Developing a Loss Prediction-based Asynchronous Stochastic Gradient Descent Algorithm for Distributed Training of Deep Neural Networks

Junyu Li
j.li.9@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Ligang He*
ligang.he@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Shenyuan Ren
shenyuan.ren@physics.ox.ac.uk
University of Oxford
Oxford, United Kingdom

Rui Mao
mao@szu.edu.cn
Shenzhen University
Shenzhen, China

ABSTRACT

Training Deep Neural Network is a computation-intensive and time-consuming task. Asynchronous Stochastic Gradient Descent (ASGD) is an effective solution to accelerate the training process since it enables the network to be trained in a distributed fashion, but with a main issue of the delayed gradient update. A recent notable work called DC-ASGD improves the performance of ASGD by compensating the delay using a cheap approximation of the Hessian matrix. DC-ASGD works well with a short delay; however, the performance drops considerably with an increasing delay between the workers and the server. In real-life large-scale distributed training, such gradient delay experienced by the worker is usually high and volatile. In this paper, we propose a novel algorithm called LC-ASGD to compensate for the delay, basing on Loss Prediction. It effectively extends the tolerable delay duration for the compensation mechanism. Specifically, LC-ASGD utilizes additional models that reside in the parameter server and predict the loss to compensate for the delay, basing on historical losses collected from each worker. The algorithm is evaluated on the popular networks and benchmark datasets. The experimental results show that our LC-ASGD significantly improves over existing methods, especially when the networks are trained with a large number of workers.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Computer vision**; **Artificial intelligence**.

KEYWORDS

Neural Networks, Distributed Training, Machine Learning

*Corresponding author.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404432

ACM Reference Format:
Junyu Li, Ligang He, Shenyuan Ren, and Rui Mao. 2020. Developing a Loss Prediction-based Asynchronous Stochastic Gradient Descent Algorithm for Distributed Training of Deep Neural Networks. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3404397.3404432

1 INTRODUCTION

Deep learning attracts attentions increasingly because of its impressive effectiveness in the areas of image recognition and speech processing [12, 20, 24]. As the size of the training dataset and the complexity in the architecture of the training network raises, training Deep Neural Network (DNN) with Stochastic Gradient Descent (SGD) on a single machine is getting harder. The demands for working on large-scale datasets and models motivate the research of distributed training strategy [1, 3, 13]. Synchronous Stochastic Gradient Descent (SSGD) and Asynchronous Stochastic Gradient Descent (ASGD) [7] are two popular solutions for this purpose.

In SGD, the neural network runs on a single machine and the gradients, which are used to update the model, are computed every time after a batch of data is processed by the network. Different from SGD, a set of workers is used in SSGD. Each worker holds the same model and computes the gradients synchronously. The gradients are then averaged on a parameter server for weight updating. However, there is a synchronous barrier in SSGD training method. The gradient averaging operation has to wait for all workers to finish their computations. The training process will slow down if any worker is delayed.

The ASGD breaks the synchronous barrier to accelerate the distributed training process. In the ASGD, each worker individually computes the gradient on the model retrieved from a parameter server and updates the model in the parameter server asynchronously [18]. Nevertheless, the main drawback of the ASGD is the delayed updating [19]. It dues to the fact that the gradients computed by each worker are commonly based on different states of the network. In other words, the workers cannot always run with the latest state of the network since the weights are continually updated, and each worker uses its retrieved version of the network to compute the gradients independently.

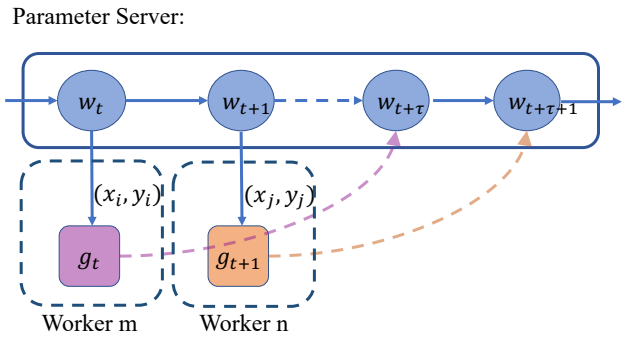


Figure 1: ASGD weight updating procedure

Due to such drawback of the ASGD, [27] proposed a delay compensation algorithm named DC-ASGD. It utilizes a cheap approximation of the Hessian matrix to compensate for the delay in the gradient computation performed by local workers. The approximation can produce a better result than both the ASGD and the SSGD. However, the effective approximation is limited as the DC-ASGD is utilizing the second-order partial derivatives, resulting in the DC-ASGD only working well for a short period of delay. The training performance in the situation of high delays cannot be guaranteed.

In this paper, we propose a novel method, called Asynchronous Stochastic Gradient Descent with Loss Compensation (LC-ASGD), to release the restriction in the DC-ASGD. Our method LC-ASGD works similarly as ASGD in the sense that there is no barrier for workers, so that enables each worker to update the network once it finishes the local training. The LC-ASGD, which differs from ASGD, applies additional Recurrent Neural Networks (RNN) to predict the loss in the existence of delay (i.e., compensate the loss due to the delay). It allows workers to use more accurate loss values to compute the gradients. Comparing with DC-ASGD, the loss compensations are not limited to the effective duration of the second-order approximation. Rather, the accuracy of loss compensation is determined by the performance of the predictor.

Moreover, the default batch normalization [9] designed for single machine training does not work very effectively in distributed training. To further improve the performance of distributed training, we extend the default Batch Normalization (BN) scheme to Asynchronous Batch Normalization (Async-BN). We have conducted comprehensive experiments, in which the results show that: (1) the LC-ASGD always produces better results than SSGD, ASGD and DC-ASGD, especially when there are a large number of workers in the system; (2) the model accuracy achieved by LC-ASGD is very close to the accuracy obtained by the sequential SGD when the number of workers is large, and is even better than SGD when the number of workers is small.

2 MOTIVATIONS

SSGD is a straightforward distributed implementation of SGD. The workloads are simply split among the workers at every iteration. The parameter server collects and averages the gradients calculated by the workers to update the weights of the neural network.

$$\omega_{t+1} = \omega_t - \gamma \cdot \frac{1}{M} \cdot \sum_{j=1}^{M} (\frac{1}{b} \sum_{i=1}^{b} \nabla_{\omega_t} \ell(f_{\omega_t}(x_{i,j}), y_{i,j})) \quad (1)$$

Formula 1 represents how SSGD works, where ω_t is the network weight at time t, γ is a learning rate, M is the number of workers, and b is the batch size of the data. ℓ is a loss function for evaluating the difference between the network output $f_{\omega_t}(x_{i,j})$ and the label $y_{i,j}$ corresponding to the input $x_{i,j}$. The workers compute gradients on their own, while the server updates the weight of the training network by taking as input the gradients computed by each worker.

