Available online at www.sciencedirect.com

ScienceDirect

JPDC

ELSEVIER

Journal of Parallel and Distributed Computing 00 (2023) 1–27

Accelerating Distributed Machine Learning with Model Compression and Graph Partition

Yubin Duan, Jie Wu*

Department of Computer and Information Sciences, Temple University, PA 19122, USA

Abstract

The rapid growth of data and parameter sizes of machine learning models makes it necessary to improve the efficiency of distributed training. It is observed that the communication cost usually is the bottleneck of distributed training systems. In this paper, we focus on the parameter server framework which is a widely deployed distributed learning framework. The frequent parameter pull, push, and synchronization among multiple machines leads to a huge communication volume. We aim to reduce the communication cost for the parameter server framework. Compressing the training model and optimizing the data and parameter allocation are two existing approaches to reducing communication costs. We jointly consider these two approaches and propose to optimize the data and parameter allocation after compression. Different from previous allocation schemes, the data sparsity property may no longer hold after compression. It brings additional opportunities and challenges for the allocation problem. We also consider the allocation problem for both linear and deep neural network (DNN) models. Fixed and dynamic partition algorithms are proposed accordingly. Experiments on real-world datasets show that our joint compression and partition scheme can efficiently reduce communication overhead for linear and DNN models.

Keywords: Data sparsity, distributed machine learning, graph partition, parameter server framework

1. Introduction

The efficiency of distributed machine learning systems becomes more and more important with the rapid growth of the training data volume and model parameter sizes. In the big data era, both sizes of training data and machine learning models can be extremely large. For example, the Alibaba click-through rate prediction dataset contains more than 12 billion data instances [1]. The BERT-large [2] model for natural language processing has 345 million parameters. A single machine may be infeasible to store all model parameters or lack sufficient computation power when training large models. It is common to train large

☆This research was supported in part by NSF grants CNS 2128378, CNS 2107014, CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, and CNS 1651947.

*Corresponding author

Email addresses: yubin.duan@temple.edu (Yubin Duan), jiewu@temple.edu (Jie Wu) models in distributed systems. However, the distributed training brings additional communication overhead. Worker machines participating in the training procedure need to maintain the parameters of the learning model. Therefore, it is necessary to reduce the communication overhead and improve the training efficiency.

In this paper, we focus on reducing the communication overhead for the parameter server framework [3], which is a distributed machine learning framework and is widely discussed and deployed in academia [4] and industry [5]. Logically, the parameter server structure contains a server node and multiple worker nodes. The server node maintains the whole parameter set. Training data is allocated into worker nodes. During training, worker nodes iteratively *pull* parameters from the server, *compute* parameter updates with local data, and *push* updates to the server. The server would aggregate parameter updates from all workers and modify the parameter set accordingly. In practice, using a single machine as the server node may cause congestion on its network interface. Therefore, efficient implementation usually partitions the parameter set into multiple machines [6]. Specifically, a machine would act as both worker and server nodes as illustrated in Fig. 1. This approach helps to distribute the communication workload for all worker machines. However, the communication overhead still can become a bottleneck of the system performance.

The large network traffic volume usually is the bottleneck of the parameter server framework. In particular, when training large machine learning models, the communication overhead significantly enlarges the overall training time. Even after distributing the communication workload, the communication overhead can take a large portion of the total training time. [7] shows that training with 300 GB data would cause about 4 TB network traffic volume when parameters are randomly allocated. Compared to the computation time consumption, the communication overhead is large even with high-speed network interfaces, such as 10 Gigabit Ethernet. Therefore, in addition to improving the power of computation units, shrinking the network traffic volume can efficiently accelerate the training process and improve the system performance.

Existing efforts made to reduce the network communication cost mainly include optimizing the model compression [8, 5] and partition [7, 9]. Both approaches utilize the sparsity of training data. Specifically, there usually are many zero entries in each data sample. One popular compression approach is applying hash functions on sparse data samples. After hashing, the size of every data instance is reduced. The model size can be significantly reduced accordingly [8]. However, the drawback of this approach is that the model accuracy drops after hashing. Moreover, a larger compression ratio usually means a more severe accuracy loss, which restricts the application scenarios of the model compression approach. The model partition approach reduces the network communication cost by mitigating the inter-machine network communication traffic into inner-machine communication. Specifically, training with a sparse data instance usually only

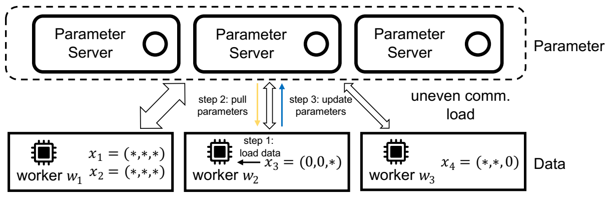


Figure 1. Communication patterns in the parameter server framework.

impacts and updates a small subset of parameters. Allocating data samples and their required parameters in the same machine can reduce the network traffic without harming the model accuracy. Considering that the inner-machine communication bandwidth is usually much greater than the inter-machine communication bandwidth, optimizing the data and parameter allocation can reduce the communication time consumption during training. However, the benefits are not significant when the number of partitions is too small or too large. Additionally, existing model partition methods only work for linear machine learning models and cannot be used for deep learning.

