Alexandria Engineering Journal (2022) 61, 12005–12017

HOSTED BY

ELSEVIER

Alexandria University

Alexandria Engineering Journal

www.elsevier.com/locate/aej
www.sciencedirect.com

An efficient algorithm for data parallelism based on stochastic optimization

Khalid Abdulaziz Alnowibet [a], **Imran Khan** [b], **Karam M. Sallam** [c], **Ali Wagdy Mohamed** [d,e,*]

[a] *Statistics and Operations Research Department, College of Science, King Saud University, PO Box 2455, Riyadh 11451, Kingdom of Saudi Arabia*
[b] *Department of Electrical Engineering, University of Engineering & Technology, Peshawar 814, Pakistan*
[c] *School of IT and Systems, University of Canberra, ACT 2601, Australia*
[d] *Operations Research Department, Faculty of Graduate Studies for Statistical Research, Cairo University, Giza 12613, Egypt*
[e] *Department of Mathematics and Actuarial Science, The American University in Cairo, New Cairo, Egypt*

Received 8 April 2022; revised 18 May 2022; accepted 30 May 2022

KEYWORDS

Stochastic optimization;
Data analysis;
Queuing theory;
Network analysis

Abstract Deep neural network models can achieve greater performance in numerous machine learning tasks by raising the depth of the model and the amount of training data samples. However, these essential procedures will proportionally raise the cost of training deep neural network models. Accelerating the training process of deep neural network models in a distributed computing environment has become the most often utilized strategy for developers in order to better cope with a huge quantity of training overhead. The current deep neural network model is the stochastic gradient descent (SGD) technique. It is one of the most widely used training techniques in network models, although it is prone to gradient obsolescence during parallelization, which impacts the overall convergence. The majority of present solutions are geared at high-performance nodes with minor performance changes. Few studies have taken into account the cluster environment in high-performance computing (HPC), where the performance of each node varies substantially. A dynamic batch size stochastic gradient descent approach based on performance-aware technology is suggested to address the aforesaid difficulties (DBS-SGD). By assessing the processing capacity of each node, this method dynamically allocates the minibatch of each node, guaranteeing that the update time of each iteration between nodes is essentially the same, lowering the average gradient of the node. The suggested approach may successfully solve the asynchronous update strategy's gradient outdated problem. The Mnist and cifar10 are two widely used image classification benchmarks, that are employed as training data sets, and the approach is compared with the asyn-

* Corresponding author at: Department of Mathematics and Actuarial Science, The American University in Cairo, New Cairo, Egypt.
E-mail addresses: knowibet@ksu.edu.sa (K. Abdulaziz Alnowibet), imran_khan@uetpeshawar.edu.pk (I. Khan), karam.sallam@canberra.edu.au (K.M. Sallam), aliwagdy@aucegypt.edu (A. Wagdy Mohamed).

Peer review under responsibility of Faculty of Engineering, Alexandria University.

https://doi.org/10.1016/j.aej.2022.05.052
1110-0168 © 2022 THE AUTHORS. Published by Elsevier BV on behalf of Faculty of Engineering, Alexandria University
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

chronous stochastic gradient descent (ASGD) technique. The experimental findings demonstrate that the proposed algorithm has better performance as compared with existing algorithms.

© 2022 THE AUTHORS. Published by Elsevier BV on behalf of Faculty of Engineering, Alexandria University This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

In the past ten years, deep learning (DL) [1–5] algorithms have been widely used in image classification, speech recognition, natural language processing, autonomous driving, and other fields and have had a huge impact [6–9], which is largely due to deeper neural network models and relatively massive training data. However, complex network models and massive training data greatly increases the training overhead of neural networks [10–16].

Since large-scale deep neural network training is limited by limited computing resources, researchers have carried out a lot of work to accelerate the neural network training using distributed computing frameworks [17–22]. For example, literature [23–24] established a cluster of multiple commercial CPUs, and perform parallel training on data on top of it. Literature [25–26] proposed a heterogeneous computing platform composed of nodes that integrate multiple GPUs for parallel acceleration. Reference [27] used high-performance computing clusters for training, etc.

In order to enable neural network algorithms to run under the distributed computing framework, researchers usually adopt corresponding parallel models. Currently popular deep neural network parallel models are mainly divided into two types: model parallelism [28] and data parallelism [29].

Model parallelism means that different hardware (CPU/GPU) in the distributed system is responsible for different parts of the neural network model. As shown in Fig. 1, different network layers in the neural network model are assigned to different hardware for execution, or within the same layer. Different parameters are assigned to different hardware [30]. Data parallelism means that each node in a distributed system has a copy of the same model, as shown in Fig. 2, each node is assigned different training data, and then parameter server combines the calculation results of all machines in a certain way, and finally completes the parameter update and broadcast.

In most cases, the communication and synchronization overhead brought by model parallelism far exceeds that of data parallelism, and the acceleration is relatively low. Therefore, most of the current research work is concentrated in the field of data parallelism. The stochastic gradient descent (SGD) algorithm [31] has many advantages such as simple implementation, fast convergence speed, and high operating efficiency, so it has been widely used in deep neural network algorithms. Deep learning distributed strategies can be divided into research based on parameter distribution and model consistency-based research. The research of parameter distribution form includes two categories: decentralized [32–33] and centralized [34–35]. In the decentralized method, each worker node maintains a local parameter and The specified communication graph performs parameter transfer and update. In the centralized approach, a central node called a parameter server maintains the global parameters, while the rest of the worker nodes do the computational work. The model consistency-based research includes the synchronous update model [36–37], the asynchronous update model [38] and the outdated synchronous update model [39]. In the synchronous update model, the SSGD (synchronization-SGD) algorithm is based on the SGD algorithm simplest and most straightforward distributed implementation was first proposed by [36]. In this algorithm, the host simply divides and maps the data to the corresponding worker nodes, and by using the display fence for forced synchronization, the host can ensure that each worker node uses the same model parameters for gradient calculation, but each node is forced to wait for the slowest node in each iteration. The cost of this synchronization strategy is to reduce the scalability of the SSGD algorithm and runtime performance. To address this problem, reference [38]

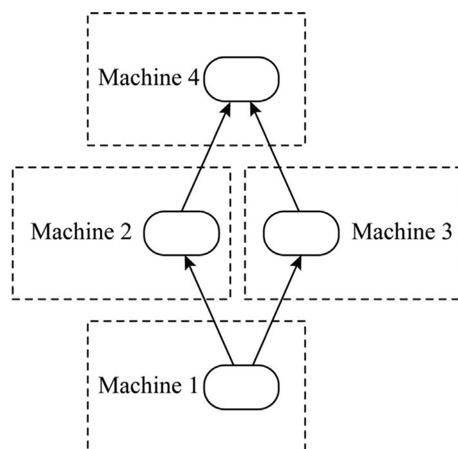


Fig. 1 Parallelism model.

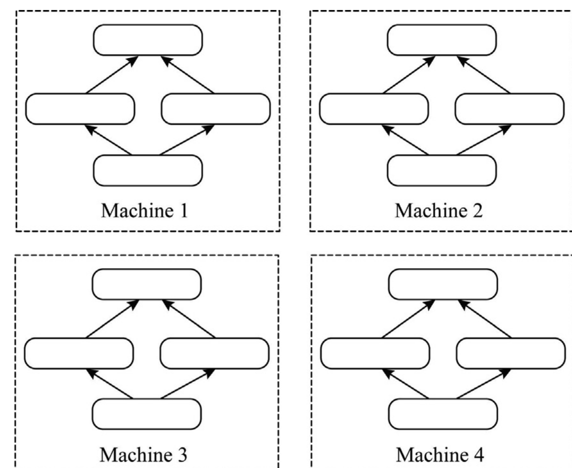


Fig. 2 Parallelism data.

proposed an asynchronization-SGD (ASGD) algorithm based on the asynchronous update model, which overcomes the above shortcomings by eliminating the synchronization barriers between nodes. However, this asynchronous behavior inevitably introduces "gradient obsolescence" to the entire system. Since it takes time to calculate the gradient when a node calculates the gradient value and submits it to the server to update the parameters, the global parameters may have been updated several times. Therefore, the updated parameters will have a certain amount of outdated. Therefore, it is not the latest convergence direction. This phenomenon is called "gradient outdated". Therefore, under the premise of determining the number of iterations, the model trained by ASGD is often much less effective than the model trained by SSGD, and the serious gradient value outdated will significantly slow down. The convergence speed of the network model even stops converging completely. The outdated synchronous update model [39] is a trade-off between the synchronous and the asynchronous update models. When the fast-running worker nodes reach a certain gradient staleness, they will be forced to perform global synchronization operations to mitigate the effects of gradient staleness issues.