The drawback of SSGD is that the server has to synchronize with all workers to update the network weights. If any worker straggles for any reason, for example, a varied computing power or an abnormal communication latency, then the weight updating process on the server will be suspended until the worker finishing its jobs.

ASGD, which is asynchronous version of distributed SGD, breaks the synchronous barrier in SSGD to accelerate the training process further. In ASGD, each worker computes its gradients and sends the results to the server individually. Once the server receives the local results from a worker, it update the model asynchronously.

The weight updating strategy of ASGD is illustrated in Figure 1. A worker (assume it is worker m in Figure 1) obtains a version of the network (i.e., the network weights w_t as in Figure 1) and performs its local computations. The local computations involve a forward propagation computation and a set of calculations for the gradients. After worker m finishes the local computations, it sends the gradients back to the parameter server, which uses the gradients calculated by worker m to update the weights of the global network. However, while worker m performs its local computation, other workers may have obtained a different version of the network to complete their local computations, based on which the server has updated the network to a new version that is different from the version that Worker m obtained to calculate the gradients. The gradients calculated by worker m based on the weight w_t will be used to update the weight $w_{t+\tau}$ rather than w_t. This is the delay in updating the weights in ASGD.

Formula 2 represents how the network is updated as described above in ASGD, in which g_m is the local gradient calculated by worker m based on the network weights at time t, $\omega_{t+\tau}$ is the network weights at time $\omega_{t+\tau}$.

$$\omega_{t+\tau+1} = \omega_{t+\tau} - \gamma \cdot g \frac{1}{b} \sum_{i=1}^{b} \nabla_{\omega_t} \ell(f_{\omega_t}(x_i), y_i) \quad (2)$$

The delay in updating the weights may cause the result that the training by ASGD cannot achieve the same effect as the training by SGD. Namely, ASGD does not perform as well as SGD in terms of the accuracy of the trained model.

Due to the limitation of ASGD, the work [27] proposed a delay compensation algorithm named DC-ASGD, aiming to use a cheap approximation to compensate for the delay in the gradients computation performed by local workers. Applying such compensation can produce a better result than both the ASGD and the SSGD. However, as DC-ASGD utilizes the second-order partial derivatives, the accurate duration of the approximation is limited. It means that DC-ASGD would only work well for a short period of delay.

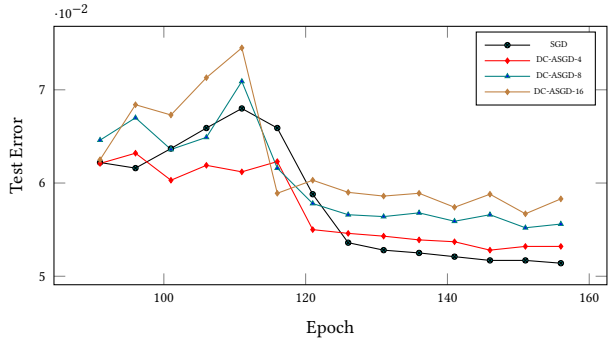


Figure 2: Performance of DC-ASGD training ResNet-18 on CIFAR-10 w.r.t. number of workers

$$\omega_{t+\tau+1} = \omega_{t+\tau} - \gamma \cdot (g_m + \lambda_t g_m \otimes g_m \otimes (w_t - w_{bak}(m))) \quad (3)$$

Formula 3 demonstrates how the DC-ASGD method works, where λ_t is a variance control parameter, w_{bak} is a backup model that the worker is using, $\otimes$ indicates the element-wise product, and g_m is the gradients computed by the worker that is as same as the ASGD.

We re-built the DC-ASGD and investigate the learning curve of the DC-ASGD on a popular DNN ResNet-18 with 4, 8 and 16 workers. The performance of the DC-ASGD on the CIFAR-10 benchmark dataset is shown in Figure 2. It can be observed that although the performance of DC-ASGD approximates that of SGD; however, the error rate obviously raises along with the amount of worker raising. According to the above analyses, we find that DC-ASGD works fine with conditions of low delay; however, the performance drops significantly when the delay is high.

3 RELATED WORKS

The SGD algorithm has been demonstrated to be very useful in training a variety of DNNs. Some related works have been proposed to speed up the efficiency of the SGD training through the parallel and distributed training [15, 16]. Hogwild! [18] is a lock-free approach for parallelised SGD, which enables the computing units to access a shared memory where the network parameters are stored. Hogwild can achieve a near-optimal convergence rate for certain problems. The work in [7] built the DistBelief framework, which adopts ASGD to train deep networks in a distributed manner. The convergence performance of ASGD has been extensively examined in many works [4, 26, 27]. In addition, several works [11, 17] developed frameworks to adopt the distributed training algorithm to accelerate the training efficiency further.

A recent work [23] proposed an Error Compensated Quantized Stochastic Gradient Descent (ECQ-SGD) algorithm to improve the training efficiency in a distributed training scenario. It quantises local gradients to reduce the communication overhead and utilizes the accumulated quantisation error to speed up the convergence. Comparing with other related works [2, 22], the ECQ-SGD applies an error compensation technique to achieve a state-of-art compression performance. However, the work focuses on reducing communication bandwidth to speed up training progress, which has a different focus from our work.

A rising approach [10, 21, 25] designed for federated learning [14], which enable devices collaboratively learn a shared prediction model while keeping all the training data locally, was proposed to determine an optimal trade-off between local update and global parameter aggregation to minimize the loss function with a given resource budget. Although the experiments show that the proposed trade-off controlling algorithm performs a good result near to the optimum with various machine learning models and different data distributions, nevertheless, such algorithms are suitable for the dataset that is stored individually and privately. In our case, we consider all of the workers that not only share the model but also use the same data.

The DC-ASGD [27] is the notable work closely related to the method proposed in this paper. The DC-ASGD utilizes the Hessian matrix to compensate for the delay approximately. As introduced in Section 1 and Section 2, the DC-ASGD estimates the distance in model version between the workers and the server, and then utilizes a cheap approximation of the second-order partial derivatives to compensate the delayed gradients. However, according to the analysis of the re-building experiments, such compensation works well when the delay is low. As the number of workers increases, the performance degrades significantly.

4 DISTRIBUTED TRAINING WITH LOSS COMPENSATION

We propose LC-ASGD to address a crucial problem in ASGD that is the delayed updating of weights. In LC-ASGD, the trend of loss values during training is modeled as a time series that is called the *loss time series*. An LSTM Recurrent Neural Network (RNN) is built as a loss predictor to forecast the future values in loss time series based on both current loss computed by the workers and historical values, which is the basis of our loss compensation. Moreover, to define the future step for the loss predictor, another LSTM network is built as a future step predictor that takes multivariate input data including computing cost of the worker, communication cost between the server and the worker, and the interval of the worker interacting with the server.