In this paper, we jointly consider the model compression and partition. In this approach, we can use a smaller compression ratio to achieve the same level of network traffic reduction, compared with merely applying the model compression. It helps to reduce the sacrifice of model accuracy when optimizing the network traffic. However, we cannot simply apply existing model partition algorithms after hashing. Model partition algorithms usually rely on the sparsity of training data, while the sparsity property may no longer hold after hashing. We need to develop an efficient model partition algorithm regardless of the sparsity of training data. The general model partition problem is NP-hard [7]. Additionally, we investigate the model partition problem for deep neural networks (DNNs). The challenge of partitioning DNN model parameters arises in the complex data and parameter correlation. Unlike training linear models, the parameter access locality in DNN training is unknown before training.

Instead of directly combining existing model compression and partition methods, we adapt the model partition algorithm for training data after hashing. Moreover, we propose to adaptively update parameter allocations when model pruning is applied to the training procedure. A dynamic parameter partition and allocation algorithm is proposed for deep learning models. Specifically, model compression, i.e. hashing, would lower the number of zero entries in each data instance, and change the ratio between data and parameter sizes. We investigate different heuristics for parameter partition, and our scheme can adaptively choose the proper heuristic to reduce the partition overhead. Additionally, we show the approximate ratio of our partition algorithm based on the submodular property of network communication cost function. Moreover, we propose a dynamic parameter partition for DNN models. It would estimate the parameter access locality during training and adjust the parameter allocation accordingly.

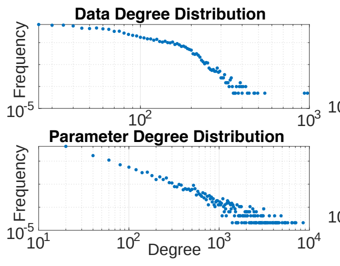
Our contributions are summarized as follows:

- We investigate the communication overhead of the training process in parameter server frameworks. We formulate an optimization problem that minimizes the bottleneck traffic volume during training.
- We jointly consider the model compression and partition methods to reduce the communication overhead. We investigate different heuristics to efficiently partition linear models.
- We propose a dynamic parameter partition scheme for DNN models. The scheme estimates the parameter access pattern and dynamically updates the model parameter partition and allocation during training.
- We evaluate our approach with real-world datasets. Our scheme can significantly reduce the network bottleneck traffic volume, improve the training efficiency, and accelerate the training process.

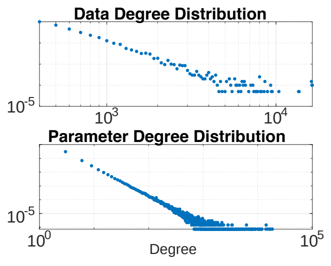
The remainder of the paper is organized as follows. Related works are reviewed in Section 2. Training procedures of machine learning models, hashing functions, network models, and problem formulations are introduced in Section 3. The method to allocation parameters for linear models is explained in Section 4. The dynamic partition method for the DNN models is shown in Section 5. Experiment settings and results are illustrated and analyzed in Section 6. Our paper is concluded in Section 7.

2. Related Work

DNN model simplification is a common approach to accelerate DNN training or inference jobs by modifying and simplifying DNN models. This approach investigates the trade-off between computation workload and model accuracy. There are some small-scale DNN models that are specially designed to be used on mobile devices. For example, [10] proposed a new small-scale DNN based on speaker verification. MobileNets [11] presented an efficient model for mobile computer vision applications. YOLO-lite [12] developed a real-time object detection DNN model that could run on mobile devices without GPUs, such as laptops or cellphones. In addition to designing a dedicated DNN, some systems could adaptively adjust the DNN size according to resource limitations. MCDNN [13] presented an optimizing compiler that can systematically trade off inference accuracy for resource usage. Taylor *et. al.* [14] proposed an adaptive scheme to select



(a) The rcv1.binary dataset



(b) The news20 dataset

Figure 2. Node degree distribution.

the most efficient DNN from a DNN pool at runtime. Wang *et. al.* [15] proposed to adaptively adjust the accuracy requirement based on the wireless network conditions. Wu *et. al.* [16] proposed an adaptive strategy to dynamically adjust the configuration. In this paper, we propose to reduce the communication cost of distributed model training by jointly optimizing model compression and parameter allocation.

Specifically, model pruning [17, 18, 19, 20, 21] can gradually reduce the DNN model size during the training process. Han *et. al.* [17] proposed the idea of learning connection importance during training and pruning unimportant connections. Li *et. al.* [18] proposed an optimization problem to automatically choose the thresholds in layer-wise magnitude-based pruning. Furthermore, Liu *et. al.* [19] proposed a framework to automatically determine hyperparameters for the pruning process. [22] jointly considered model compression and network architecture optimization and introduced a compiler-level automatic code generation framework. Wimmer *et. al.* [23] presented interspace pruning which used filters constructed in a dynamic interspace to avoid potential bias caused by standard unstructured pruning methods. Ma *et. al.* [24] presents a scheduled grow-and-prune method. In their method, a subset of DNN layers is grown dense first and then pruned back to sparse after several training iterations. The proposed method can efficiently compress model size and preserves the model quality at the same time. Different from those methods, we propose to dynamically partition and allocate DNN model parameters to accelerate distributed training by reducing communication costs. Our dynamic model partition and allocation approach can be combined with the model compress methods to further reduce the training cost.

