Distributed Framework for Accelerating Training of Deep Learning Models through Prioritization

Tian Zhou
Dept. of Electrical and Computer Engineering
UMass Amherst, USA
tzhou@umass.edu

Lixin Gao
Dept. of Electrical and Computer Engineering
UMass Amherst, USA
lgao@ecs.umass.edu

2021 IEEE International Conference on Cloud Engineering (IC2E) | 978-1-6654-4970-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/IC2E52221.2021.00035

***Abstract*—Machine learning models such as deep neural networks have been shown to be successful in solving a wide range of problems. Training such a model typically requires stochastic gradient descent, and the process is time-consuming and expensive in terms of computing resources. In this paper, we propose a distributed framework that supports the prioritized execution of the gradient computation. Our proposed distributed framework identifies important data points through computing or estimating the priority for each data point. We evaluate the proposed distributed framework with several machine learning models including multi-layer perceptron (MLP) and convolutional neural networks (CNN). Our experimental results show that prioritized SGD accelerates the training of machine learning models by as much as 1.6X over that of the mini-batch SGD. Further, the distributed framework scales linearly with the number of workers.**

I. Introduction

In recent decades, machine learning models such as deep neural networks are successful in many fields such as computer vision [1], [2], speech recognition [3], and natural language processing [3]. For example, a deep-learning model developed in [2] is able to identify human faces with an accuracy of over 99%. Training a deep neural network is usually time-consuming and expensive in terms of computing resources due to its increasing size. For example, Lecun et al. show that identifying handwritten digits requires a deep neural network with around 1 million parameters [4]. Later, Krizhevsky et al. recognized images with a deep neural network consisting of 60 million parameters and won the ImageNet competition [1]. More recently, Deepface successfully classified human faces with a deep neural network containing 120 million parameters [2]. As the model becomes more complex, the corresponding training time increases dramatically. Despite the rapid increase of computing power in recent decades, the training time increases from several hours to several days, or even several weeks [4]–[6].

To accelerate the training process of deep neural networks, both hardware and software approaches have been proposed. Graphics Processing Units (GPU) are used to train deep learning models. Many distributed frameworks, including Torch [7], Tensorflow [8], Caffe [9] and MXNet [10], support the training of machine learning model through GPUs. Google's specialized hardware, Tensor Processing Unit (TPU) [5], targets at deep neural network models training. Others have accelerated the training and testing of deep learning models with customized hardware, such as field-programmable gate array (FPGA) [11]–[13].

In addition to hardware approaches, researchers have proposed a series of training algorithm such as RMSProp [14], ADAM [15], Nadam [16] and AdaGrad [17] to accelerate the gradient descent process. The acceleration stems from two major ideas. First, instead of simply updating the model parameter with gradients, they consider the momentum as well. Second, instead of a fixed learning rate for every parameter, each parameter has its own learning rate. And the rates are adaptively adjusted during the training process. Other researchers accelerate the training process by pruning or augmenting the training data set [18]–[20].

In this paper, we propose a distributed framework that accelerates the training process of deep learning models through prioritizing the execution of gradient descent. The proposed distributed framework automatically identifies and utilizes the most important data points to update the gradient of the model parameters. We quantify the importance of a data point as how much that data point is able to reduce the loss function value. Using the quantified importance as the priority value for each data point, our distributed framework selects a set of data points with the highest priority values as a mini-batch. Only the gradients of the selected data points in the batch are used to update the model parameters.

The challenge of adopting priority in stochastic gradient descent is that the computation of priority values is extremely time-consuming. To reduce the overhead of the prioritized execution, we propose to compute priority values only for a subset of data points and estimate priority values for the rest. The distributed framework learns the trend of priority values for each data point online and estimates the priority values with the learned trends. Our distributed framework can be applied to a large variety of deep learning models solved with stochastic gradient descent, including multi-layer perceptron (MLP), convolutional neural network (CNN), and recurrent neural network (RNN).

We implement a distributed framework to facilitate deep neural network training in the cloud environment. To evaluate the proposed distributed framework, we implement several

978-1-6654-4970-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IC2E52221.2021.00035

deep neural network models including multi-layer perceptron and convolutional neural network using the framework. Our experimental results show that the distributed framework accelerates the training of these deep neural network models by about 1.6X over that of the typical SGD. We further show that our framework scales linearly with the number of workers.

The outline of this paper is as follows. In Section II, we review the gradient descent algorithm and derive the priority value for each data point. Then, we propose our prioritized framework for gradient descent (PGD) in Section III. In Section IV, we elaborate how to estimate priority values. We evaluate the distributed framework in terms of the impact of parameters, training time and scalability in Section V. Finally, we describe related works and conclude our paper in Section VI and Section VII respectively.