Our LC-ASGD aims to compensate for the loss caused by the delay. To achieve this, the order in which the workers finish their local computations is also modeled as a time series that is called the *worker time series*. It determines the order in which the weights of the network are updated in the parameter server and also the version of the network each worker obtains to start its local computation. If k numbers of other workers update the network weights in the parameter server before the worker m completes its local computation with the network w_t, we call that the latest network is k_m steps away from w_t. The larger value of k_m, the higher delay is experienced by worker m. We make use of the worker time series to predict how many steps the network version has evolved between the time when a work obtains a network version and the time when the work completes its local computation and sends back updates (i.e. gradients). We build another LSTM Recurrent Neural Networks (RNN) as a step predictor.

The loss caused by the delay is compensated as follows. When the parameter server receives a loss value computed by a worker m, it invokes the step predictor to predict the number of steps

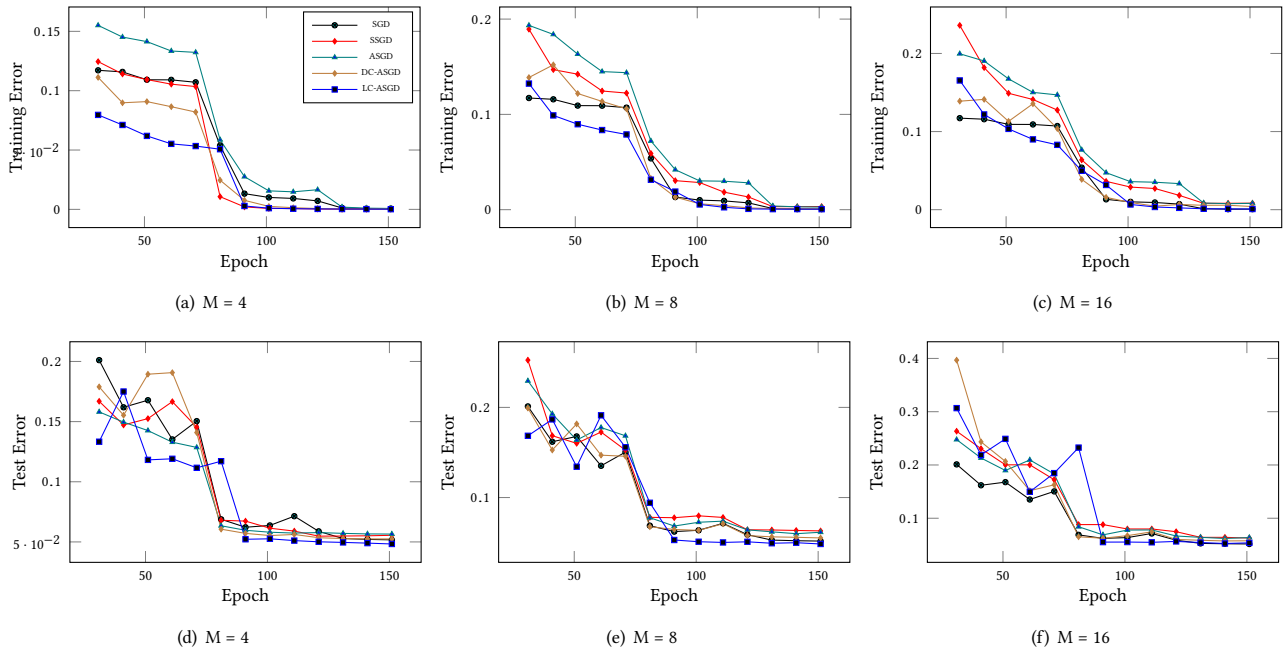


Figure 3: Error rates of the global model ResNet-18 with Async-BN as the training progresses on CIFAR-10

Algorithm 1 The computations performed by a worker, m

Initialize:
$state_m = \{loss : 0, mean : \{\}, var : \{\}, t_{comm} : 0, t_{comp} : 0\}$,
$z \in \{1, 2, ..., Z\}$, t_0, t_1

1: Pull w_t from the parameter server at timestamp t_0
2: Receive the weights w_t at timestamp t_1
3: Record the pulling time cost $state_m[t_{comm}] = t_1 - t_0$
4: Compute loss $\ell_m = \ell(f_{w_t}(x_i),\ y_i)$
5: Record the local loss $state_m[loss] = \ell_m$
6: Store mean μ_z in each BN layer bn_z into $state_m[mean]$
7: Store variance σ_z in each BN layer bn_z into $state_m[var]$
8: Push all recordings $state_m$ to the parameter server
9: Receive loss compensation ℓ_{delay} from the parameter server at timestamp t_2
10: Compute gradient $g_m = \nabla_{w_t}(\ell_m + \lambda \cdot \ell_{delay})$, finishing at timestamp t_3
11: Record computational time cost $state_m[t_{comp}] = t_3$ - t_2
12: Push the gradients g_m to the parameter server

that will be experienced by the worker. Assuming the predicted number of steps is k_m, the server then invokes the loss predictor to predict the loss value at the k_m-th step ahead, which is the loss value after compensating the delay corresponding to the k_m steps. The compensated loss value is then sent to the worker for its local computation.

As presented above, there are four main components in our LC-ASGD: the workers, the parameter server, the loss predictor and the step predictor. Next, we present these four components and the relevant algorithms in more detail.

4.1 Worker

Algorithm 1 outlines the computations performed by each worker in the distributed training. At the beginning of each iteration, the worker requests the latest network weights w_t from the parameter server (Line 1). The time consumption of pulling the network parameters is calculated by the difference between two timestamps t_0 and t_1. It is stored as t_{comm} in a data collection $state_m$ (Line 3). Based on the retrieved network from the parameter server, the worker takes a batch of training data and performs the forward propagation to compute a loss value ℓ_m (Line 4), following Formula 4. Along with the forward propagation, the mean μ and variance σ of each BN layer are also updated according to the input data.

$$\ell_m = \ell(f_{w_t}(x_i),\ y_i) \tag{4}$$

ℓ_m, μ_z and σ_z are saved in $state_m$ (Line 5, 6 & 7), which will be sent to the parameter server (Line 8). The loss ℓ_m is used to calculate a compensation value, while μ and σ are accumulated to a global mean and a global variance of each BN layer. The reason of doing accumulations for BN layers across all workers is that we found such optimizations can deliver better and more stable performance for the distributed training in many cases.