In addition to compressing the model size, gradient compression can accelerate the DNN training process by reducing the communication overhead. Zhang *et. al.* [25] proposed a compression algorithm for distributed gradient descent with a guarantee on model convergence. Abdelmoniem *et. al.* [26] introduced an efficient compression technique to minimize the extra computation overhead for gradient compression.

Shi *et. al.* [27] further investigated the trade-off between computation and communication in gradient sparsification. Abrahamyan *et. al.* [28] exploited a neural network to build an autoencoder for identifying common information in gradients during distributed training. The captured common information can be used to avoid communication redundancy and improve communication efficiency. Yu *et. al.* [29] introduced a novel optimization method, which overlaps the gradient compression phase with part of the communication phase and hides the time consumption of the gradient compression. Different from those approaches, we consider the correlations between training data and model parameters and investigate their allocation problem, which can further reduce the communication cost beyond the compression.

3. Model

In this section, we first introduce the training procedures of machine learning models. Then, we briefly explain the application of hashing techniques in machine learning model compression. Moreover, we show the network communication pattern in the parameter server framework and identify the feasibility of compressing network traffic in this framework. Eventually, we formulated our problem with the objective of reducing the communication cost during distributed training.

3.1. Training Machine Learning Models

Training machine learning models, including Deep Neural Network (DNN) models, usually can be treated as a procedure of minimizing loss functions. *Loss functions* are used to evaluate the performance of machine learning models on the given training dataset. Typical loss functions can be categorized into two groups: regression loss and classification loss. For example, in regression models, the mean square error or L2 loss [30] can be used to measure the different model predictions and actual values. In classification models, the cross-entropy loss [31] is widely used to quantify the probability divergence between the predicted class and the actual labels. Training algorithms, such as stochastic gradient descent, would iteratively update the parameter weights in loss functions such that the function value on the training dataset is reduced. Training terminates when the loss value cannot be further reduced or an iteration threshold is reached. We notice that there are many zero or near-zero weight updates during training iterations when training machine learning models for click-through rate prediction [32]. Motivated by this observation, we study the correlation between training data and parameter updates with the objective of reducing the communication volume during training.

Formally, we use $D = \{d_1, d_2, \ldots, d_n\}$ to denote the set of n training data samples, where d_i is the i-th data vector. We focus on supervised learning models in this paper. Therefore, the *data vector* $d_i = (x_i, y_i)$, meaning it consists of a feature vector x_i and a groundtruth label y_i. The feature vector x_i is a vector that quantifies the input data. For example, the feature represents the occurrence frequency of a word in text classification applications. We use $P = \{p_1, p_2, \ldots, p_m\}$ to denote the *parameter* set of machine learning models. Each parameter p_i is a scalar value that is used to adjust the importance or weight of each feature in the loss function. In *linear models*, the parameter set usually consists of weights for features in x_i plus a bias term. Each feature is associated with a weight as its coefficient. Therefore, the size of parameters m usually equals the dimension of x_i plus one for the bias term. In *DNN models*, the parameter value represents the weights of connections between neurons. The number of parameters is mainly determined by the number and types of layers in the DNN model, and may greatly exceed the number of features. In a training iteration, not all parameters will be updated. In contrast, for both linear and DNN models, gradient vectors which are used for parameter updating are usually sparse [9, 17, 33].

We use bipartite graphs to model the correlation between data and parameters. If a training iteration using data d_i will update the value of parameter p_j, then we denote that d_i and p_j are correlated. For linear models, the correlation graph could be estimated without pre-training the model when training with the parameter server framework introduced in [6]. Zero elements in a data vector d_i will not update the value of the corresponding parameter during training. According to this property, we build a bipartite graph $G = (D \cup P, E)$ to represent the correlation between data samples and model parameters. There are two types of vertices which represent the data D and parameter P, respectively. The edge set $E \subseteq D \times P$ contains an edge between d_i and p_j if they are correlated. The graph G can be built before training by scanning nonzero entries for all $d_i \in D$. For DNN models, the correlation is *dynamic* and hard to predict since the loss function of DNN models is a *compound* function. Because of the chain rule, the partial derivative $\partial Loss/\partial p_i$ of parameter p_i depends on both data values and the value of other parameters. The gradient of the loss function relies on data values and current parameter states.