II. Priority in Gradient Descent

In this section, we start with a review of the gradient descent algorithm and stochastic gradient descent algorithm from the perspective of optimization problems. Base on that, we quantify the importance of a data point in terms of its contribution to the loss function and then derive the priority value of a data point.

A. Review of Gradient Descent

Machine learning problems are typically expressed with optimization problems. An optimization problem can be formulated as a problem of finding the optimal parameter W to maximize or minimize a loss function $L(W)$. The loss function L is usually a summation among individual losses L_i among a data set $D = \langle d_i \rangle$, *i.e.,* $L(W) = \frac{1}{|D|}\sum_D L_i(W)$.

Gradient Descent (GD), Stochastic Gradient Descent (SGD) and mini-batch SGD, search the model parameter by recursively updating current W using the gradient g of the loss function as shown in Equation (1):

$$W \leftarrow W + \eta g \tag{1}$$

where $g = \frac{\partial}{\partial W} L(W; D)$. We use g_i to represent the gradient of an individual data point d_i. It is defined as in the following equation.

$$g_i = \frac{\partial}{\partial W} L_i(W; d_i) \tag{2}$$

GD calculates g by averaging all gradients from the data set as in the following equation.

$$g = \frac{1}{|D|} \sum_{d_i \in D} g_i \tag{3}$$

The typical SGD simplifies the gradient calculation by using one data point each time as in the following equation.

$$g = g_i \tag{4}$$

The mini-batch SGD lies between GD and SGD. It calculates g using the average gradient from a subset of data points $S \subset D$ as in the following equation.

$$g = \frac{1}{|S|} \sum_{d_i \in S} g_i \tag{5}$$

Because the mini-batch SGD is usually faster than GD and stabler than the typical SGD, it is one of the most wildly used gradient-based optimizing algorithm. In the rest of this paper, we use the term SGD to represent the mini-batch SGD.

B. Priority of Data Points

In gradient descent algorithms, data points are considered to be equally important in calculating the gradient. So usually, data points are used to update the model parameter in a round-robin fashion.

However, this assumption is not true. The gradient of each data point does not necessarily contribute equally to model parameters. We quantify the importance of a data point with its impact on the loss function. According to the definition of differentiation, when the model parameter is changed by ΔW, the change of the loss function can be approximated with the inner product of its gradient $\partial L/\partial W$ and the change of model parameter, *i.e.,* $\Delta L = \partial L/\partial W \cdot \Delta W$.

We define the term $\Delta L(d_i; W, \eta)$, $\Delta L(d_i)$ for short, as the difference on the loss function L when only the data point d_i is used to update the model parameter. In that case, the difference of the model parameter $\Delta W_i = \eta g_i$. It changes the loss function value as the following equation.

$$\Delta L(d_i) = \frac{\partial L}{\partial W} \cdot \Delta W_i = \eta \left(\sum_{d_j \in D} g_j \right) \cdot g_i \tag{6}$$

Based on that, we quantify the impact of data point d_i as Equation (6).

The impact of data points on the loss function can be used to select the most important data points. The term η can be omitted in calculating priorities, since this scalar factor η is applied to every data point. We use $\bar{g}$ to represent the aggregated gradient of all data points.

$$\bar{g} = \sum_{d_j \in D} g_j \tag{7}$$

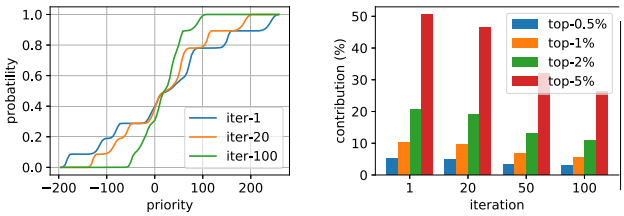
Therefore, the priority of a data point is the inner product of the gradient on that data point g_i and the aggregated gradient of all data points, as in Equation (8).

$$p_i = \left(\sum_{d_j \in D} g_j \right) \cdot g_i = \bar{g} \cdot g_i \tag{8}$$

Note that, the priority definition works for all objective functions. Different objective functions only affect how g_i is computed.

Different data points have different impacts on the loss function. We show the distribution of priority values of a multi-layer perceptron model with 100 hidden neurons in Fig. 1. This model is trained for a classification task using the MNIST data set. As we can see from Fig. 1a, the priority values are neither evenly distributed or gathered around a certain value.

The distribution suggests that a small number of top data points contribute a significant part to the loss function. Using the loss decrease made by adopting all data points as a



(a) CDF of priority (b) ΔL Contribution

Fig. 1. Distribution of priority and ΔL contribution

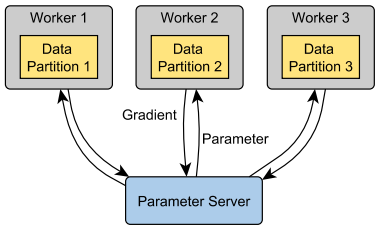


Fig. 2. Structure of PGD Framework

reference, we check how much does the top-k data points contribute to it. Shown in Fig. 1b, the top 0.5%, 1%, 2%, 5% data points contribute 5%, 8%, 15%, 32% of the loss decrease at iteration 50.