$$g_m = \nabla_{w_t}(\ell_m + \lambda \cdot \ell_{delay}) \tag{5}$$

Once the worker receives the compensated loss value from the server (Line 9), it leverages Formula 5 to combine the compensation with the current loss (Line 10). Then do the back-propagation through the network to calculate the gradients g_m. The λ here is a hyper-parameter to fine-tuning the compensated loss. Similar to t_{comm}, the time used for the gradient computation is recorded as

Algorithm 2 LC-ASGD: parameter server

Input: learning rate γ
Initialize: $t = 0$, $E_{bn_z} = 0$, $Var_{bn_z} = 1$, w_0 is initialized randomly, $iter = [\ ]$, $m \in \{1, 2, ..., M\}$, $z \in \{1, 2, ..., Z\}$
repeat
1: **if** receive $state_m$ **then**
2: Append m to $iter$
3: Predict step k_m = $stepPredictor(m, state_m[t_{comm}, state_m[t_{comp}],\ iter)$
4: Predict loss ℓ_{delay} for the next k_m steps by $lossPred(state_m[loss], k)$
5: Send ℓ_{delay} to worker m
6: Update $E_z = (1 - d) * E_z + d * state_m[mean_z]$
7: Update $Var_z = (1 - d) * Var_z + d * state_m[var_z]$
8: **else if** receive g_m **then**
9: $w_{t+1} = w_t - \gamma \cdot g_m$
10: $t = t + 1$
11: **else if** receive pull request from worker m **then**
12: Send w_t to worker m
13: **end if**
until $forever$

Algorithm 3 LC-ASGD: loss predictor

Input: loss ℓ_m (the loss received from worker m), step k_m
Initialize: ℓ_t (the latest loss of the network)
1: Train $lossPred$ with $(data = \ell_t,\ label = \ell_m)$
2: $predictions = lossPred(data = \ell_m,\ future = k)$
3: $\ell_{delay} = sum(predictions)$
4: $\ell_t = \ell_m$
Return: ℓ_{delay}

Algorithm 4 LC-ASGD: step predictor

Input: worker rank m, t_{comm}, t_{comp}, iteration recording $iter$
Initialize: $step_m = 0$, t^m_{comm}, t^m_{comp}, $m \in \{1, 2, ..., M\}$
1: Extract the last iteration $step_t$ of worker m from $iter$
2: Train $stepPred$ with $(data = \{step_m, t^m_{comm}, t^m_{comp}\},\ label = step_t)$
3: $k_m = stepPred(data = \{step_t, t_{comm}, t_{comp}\},\ future = 1)$
4: $t^m_{comm}, t^m_{comp}, step_m = t_{comm}, t_{comp}, step_t$
Return: k_m

t_{comp} (Line 11). Finally, the worker m pushes the gradients g_m to the parameter server (Line 12) to update the network.

4.2 Parameter Server

The functions performed on the parameter server are outlined in Algorithm 2. The parameter server generally receives the requests from the workers and sends the corresponding response back to the workers. A list $iter$, which is maintained in the parameter server, records the sequence of all workers that send the computing results to the server. The $iter$ is applied to derive the number of steps experienced by a particular worker. The server receives the computing results $state_m$ from worker m that contains a loss value and the updates (i.e. mean and variance) for the BN layers. Every time when the server gets the $state_m$, the worker index m will be appended into the list $iter$ (Line 2).

The step predictor is invoked to predict the number of steps (assume it is k_m) for which the network version will have evolved when the worker m finishes its computation and send the computing results back to the server (Line 3). Then, the loss value (i.e., $state_m[loss]$) and the predicted number of steps k_m are fed into the loss predictor $lossPred$ to predict the loss delay (denoted by ℓ_{delay}) at the following k_m steps (Line 4). The server sends ℓ_{delay} to worker m (Line 5) so that the worker uses the predicted loss delay to compensate for its loss value and then performs the back-propagation.

$$E_z = (1 - d) * E_z + d * state_m[mean_z] \tag{6}$$

$$Var_z = (1 - d) * Var_z + d * state_m[var_z] \tag{7}$$

Formula 6 and Formula 7 shows how the server accumulates the mean and variance sent by worker m to update the global mean E and the global variance Var for each batch normalization layer bn_z, where the z is the index of the batch normalization layer. The E and the Var are updated by calculating the new mean and variance across all workers after receiving the local mean and variance from the worker (Line 6 & 7).

$$w_{t+1} = w_t - \gamma \cdot g_m \tag{8}$$

When the server receives the gradients g_m from worker m, the server updates the network weights by Formula 8 (Line 9). Finally, the server sends the latest network parameters w_t to the worker who requests for the latest network (Lines 12).

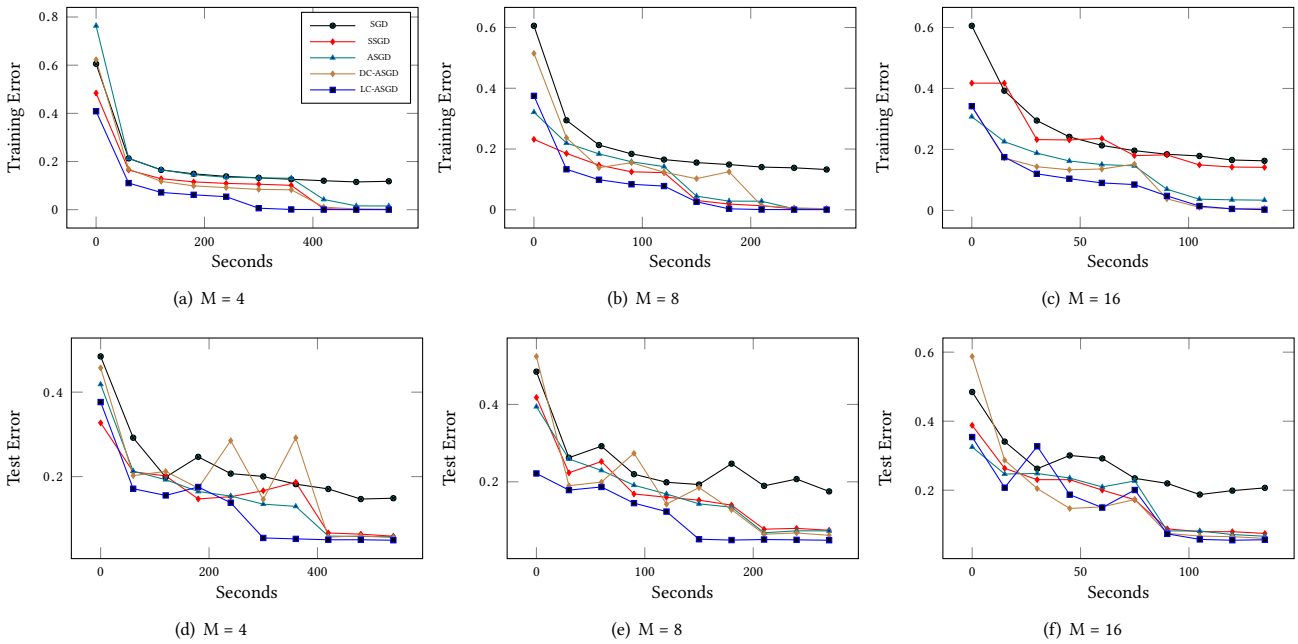
4.3 Loss Compensation Predictor

The prediction model $lossPred$ used by the loss compensation predictor is an RNN network, which resides in the parameter server. The first two layers of the RNN network are the LSTM layers while the final layer is a linear layer. The operations performed by the predictor is outlined in Algorithm 3.