3.2. Hashing and Model Compression

Considering the sparsity in training data, hashing is an efficient approach to compress machine learning models. Specifically, using a hashing function can map $d_i \in D$ into a smaller size vector d'_i. Let D' denote the dataset after mapping. Training with D' can significantly reduce model sizes with a slight sacrifice of model accuracy. More importantly, it is valid for both linear and DNN models. In addition, dynamic

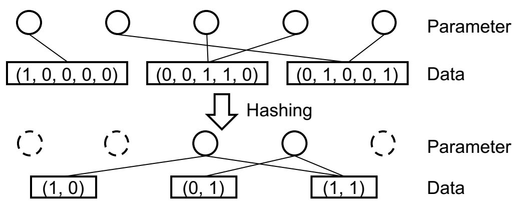


Figure 3. Hashing changes the correlation between data and parameters.

parameter pruning [19, 20] can also reduce the model size during training. The insight is that the neural connections which are not important would be dynamically deleted during the training process. Our data and parameter problem is built upon those scenarios. Our data partition algorithm introduced in Section 4 considers the case that hashing can reduce the model size. It can adaptively balance the time complexity of data and parameter partition based on their sizes and can be used for efficient data and parameter partition when hashing algorithms are applied. In addition, we can estimate the parameter access pattern for DNN models even if the model pruning techniques are applied. With the estimated parameter access patterns, we can approximate correlation graphs of data and parameters during the DNN training procedures. Our partition algorithms introduced in Section 5 would re-evaluate the correlation between data samples and parameters and perform parameter reallocation to reduce the communication traffic during training DNN models. Fig. 3 provides a high-level illustration of how hashing can change the correlation between data samples and parameters. The tuple within each data node in the figure shows which parameter will be updated when processing the data point. For example, $(1, 0, 0, 0, 0)$ means the first parameter would be changed by this data sample. Comparing the correlation graph before and after hashing, we can observe that hashing may significantly reduce the number of parameters but the number of edges in the correlation graph is less impacted. The insight is that hashing may reduce the parameter size but should not lose the information (i.e., how the model should be updated) integrated within each data sample. This example shows that the correlation graph could still be dense after hashing and optimizing the parameter reallocation can potentially reduce the inter-machine communication cost for the training process.

3.3. Network Communication in Parameter Server

With the rapid increase of data size, distributed training of machine learning models becomes necessary. In this paper, we follow the data parallelism [34, 35] and focus the network communication in the parameter server [3]. The framework of the parameter server is illustrated in Fig. 1. Let $W = \{w_1, w_2, \dots, w_k\}$ denote the set of k worker machines that are available for training. Each worker $w_i \in W$ would be allocated with

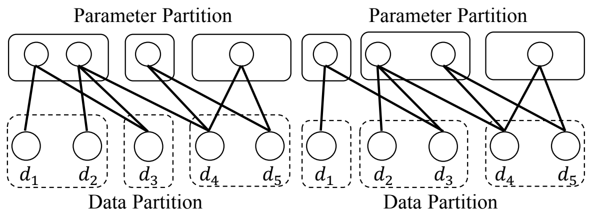


Figure 4. Data and parameter partition (left: random, right: optimal).

a subset of data from D. If the hashing algorithm proposed in [8] is used, a subset of mapped data from D' would be allocated to workers. Let $D_i \subset D$ or $D_i \subset D'$ denote the subset of data allocated to worker w_i. Worker w_i would iteratively apply training algorithms, such as stochastic gradient descent, on the subset of data D_i. Notably, each worker also acts as a node of parameter server. This configuration can utilize the high speed inner-machine communication buses and the update locality to reduce the communication overhead [6]. Worker w_i would be assigned a subset of parameters. We use P_i to denote the subset of parameters assigned to w_i. To minimize the communication cost, we need to carefully schedule the partition and allocation of dataset D and parameter set P.

When training with the parameter server framework, each iteration contains three major steps. Firstly, workers *pull* model parameters from parameter server nodes. Then, each worker calculates the parameter updates using its training data locally. Finally, workers *push* the updates to parameter server nodes. The communication overhead is caused by the pull and push operations. To reduce the cost, the parameter server supports *partially pull/push* operations. For linear models, workers would only pull/push parameters that are correlated with its data [6]. For DNN models, we can dynamically prune the model parameters during training [36]. There is no need to pull or push pruned parameters, which causes the dynamic correlations.

Definition 1. *Let $f : 2^D \times 2^P \mapsto \mathbb{R}^+$ denote the communication cost function. $f(D_i, P_j)$ denotes the communication volume between $D_i \subseteq D$ and $P_j \subseteq P$.*

For linear machine learning models, the communication cost can be theoretically formulated and has monotone and submodular properties that are useful for data and parameter partition. It is difficult to formulate the cost function for DNN models because of the complex correlation between data and parameters. We propose to estimate the communication cost during training and adaptively adjust the parameter partition accordingly.