At present, there are many studies on the gradient obsolescence problem in asynchronous gradient descent algorithms. The basic ideas of most asynchronous stochastic gradient descent algorithms are roughly the same. The main difference is that different optimization strategies are used to reduce obsolescence and the influence is caused by gradient. Reference [39] used the delayed synchronization technology on the parameter server side. For every certain number of iterations, the server will perform a forced synchronization to eliminate the influence of gradient obsolescence. Reference [40] proposed a variant of the ASGD algorithm, which dynamically adjusts the learning rate according to the gradient stale value at different times, while having strong convergence. Reference [41] proposed to use 5 % to 10% of the nodes can replace those nodes with slower performance at any time, so as to ensure the consistency of the performance of all nodes in the system as much as possible. Although, these studies have eliminated the impact of gradient obsolescence in different ways, the related algorithms are all running on a homogeneous platform with average performance of each node, the commercial-grade cloud computing environment is not fully considered. Reference [42] mentioned three typical application scenarios in commercial-grade cloud computing platforms:

(1) Network and computing heterogeneity. In a cluster consisting of thousands of commercial machines provided by different data centers, it is difficult for each machine to have the same generation of processors, which makes the performance of nodes vary.
(2) Resource competition. In a multi-user-oriented cluster, multiple application requests usually need to be processed at the same time, then these requests will inevitably compete for scarce hardware resources on the same node. Therefore, different instances of an application usually have different runtimes.
(3) Spot Instance. Amazon EC2 provides services to users by using Spot Instance. For substantial cost savings, instances with different resource requirements can run on the same cluster (e.g. m4.large has 2 cores, a1.4xlarge has 16 cores). When a user submits a task on such a cluster, the worker nodes assigned with low-performance hardware need to spend more time computing the same amount of data than the worker nodes assigned with high-performance hardware.

These three application scenarios lead to different computing power provided by different nodes. At this time, the difference in computing performance between nodes leads to the problem of gradient obsolescence. Based on the problem of computing performance differences in heterogeneous environments, reference [42] through testing, it is found that the running time of the SSP model strategy on heterogeneous clusters is about 80% slower than normal. At the same time, the learning rate is dynamically optimized on the basis of the SSP model to solve the problems caused by the heterogeneity, while reference is based on decentralized research and combined with the backup works method proposed by [43] to solve the heterogeneous problem. This paper proposes a performance-aware technology-based dynamic batch size stochastic gradient descent algorithm. It can effectively allocate the appropriate amount of data according to the different performance of each node, and finally make each node update asynchronously to achieve essentially the same number of iterations in all cases, thereby eliminating the effects of gradient staleness issues caused by performance differences.

The main work of this paper has two aspects:

(1) A model that can quickly quantify the performance of each node is proposed. This model quantifies the current performance of the node by counting and calculating the latest N running times of the node, so as to facilitate the direct evaluation of the performance of each node.
(2) Based on the above node performance quantification model, a dynamic batch size stochastic gradient descent algorithm DBS-SGD based on performance-aware technology is proposed. After obtaining the quantified value of each node's performance, the algorithm can allocate the work of each node in real time. Therefore, the running time of each iteration of each node is basically the same, thus solving the problem of gradient obsolescence caused by performance differences and eliminating the influence of gradient obsolescence on the system.
(3) The remaining of the paper is organized as follows. Section 2 explains the distributed neural network framework. Section 3 provides the algorithm design and analysis. Section 4 provides the simulation results evaluation while Section 5 concludes the paper.

2. Distributed framework based on neural network

In this section, we will analyze the relevant theories of distributed deep learning, the overall framework of parameter servers, and the synchronization strategy in distributed systems.

2.1. Deep learning

The neural network algorithm performs a nonlinear transformation $f_\theta : X \rightarrow Y$ on a set of inputs and their corresponding outputs by adjusting their own parameters , where θ is a set

of adjustable parameters (or called weights). For example, in the environment of supervised learning the image classification task below, X is the input image, and Y is the classification label corresponding to the input image. The deep neural network divides the parameter set θ into multiple layers, each layer is composed of a linear transformation and a corresponding nonlinear transformation function. The non-linear transformation functions include sigmoid, tanh functions, etc. In a feedforward deep neural network, each layer is in sequence so that the output of the $L-1$ layer is passed to the input of the L layer, and the last layer manages the output of the entire network.

As a result $\widehat{Y} = f_\theta(X).1$, the training algorithm of the neural network tries to find an optimal set of parameters θ^* that can minimize the difference between the actual result Y and the predicted result. The optimization algorithm used is usually gradient descent, which is widely used in $\widehat{Y}$ in the neural network model, the gradient descent algorithm takes the negative direction of the gradient as the search direction, and iteratively solves the optimal value or local optimal value of the objective function or loss function. Since the objective function needs to be derived, theoretically gradient descent algorithm can achieve linear convergence and can only solve differentiable convex functions. The expression of the loss function is

$$\min\left\{F(w) = \frac{1}{N}\sum_{n=1}^{N} f(w, \varepsilon_n)\right\} \tag{1}$$

Among them, ε_n represents the nth data in the dataset, w is the parameter of the entire neural network, and $f(w, \varepsilon_n)$ represents the composite loss function using the parameters w and ε_n.

The neural network model uses the back-propagation algorithm to derive the parameters in the network and update them in turn. The expression for updating the parameters in each iteration is as follows:

$$\nabla F(w_j) = \frac{1}{N}\sum_{n=1}^{N} \nabla f(w_j, \varepsilon_n) \tag{2}$$

$$w_{j+1} = w_j - \eta \nabla F(w_j) \tag{3}$$

where η is the learning rate and ∇F is the first derivative of the loss function.

Since the gradient descent algorithm needs to be calculated on the entire data set, the calculation time is long, and it is easy to fall into the local optimal solution. Therefore, there are many variants of gradient descent, such as stochastic gradient descent algorithm, batch gradient descent algorithm, etc. The expressions for stochastic gradient descent and mini-batch gradient descent are

$$w_{j+1} = w_j - \nabla F(w_j) = w_j - \nabla f(w_j, \varepsilon_n) \tag{4}$$

$$w_{j+1} = w_j - \nabla F(w_j) = \frac{1}{k}\sum_{n=1}^{k} \nabla f(w_j, \varepsilon_n) \tag{5}$$

(1) Stochastic gradient descent algorithm. Only one data is selected each time for gradient calculation, and then iterates.
(2) Mini batch gradient descent (MBGD). It is a compromise between the gradient descent algorithm and the stochastic gradient descent algorithm. It takes k data to calculate the gradient, $1 < k \ll n$.

2.2. Architecture of parameter

In the cluster environment, the parameter server architecture (as shown in Fig. 3) is currently the mainstream architecture supporting distributed parallel deep learning. Its earliest prototype is Distbelief developed by the Google team, which is a large-scale architecture abstracted from the process of continuous use.

In this architecture, the parameters and operations of worker nodes are defined as follows.

(1) λ. The number of working nodes.
(2) μ. The size of the minibatch used by each worker node to compute gradients.
(3) η. Learning rate, also known as step size.
(4) Timestamp. Use the count value of the scalar clock to represent the gradient weight timestamp i. The beginning point $i = 0$. The value of the related timestamp will grow by one with each weight change. The gradient's timestamp is the same as the weight's date.
(5) $\tau_{i,l}$. Gradient stale value of worker node l. Worker node 1 sends gradient value with timestamp j to i, $i \geq j$. In the form of $i - j$ to compute the gradient staleness $\tau_{i,l}$, where $\tau_{i,l} \geq 0$ for any i and l.

There are 4 sequential operations performed by each worker node:

(1) getMiniatch. Randomly select a mini-batch from the training set.
(2) pullWeights. Send a request to the parameter server to apply for the current parameter set.
(3) calcGradient. Calculates the gradient value based on the training error generated by the selected mini-batch samples.
(4) pushGradient. To the parameter server, provide the determined gradient.

