).

Profile	**Sync-SGD**		**Async-SGD**	
DNN Model	Incp.	Res.	Incp.	Res.
CPU util.	95%	96%	96%	98%
Memory util.	65%	98%	65%	96%
R/W num.	4/3500	80/1600	3/1500	50/2000
WU num.	931	4067	7448	32536
WU time (ms)	6.327	5.659	10.117	9.279
Ave. pull (ms)	42	53	52	63
Ave. push (ms)	1114	2534	86	104

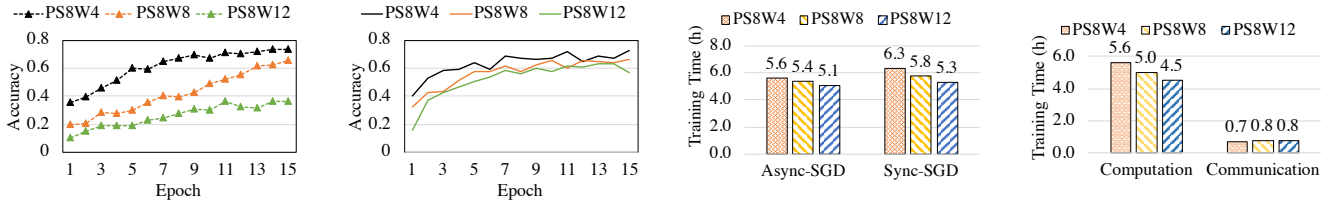
We analyze the traces of the application activities performed by PSs. We find that as workers send gradients and receive weights independently in Async-SGD, the number of weight updates (i.e., WU num.) is N times of that in Sync-SGD, where N is the number of workers (i.e., 8 in our setup). Moreover, the average time (i.e., WU time) consumed by PSs to compute new weights for Sync-SGD is shorter than that for Async-SGD. This indicates that if we do not schedule a sufficient number of PSs in the deep learning cluster, then PSs can be overloaded and degrade the overall performance, especially in Async-SGD.

We further collect the communication traces to calculate the average duration (i.e., Ave. pull or Ave. push) of a *pull* or *push* call for a worker. Both Sync- and Async-SGD have a similar average pull duration. However, Sync-SGD has an average push duration ten times longer than that of Async-SGD. This is because Sync-SGD needs to wait for all workers to finish their computation tasks before calling *push*. *This observation indicates that compute efficiency, network latency and bandwidth should be well balanced across multiple workers in Sync-SGD.* Long lag in the operations of a worker or communication congestion can deteriorate the performance of distributed deep learning.

B. Data Parallelism

Distributed deep learning takes advantage of parallel data-processing to accelerate training. Nonetheless, as the degree of parallelism increases, the accuracy might be affected due to synchronization overhead [15]. Therefore, we conduct experiments with different numbers of workers to investigate the impact of data parallelism on distributed deep learning. We fix the number of PSs (i.e., 8) but change the number of workers to 4, 8, and 12.

Fig 7-(a) shows the top-1 accuracy of ResNet-18 across training epochs for different numbers of workers using Async-SGD. *We observe that for Async-SGD (lines with triangle markers), the accuracy drops significantly on an average across all epochs by more than 0.2 when the number of parallel workers is increased from 4 to 12.* Moreover, the drop in accuracy increases further for higher epochs. The final accuracy of Async-SGD drops by around 0.35 as the



(a) Async-SGD (b) Sync-SGD (c) Overall Training Time (d) Sync-SGD Training Time

Fig. 7: Impacts of data parallelism on accuracy and training time.

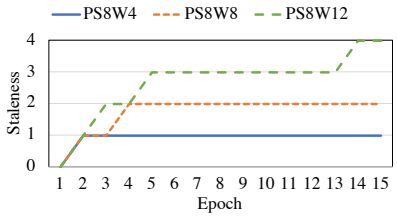


Fig. 8: Run time staleness for different data parallelism.

worker number increases. However, such an impact of an increasing number of workers on Sync-SGD is limited, which only exhibits a decrease of 0.08, see solid lines in Fig. 7-(b).