Assume that the worker m takes a batch of input data and uses the model w_t to calculate a loss value ℓ_t through the forward-propagation. In the next iteration, the loss value of the training model w_{t+1} will be ℓ_{t+1}. Then, it will be ℓ_{t+2}. Following this pattern, the loss values from each iteration can be regarded as a time serial data. We utilize an RNN to model the relations among such data so that the loss value can be predicted. To train the RNN on the server without disturbing workers' progress, we implement an online-training process the loss predictor. The loss predictor takes ℓ_t as input and uses ℓ_{t+1} as the target every time to train the loss prediction model online. This is indeed how the loss prediction model is trained.

$$\ell_{delay} = sum(lossPred(data = \ell_m,\ future = k_m)) \tag{9}$$

Specifically, the loss prediction model $lossPred$ uses ℓ_t as input (since ℓ_t is the last loss value at time t coming from the training model) to predict the loss value at the next time $t + 1$, i.e., ℓ_{t+1}. When the real loss value ℓ_m at time $t + 1$ coming to the server,



(a) M = 4 (b) M = 8 (c) M = 16

(d) M = 4 (e) M = 8 (f) M = 16

Figure 4: Error rates of the global model ResNet-18 with Async-BN w.r.t. wall-clock time on CIFAR-10

the ℓ_m act as a label to calculate difference to the prediction ℓ_{t+1} (Line 1). Such difference will be used to back-propagating through the network to do updates. The updating procedure is same to the regular training process of neural networks. Follow this training cycle, the model *lossPred* is trained every time when the loss ℓ_{t+2}, ℓ_{t+3}, ..., , ℓ_{t+n} arriving to the server.

The forward-propagating goes on k_m iterations to generate the predictions of the loss for the k_m steps in the future (Line 2), where the value of k_m is predicted by the **step predictor** to be presented in the next subsection. Following Formula 9, all predicted loss values for the k_m future steps are summed up. The total loss is then sent back to the worker m. The reason why we sum up the loss predictions made for these k_m steps is because when a worker computing the gradient at the k_m-th step, we need to calculate the sum of the partial derivative of individual loss values at these k_m steps, which equals to the partial derivative of the sum of these loss values.

4.4 Step Predictor

The step predictor *stepPred* also runs on the server. The role of the step predictor is outlined in Algorithm 4. The number of steps k_m means the number of other workers who send updates to the server while the worker m running on its local computations. According to our analysis, the value of k_m depends on a number of system statuses including the computing capacity of each worker, the network quality between each worker and the server, etc. These system statuses typically vary in practice. Therefore, we construct a multivariate step predictor to capture the complex conditions in the distributed training systems, which can potentially generate more accurate predictions for the k_m.

The step predictor consists of two LSTM layers in the front of the network and a linear layer at the end. The size of the hidden layers in the model is limited, so that reduces the training time of the network. Similar to the loss predictor *lossPred*, the step predictor also conducts the online training. We not only utilize the previous step recordings k_m but also take communication cost and computation cost into account. Thus, there are three dimensions in the input data: communication cost t_{comm} between worker m and the server, computing cost t_{comp} of worker m doing local calculations, and the value of step k_m for worker m derived from *iter* (Line 1).

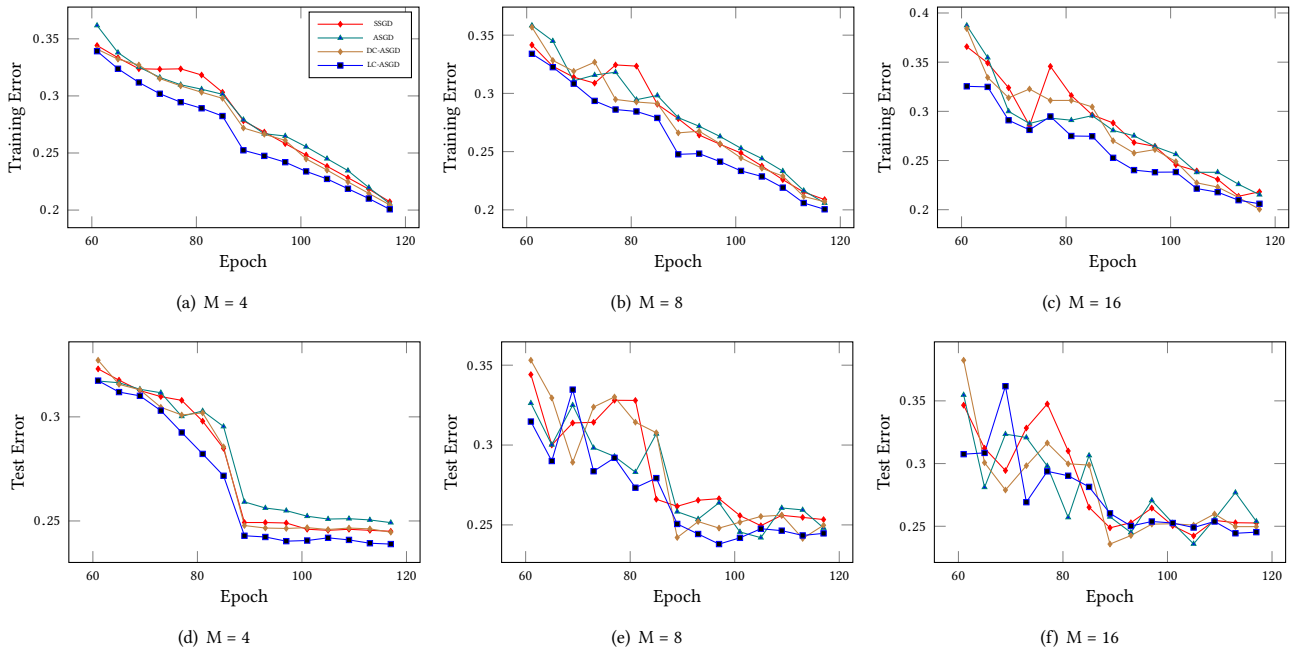
Following the online training method motioned in Section 4.3, the current step k_m first acts as a label for training the step predictor *stepPred*. Then the step predictor *stepPred* forecasts the next value of step k_m for the worker m by inputting the current value of k_m, t_{comm} and t_{comp}.

$$k = stepPred(data = \{step_t, t_{comm}, t_{comp}\},\ future = 1) \quad (10)$$

Formula 10 presents the way to make a prediction on the next step value for the worker m. It is slightly different from predicting the loss delay since predicting the next step only need to forward-propagating the step predictor for once.