The parameter server evenly retains all model parameters and performs two operations:

(1) sumGradients. Receive and aggregate the gradient values calculated by each worker node.
(2) applyUpdate. Multiply the learning rate by the calculated gradient value and update the global parameters.

On the server side, different update strategies can be used, including synchronous update strategies and asynchronous update strategies. Synchronous update strategies can also be called hard synchronization strategies (synchronous-SGD), and asynchronous update strategies can also be called soft synchronization strategies (asynchronous-SGD). Section 3 will discuss the different effects of different update strategies on the gradient.

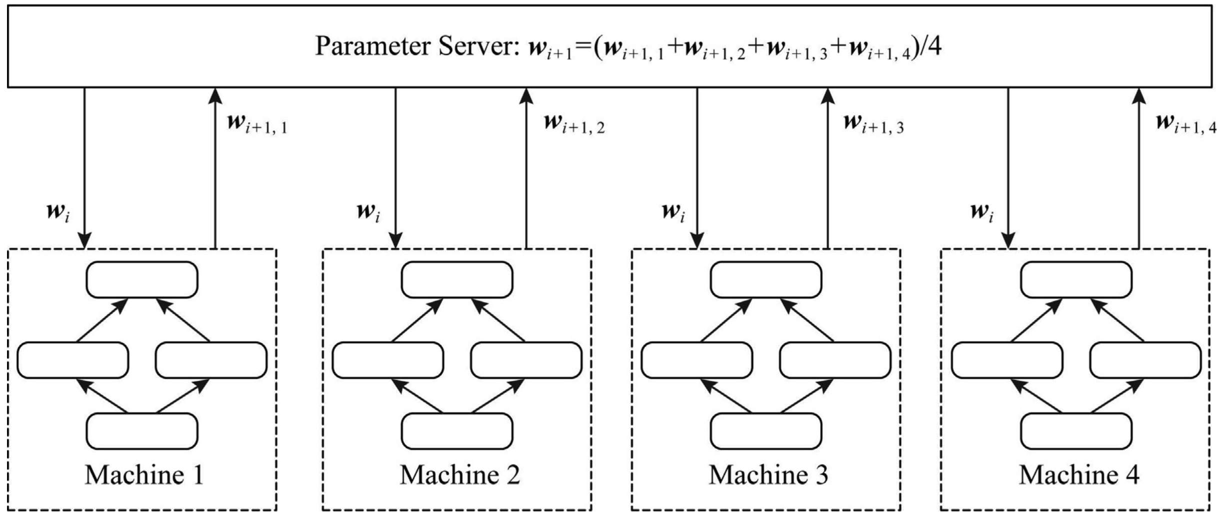


Fig. 3 Parameter architecture.

2.3. Update mechanism of parameter

The SSGD is the most intuitive implementation of the gradient descent algorithm on distributed systems. The algorithm uses the parameter averaging method [43] to process the updated value of the weight on the server side. At the same time, it can be proved that the SSGD algorithm in a distributed environment can be mathematically equivalent. For this algorithm in a stand-alone environment, the proof process is shown in Appendix A. From the formulas (A1) and (A2) in Appendix A, it can be known that the parameter server updates the i-th value by aggregating the gradient change values submitted by each node after the calculation of the i-th step is completed $+1$ step weight, thus guaranteeing that the gradient stale amount produced by the i-th step is 0, i.e. there is no gradient staleness problem, so this algorithm produces the model with the best accuracy.

The consistency of parameter updates is ensured by forced synchronization in each iteration. When there are fewer working nodes, the algorithm has better scalability. However, in large-scale clusters, the forced synchronization mechanism will cause the barrel effect, which will make the entire system becomes very inefficient, as shown in Fig. 4.

To address the aforementioned issues, the soft synchronization technique recommends transferring the parameter computation process from the parameter server to the node, making parameter updating easier.:

$$\Delta w_{i,j} = \alpha \nabla L_j \tag{6}$$

$$w_{i+1} = \boldsymbol{w_i} - \frac{1}{n}\sum_{j=1}^{N} \Delta w_{i,j} = \frac{1}{n}\sum_{j=1}^{n} w_i - \alpha \nabla L_j = \frac{1}{n}\sum_{j=1}^{n} w_{i,j} \tag{7}$$

When the server still uses the synchronization method to update the parameters, this method is completely equivalent to the parameter averaging method. However, after the server relaxes the synchronization constraints, when the server receives the new parameters calculated by any one node, it immediately updates the global parameters, instead of waiting for the calculation results of all nodes. This strategy becomes the ASGD scheme, as shown in Fig. 5. It has two main advantages:

(1) The data throughput of the entire distributed system is increased, and each worker node can devote more time to computing instead of waiting for other nodes;
(2) Compared with the synchronous update method, each node can obtain the information (parameter update value) of other nodes faster.

The asynchronous update strategy not only obtains the above advantages, but also brings new overhead. When the

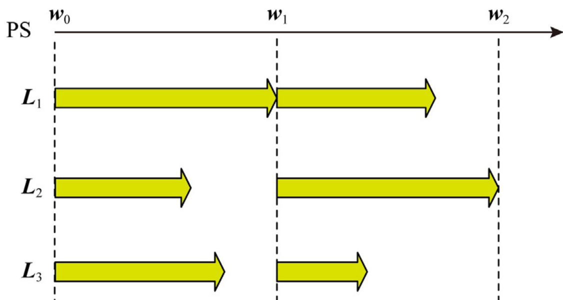


Fig. 4 Parameter update flow via synchronous mechanism.

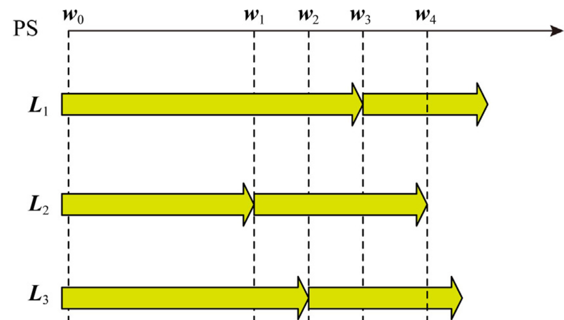


Fig. 5 Parameter update flow via asynchronous mechanism.

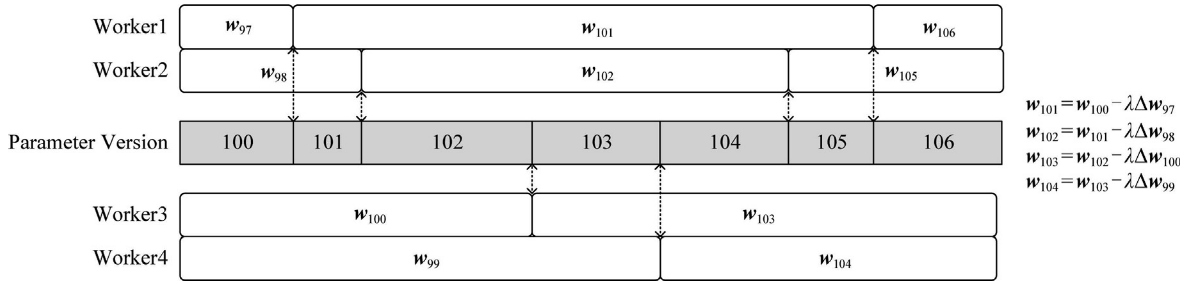


Fig. 6 Global update mechanism.

global parameter adopts the asynchronous update strategy, the problem of gradient obsolescence also follows. Because the calculation of the gradient itself takes time, when a node When the gradient value is calculated and submitted to the parameter server, the global parameters maintained by the parameter server may have been updated many times. The specific process is shown in Fig. 6.

As can be seen from the parameter update process shown in Fig. 6, the value of w_{101} is calculated based on w_{97}, not the most recent value of w_{100}. The difference between the subscripts of the two weights is greater than one, which means that the gradient of update is not based on the latest parameters, so the gradient is outdated. Reference [27] discussed that the minimum average outdated value of the gradient under this model is the number of working nodes n, in fact, due to the different number of iteration clock cycles of each node, the gradient outdated value is generally greater than n, and the gradient outdated problem is more serious due to the increase in the number of nodes in the system and the increase in the performance difference between nodes.