This observation is consistent with our discussion of staleness in Sec. II. Async-SGD experiences higher staleness with the number of workers increases. Out of many workers, some compute gradients using model parameters that may be several iterations behind the most updated model parameters, which consequently reduces the accuracy. Furthermore, as the staleness is proportional to the number of workers, more workers mean a higher possibility of staleness. To verify our observation, we plot the run time staleness across different worker numbers for Async-SGD in Fig. 8. We observe that the staleness of 12 workers (i.e., PS8W12) reaches 4 at epoch 13, while the maximum staleness of 8 and 4 workers is 2 and 1, respectively. The number of staleness is expected to increase if we run more epochs.

We also compare the training time for ResNet-18, as shown in Fig. 7-(c). *When we change the worker number from 4 to 12, the training time decreases accordingly for both Sync- and Async-SGD because high parallelism offers fast data processing speed.* We further breakdown the training time to communication time and computation time for Sync-SGD, see Fig. 7-(d). We denote the communication time as to how long workers are waiting for PSs to send the latest updated parameters, and denote the computation time as the time that workers spend to finish the forward and backward paths. We observe that the computation time decreases as the number of workers increases, while the communication time across different worker numbers is similar, as shown in Fig. 7-(d). We also notice that Async-SGD takes less time to train the same model, Fig. 7-(c). The reason is that Async-SGD updates model parameters with each worker asynchronously without waiting for the other. Although we observe that Async-SGD has higher communication congestion in Tab. II, the overall training time of Async-SGD is shorter than Sync-SGD, as the bottleneck of the performance is worker's throughput. The reason we do not breakdown training time for Async-SGD is that the training behavior of each worker is heterogeneous, which makes the overlap between computation and communication non-deterministic and hard to observe at the system level.

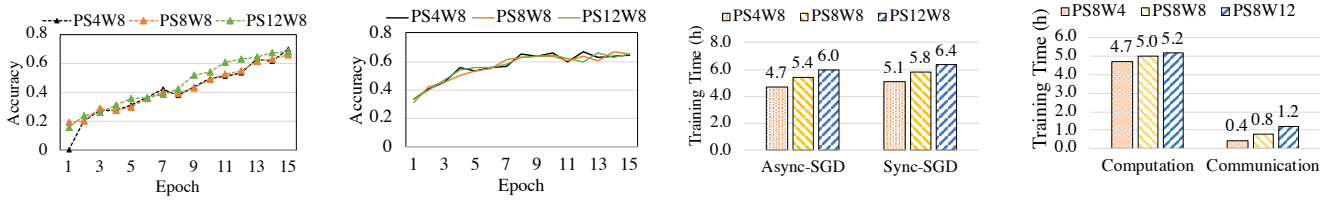
C. Parameter Distribution

We conduct similar experiments with different PS numbers. Our observation is that various PS numbers have the same convergence trends, which means parameter distribution has limited impacts on model accuracy under both Sync- and Async-SGD algorithms, see Fig. 9-(a) and (b). We anticipate this is because model accuracy is determined by gradients calculated on workers, not by weights-updating executed on PSs. The mathematical proof aligning with our real systems observation can be found in [12].

Fig. 9-(c) further shows the overall training time. When the PS number increases, the training time increases for both Sync- and Async-SGD. We break the overall training time to computation and communication for Sync-SGD, as shown in Fig. 9-(d). We observe that both computation and communication time increases. This is because increasing the PS number incurs higher communication overhead among PSs and increases CPU intensity on each VM as more PS processes are launched. Furthermore, because PS uses the distributed key-value store fundamentally, it may take longer for each PS to query and find the desired potion of parameters if the number of PSs increases. We also observe that Async-SGD has shorter training time than Sync-SGD, which is consistent with the observation in Fig. 7-(c).