III. Prioritized Distributed Framework for Gradient Descent

In this section, we propose a distributed framework to support the prioritized execution of gradient descent. First, we give an overview of our framework and the workflow of each worker in Section III-A. Since the priority value is crucial to the prioritized execution, we then introduce an efficient computation of priority values in Section III-B. After that, we summarize the workflow in Section III-C.

A. Framework Overview

We design a distributed framework for the prioritized execution of GD. We distribute data points and corresponding calculation of priority and gradient among workers. They report the gradient to a parameter server that maintains the latest model parameter. We illustrate the general structure of our framework in Fig. 2. The parameter sever updates the model parameter using the gradients received from workers and sends back the updated model parameter to workers. Then, workers compute new gradients using the updated model parameter.

The majority of the computation task in the system is the computation of gradient performed on each worker. As discussed in Section II, data points with high priority values contribute more in decreasing the loss function value. Instead of selecting a mini-batch in a Round-Robin manner, updating the model parameter using the data points with the highest priority values is able to decrease more of the loss function.

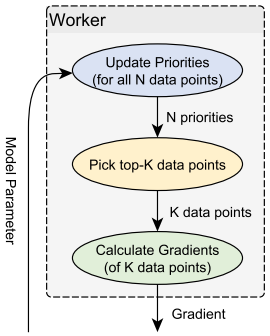


Fig. 3. Basic Workflow of Worker

Therefore, instead of uniformly sampling data points in the typical SGD, we propose to select data points with the highest priority values in each iteration. We refer to this framework as *Prioritized Gradient Descent (PGD)*.

Fig. 3 illustrates the main workflow of a worker. Different from typical SGD which constructs a mini-batch in Round-Robin, the workers in our framework select the data points with top priority values to form a mini-batch in each iteration. So at the beginning of each round, a worker updates the priority value of all its local data points. Then, it picks top-k data points as a mini-batch from all local data points, where k is predefined and indicates the batch-size. In the rest of this paper, denote k as a ratio of data points instead of an integer. After that, the worker performs the actual gradient computation task and sends out the aggregated gradient to the parameter server.

B. Update of Priority Value

A worker needs to know the priority value of each data point before it calculates gradients in an iteration. The most straight-forward method is to apply Equation (8) for all data points in each iteration. However, it is not feasible to compute priority values of all data points from scratch in each iteration. Calculating priority values can be time-consuming, because according to Equation (8) we need to calculate the gradient of a data point to get its priority value. The computation resources consumed for this cannot cover the gain from prioritized execution. Therefore, as shown in Fig. 4, our framework only computes the priority values of some data points and estimates the rest in each iteration. In the next iteration, our framework choose another set of data points in Round-Robin to compute priority values. We define the ratio of data points chosen to compute priority values as *renewal ratio* and denote it with r. We should set $r < k$ to make sure the benefit of prioritized execution is not covered by the additional cost of computing priority values.

Since the aggregated priority value of all data points is needed for priority calculation, we initialize computation by calculating the gradient of all data points in the first iteration. Then, we aggregate them and then apply Equation (8) to initialize the priority of all data points. After that, the framework

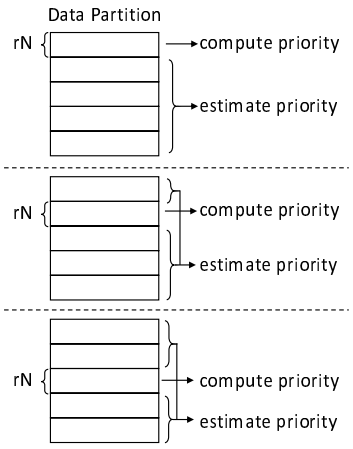


Fig. 4. Update of Priorities

picks the most prioritized data points and uses their gradients to update the model parameter.

1) Calculating Priority Value: In order to compute priority values, we need to know the aggregated gradient of all data points, *i.e.,* $\bar{g}$, according to the definition of priority. But it is not reasonable because we want to use the priority to choose data points for gradient calculation. So we approximate the aggregated gradient term with the one in the last iteration. It is because in SGD the model parameter is updated gradually, so that the corresponding gradient also changes slowly in consecutive iterations.

The memory can also be a problem in computing the priority values. Since aggregating all gradient is time-consuming, we can incrementally update it using the newly computed gradient in each iteration. A straightforward method to do so is to store old gradients of all data points and use them to compute the delta of gradients. However, it requires the memory of $O(N \cdot M)$, where N is the number of data points and M is the number of parameters of the model. For example, if we want to train a face recognition CNN model with 100,000 single-precision parameters using a data set with 1 million images, we will need as much as 400GB memory for storing previous gradients.