In addition, there is an n-soft synchronous update strategy based on ASGD strategy. As shown in Fig. 7, for gradient computation, each worker node acquires the most recent parameter values from the parameter server and pushes the computed results to the server. After collecting at least $c = \lfloor \lambda/n \rfloor$ parameters, the parameter server conducts a global update (n:1 to λ), then the parameter update strategy of the n-soft synchronization strategy is:

$$c = \lfloor \lambda/n \rfloor \tag{8}$$

$$\nabla w_i = \frac{1}{c} \sum_{l=1}^{c} \nabla w_l \tag{9}$$

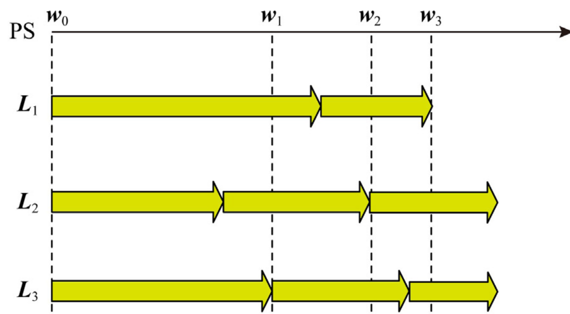


Fig. 7 Update mechanism based on top-k

$$w_{i+1} = w_i - \alpha \nabla w_i \tag{10}$$

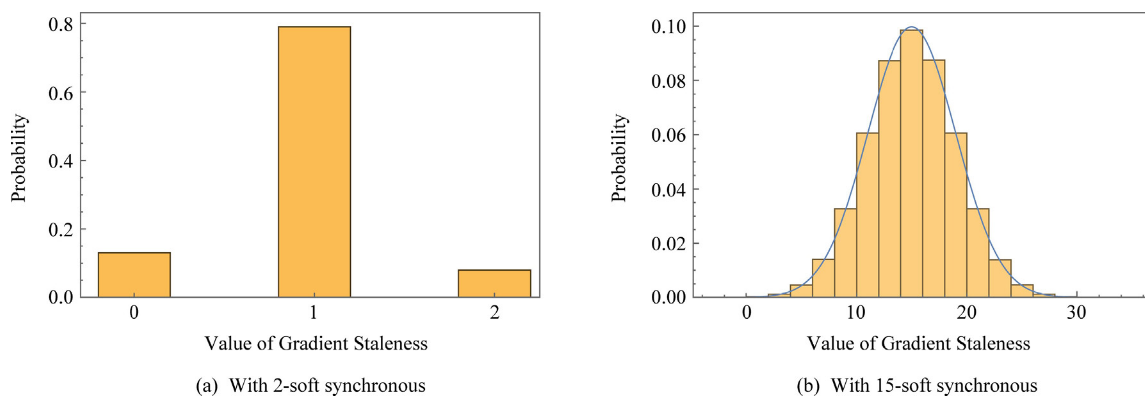
This update strategy can better solve the gradient obsolescence problem caused by the asynchronous update strategy, but it is only suitable for the case where the performance of each worker node is relatively average. Section 3 will quantitatively analyze the gradient obsolescence problem of different strategies.

3. Algorithm design

The SSGD algorithm mentioned in Section 2 uses an explicit synchronization barrier. In the process of each step forward iteration, the parameter server will perform a forced aggregation operation, so the parameters obtained by each worker node from the server side are all is the latest, that is, the gradient outdated value is 0. But when the update strategy becomes asynchronous, the gradient outdated problem is inevitably introduced. In order to solve this problem, we must first analyze the causes of gradient outdated.

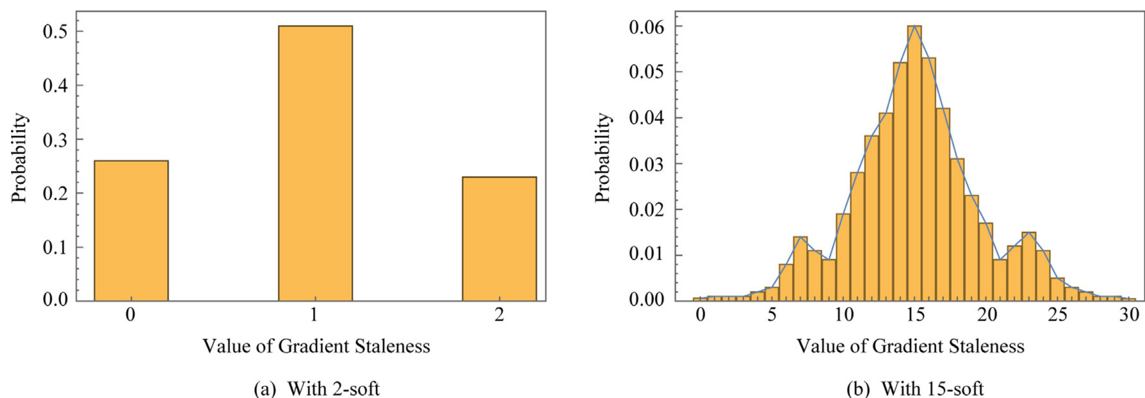
3.1. Obsolescence performance of inconsistent node

Assuming that there are 30 working nodes in the parameter server system, that is, $\lambda = 30$. Then for the 1-soft update strategy, the server needs to collect the gradient values from the working nodes 30 times, and then perform parameter reduction and update operations. Similarly, For the 2-soft update strategy, the parameter server needs to collect $30/2 = 15$ gradient values from worker nodes before updating the parameters. According to [27,40], for each worker node, no matter how the size of its *batch_size* and the size of the training model change, the empirical experimental results similar to Fig. 8 can be obtained. From Fig. 8(a), it can be seen that for the 1-soft update strategy, the gradient value is out of date and is 0, 1, 2. For the 15-soft update strategy in Fig. 8(b), the average gradient stale value becomes 0 to 30. If it is a 30-soft synchronization strategy, the parameter server receives computations from any worker node After the result, the parameters are updated, which is equivalent to the ASGD algorithm. Therefore, for this algorithm, the outdated value of the gradient is 0 to 60, which is Gaussian distribution. It can be seen that in the ASGD algorithm, the increase in the number of working nodes leads to the gradient increase of obsolete value. As can be seen from Fig. 8(a)-(b), for the n-soft strategy, the average gradient obsolete value is close to n. This is based on the situation that the performance of each worker node is completely consistent. If the performance of each working node



(a) With 2-soft synchronous (b) With 15-soft synchronous

Fig. 8 Two and fifteen soft gradient distributions with synchronous mechanism.



(a) With 2-soft (b) With 15-soft

Fig. 9 Two and fifteen soft gradient distributions with different mechanisms.

is quite different, the average gradient stale value generated by the algorithm will become larger, and the gradient stale value will change from a relatively stable Gaussian distribution to a more random distribution. As shown in Fig. 9(a)-(b), the gradient value is at nAt the same time, where the probability of occurrence of the two sides is low, the probability of their occurrence has increased instead. This will cause the loss function to vibrate violently during the convergence process, which will affect the final convergence efficiency and results. The idea of DBM-SGD algorithm is to dynamically allocate different workloads to each node according to the real-time performance of each node, that is, the value of *batch_size*. Through this operation, the time of each iteration of each node is basically the same. Then in this case, the distribution of the gradient outdated values of the entire system will regress to the normal Gaussian model again, the average gradient outdated amount will be reduced to n, tend to be stable, and finally converge to normal.

3.2. Influence of batch size on the performance

In previous studies [31,33,39–40], the value m of *batch_size* is usually set to a fixed value, so, the amount of data trained by each node is the same, and m is generally selected temporarily or through empirical testing. Reference [44] proposed that dynamically adjusting the value of *batch_size* in the process of neural network training can accelerate the convergence of the algorithm, which indicates that the value of *batch_size* is actually a key variable, and its changes will have intricate effects on the performance of the neural network. On the one hand, the variance (that is, the degree of convergence) of the stochastic gradient decreases linearly with the value of m, then a larger batch value can obtain more specific gradient information, thereby reducing the overall number of convergence steps. On the other hand, each time the computational cost of iteration increases linearly with the value of m, so, the variance and time cost can be traded off linearly. From the formula of the stochastic gradient descent algorithm for each update of the parameters, it can be seen that the stochastic gradient descent algorithm only uses 1 value calculated from the sample data is used for parameter update, while the gradient descent algorithm uses the value calculated from all the data to update. Reference [45] mathematically proved that the final convergence trend of the two is consistent. The gradient descent algorithm that updates the sample has the optimal direction of convergence (the gradient is the steepest), while the stochastic gradient descent algorithm using 1 sample data has large fluctuations in the direction of convergence each time. Because it is not the best way to go, so the result of this jitter will lead to an increase in the overall number of iterations of the function on the one hand, and on the other hand, make the function have a high probability to jump out of the local optimal area, so as to find a better solution.