5 EXPERIMENTS

In this section, we present the evaluation results for the proposed LC-ASGD algorithm. The experiments were carried out on a cluster where every node is equipped with an NVIDIA Tesla V100 GPU. Each node acts as a worker. The parameter server is equipped with two additional GPUs to accelerate the training process of loss predictor and step predictor. We tested our algorithm on ResNet [8] with the benchmark datasets CIFAR-10 and ImageNet. In addition to LC-ASGD, we also implemented several popular existing distributed



(a) M = 4 (b) M = 8 (c) M = 16

(d) M = 4 (e) M = 8 (f) M = 16

Figure 5: Error rates of the global model ResNet-50 with Async-BN as the training progresses on ImageNet

training algorithms including SSGD, ASGD and DC-ASGD, which have been used in many previous works as baselines [5–7]. For the sake of fairness, all experiments based on the same randomly initialized model, and worked with the same scheduling for learning rate.

5.1 Experiment Results on CIFAR-10

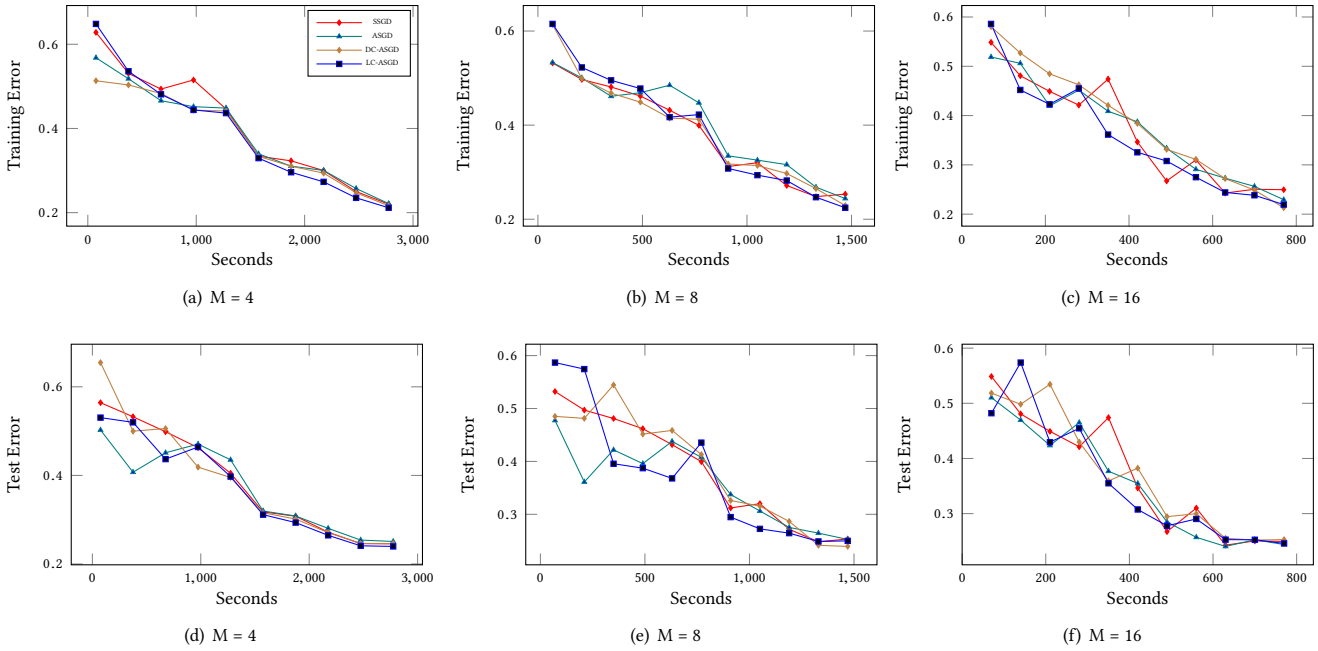
The CIFAR-10 dataset consists of 60000 color images in 10 classes, with 6000 images per class. We used 50000 images for training and 10000 images for the test. For all the algorithms under investigation, we ran the training for 160 epochs with a mini-batch size of 128 and the cross-entropy loss function. An initial learning rate of 0.3 was used and then was divided by ten after 80 and 120 epochs, following the same practice presented in [8]. Moreover, the hidden sizes we used for loss predictor and step predictor were 64 and 128, respectively. The network architecture is constructed following the literature [8]. We implemented SGD, SSGD, ASGD, DC-ASGD and LC-ASGD with the same settings as above for a fair comparison. The sequential SGD algorithm was regarded as a performance baseline to evaluate the distributed methods.

Figure 3 shows the learning curve of training error rate and test error rate of the network ResNet-18 on CIFAR-10 dataset as the training progresses. Table 1 details the final test rates of all the algorithms when they run with different numbers of workers. Table 1 also shows the performance degradation over the baseline algorithm. The following observations can be made from figure 3 and table 1: (1) SGD (the sequential method) delivered a test error of 5.15% in our setting (the test error of SGD reported in [8] was 8.75%). (2) Our method LC-ASGD (with Async-BN) achieved the lowest error. When training with 4 workers and 8 workers, LC-ASGD achieved 4.87% and 4.96% of error rate, respectively. They were even better than SGD, even though LC-ASGD was not designed to beat sequential SGD. In addition, LC-ASGD delivered a test error of 5.52% with 16 workers. (3) Although all the distributed algorithms lost the accuracy as the number of workers increased, LC-ASGD demonstrated the lowest degradation. As can be seen from Table 1, the performance degradation of DC-ASGD, SSGD and ASGD with 16 workers were 13.20%, 20.39% and 24.47% respectively, while that of LC-ASGD (with Async-BN) was only 7.18% in the worst case.

Figure 4 presents the convergence rate of the algorithms, namely, the change in training/test error rate over the time.

Combining Figure 4 and Table 1, it can be observed that the convergence rate of these five algorithms were different from their error rates. Although ASGD had the worst error rate among these algorithms, it converges very fast, nearly reaching a linear speed-up comparing with SGD in terms of throughput. SSGD was slightly slower than ASGD due to the synchronization barrier. DC-ASGD and LC-ASGD both struck a good balance in error rate and convergence speed. Their convergence speeds were similar to that of ASGD. We noticed that LC-ASGD took a longer time than SSGD in the case of 16 workers. This is because that two additional RNN models were running on the server to predict the loss and the steps. The loss predictor and the step predictor took slightly more time when the number of workers increased.