Therefore, we update the aggregated gradient via a memory-efficient approximate method. Instead of performing the accurate incremental update, we employ a proportional idea to do an approximate update. We approximate the old gradient of a mini-batch with n data points with $\frac{n}{N}\bar{g}$ where N is the total number of data points. So that we only store the aggregated gradient. When we get the local aggregated gradient g_n of a new mini-batch with n data points, we update the aggregated gradient $\bar{g}$ as Equation (9).

$$\bar{g} = \bar{g} - \frac{n}{N}\bar{g} + g_n = (1 - \frac{n}{N})\bar{g} + g_n \tag{9}$$

We can aggressively update the gradient during the phase of calculating priority values. The worker needs to calculate the gradient of a data point in order to compute its priority according to Equation (8). Usually, the cost of calculating the gradient of the data point is much larger than the cost of calculating the priority with the given gradient. Therefore, instead of simply using the gradient for priority calculation, we accumulate them for updating the model parameter. As a result, $(k + r)N$ data points are used to update the model parameter in each iteration.

2) Estimate Priority Value: Instead of computing the actual priority values using Equation (8), we estimate priority values for data points that are not selected to calculate priority values. We estimate the priority value of a data point using its computed priority values in history. In our framework, the priority value of a data point is computed at most every $\lceil 1/r \rceil$ iterations. We employ an online learning method to estimate the priority value for each data point. We will elaborate the learning and estimation method in Section IV.

C. Workflow Summary of Workers

We show the workflow of a worker in Alg. 1. At the beginning of each iteration, we update the priority value P of all data points. The worker selects rN data points in the Round-Robin manner to calculate priority values as shown at Line 9, where N is the number of data points. Then, it learns the parameter for priority estimation u_i using the calculated priority as shown at Line 10. For the priority values of un-selected data points, the worker estimates them using the estimation parameter as shown in Line 10. The learning and estimating will be elaborated later in Section IV. Meanwhile, the gradients used for calculating priority are aggressively accumulated in variable G as shown in Line 11.

Then, the worker picks the top-(kN) data points by their priorities and then calculates the gradient of picked data points. The calculated gradients are accumulated into the variable G. Since the priority calculation is much cheaper than the gradient calculation, the worker computes the priority value for each calculated gradient during this phase. The priority value is used to learn the estimation parameter which will be used in the next iterations. This aggressive update of priority values is shown with Line 20-21. After that, the variable G accumulates every gradient calculated in this iteration and all calculated gradients are used to update the estimation parameter. At the end of each iteration, the worker sends G to the parameter server and updates the aggregated gradient $\bar{g}$ using G with Equation (9) by setting $n/N = r + k$.

IV. Priority Value Estimation

In this section, we describe how priority values are estimated for data points that are not selected to calculate their priority in an iteration. We introduce how to learn the priority trend of a data point using its history priority values and how to the estimation using the learned trend.

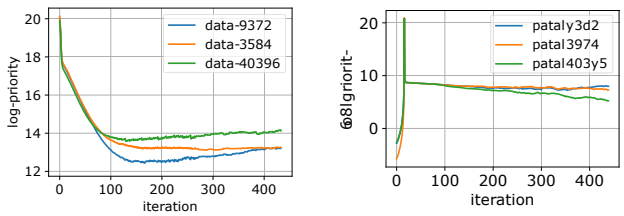
We need to construct an estimation function which outputs the priority value of a data point for any given iteration number. We estimate from the data historical data. We assume that the gradient function is continuous with the model parameter. Then, according to Equation (8), the priority value of a data

Algorithm 1 Workflow of a Worker

procedure WorkerTask($D, r, k, W, U, \bar{g}, it$)
Require: data set D, renewal ratio r, top ratio k
Require: model parameter W, estimation parameters $U = \langle u_1, \cdots u_{|D|} \rangle$
Require: aggregated gradient $\bar{g}$, current iteration number it
  $G \leftarrow \mathbf{0}$
  // update priority
  $S \leftarrow$ select $r|D|$ data points in Round-Robin
  $P = \langle p_1, \cdots p_{|D|} \rangle \leftarrow \mathbf{0}$
  for each $d_i \in D$ **do**
    if $d_i \in S$ **then**
      $g_i \leftarrow gradient(W, d_i)$
      $p_i \leftarrow \bar{g} \cdot g_i$ ▷ calculate priority
      $u_i \leftarrow learn(u_i, p_i, it)$ ▷ learn priority
      $G \leftarrow G + g_i$ ▷ aggressive gradient update
    else
      $p_i \leftarrow estimate(u_i, it)$ ▷ estimate priority
  // pick top-k
  $T \leftarrow$ pick the top $k|D|$ data points by priority P
  // calculate gradient
  for each $d_i \in T$ **do**
    $g_i \leftarrow gradient(W, d_i)$
    $G \leftarrow G + g_i$
    $p_i \leftarrow \bar{g} \cdot g_i$ ▷ aggressive priority update
    $u_i \leftarrow learn(u_i, p_i, it)$ ▷ aggressive prio. learning
  $\bar{g} \leftarrow (1 - r - k)\bar{g} + G$
  send G to the parameter server
  return $U, \bar{g}$