The mini batch SGD algorithm compromises the value of *batch_size* of each node ($1 < k \ll n$), and each node does not have to calculate the entire training sample set, which saves

time and does not make each node due to the randomness of a single data. There are too many fluctuations in the direction of the second convergence. From this, the basic idea of the DMB-SGD algorithm can be obtained as follows: the first choice is to count the performance indicators of each node's recent N runs (in this algorithm, each node runs the neural network once calculated wall time is used as the performance parameter of the node), and then different workloads are allocated to each node in real time, so that under the asynchronous update strategy of each node. The wall time of each node for gradient calculation is roughly the same. Keep the overall gradient stale value in a low range ([27] have proven that ideally, in a distributed environment, the gradient stale value is at least n, where n is the number of worker nodes in the distributed system), so that while maximizing the use of the computing power of each working node, it will not cause a greater impact on the convergence trend and iteration times of the entire model.

According to the analysis, it can be concluded that, in each iteration of neural network training, the size of the training data, that is, the value of *batch_size* determines the accuracy of the convergence direction of this iteration. The smaller the value of *batch_size*, the greater the volatility of convergence. If *batch_size* = 1, the equivalent for SGD algorithm. The larger the value of *batch_size*, the more accurate the direction of convergence. If the value of *batch_size* is n, it is equivalent to the GD algorithm. Therefore, the value of *batch_size* is dynamically adjusted in the process of each iteration (whether distributed or stand-alone) does not affect the overall convergence trend.

3.3. Performance evaluation of DBS-SGD scheme

Combined with the analysis of the value of *batch_size* in Section 3.2, in a cluster that uses processors as computing resources, the performance-aware model can predict and obtain the current performance data of the node by analyzing the working state of the node. The factors that affect the operation of the program in a single node are: There are many, including hardware architecture, operating system, compilation environment, programming framework, algorithm, etc. The more parameters involved in the model, the more accurate the predicted results, but at the same time the cost of collecting data will also increase. The goal is to predict the operating efficiency of worker nodes in real time, so the timeliness of data should be given priority. Secondly, in heterogeneous application scenarios, the underlying hardware architectures are often different, so, the overhead of data collection also increases. It can be seen that, the parameters selected in the performance-aware model should have both timeliness and accessibility. The performance evaluation of a single node is mainly represented by relatively intuitive parameters (the number of iterations and running time). The main reason for the outdated gradient is that there are too many nodes. In addition, the computing performance between nodes is inconsistent, which makes the time interval for the nodes to update the parameters. The main application scenario of this paper is the latter case. In order to minimize the impact of gradient obsolescence, the iteration time of each node should be consistent, rather than the final number of iterations. At the same time, the running time is easy to obtain and can more intuitively reflect the current running efficiency of the node. Therefore, it is more reasonable to choose the running time of one node iteration as the quantitative standard. From the Nth round (that is, the average running time is not calculated during the initial operation), and then each iteration takes the average running time of the node for the most recent N times (after analysis of the results of many experiments, the statistical effect is better when the value of N is set to 5 to 10) as the current performance index of each node, in order to quantitatively evaluate the computing performance of all nodes.

Combined with the above performance perception model, the operations performed by the DBS-SGD algorithm are as follows: each node uses a set of vectors of length N to save the working state of the local node, and the vector saves the running time sequence of the last N iterations of the local node, and each round of iterations completion, add the running time of the latest round to the vector to replace the running time of the oldest round. If the local node is the benchmark node, then it needs to send this value to the server while counting the running time of the latest N iterations. The running time of the benchmark is broadcast to other worker nodes through the server. In the deep learning algorithm, the amount of computation for each node to compute one instance process is the same. So the time used for each iteration is proportional to the amount of training data. In the algorithm, node 0 is used as the benchmark. Each iteration of the local node is compared with the average running time of the latest N rounds of node 0. If the ratio falls within a certain range, it means that the performance of the node is similar to that of the benchmark node. Its *batch_size* does not need to be adjusted, otherwise the *batch_size* of the node needs to be enlarged or reduced in equal proportion. Through the scheduling of the algorithm, the running time of all nodes in the whole system is roughly maintained at the same level as the running time of node 0, which ultimately avoids the gradient outdated problem caused by the large performance difference between nodes. The algorithm also needs to add a 1-dimensional vector on the parameter server side to save the running time of node 0. Because the running time of each iteration in the parameters are relatively intuitive, it is very convenient to obtain and count them, and it is easy to ensure their timeliness. At the same time, the calculation operation of quantizing nodes is not complicated. So even if performance quantification is required in each iteration, the increased computational overhead is also negligible relative to the calculation process of deep learning itself. Based on these theories and models, Algorithms 1 and 2 can be obtained.

Algorithm 1: Working node algorithm

Input: CPU clock count of the last m times of the current node
Output: The *batch_size* value to be used by the current node in the next round
1: Receive the latest global parameters and the last m computation time of node 0 from the server
2: for *iteration number* i to N
3: S = *sum*(calculation time of the last m times of node 0)/sum (calculation time of the current node of the
last m times)
4: If current node $\neq 0$
5: if $|1 - S| \geq A$
6: $batch_size = X \times S$

a (*continued*)

Algorithm 1: Working node algorithm

7: else
8: *batch_size* remains unchanged
9: end if
10: else
11: *batch_size* = X
12: Send the current running time of node 0 to the server
13: end if
14: Apply the *batch_size* and the received global parameters for batch gradient calculation
15: Push the number of CPU clocks of the latest iteration of the current node into the stack, and pop the latest $m + 1$ data from the stack
16: Transfer the calculation results to the server for global parameter update
17: End for

Algorithm 2: Server-side algorithm

1: While true do
2: if received request ≠ node 0
3: Send the last m iteration time of node 0, and send the latest updated global parameters
4: end if
5: If receives the information of node 0
6: Push the time of the latest iteration of node 0 into the stack, push the latest m + 1 data out of the stack, and send the latest updated global parameters
7: end if
8: end while

Note: The values of X and m need to be given by experiment to give better values, not the smaller or the bigger the better. A is a constant.

Ideally, the proposed algorithm can make each node generate an implicit fence, which can approach or achieve the effect of synchronous update under the constraints of the explicit synchronous fence.

4. Simulation results

In order to test the efficiency of the algorithm, this paper evaluates the efficiency of the single machine running and using the ASGD model and the algorithm optimized method on the Mnist and cifar10 benchmarks.

The test environment of the experiment is Intel ® Core TM i7-6700 CPU with a core frequency of 3.40 GHz and a memory of 8 GB. The software used in the experiment is tensorflow 1.6-CPU version. In order to simulate a distributed operating environment with large performance differences, this experiment adopts container mechanism to limit the number of CPU resources or CPU cores that the container can use through -c -- cpu - share or -- cpus. In the experimental test, the distributed training scenarios of 4 and 8 nodes are simulated respectively, in order to simulate a heterogeneous environment, half of the node CPUs are configured with 1 CPU load (the docker configuration parameter is -- cpus = 1), and the other half of the node CPUs are configured with 1.5 times the CPU load (the docker configuration parameter is -- cpus = 1.5), indicating that the CPU load of the first half of the nodes is about 70% of that of the second half of the nodes. The memory configuration is to allocate 5 GB of memory evenly to each container. The container system creates a unique IPC namespace for each container by default. There is no shared memory between them, and the shared memory mechanism is not used in the experiment in order to simulate the distributed environment more realistically.

The Mnist (mixed National Institute of Standards and Technology) dataset comes from the National Institute of Standards and Technology in the United States. It is a handwritten dataset and is a very general dataset for classification testing. It includes a training image set, 1 A training label set, a test image set and a test label set, in which the 50,000 training data consists of 28 × 28 pixel handwritten digital pictures, and the test data contains 10,000 pictures.