We utilize the fixed data and parameter correlation to formulate the function f for linear machine learning models. Based on the bipartite graph G, we can find the correlation set of each vertex. Formally, for a data sample $d_i \in D$, its correlation set is $\{p \in P|(d_i, p) \in E\}$. Symmetrically, $\{d \in D|(d, p_j) \in E\}$ is the correlation set of the parameter $p_j \in P$. Moreover, for a subset $D_i \subseteq D$ or $P_j \subseteq P$, its correlation set is the union of the correlation set of its every element. Let $\delta : 2^D \cup 2^P \mapsto 2^P \cup 2^D$ denote the correlation set function. Notably, G is a bipartite graph. The correlation set of a subset of D must be a subset of P, and vice versa. Then, $\delta(D_i) = \{p \in P|d \in D_i, (d, p) \in E\}$ and $\delta(P_j) = \{d \in D|p \in P_j, (d, p) \in E\}$. For example, the vertex d_3 shown in Fig. 4 is connected with two parameter nodes p_1 and p_2. Accordingly, $\delta(\{d_3\}) = \{p_1, p_2\}$. For a subset of vertices $\{p_1, p_2\}$, its correlation set $\delta(\{p_1, p_2\}) = \{d_1, d_2, d_3, d_4\}$.

We can formulate the communication volume for training linear machine learning models based on δ. Formally, the communication cost $f(D_i, P_j) = |\delta(D_i) \cap P_j|$, where $| \cdot |$ represents the cardinality of a set. $\delta(D_i)$ represents the set of parameters required when training with data D_i. Its intersection with P_j means the set of parameters that should be transmitted between w_i and w_j during training. The cardinality of the intersection set gives the communication volume between D_i and P_j. Recall that each worker machine maintains a subset of parameters. For a worker w_i, it needs to pull/push parameters correlated with its data D_i during training. $\delta(D_i)$ is the set of parameters required by w_i. Among them, $f(D_i, P_i) = |\delta(D_i) \cap P_i|$ is the number of parameters that are located in itself. Those parameters are transmitted via the system bus and are not counted for the network communication volume. The rest $f(D_i, P \setminus P_i) = |\delta(D_i) \cap (P \setminus P_i)|$ is the volume of parameters stored in other worker machines, which contributes to the network communication volume. What's more, other workers need to pull/push parameters through w_i. The number of parameters located at w_i and needed by other workers is $f(D \setminus D_i, P_i) = | \bigcup_{i \neq j} \delta(D_i) \cap P_j|$, which also should be added to the inter-machine communication cost. Above all, the network communication volume of worker w_i is $f(D_i, P \setminus P_i) + f(D \setminus D_i, P_i) = |\delta(D_i) \cap (P \setminus P_i)| + |\delta(D \setminus D_i) \cap P_j|$. The meanings of the notations we introduced are summarized in Table 1.

3.4. Problem Formulation

We aim to reduce the communication overhead for distributed machine learning. Notably, the communication among worker machines is performed simultaneously. Instead of minimizing the overall communication volume among all workers, we should focus on balancing the communication workload among available workers, i.e., minimizing the largest communication cost among workers. According to the analysis in Section 3.3, the network communication volume of worker w_i is $f(D_i, P \setminus P_i) + f(D \setminus D_i, P_i)$. We can adjust

Table 1. Table of Notations

Notations	Description
W, w_i	the worker set and the i-th worker
k	the number of worker machines
D, D_i	the data set and the subset of data allocated to w_i
n	the number of data nodes
P, P_i	the parameter set and the subset of parameter allocated to w_i
m	the number of parameter nodes
E	the edge set $E = D \times P$
$f(D_i, P_i)$	the the communication volume between $D_i \subseteq D$ and $P_j \subseteq P$
$\delta(\cdot)$	the correlation set function $\delta : 2^D \cup 2^P \mapsto 2^P \cup 2^D$
$\lvert \cdot \rvert$	the cardinality of a set
$f(D')$	the cardinality of neighbor set of D'

the communication cost by optimizing the data and parameter partition. Therefore, it leads to our partition problem. Formally, the formulation of our problem can be shown as the following equations.

$$\min \max_i (f(D_i, P \setminus P_i) + f(D \setminus D_i, P_i)) \tag{1}$$

$$s.t. \ \bigcup_{i=1,\ldots,k} D_i = D \tag{2}$$

$$\bigcup_{i=1,\ldots,k} P_i = P \tag{3}$$

Among them, Eq. (1) is the objective equation. For linear machine learning models, the closed-form formulation of f is shown in Section 3.3. For DNN models, the function value can be estimated during training. Eq. (2) is the data partition constraint, which means that all data should be partitioned and allocated into k workers. If hashing methods are applied to the training data set, then D refers to the data after hashing. Eq. (3) shows the parameter partition constraint. The whole parameter set P should be covered in the parameter partition.

4. Parameter Allocation for Linear Models

To efficiently reduce the communication cost, existing approaches either use hashing techniques to reduce the sizes of linear models [8, 37], or utilize graph partition methods to mitigate the inter-machine communication into inner-machine communication [9, 6]. In this section, we jointly consider the impact of hashing and graph partition to efficiently shrink the communication cost for linear machine learning models while limiting the impact on the model accuracy.

4.1. Submodular Optimization Framework

As we introduced in the previous section, linear models have fixed data-parameter correlations. Specifically, before training, we can obtain the subset of parameters that would be updated by each data sample. Based on the correlation graph G, we can reduce the communication overhead by optimizing the data and parameter partition among worker machines. Sparsity and submodularity are useful properties for the partition. The sparsity of the correlation graph may be changed by model compression. Therefore, we first focus on the submodularity and present the optimization framework. Then, we adaptively adjust the heuristics used in partition for different sparsity. The submodularity of our communicaiton cost function is shown in the following theorem.