is a continuous function with the model parameter as the argument. Therefore, we employ the extrapolation method to predict the priority values in terms of iteration number given history priority values. Since we get a new priority value of any data at most every $\lceil 1/r \rceil$ iterations, we will not extrapolate values too far away from known records. Fig. 5 demonstrates the priority values during the training process of an MLP model and a CNN model on the MNIST data set. We randomly select 3 data points and show their priority values with different colors. As we can see, the priority values change drastically in the first several iterations and then decrease at a relatively stable pace. For example, shown in Fig. 5a, priority values of MLP decrease linearly in log-scale between iterations 10 and 100. From iteration 100, the priority values of some data points bump around certain values and the priority values of some other data points follow some new straight lines whose slopes are much smaller.

Now, the question is which fitting function to use for the extrapolation. Since the estimation operation is executed frequently in our framework, the fitting function should be simple and fast. Therefore, we design a specific fitting function for our estimation. Our experiments on various types of machine learning models show that the priority values of each data point are piece-wise linear in log-scale in terms of the number



(a) MLP (b) CNN

Fig. 5. Priority Values over Time

of iterations. Therefore, we use the following equation to approximate the priority value of data point d_k in the iteration between j and i ($j \leq i$).

$$p_i^k = e^{a_i^k i + b_i^k} = p_j^k e^{a_i^k (i-j)} \tag{10}$$

a_j^k is a constant factor to be learned. The empirical result shows that the factor is relatively stable in consecutive iterations. We can calculate a_j^k using the following equation.

$$a_i^k = \frac{\ln p_i^k - \ln p_j^k}{i - j} \tag{11}$$

In addition to Equation (10), we also apply an exponential average method for a_i^k using the following equation,

$$\hat{a}_i^k = \lambda a_i^k + (1 - \lambda) a_j^k \tag{12}$$

where λ is a factor controlling how much the new value contributes. Empirically, we set $\lambda = 0.8$ to give more attention to the new values, so that we mitigate the impact of random bumps on the priority values which are caused by the mini-batch update manner. We discribe the *learn* and *estimate* procedure in Alg. 2.

Algorithm 2 Priority Estimation

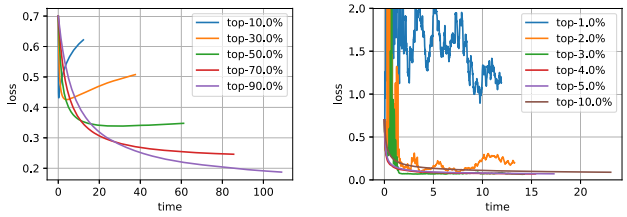
procedure learn($u_k = \langle \ln p_j^k, a_j^k, j \rangle, p_i^k, i$)
  $a_i^k = \left(\ln p_i^k - \ln p_j^k\right) / (i - j)$
  $a_i^k = \lambda a_i^k + (1 - \lambda) a_j^k$
  return $u_i = \langle \ln p_i^k, a_i^k, i \rangle$
procedure estimate($u_k = \langle \ln p_j^k, a_j^k, j \rangle, i$)
  return $\exp\left(a_j^k (i - j) + \ln p_j^k\right)$

V. Evaluation

In this section, we evaluate our PGD framework in terms of the impact of parameters, convergence time and scalability.

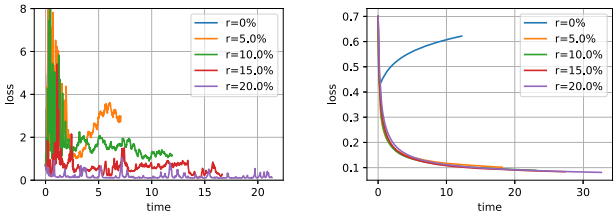
A. Data Set and Setting

We evaluate our framework with three types of machine learning models, including logistic regression (LR), multilayer perception (MLP) and convolutional neural network (CNN). Both synthetic data and the real-world data set are employed in the evaluation. In considering the uncertainty in real-world data, we artificially add a random noise following the Gaussian



(a) Given small renewal ratio (r=0.1%) (b) Given large renewal ratio (r=10%)

Fig. 6. Impact of Top Ratio k



(a) Given small top ratio (k=1%) (b) Given large top ratio (k=10%)

Fig. 7. Impact of Renew Ratio r

distribution to each dimension of the synthetic data. We adopt the MNIST image data set. We implement our system using C++ with OpenMPI. The source code is publicly accessible on GitHub [1]. We evaluate it on a local Linux cluster. In the following experiments, we use 4 workers by default.