The cifar10 dataset has a total of 60,000 color images, which are 32 × 32 pixels and are divided into 10 classes with 6,000 images in each class. 50,000 images in each class are used for training, constituting 5 training batches, each batch has 10,000 images. The other 10,000 images are used for testing, forming a separate batch. The data in the test batch is taken from each of the ten categories, and 1,000 images are randomly selected for each category. After extraction, the remaining random permutations make up the training batch. The number of different types of images in a training batch is not necessarily the same. In general, there are 5,000 images in each type of training batch. It should be noted that these ten types are all are independent and do not overlap.

For the Mnist dataset, a fully connected neural network is constructed, which includes two hidden layers and 1 output layer, the hidden layer contains 200 neurons, and the output layer is a 10-way output function softmax(), the function generates the corresponding probability distribution over 10 output classes, the learning rate is set to 0.001, and the loss function is optimized. The optimization function is adamoptimizer(). For setting the *batch_size*, it is set to 256. Each worker node is visited in order to read *batch_size* data from the dataset as sub-datasets.

For the cifar10 dataset, the original data is first preprocessed, the 32 × 32 image is resized from the center to a size of 24 × 24, and then operations such as inversion, adjustment of brightness, and contrast are performed to increase the capacity of the dataset to ensure Better learning effect. Then, the LeNet convolutional neural network is also used, with two convolutional layers. Each convolutional layer is connected to a pooling layer, and the size of each convolutional layer convolution kernel is five, and the number of kernels is 5. The kernel size of the pooling layer is 2, and the step size is 2. The method is to take the maximum value of the region, and then a fully connected 2-layer neural network, including 128 neurons, and an output layer with 10 output functions. The softmax() function generates probability distributions over 10 output classes, the learning rate is set to 0.0001, and the loss function is optimized. The optimization function is adamoptimizer(). For the *batch_size* setting, similar to the Mnist dataset, just each worker node randomly reads *batch_size* data from the dataset as sub-datasets in order.

The size of the Mnist dataset is about 55 MB (about 11 MB after compression), the cifar10 dataset is about 180 MB (about 140 MB after compression), and the LeNet model contains about 60,000 parameters, and the data size is about 240 KB. The memory space given to docker is much larger than the storage space required for datasets and network models, and the entire training process can be performed in memory, so there is no delay in data access between memory and hard disk.

The simulated network between nodes is a 2-layer switching network, and its network delay is between 0.04 and 0.08 μs. At the same time, the calculation time of one iteration of the neural network needs at least 0.15 s (the LeNet model is used for both data sets in the experiment). So the overhead due to network communication between containers is negligible relative to computation.

First, the performance prediction model is tested. On the node, the fully associative neural network and the LeNet network are used for the Mnist data set. The LeNet network is used for the cifar10 data set. The test results are shown in Table 1. In the cifar data, set the number of iterations to 1000, we can see that the running time of the program is linearly related to the value of *batch_size*, and the value of *batch_size* doubles, the running time of the program also doubles. The same phenomenon can also be found on the Mnist dataset. It is observed that on the Mnist data set, when using the fully associative network test, the test results are not very accurate when the number of iterations is set to 1000. The reason for this phenomenon is that the Mnist data set is a black and white image, and the structure is relatively simple. Compared with the convolutional neural network (CNN) network, the amount of computation is relatively small, and the number of iterations set at the end is also lower. Therefore, the running time of the program is very short and thus, the results are not stable. When the number of iterations is set to 2000, it can be seen that the results have returned to normal. This test result is consistent with the theoretical analysis of the previous performance prediction model.

The data set used in Figs. 10–13 is Mnist, which is compared with the proposed, ASGD, SSGD, and *n*-soft algorithm in the case of 4 nodes, where the parameter in the *n*-soft algorithm is set to 2, that is, each server receiving the update values of 2 nodes, the global parameters are updated once, and the SSGD algorithm based on the synchronous update strategy has a gradient staleness of 0, which is used as a benchmark to test the final convergence accuracy of the remaining algorithms. The value of *batch_size* in the four algorithms is set is 400, and all use the LeNet model for training. In Figs. 10-11, ASGD-0.78 indicates that the percentage of the lowest performance node and the highest performance node after performance quantification is 78% among the 4 nodes. From Fig. 10, we can see the optimization of the proposed algorithm, within the same number of iteration steps, its convergence speed is faster than ASGD and *n*-soft algorithm, and slightly lower than SSGD algorithm. Fig. 11 shows that after algorithm optimization, the loss function value is the same as the number of iterations is reduced by 60%. Fig. 12 shows the speedup achieved by the algorithm when scaled to 8 nodes. It can be seen in Fig. 13 that in the heterogeneous environment, after the algorithm optimization, the accuracy convergence curve

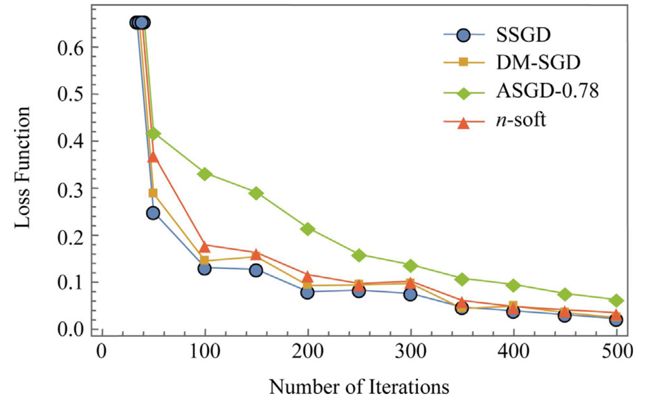


Fig. 10 Comparison of loss function.

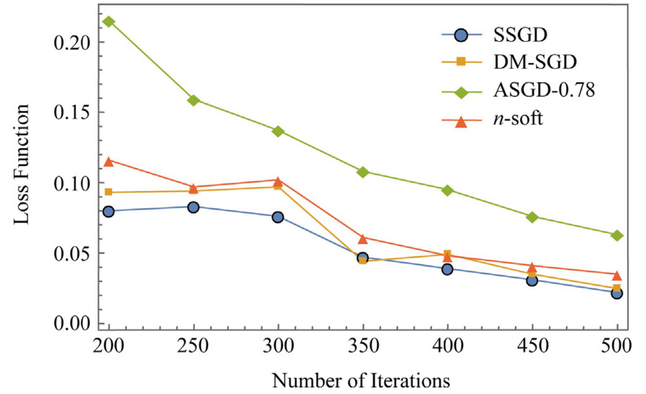


Fig. 11 Comparison of loss function with Mnist 60% reduction.

Table 1 Comparison of running time of algorithms.

block_size	Running time (s)			Running time (min)
	No. of iterations = 1000	No. of iterations = 2000	No. of iterations = 200	No. of iterations = 1000
	Mnist full		Mnist LeNet	Cifar 10 LeNet
100	2.14	4.34	33.26	2.28
200	3.16	6.27	63.50	4.50
300	6.08	11.84	122.76	9.26
400	8.43	16.65	186.19	14.10
500	10.45	21.06	246.79	18.42

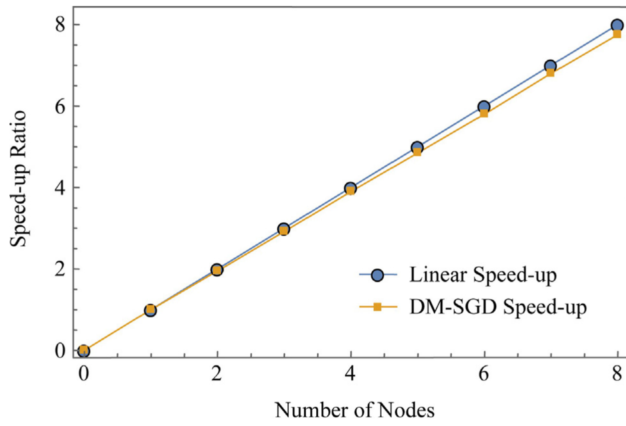


Fig. 12 Comparison of ratio of speed-up versus number of nodes.