Theorem 4.1. *Given a data set $D_i \subseteq D$, the communication cost function f is monotone and submodular w.r.t. parameter set P_j and vice versa.*

Proof: We first show that f is monotone w.r.t. variable P_j. For a constant $D_i \subseteq D$, $f(D_i, P_a) \leq f(D_i, P_b)$ if $P_a \subseteq P_b \subseteq P$. It can be proved by definition. For any $P_a \subseteq P_a \subseteq P$, $|\delta(D_i) \cap P_a| \leq |\delta(D_i) \cap P_b|$. Hence, $f(D_i, P_a) \leq f(D_i, P_b)$. Additionally, f is monotone w.r.t. variable D_i. Specifically, for a constant $P_j \subseteq P$, $f(D_a, P_j) \leq f(D_a, P_j)$ if $D_a \subseteq D_b \subseteq D$. For $D_a \subseteq D_b \subseteq D$, $\delta(D_a) = \bigcup_{d \in D_a} \delta(\{d\}) \subseteq \bigcup_{d \in D_a \cup (D_b \setminus D_a)} \delta(\{d\}) = \delta(D_b)$. Therefore, $\delta(D_a) \cap P_j \subseteq \delta(D_a) \cap P_j$ and $|\delta(D_a) \cap P_j| \leq |\delta(D_b) \cap P_j|$.

Then we show the submodularity. Given a constant $D_i \subseteq D$, $f(D_i, P_j) = |\delta(D_i) \cap P_j|$ is a modular set function w.r.t. P_j. Every modular set functions is a special type of submodular function. Therefore, $f(D_i, P_j) = |\delta(D_i) \cap P_j|$ is submodular w.r.t. P_j. On the other hand, $f(D_i, P_j)$ is also submodular w.r.t. variable D_i when $P_j \subseteq P$ is treated as a constant. Formally, for $D_a \subseteq D_b \subseteq D$, $f(D_i, P_j)$ is submodular to D_i if and only if $f(D_a \cup \{d\}, P_j) - f(D_a, P_j) \geq f(D_b \cup \{d\}, P_j) - f(D_b, P_j)$, where $d \in D$. By definition, $f(D_a \cup \{d\}, P_j) - f(D_a, P_j) \geq f(D_b \cup \{d\}, P_j) - f(D_b, P_j)$ is equivalent to:

$$|\delta(D_a \cup \{d\}) \cap P_j| - |\delta(D_a) \cap P_j| \geq |\delta(D_b \cup \{d\}) \cap P_j| - |\delta(D_b) \cap P_j| \tag{4}$$

By definition, we have $\delta(D_a \cup \{d\}) = \delta(D_a) \cup \delta(\{d\})$. Therefore, $\delta(D_a \cup \{d\}) \cap P_j = (\delta(D_a) \cup \delta(\{d\})) \cap P_j = (\delta(D_a) \cap P_j) \cup (\delta(\{d\}) \cap P_j)$. Associated with the property of cardinality of intersection, we have $|\delta(D_a \cup \{d\}) \cap P_j| = |(\delta(D_a) \cap P_j) \cup (\delta(\{d\}) \cap P_j)| = |\delta(D_a) \cap P_j| + |\delta(\{d\}) \cap P_j| - |(\delta(D_a) \cap P_j) \cap (\delta(\{d\}) \cap P_j)| =$

$|\delta(D_a) \cap P_j| + |\delta(\{d\}) \cap P_j| - |\delta(D_a) \cap \delta(\{d\}) \cap P_j|$. Similarly, $|\delta(D_b \cup \{d\}) \cap P_j| = |\delta(D_b) \cap P_j| + |\delta(\{d\}) \cap P_j| - |\delta(D_b) \cap \delta(\{d\}) \cap P_j|$. Plugging in those equations, Eq. (4) is converted to $|\delta(\{d\}) \cap P_j| - |\delta(D_a) \cap \delta(\{d\}) \cap P_j| \geq |\delta(\{d\}) \cap P_j| - |\delta(D_b) \cap \delta(\{d\}) \cap P_j|$. After subtracting $|\delta(\{d\}) \cap P_j|$ and multiplying -1 on both sides, Eq. (4) is eventually converted to:

$$|\delta(D_a) \cap \delta(\{d\}) \cap P_j| \leq |\delta(D_b) \cap \delta(\{d\}) \cap P_j| \tag{5}$$

We have shown that $\delta(D_a) \subseteq \delta(D_b)$ if $D_a \subseteq D_b \subseteq D$. After intersection with $\delta(\{d\})$ and P_j, we have $\delta(D_a) \cap \delta(\{d\}) \cap P_j \subseteq \delta(D_b) \cap \delta(\{d\}) \cap P_j$ and Eq. (5) holds. By definition, $f(D_i, P_j)$ is submodular w.r.t. D_i. ■