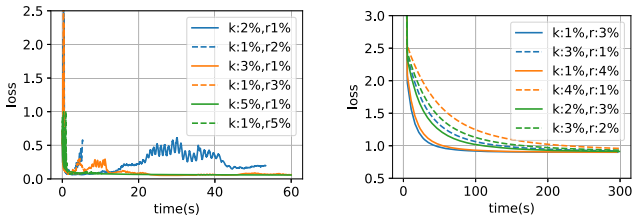
B. Impact of Parameters

There are two key parameters in the PGD framework, the renewal ratio k and the top ratio r. We examine their individual impacts on training time and the impact of their combinations.

1) Top Ratio: First, we demonstrate the impact of the top ratio k in Fig. 6. In this experiment, we show the training curve of the first 10000 iterations a 1000-dimension logistic regression model under a synthesized data set. There are 10,000 data points in the synthesized data set. The impact of the top ratio k is conditioned on the renewal ratio r. When only a small amount of data points renews their priority values in each iteration, shown in Fig. 6a, small top ratios lead to divergence. While given a relatively larger renewal ratio like 10%, shown in Fig. 6b, large top ratios usually make the training slower. However, the top ratio should be set too small. It leads to big bumps or even divergence.

2) Renewal Ratio: Similar to the top ratio, the impact renewal ratio r is also conditional on the setting of the top ratio k. We show the training curve of the same logistic regression model using the same data set for renewal ratio in Fig. 7. Illustrated in Fig. 7a, when the top ratio k is small, the training curve is not stable. But in general, the larger r is, the stabler the training curve is. And the final loss value is also lower with larger r. As a trade-off, the running time increases when a larger renewal ratio is adopted. Meanwhile, the priority estimation is more accurate, which improves the quality of the selected top-k data points. On the other hand, shown in Fig. 7b, given a large top ratio, like 10%, the renewal ratio, except some extreme values, just slightly affects the convergence speed. It is because that when k is large enough, the average gradient of the top-k data points will be very close to the average gradient of all data points, even if the top-k data points are selected totally randomly.

3) Top Ratio and Renewal Ration Combination: We also examine the relationship between k and r when $k + r$ is fixed, *i.e.,* the same amount of gradients are calculated in each iteration. In this scenario, the running time for each iteration is roughly the same. In Fig. 8, we compare the training time of several k, r combinations for the LR model on the synthesized data set and a CNN model on the real-world MNIST data set. In both experiments, the combinations with smaller k and larger r (solid lines in the figure) lead to better results in terms of both training time and stability. The combinations with larger k and smaller r (dashed lines in the figure) bump drastically in the for the LR model and converge slower in the CNN model.



(a) Logistic regression (b) convolutional neural network

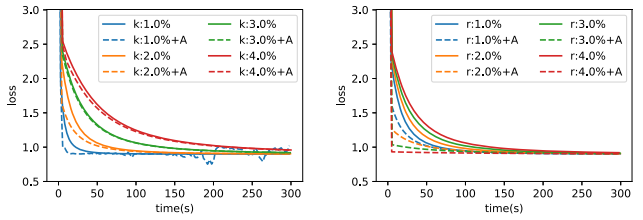
Fig. 8. Impact of the Combination of Top Ratio k and Renewal Ratio r

As a conclusion, we can determine the total number of processed data points in each iteration *i.e.,* $k + r$ using the techniques people determine the best mini-batch ratio for SGD. And based on that, we should make the top ratio k smaller than the renewal ratio r.

C. Aggressive Optimization

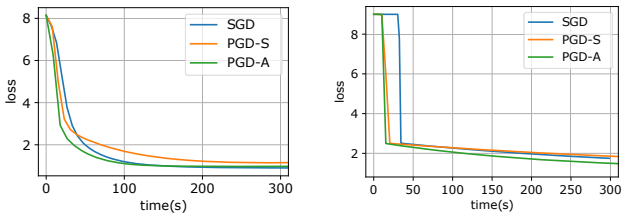
Our PGD framework aggressively utilizes all calculated gradients. We compare the performance of PGD with the aggressive optimization and a basic version without it in Fig. 9 using the CNN model with the MNIST data set. Because the aggressive version updates the priority values more frequently, the quality of selected top-k data points is higher than those without the aggressive optimization. As we can see that in both experiments, the aggressive versions (dashed lines, annotated with a suffix "-A") converges 10%-500% faster than the basic version (solid lines). And with the help of the aggressive update, the larger the renewal ratio r is, the faster the framework is.