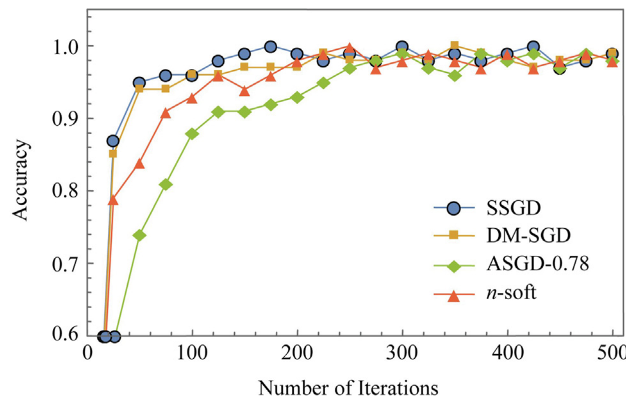


Fig. 13 Comparison of accuracy of the algorithms.

of the neural network model is the steepest and reaches the optimum in fewer iteration steps. The trend is close to that of the SSGD algorithm based on the synchronization strategy.

For the cifar data set, a comparison test with the ASGD, SSGD, and *n*-soft algorithms was also done, where the parameter in the *n*-soft algorithm is set to 2. That is, the server updates the global once every time it receives the update value of any two nodes. Like the experiment on the Mnist dataset, the SSGD algorithm based on the synchronization strategy is also used as the benchmark to test the final convergence accuracy of the other algorithms. The value of *batch_size* in the four algorithms is set to 400, and the LeNet model is used for training. From Fig. 14, we can see that in the case of four working nodes, the greater the performance difference between the nodes, the more frequently the jitter occurs during the convergence process, and the larger is the jitter amplitude. The serious gradient outdated problem leads to frequent jitter in the convergence process and affects the final convergence accuracy. After the algorithm is optimized, the convergence process of the neural network appears to be relatively stable, and the convergence accuracy is also improved by 10%, its effect is better than ASGD and *n*-soft algorithm, slightly lower than SSGD algorithm based on synchronization strategy. It can also be seen in Fig. 15 that in heterogeneous environment, the loss function value of neural network after algorithm optimization is also 10% decrease, its convergence trend is also faster than that of ASGD and *n*-soft algorithm, which is close to the convergence curve of SSGD algorithm based on synchronization strategy. In Fig. 16, the speedup achieved when the algorithm is extended to 8 nodes is shown, which is close to linear acceleration.

The overhead generated by using this algorithm is described in Section 3. Only each time the node communicates with the parameter server, an additional set of vectors about the latest m iterations of node 0 is transmitted, and its data size of the *batch_size* is less than 1% relative to the global parameters. At the same time, the calculation amount of the dynamic *batch_size* on the working node is also less than 1% relative to the calculation amount of the deep neural network algorithm. Therefore, the overhead generated by the proposed algorithm is very small, which can be ignored, and the test results on 2 datasets also indicated. The image size used in the Mnist dataset is small, and the neural network structure for classification is relatively simple. In this case, the proposed algorithm can still achieve approximately linear acceleration, indicating that the overhead generated has little effect on the overall convergence. For cifar10 with a larger dataset and a more complex convolutional neural network, the overhead

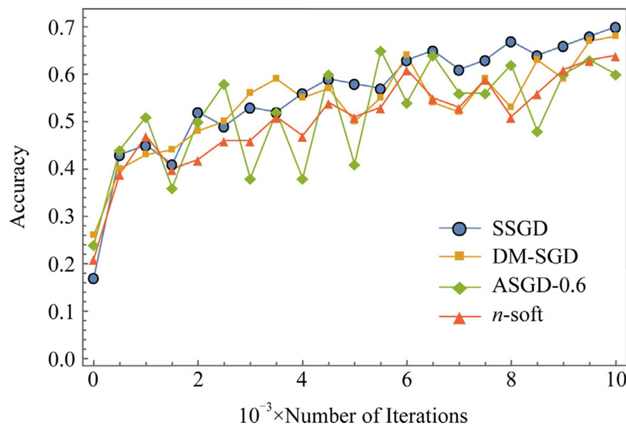


Fig. 14 Accuracy comparison of the algorithms versus number of iterations for cifar 10.

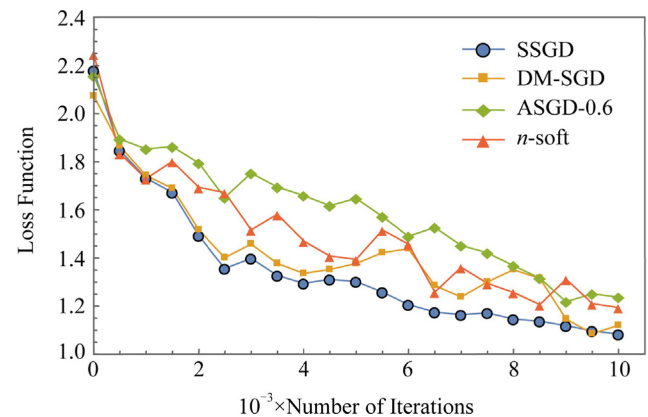


Fig. 15 Comparison of loss function versus number of iterations.

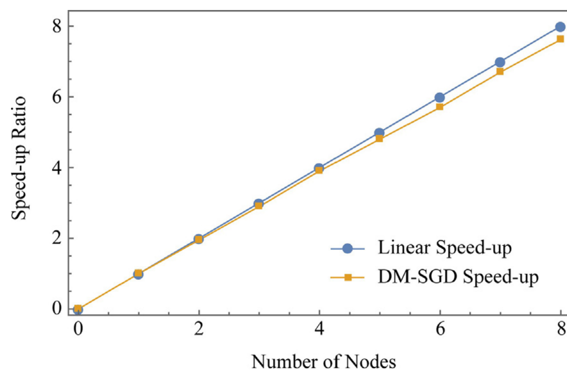


Fig. 16 Comparison of cifar 10 speedup ratio versus number of nodes.

generated by the proposed algorithm accounts for ratio will be smaller so that it does not affect the convergence.

It can be seen from the experiment that compared with the asynchronous gradient descent and the *n*-soft algorithms, the proposed algorithm removes the waiting time and makes full use of all computing resources. However, the proposed algorithm reduces the impact of gradient obsolescence through reasonable task allocation, so we can see faster convergence and better performance in the same amount of time.

5. Conclusion

In order to solve the problem of gradient obsolescence caused by the parallel process of deep learning data, the proposed model solves the problem that the performance of each node in distributed systems is not easy to quantify. The convergence of the algorithm is theoretically proved. It detects the computing performance of each working node in real time through the performance-aware model, and dynamically allocates the corresponding workload to the working nodes with different computing performance. Thus, it solves the gradient outdated problem and making the neural network converge and the speed is accelerated. The experimental results show that the proposed algorithm makes the single iteration time of each node in the distributed system consistent, and solves the problem of gradient obsolescence. Compared with the traditional ASGD and improved *n*-soft algorithms, the proposed algorithm has a faster convergence speed and stability of the neural network under the same number of iterations, and the algorithm can obtain a nearly linear speedup ratio.

Future work includes two directions: (1) how to assign corresponding weights to the unequal tasks of each node to better quantify the contribution of each node in each submission of gradient updates; (2) combine the first research direction, the algorithm is applied to the synchronous update strategy, which can make the number of neural network convergence to reach the theoretical minimum.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors present their appreciation to King Saud University for funding the publication of this research through Researchers Supporting Program (RSP-2021/305), King Saud University, Riyadh, Saudi Arabia.