The above results are to be expected. Since ASGD suffers from the delayed gradients updating and the situation becomes worse as the number of workers increases. In SSGD, when the number of workers is increased, it is equivalent to increasing the batch size of the training data. However, increasing the batch size hurts the training performance of DNN in general.



(a) M = 4 (b) M = 8 (c) M = 16

(d) M = 4 (e) M = 8 (f) M = 16

Figure 6: Error rates of the global model ResNet-50 with Async-BN w.r.t. wall-clock time on ImageNet

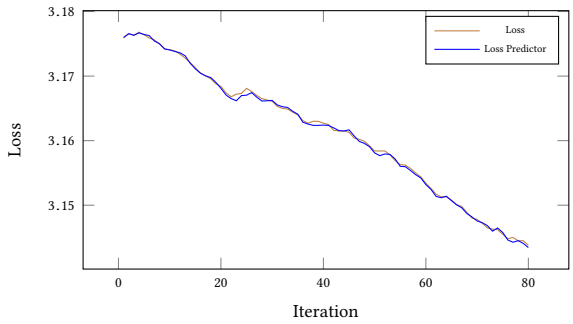


Figure 7: Performance of the loss predictor for ResNet-50 w.r.t. number of iterations on ImageNet training with 16 workers

5.2 Experiment Results on ImageNet

To further evaluate the performance of LC-ASGD with large-scale tasks, we tested these algorithms with the ImageNet dataset. The latest version of ImageNet contains more than 14 millions of annotated images. The data is split into 27 high-level categories, in which each contains up to 3822 subcategories. In the experiments, we adopted the same settings presented in [8], used ResNet-50(V2) with a mini-batch size of 128 and performed the training for 120 epochs being reduced by ten times at the 60th and 90th epoch. Because ImageNet is very large, training with the sequential method SGD on a single machine takes too long. Thus, we only ran the experiments with distributed algorithms.

Figure 5 and Table 1 show the Top-1 error of the SSGD, ASGD, DC-ASGD and LC-ASGD algorithm training ResNet50 with 1-crop ImageNet. The result of SSGD with 4 workers was used as a baseline performance which other algorithms' performance was compared against. The results of the experiments with ImageNet were similar to those with CIFAR10. Our LC-ASGD with Async-BN still produced the best performance among all other distributed algorithms. Specifically, when training with 4, 8 and 16 workers, it delivered 23.86%, 24.07% and 24.82% of test error rate respectively. The performance of other algorithms dropped from 3.02% to 5.39% when training with 16 workers, whilst the proposed LC-ASGD degraded only by 1.35%.

Figure 6 shows the convergent rate of four distributed algorithms. The results were similar to those of training ResNet18 with CIFAR10. SSGD was slowed down due to the synchronization barrier, while ASGD and DC-ASGD had a similar convergence rate. Although our LC-ASGD spent some extra time in training the additional RNNs (for the loss predictor and step predictor), it still demonstrated an excellent trade-off between error rate and convergence speed.

Figure 7 compares the loss calculated by the workers and the loss predicted by the *LossPredictor* in the experiments of training the ResNet-50 with ImageNet by 16 workers. Overall, the predictor worked effectively in terms of accuracy. The curve of the prediction largely overlapped the curve of the actual loss values. On the one hand, the actual loss value started at 3.176, and then dropped gradually to 3.144. On the other hand, the prediction also started at 3.176, and decreased to a value that was only slightly more than 3.143 at the end of the period. The reason why the predictor had such outstanding performance is because RNN is a robust model for dealing with the time series problems. The previous state was used as the feedback to preserve the memory of the network over time. The model learned from previous values and trends, so as to make accurate predictions based on the current state.

Moreover, we also evaluated the performance of the step predictor, which is shown in Figure 8. The brown curve is the order in

Table 1: Training performance of ResNet-18 on CIFAR-10 and ResNet-50 on ImageNet.

# Workers	Algorithm	CIFAR-10 BN		CIFAR-10 Async-BN		ImageNet BN		ImageNet Async-BN	
		Test Error (%)	Perf. Deg. (%)	Test Error (%)	Perf. Deg. (%)	Test Error (%)	Perf. Deg. (%)	Test Error (%)	Perf. Deg. (%)
1	SGD	5.15	Baseline	5.15	Baseline	-	-	-	-
4	SSGD	5.67	10.10	5.57	8.16	24.61	Baseline	24.49	Baseline
	ASGD	5.73	11.26	5.65	9.71	24.99	1.54	24.90	1.67
	DC-ASGD	5.33	3.50	5.22	1.36	24.53	-0.33	24.46	-0.12
	LC-ASGD	**4.98**	**-3.3**	**4.87**	**-5.44**	**23.91**	**-2.84**	**23.86**	**-2.57**
8	SSGD	6.19	20.19	6.01	16.70	25.24	2.56	25.11	2.53
	ASGD	6.38	23.88	6.27	21.75	25.71	4.47	25.64	4.70
	DC-ASGD	5.72	11.07	5.58	8.35	25.98	5.57	24.89	1.63
	LC-ASGD	**5.11**	**-0.78**	**4.96**	**-3.69**	**24.17**	**-1.79**	**24.07**	**-1.71**
16	SSGD	6.41	24.47	6.20	20.39	25.80	4.84	25.62	4.61
	ASGD	6.59	27.96	6.41	24.47	25.96	5.49	25.81	5.39
	DC-ASGD	6.05	17.48	5.83	13.20	25.41	3.25	25.23	3.02
	LC-ASGD	**5.76**	**11.84**	**5.52**	**7.18**	**24.99**	**1.54**	**24.82**	**1.35**

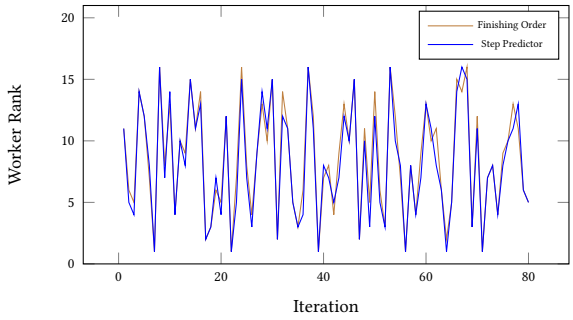


Figure 8: Performance of the step predictor for ResNet-50 w.r.t. number of iterations on ImageNet training with 16 workers

which the workers finished their local computations and sent the results back to the server (which was recorded in *iter* and was used to derive the actual values of step k_m for the worker m). The blue curve is the predictions made by *StepPredictor* in a period of training ResNet-50 with ImageNet by 16 workers. Although the order of workers was generally regular, the variance still occurred during the training. The variance was typically caused by the reasons discussed previously, for example, the changes in computing workload and/or the communication status in the workers. As we can see from the figure, the forecast made by the *StepPredictor* were very accurate.