[1] Source Code: https://github.com/yxtj/FAB



(a) With fixed top ratio (r=1%) (b) With fixed renewal ratio (k=2%)

Fig. 9. Comparison for PGD with and without Aggressive Optimization



(a) MLP (b) CNN

Fig. 10. Convergence Speed of PGD Compared with SGD

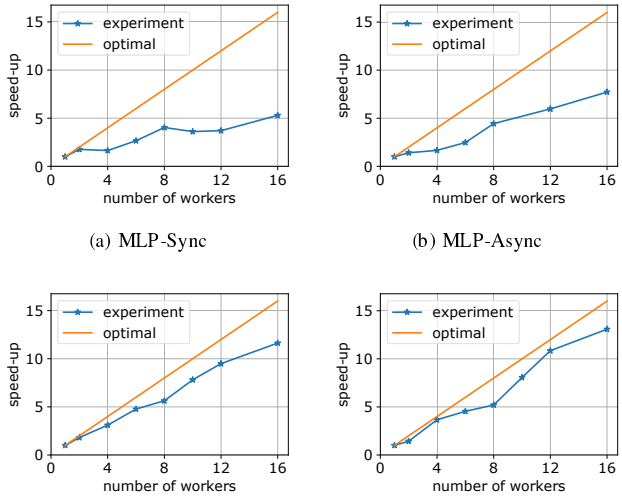
D. Convergence Time

We evaluate the convergence time of PGD for the MLP and CNN model using the real-world MNIST data set. In this experiment, we adopt a 3-layer MLP with 300 hidden neurons. We design a CNN model containing two 5×5 convolution layers activated with the ReLU function, two 5×5 max-pooling layers and a fully-connected layer. There are 10 and 20 convolution kernels in the two convolutional layers respectively. Based on the parameter impact discussion above, we set the top ratio and renewal ratio to 1% by some profiling results. Correspondingly, we set the mini-batch size to 2% for SGD in comparison.

Shown in Fig. 10, PGD converges 2X-5X faster than the typical SGD at the beginning of the training process. It is consistent with the ΔL contribution we shown in Fig. 1. With the training process, the difference in loss function contribution among data points becomes smaller. Therefore, starting from a certain time, the overhead of prioritization overwhelms its benefit. As a result, the basic PGD without aggressive optimization (annotated with PGD-S) becomes slower than SGD from that moment. While the aggressive version (annotated with PGD-A) is able to make use of all computed gradients. Compared with calculating gradient, selecting top-k data points takes very little time. Therefore, in the worst case, PGD is able to perform almost the same as SGD. Overall, PGD converges up to 1.6X faster than SGD.

E. Scalability

Shown in Fig. 11, PGD scales linearly to the number of workers. In the experiments, we test the scalability using both synchronous parallelization and asynchronous parallelization



(a) MLP-Sync (b) MLP-Async

(c) CNN-Sync (d) CNN-Async

Fig. 11. Scalability of PGD

on the MLP and CNN model. We calculate the speed-up by comparing the time spent in reaching the identical loss value with the time of the 1-worker PGD case. For both models, the asynchronous mode scales better than the corresponding synchronous mode. The CNN model scales better than the MLP model. It is because that each worker performs top-k selection locally and relatively the MLP model spends more time in the priority selection. Actually, calculating the gradient for one data point on CNN consumes more time than MLP, while the prioritization time for each data point is fixed. As a result, given the same amount of total running time, the CNN model spends more time updating the model parameter.

VI. Related Works

In recent years, many efficient or dedicated devices are designed or adopted to accelerate the training process. Similar to the 3D rendering, in the training of machine learning models, especially deep neural networks, identical operations are taken on a huge amount of floating-point data. Graphics Processing Units (GPU) are designed with high parallelism for higher throughput of floating-point computation. Therefore, GPUs are employed by the machine learning community and widely used by many popular systems, including Torch [7], Tensorflow [8], Caffe [9] and MXNet [10]. Compared with CPUs, GPUs reduce the training time by 10X-100X. The field-programmable gate array (FPGA) is a highly customized hardware with high parallelism and low cost. There are works about accelerating the training and testing of machine learning models with FPGA [11]–[13]. Google's specialized hardware, Tensor Processing Unit (TPU), is reported to be 15X-30X faster than CPUs and GPUs and 10X-80X more power-efficiency [5].

Some researchers work on improving training algorithms. NAG [21] accelerates SGD in the relevant direction and

dampens oscillations by adopting the momentum of previous gradients. Recently, researchers focus on the adaptive learning rates and propose many related algorithms, including RMSProp [14], ADAM [15], Nadam [16] and AdaGrad [17]. For example, AdaGrad and RMSProp calculate the learning rates according to the previous gradients. Nadam combines the adaptive learning rates ideas of Adam and the Nesterov momentum from NAG. While most algorithms focus on the learning rate, some other researchers report that in some cases instead of decaying the learning rate, we can get the same learning curve by increasing the batch size [22].