Funding statement

The research publication is funded by Researchers Supporting Program at King Saud University, (Project# RSP-2021/305)

References

[1] M.A. Garadi, A. Mohammed, A.K. Ali, X. Du, I. Ali, et al, A survey of machine and deep learning methods for internet of things (IoT) security, IEEE Commun. Surv. Tutorials 22 (3) (2020) 1646–1685.
[2] S. Li, L. Li, B. Xu, Y. Feng, H. Zhou, Research of a reliable constraint algorithm on MIMO signal detection, Int. J. Embedded Syst. 12 (2) (2020) 13–26.
[3] K. Shah, M. Irfan, A. Ullah, Q. Mdallal, K. Ansari, et al, Computational study on the dynamics of fractional order differential equations with application, Chaos Solitons Fractals 157 (4) (2022) 1813–1837.
[4] K. Shah, F. Jarad, T. Abdeljawad, Stable numerical results to a class of time-space fractional partial differential equations via spectral method, J. Adv. Res. 25 (7) (2020) 39–48.
[5] K. Shah, H. Naz, M. Sarwar, T. Abdelawad, On spectral numerical method for variable-order partial differential equations, AIMS Mathematics, 7 (6), pp. 10422–10438.
[6] S. Bushnaq, K. Shah, S. Tahir, K. Ansari, M. Sarwar, et al, Computation of numerical solutions to variable order fractional differential equations by using non-orthogonal basis, AIMS Mathematics 7 (6) (2022) 10917–10938.
[7] S. Bashir, M. H. Alsharif, I. Khan, M. A. Albreem, A. Sali, B. Mohd Ali, W. Noh, MIMO-terahertz in 6G nano-communications: channel modeling and analysis, Comput. Mater. Continua 66 (1) (2020) 263–274.
[8] A. Silva, S. Teodore, R. Dinis, A. Gameiro, Iterative frequency-domain detection for IA-precoded MC-CDMA systems, IEEE Trans. Commun. 62 (4) (2014) 1240–1248.
[9] A.I. Taloba, An artificial neural network mechanism for optimizing the water treatment process and desalination process, Alexandria Eng. J. 61 (12) (2022) 9287–9295.
[10] S.S.I. Ismail, R.F. Mansour, R.M. Abd El-Aziz, A.I. Taloba, A. D. Doulamis, Efficient E-Mail Spam Detection Strategy Using Genetic Decision Tree Processing with NLP Features, Comput. Intell. Neurosci. 2022 (2022) 1–16.
[11] D. Castanheira, A. Silva, A. Gameiro, Set optimization for efficient interference alignment in heterogeneous networks, IEEE Trans. Wireless Commun. 13 (10) (2014) 5648–5660.
[12] S. Teodoro, A. Silva, R. Dinis, F. Barradas, P.M. Cabral, A. Gameiro, Theoretical analysis of nonlinear amplification effects in massive MIMO systems, IEEE Access 7 (2019) 172277–172289.
[13] F. Jameel, T. Ristaniemi, I. Khan, B.M. Lee, Simultaneous harvest-and-transmit ambient backscatter communications under Rayleigh fading, EURASIP J. Wireless Commun. Network. 19 (1) (2019) 1–9.
[14] Q. Alsafasfeh, O.A. Saraereh, A. Ali, L.A. Tarawneh, I. Khan, et al, Efficient power control framework for small-cell heterogeneous networks, Sensors 20 (5) (2020) 1–14.
[15] K.M. Awan, M. Nadeem, A.S. Sadiq, A. Alghushami, I. Khan, et al, Smart handoff technique for internet of vehicles

communication using dynamic edge-backup node, Electronics 9 (3) (2020) 1–17.
[16] W. Shahjehan, S. Bashir, S.L. Mohammed, A.B. Fakhri, A. Adebayo Isaiah, I. Khan, P. Uthansakul, Efficient modulation scheme for intermediate relay-aided IoT networks, Appl. Sci. 10 (6) (2020) 2126.
[17] B.M. Lee, M. Patil, P. Hunt, I. Khan, An easy network onboarding scheme for internet of things network, IEEE Access 7 (2018) 8763–8772.
[18] O.A. Saraereh, A. Alsaraira, I. Khan, B.J. Choi, A hybrid energy harvesting design for on-body internet-of-things (IoT) networks, Sensors 20 (2) (2020) 1–14.
[19] T. Jabeen, Z. Ali, W.U. Khan, F. Jameel, I. Khan, G.A.S. Sidhu, B.J. Choi, Joint power allocation and link selection for multi-carrier buffer aided relay network, Electronics 8 (6) (2019) 686.
[20] W. Wang, M. Jacob, X. Mou, Y. Shi, Y. Eldar, A survey of deep learning techniques for cybersecurity in mobile networks, IEEE Commun. Surv. Tutorials 23 (3) (2021) 1920–1955.
[21] Y. Lecun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.
[22] R. Zhou, F. Liu, C. Gravelle, Deep learning for modulation recognition: a survey with a demonstration, IEEE Access 8 (2020) 67366–67376.
[23] K. He, X. Zhang, S. Ren, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, USA, 2016, pp. 770–778.
[24] S. Hoermann, M. Bach and K. Dietmayer, Dynamic occupancy grid prediction for urban autonomous driving: a deep learning approach with fully automatic labeling, in: IEE International Conference on Robotics and Automation, New York, USA, pp. 2056–2063, 2018.
[25] M. Aspri, G. Tsagkataki, P. Tsakalides, Distributed training inference of deep learning models for multi-modal land cover classification, Remote Sensing 12 (17) (2020) 1–19.
[26] Y. Ko, S. Kim, SHAT: a novel asynchronous training algorithm that provides fast model convergence in distributed deep learning, Appl. Sci. 12 (1) (2022) 1–13.
[27] F. Seide, H. Fu, J. Droppo, 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs, in: IEEE Annual International Conference of the Speech Communication Association, Hong Kong, 2014, pp. 1058–1062.
[28] B. Guo, Y. Liu, C. Zhang, A partition based gradient compression algorithm for distributed training in AIoT, Sensors J. 21 (6) (2021) 1–18.
[29] T. Knez, O. Machidon, V. Pejovic, Self-adaptive approximate mobile deep learning, Electronics 10 (23) (2021) 1–17.
[30] A. Coates, B. Huval, T. Wang, Deep learning with cots HPC systems, in: International Conference on Machine Learning, New York, USA, pp. 1337–1345, 2013.
[31] P. Faerber, K. Asanovic, Parallel neural network training on multi-spert, in: International Conference on Algorithms and Architectures for Parallel Processing, Glasgow, UK, pp. 659–666, 1997.
[32] V. Niculescu, R. Stefanica, Tries-based parallel solutions for generating perfect crosswords grids, Algorithms 15 (1) (2022) 1–14.
[33] F. Lopes, J. Ferreira, M. Fernandes, Parallel implementation on FPGA of support vector machines using stochastic gradient descent, Electronics 8 (6) (2019) 1–14.
[34] H. Gao, M. Lee, G. Yu, Z. Zhou, A graph neural network based decentralized learning scheme, Sens. J. 22 (3) (2022) 1–18.
[35] J. Jiang, L. Hu, C. Hu, J. Liu, Z. Wang, BACombo-bandwidth-aware decentralized federated learning, Electron. J. 9 (3) (2020) 1–15.
[36] Z. Song, Y. Gu, Z. Wang, G. Yu, DRPS: efficient disk-resident parameter servers for distributed machine learning, Front. Comput. Sci. 16 (2022) 975–987.
[37] H. Cui, H. Zhang, G. Ganger, GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server, in: European Conference on Computer Systems, New York, USA, 2016, pp. 1–6.
[38] J. Langford, A. Smola, M. Zinkevich, Slow learners are fast, in: International Conference on Neural Information Processing Systems¸ Washington DC, USA, pp. 2331–2339, 2009.
[39] F. Iandola, M. Moskewicz, K. Ashraf, FireCaffe: near-linear acceleration of deep neural network training on computer clusters, in: IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, USA, pp. 2592–2600, 2016.
[40] S. Zheng, Q. Meng, T. Wang, Asynchronous stochastic gradient descent with delay compensation, in: International Conference on Machine Learning, New York, USA, pp. 4120–4129, 2017.
[41] J. Zhang, H. Tu, Y. Ren, J. Wan, L. Zhou, et al, An adaptive synchronous parallel strategy for distributed machine learning, IEEE Access 6 (2018) 19222–19230.
[42] D. Niu, T. Liu, T. Cai, S. Zhou, The asynchronous training algorithm based on sampling and mean fusion for distributed RNN, IEEE Access 8 (2019) 62439–62447.
[43] V. Campos, F. Sastre, M. Yagües, M. Bellver, X. Giró-i-Nieto, J. Torres, Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster, Procedia Comput. Sci. 108 (2017) 315–324.
[44] Q. Zhou, S. Guo, Z. Qu, P. Li, L. Li, et al, Petrel: heterogeneity-aware distributed deep learning via hybrid synchronization, IEEE Trans. Parallel Distrib. Syst. 32 (5) (2021) 1030–1043.
[45] R. Kennedy, T. Khoshgoftaar, F. Villanustre, T. Humphrey, A parallel and distributed stochastic gradient descent implementation using commodity clusters, J. Big Data 6 (2019) 1187–1194.