The prediction error of the two predictors in LC-ASGD is mainly due to two reasons. The error rate is unstable when there are not enough data to train the predictor, or when the network state is changing significantly. This situation generally occurs at the beginning of the training process or when the learning rate is tuned.

5.3 Evaluation on Asynchronous Batch Normalization

In our LC-ASGD, we proposed an asynchronous batch normalization strategy to improve the training accuracy in distributed learning. Table 1 compares the training accuracy between the proposed asynchronous batch normalization (denoted by Asysc-BN) with the regular batch normalization (denoted by BN) in literature. It can be observed that the test error rate of the models training with Async-BN are generally better than that with the regular BN. As the number of workers increased, The advantage of Async-BN is more prominent over the regular BN. The reason why Async-BN outperforms the regular BN is because the difference in how the batch information is exchanged and updated between the workers and the server. When training with the regular BN, the parameter server replaces the mean and variance of all BN layers using the parameter values received from the latest worker. Our Async-BN strategy updated the mean and the variance by accumulating all the updates from the workers and re-calculating the global mean and variance. Consequently, the parameters that the worker retrieved from the server contained the accumulated mean and variance. This way, the mean and variance that the workers used to start their training is more consistent.

5.4 Parameter Server Overhead Analysis

As aforementioned in Algorithm 3 and Algorithm 4, two RNN models ran on the parameter server in our LC-ASGD to provide the predictions for the loss compensation. The overhead mainly depends on the complexity of the prediction model. Also there is a trade-off between the complexity and the accuracy when designing the prediction model. We conducted the experiments to evaluate the overhead incurred by both loss predictor and the step predictor, where delivered the prediction performance presented in Figures 7 and 8.

Table 2 (with CIFAR-10) and Table 3 (with ImageNet) show the average time spent by the online loss predictor and step predictor in a training iteration when training with CIFAR-10 and ImageNet, respectively. It can be seen from the tables that the time cost is steady for different datasets. Also, the time cost increases slightly as the number of workers increases, which is to be expected since when there are more workers, more input data to the prediction models and hence more training time. Overall, the overhead incurred by the predictions is low compared with the training time of an iteration. The overhead accounts for around 8% for CIFAR-10, while it is slightly more than 1% for ImageNet. These results indicate that our loss and step predictor can make accurate predictions with relatively low overhead.

Moreover, according to the learning curve shown in Figure 4 and Figure 6, our LC-ASGD achieved the excellent convergence rates while delivering lower error rates. It beats SSGD as it removes the

Table 2: Average time of a training iteration on CIFAR-10.

# Workers	4	8	16
Loss Pred. (ms)	1.28	1.29	1.30
Step Pred. (ms)	1.37	1.43	1.48
Total Training (ms)	32.23	32.84	34.64
Overhead (%)	8.22	8.28	8.03

Table 3: Average time of a training iteration on ImageNet.

# Workers	4	8	16
Loss Pred. (ms)	1.27	1.29	1.33
Step Pred. (ms)	1.36	1.45	1.50
Total Training (ms)	183.23	185.68	188.71
Overhead (%)	1.44	1.48	1.50

synchronization barrier, and nearly reached the convergence rate of ASGD when it runs with 4 workers. These results also reflect that the loss and step predictor do not incur heavy overhead during the training.

6 CONCLUSION

In this paper, we discussed the issue of synchronization barrier in SSGD, the delayed gradient updating in ASGD and the limitation of DC-ASGD. These problems motivate us to propose a novel distributed training algorithm called LC-ASGD. LC-ASGD compensates for the loss delay seen in ASGD. It constructs the RNN models to predict the delayed steps and further predict the value of the loss function at a specific step into the future. An asynchronous batch normalization strategy is also proposed in LC-ASGD. The proposed LC-ASGD is evaluated on popular deep neural networks with the widely used datasets. The experimental results demonstrate that LC-ASGD is able to train the models to the outstanding accuracy compared with the existing distributed training algorithms, especially when training with a large number of workers. For future work, we plan to extend LC-ASGD to address the training problem with distributed data, i.e., different workers train the models with different subset of input data.

ACKNOWLEDGMENTS

This research is partially supported by Worldwide Byte Security Information Technology Co. LTD, CCF-Huawei DBIR2019001A, and Guangdong project 2018B030325002.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.

[3] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *Siam Review* 60, 2 (2018), 223–311.

[4] Sorathan Chaturapruek, John C Duchi, and Christopher Ré. 2015. Asynchronous stochastic convex optimization: the noise is in the noise and SGD don't care. In *Advances in Neural Information Processing Systems*. 1531–1539.

[5] Kai Chen and Qiang Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *2016 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE, 5880–5884.

[6] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[9] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[10] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[11] Matthias Langer, Ashley Hall, Zhen He, and Wenny Rahayu. 2018. MPCA SGD—A Method for Distributed Training of Deep Learning Models on Spark. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2540–2556.

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[13] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*. 19–27.

[14] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).

[15] Qi Meng, Wei Chen, Jingcheng Yu, Taifeng Wang, Zhi-Ming Ma, and Tie-Yan Liu. 2016. Asynchronous Accelerated Stochastic Gradient Descent. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (New York, New York, USA) *(IJCAI'16)*. AAAI Press, 1853–1859. http://dl.acm.org/citation.cfm?id=3060832.3060880

[16] Yuewei Ming, Yawei Zhao, Chengkun Wu, Kuan Li, and Jianping Yin. 2018. Distributed and asynchronous Stochastic Gradient Descent with variance reduction. *Neurocomputing* 281 (2018), 27 – 36. https://doi.org/10.1016/j.neucom.2017.11.044

[17] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. 2015. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051* (2015).

[18] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.

[19] Anand Srinivasan, Ajay Jain, and Parnian Barekatain. 2018. An analysis of the delayed gradients problem in asynchronous sgd. (2018).

[20] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. ACM, 303–314.

[21] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. 2019. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1205–1221.

[22] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[23] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. 2018. Error compensated quantized SGD and its applications to large-scale distributed optimization. *arXiv preprint arXiv:1806.08054* (2018).

[24] Wayne Xiong, Lingfeng Wu, Fil Alleva, Jasha Droppo, Xuedong Huang, and Andreas Stolcke. 2018. The Microsoft 2017 conversational speech recognition system. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 5934–5938.

[25] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 1–19.

[26] Shen-Yi Zhao and Wu-Jun Li. 2016. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[27] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Asynchronous stochastic gradient descent with delay compensation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 4120–4129.