D. Batch Size

We change batch size from 16 to 512 to train Inception-v1 under both Sync- and Async-SGD. We set 4 workers and 4 PSs. The training time (left side y-axis) and accuracy (right side y-axis) after three epochs are shown in Fig. 10. *Training time decreases as we increase the batch size.* However, this inverse relationship is not linear. Training time improvement gradually decreases. As shown in Fig. 10, the maximum relative improvement rates are 51% and 54% for Sync- and Async-SGD, respectively, when the batch size increases from 16 to 32. Smaller batch size leads to more updates within an epoch, which increases the workload on PS and results in higher execution time. This observation indicates that when



(a) Async-SGD (b) Sync-SGD (c) Overall Training Time (d) Sync-SGD Training Time

Fig. 9: Impacts of parameter distribution on accuracy and training time.

we set a smaller batch size, it is beneficial to allocate more PSs to the cluster. We also observe that Sync- and Async-SGD have similar training time for each epoch across different batch sizes. However, Async-SGD achieves marginally lower training time because of the asynchronous weight updates.

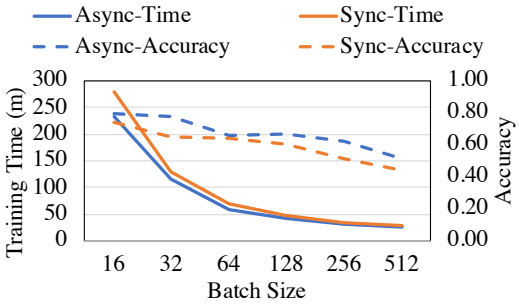


Fig. 10: Training time and accuracy of Inception-v1.

We further observe that *larger batch size leads to lower accuracy*. Particularly, when we set batch size as 16, the accuracy after three epochs can reach 0.79 and 0.74 for Sync- and Async-SGD, respectively. However, when we increase the batch size to 512, the accuracy drops to 0.52 and 0.44. We explain that when batch size increases, the number of iterations within an epoch decreases, which consequently cannot provide enough weight updates to achieve high accuracy. This can also explain why Async-SGD has higher accuracy than Sync-SGD after three epochs, as the number of updates of Async-SGD is N times as many as that of Sync-SGD.

Our experimental results indicate there is a trade-off between training time and accuracy across different batch sizes. Although the larger batch size provides faster training speed, it inversely degrades accuracy. When we have a large batch size (e.g., 512), we have to run more epochs to obtain the same accuracy as the small batch size (e.g., 16), which unfortunately may expand the total training time.

E. Idleness of Computing Resources

Distributed deep learning applications speedup training procedure by resolving task dependencies and parallelizing communication and computation of consecutive layers. However, there is a strong dependency between the backward path of the previous iteration with the forward path of the next iteration. The next iteration requires the output of the previous iteration as an input. While the next iteration is waiting for the completion of the previous iteration, the idleness of computing resources on the worker side unavoidably exhibits.

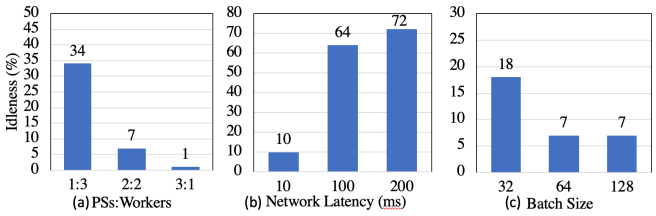


Fig. 11: Impact of configurations on idleness.

To investigate idleness under different configurations, we use the idleness ratio, which is the ratio of the idle period where workers are waiting for updated model parameters from PSs to the training time of the current iteration, to measure the percent of idle computing resources. We first conduct experiments across different ratios of PSs to workers (e.g., 1:3, 2:2, and 3:1) on Inception-v1. Fig. 11-(a) shows that *the idleness decreases as the ratio of PS to worker increases* (i.e., having less workers than PSs). This is because the workload on each worker increases when the number of workers is reduced. Then, the idle period becomes relatively shorter when comparing with the calculation time on workers, therefore the idleness ratio deceases. We further change the batch size to 32, 64, and 128 on Inception-v1, as shown in Fig. 11-(b). Similarly, *the idleness decreases upon increasing the batch size* because more workload is assigned to each worker when the batch size increases. The last factor that may affect idleness is network latency that can incur different communication time. As shown in Fig. 11-(c), we manually change the network latency in virtual machine configurations from 10ms to 100ms and 200ms. *The idleness increases dramatically from* 10% *to* 72% *upon increasing the network latency from 10ms to 200ms.* This is because longer network latency causes longer waiting (idle) time on workers.