To simultaneously partition both data and parameter sets is a graph partition problem, which is NP-hard[7]. Therefore, we partition them in sequence. We partition the correlation graph with a two-step heuristic. Because the data vertex and the parameter vertex in the correlation graph is symmetric, we can partition either D or P first. W.l.o.g., we assume D is partitioned first. With the submodularity shown in Theorem 4.1, we can apply the submodular optimization techniques introduced in [38] to partition D. The optimization framework iteratively samples unassigned elements and greedily chooses a subset of samples such that the cost increment is minimized. To achieve the efficient partition, we adapt the optimization framework to fit a different graph density caused by model compression or hashing. As shown in the proof of Theorem 4.1, $f(D_i, P_j)$ is a modular function w.r.t. P_j. With the property, we present a greedy algorithm for parameter partition.

4.2. Data and Parameter Partition

We present the detailed procedures of data and parameter partition in this subsection. Notably, we propose to adaptively adjust the size of the data sampled in the submodular optimization framework. The time complexity for data partition is exponential to the data sampling size. We propose to adaptively adjust the time complexity such that data and parameter partitions have the same big-O complexity. The insight is that hashing will reduce the number of model parameters and we don't want the data partition to become the scheduling bottleneck.

The procedures of data partition are shown in Alg. 1. We first initialize k data partitions into empty sets in line 1. The following loop implements the submodular optimization. Line 3 randomly samples from unassigned data instances and stores the sampled data in set S. During sampling, each data instance is chosen randomly with probability $n/(k|D|)$, where $|D|$ indicates the number of unassigned data instances. S is valid if its cardinality is smaller than or equals to $2n/k$. In line 5, we greedily choose the data partition D_i

Algorithm 1 Data Partition

```
Input:   Correlation graph $G$, number of partitions $k$
Output:  Data partition $D_i, i = 1, 2, \dots, k$
1: $D_i \leftarrow \emptyset$ $1 \le i \le k$
2: while $D$ is not empty do
3:     Sample a random $S \subseteq D$ with probability $\frac{n}{k|D|}$
4:     if $|S| \le 2n/k$ then
5:         Find the data partition $D_i$ that has the smallest size
6:         $S^* \leftarrow \arg\min_{S' \subseteq S, |S'| < \alpha} f(D_i \cup S', P) - \beta|D_i \cup S'|$
7:         if $f(D_i \cup S^*, P) - \beta|D_i \cup S^*| < 0$ then
8:             $D_i \leftarrow D_i \cup S^*, D \leftarrow D \setminus S^*$
9: return $D_i, 1 \le i \le k$
```

that has the smallest size to assign data combinations in a valid S. Line 6 finds the best data combination for D_i. Let S^* denote the best subset chosen in line 6. Specifically, from all possible subsets of S, the one that minimizes the value of $f(D_i \cup S', P) - \beta|D_i \cup S'|$ is chosen, where S' represents a subset of S, α limits the size of S', and *beta* is a hyperparameter of the submodular optimization[38]. Notably, testing all possible subsets S' costs exponential time. We adjust the time complexity by limiting the size of S'. If S^* can bring sufficient benefits as shown in line 7, then we assign S^* to D_i and remove vertex in S^* from D. Finally, line 9 returns the data partition result. We will show how to determine the value of α after introducing the parameter partition algorithm.

The procedures of parameter partition are shown in Alg. 2. Line 1 initializes parameter partitions into empty sets. Line 2 defines cost functions for parameter partitions. For partition P_j at for worker machine w_j, the cost is the network communication volume $f(D_i \cup S', P) - \beta|D_i \cup S'|$ for worker w_j as defined in Section 3.3. Then, parameters are greedily assigned in the loop of lines 3-6. The worker machine w_j with the lowest communication cost is chosen in line 4. Then, an unassigned parameter p^* that introduces the smallest cost increase is picked in line 5. Line 6 puts p^* into P_j and removes p^* from P. The while loop terminates when all parameters are assigned. Finally, $P_j, 1 \le j \le k$ are returned as the parameter partitions for worker machines.

Algorithm 2 Parameter Partition

```
Input: Parameter set $P$, data partition $D_i$, and $k$
Output: Parameter partitions for $k$ worker machines
1: Initialize parameter partition $P_j \leftarrow \emptyset$ $1 \leq j \leq k$
2: Initialize cost function for worker $w_j$, $cost(P_j) \leftarrow f(D_i, P \setminus P_i) + f(D \setminus D_i, P_i), \forall 1 \leq j \leq k$
3: while $P$ is not empty do
4:     $j \leftarrow \arg\min_j cost(P_j)$
5:     $p^* \leftarrow \arg\min_{p \in \delta(D_j)} cost(P_j \cup p) - cost(P_j)$
6:     $P_j \leftarrow P_j \cup p^*$, $P = P \setminus p^*$
7: return $P_j, 1 \leq j \leq k$
```

4.3. Properties

Because of the submodular property shown in Theorem 4.1, the performance of data partition shown in Alg. 1 has an approximation ratio according to the submodular optimization theory [38]. In particular, when the number of workers is much less than the number of data samples (which is a common case in real-world applications), Alg. 1 is $\Theta(n/\log n)$-approximate w.r.t. the worst communication cost $\max_i f(D_i, P_j)$ as explained in the following theorem.