The importance difference among data points is noticed by many researchers [20], [23], [24]. The most common idea is to filter out the low-quality data points before feeding them into the training process. Millard *et al.* analyze the importance of data points and proposed a criterion of data selection for image classification with the random forest model in [23]. Similarly, Gong *et al.* investigate the influence of data selection for GPS data [24]. AutoClean is an iterative data cleaning system that automatically learns the quality of data points [20]. It applies machine learning technologies to greatly reduce the human interaction while data cleaning.

Instead of pruning out low-quality data points, some researchers utilize the data points discriminatively according to their importance. Jiang *et al.* prioritize candidate training samples as functions of their test trial performance, including correctness and confidence [19]. They perform a test process on a partially trained model. Using domain knowledge like signal-noise-ratio, Sivasankaran *et al.* proposed a weighting algorithm for acoustic model [18].

Some researchers adopt the online decision about priority values in some fields, especially the deep reinforcement learning [25]–[27]. A deep Q-network learns how to take actions given a certain state from history experiences. The reward of an experience is quantified based on its TD-error [26]. Instead of uniformly sampling the history, Schaul *et al.* report that by sampling history experiences with probabilities based on their rewards, the training time is reduced and the test performance of the trained model is improved [25].

VII. Conclusion

In this paper, we show that during the training process of a machine learning model, the contribution of different data points can be highly different. We quantify the contribution of a data point as how much this data point decreases the loss function. Based on that, we find that a small portion of data points contributes most decrease of the loss function value. Therefore, we propose a Prioritized Gradient Descent (PGD) framework which updates the model parameter using the data points with the highest contribution. To overcome the challenge of the overhead in computing the priority values, we design a priority value estimation mechanism. It learns and estimates the priority values of all data points online. We implement our distributed framework and evaluate it with both synthesized and real-world data set. Our evaluation results show that PGD gets up to 1.6X speedup over the typical gradient descent and scales linearly.

VIII. Acknowledgment

This work was supported in part by National Science Foundation Grants CNS-1815412 and CNS-1908536.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] O. M. Parkhi, A. Vedaldi, A. Zisserman *et al.*, "Deep face recognition." in *bmvc*, vol. 1, no. 3, 2015, p. 6.

[3] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[4] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS workshop*, no. CONF, 2011.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[10] 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 preprint arXiv:1512.01274*, 2015.

[11] D. Unnikrishnan, S. G. Virupaksha, L. Krishnan, L. Gao, and R. Tessier, "Accelerating iterative algorithms with asynchronous accumulative updates on fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013, pp. 66–73.

[12] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2016.

[13] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 110.

[14] G. Hinton, "Neural networks for machine learning (lecture 6)," rMSprop.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[16] T. Dozat, "Incorporating nesterov momentum into adam.(2016)," *Dostupné z: http://cs229. stanford. edu/proj2015/054_report. pdf*, 2016.

[17] L. Luo, Y. Xiong, and Y. Liu, "Adaptive gradient methods with dynamic bound of learning rate," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg3g2R9FX

[18] S. Sivasankaran, E. Vincent, and I. Illina, "Discriminative importance weighting of augmented training data for acoustic model training," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 4885–4889.

[19] Z. Jiang, X. Zhu, W.-t. Tan, and R. Liston, "Training sample selection for deep learning of distributed data," in *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2017, pp. 2189–2193.

[20] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "Activeclean: interactive data cleaning for statistical modeling," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 948–959, 2016.
[21] Y. Nesterov, "A method for unconstrained convex minimization problem with the rate of convergence o (1/k^ 2)," in *Doklady AN USSR*, vol. 269, 1983, pp. 543–547.
[22] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.
[23] K. Millard and M. Richardson, "On the importance of training data sample selection in random forest image classification: A case study in peatland ecosystem mapping," *Remote sensing*, vol. 7, no. 7, pp. 8489–8515, 2015.
[24] L. Gong, R. Kanamori, and T. Yamamoto, "Data selection in machine learning for identifying trip purposes and travel modes from longitudinal gps data collection lasting for seasons," *Travel Behaviour and Society*, vol. 11, pp. 131–140, 2018.
[25] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.
[26] J. Zhai, Q. Liu, Z. Zhang, S. Zhong, H. Zhu, P. Zhang, and C. Sun, "Deep q-learning with prioritized sampling," in *International conference on neural information processing*. Springer, 2016, pp. 13–22.
[27] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, "Distributed prioritized experience replay," *arXiv preprint arXiv:1803.00933*, 2018.