The observed idleness is evident for both Sync- and Async-SGD. We can shrink the idleness by increasing the workload on workers and deploying the deep learning cluster with low network latency. However, we remark that such idleness cannot be completely eliminated due to the aforementioned strong dependency between two consecutive iterations, no matter Sync- or Async-SGD is used.

V. Related Work

Real-world deployments of distributed deep learning infrastructure are highly heterogeneous. The different categories

of methods introduced to train DNN models efficiently on distributed infrastructure include - model consistency (e.g., synchronous, or asynchronous), parallelism of computation and communication (e.g., scheduling), parameter distribution (e.g., centralized PS, or decentralized all-reduce), parameter compression (e.g., quantization), and optimization algorithms (e.g., hyper-parameter search) [5], [11].

Although Sync-SGD with PS ensures model consistency, the synchronization overhead and idle time wasted by waiting for slow workers hinder scaling [11]. Recent work suggests the use of backup workers to accelerate Sync-SGD [15]. Other researchers attempt to circumvent the above issues by utilizing Async-SGD, but introduces model inconsistency due to stale gradients and reduces convergence speed. Using staleness-aware Async-SGD can reduce convergence time [13].

Another research direction is to improve training efficiency through efficient scheduling of communication and computation. One interesting work [6] proposes the parameter slicing and priority-based parameter update to reduce communication delay. [2], [7] propose to reduce training time by dynamically allocating computational resources in a cluster during runtime based on resource-performance models. Some researchers also examine hyper-parameter based optimization techniques. These techniques include adaptive learning rate in a PS-worker architecture with Async-SGD, and dynamic learning rates to account for heterogeneous workers [3].

VI. Conclusion

This paper begins up posing a simple question for investigation: "how to obtain the maximum throughput without sacrificing accuracy with limited physical computing resources?". By conducting experiments in a distributed deep learning cluster, We revealed that training of deep learning models may have drastically varying behaviors of model accuracy and training time with respect to different model consistency methods, application configurations, and system specifications. We thus delivered insights and guidelines of how to take full advantage of distributed deep learning clusters in deploying deep learning applications. For example, there is a tradeoff between data parallelism and computation load on PS, which needs to be considered when configuring the batch and mini-batch sizes. Idleness on workers is caused by the strong dependency between consecutive iterations, which exhibits under both Sync- and Async-SGD and should be taken into account for improving resource utilization. In the future, we plan to leverage our insights to design new techniques and solutions that aim to speed up the training procedure and meanwhile maintain high model accuracy, and conduct experiments on more complicated models with larger datasets.

References

[1] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100, 2018.

[2] Y. S. L. Lee, M. Weimer, Y. Yang, and G.-I. Yu, "Dolphin: Runtime optimization for distributed machine learning," in *Proc. of ICML ML Systems Workshop*, 2016.

[3] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 463–478. [Online]. Available: https://doi.org/10.1145/3035918.3035933

[4] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, vol. 6, 2013, p. 2.

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3320060

[6] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," *CoRR*, vol. abs/1905.03960, 2019. [Online]. Available: http://arxiv.org/abs/1905.03960

[7] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190517

[8] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze, "Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud," in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 82–97.

[9] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[10] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[11] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," *arXiv preprint arXiv:2003.06307*, 2020.

[12] M. Zinkevich, A. J. Smola, and J. Langford, "Slow learners are fast," in *NIPS*, 2009.

[13] A. Odena, "Faster asynchronous sgd," *ArXiv*, vol. abs/1601.04033, 2016.

[14] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *IJCAI*, 2017.

[15] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," in *International Conference on Learning Representations Workshop Track*, 2016. [Online]. Available: https://arxiv.org/abs/1604.00981

[16] P. Radiuk, "Impact of training set batch size on the performance of convolutional neural networks for diverse datasets," *Information Technology and Management Science*, vol. 20, pp. 20 – 24, 2017.

[17] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *ArXiv*, vol. abs/1512.01274, 2015.

[18] X. Zhang, Y. Wang, and W. Shi, "pcamp: Performance comparison of machine learning packages on the edges," *ArXiv*, vol. abs/1906.01878, 2018.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM '14*, 2014.

[21] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/ kriz/cifar.html

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.