Theorem 4.2. *Let D_i^* denote the optimal data partition on for worker w_i that minimize the worst communication cost $\max_i f(D_i^*, P)$. Then, the data partition D_i generated by Alg. 1 guarantees that $\max_i f(D_i, P) \leq \Theta(n/\log n) \max_i f(D_i^*, P)$.*

Proof: In Alg. 1, random sampling is used to reduce the time complexity. According to [38], the data partition D_i found by Alg. 1 with random sampling still satisfies the submodular property when $k = \Theta(n/\log n)$. It is for arbitrary partitions D_i and D_j found by Alg. 1, we have $f(D_i, P) + f(D_j, P) \geq f(D_i \cup D_j, P) + f(D_i \cap D_j, P)$, so as the optimal partition D_i^* and D_j^*. Moreover, $D_i^* \cap D_j^* = \emptyset$ since duplicated or overlapped data allocation can only enlarge the communication cost and would not be the optimal solution. Therefore, we have that $f(D_i^*, P) + f(D_j^*, P) \geq f(D_i^* \cup D_j^*, P) + f(D_i^* \cap D_j^*, P) = f(D_i^* \cup D_j^*, P) + f(\emptyset, P) = f(D_i^* \cup D_j^*, P)$.

Repeatedly apply the property on all k data partitions, we have $\sum_{i=1}^{k} f(D_i, P) \geq f(D_1^* \cup D_2^* \cup \cdots \cup D_k^*, P) = f(D, P)$. In addition, $f(D, P) \geq \max_i f(D_i)$ since $f(D, P)$ contains all correlations between D and P. Even if we need to transmit all of them cross machines, the communication cost is $f(D, P)$. Combining those properties, we have the inequality $\sum_{i=1}^{k} f(D_i^*, P) \geq \max_i f(D_i, P)$ or $\max_i f(D_i, P) \leq \sum_{i=1}^{k} f(D_i^*, P)$.

By definition, we also have $\max_i f(D_i^*, P) \geq f(D_i^*, P), \forall 1 \leq i \leq k$. Therefore $\sum_{i=1}^{k} f(D_i^*, P) \leq k \cdot \max_i f(D_i^*, P)$. Above all, we have $\max_i f(D_i, P) \leq \sum_{i=1}^{k} f(D_i^*, P) \leq k \cdot \max_i f(D_i^*, P)$, when $k = \Theta(n/\log n)$. It is $\max_i f(D_i, P) \leq \Theta(n/\log n) \max_i f(D_i^*, P)$. ■

Theorem 4.2 shows that using random sampling in Alg. 1 still provides performance bounds because of the submodular property. Another advantage of our approach is that it can balance the time complexity of data and parameter partition by adjusting the sample rate in the data partition. We can adjust the sample rate in Alg. 1 to fit the time complexity for parameter partition. Therefore, the overall time complexity is mainly determined by the time complexity of the parameter partition, which is analyzed in Theorem 4.3.

Theorem 4.3. *The time complexity of parameter partition is $O(nm^2)$, where $n = |D|$ and $m = |P|$. D and P can either be original data and parameter sets or the sets after hashing.*

Proof: The time complexity of Alg. 2 mainly comes from the while loop in lines 3-6. In particular, line 4 greedily finds the current parameter partition that has the smallest cost. It takes $O(k)$ time. Line 5 searches for the best assignment that minimizes the cost increment. Notably, we need to calculate the value of $cost(P_j \cup p)$ for each candidate $p \in \delta(D_i)$. It cannot be done in constant time. Instead, it costs at most $O(n)$ time to recalculate the number of data samples that are correlated with p. The number of candidates $p \in \delta(D_i)$ is at most $O(m)$. Therefore, line 5 takes $O(nm)$ time to find the parameter that introduces the lowest cost increment. Hence, each loop round takes $O(k + nm)$ time. Considering $k < m$ and $k < n$ in most cases, $O(k + nm) = O(nm)$. Moreover, the while loop at most takes $O(m)$ rounds to terminate. Therefore, the overall time complexity of the parameter partition shown in Alg. 2 is $O(m) \cdot O(nm) = O(nm^2)$. ■

5. Dynamic Partition for DNN Models

For DNN models, parameter pruning technologies [33, 17, 39] are widely used to reduce the communication overhead during training. They notice that there are redundant connections in DNNs and dropping unimportant connections during training will not affect the model accuracy [17]. After pruning, the size of effective model parameters is significantly reduced. Applying parameter pruning technologies in the parameter server framework will cause parameter access locality. The important parameters are frequently accessed and updated during training, while unimportant parameters are dropped. The pruning may unbalance the communication workload among workers. Additionally, when updating parameter weights in each iteration, the gradient vector is sparse when the model pruning and sparsification methods are applied. We

Algorithm 3 Building Correlation Graph for DNN Models

```
Input: $D_t \subseteq D$, set of data samples for training batch $t$
Output: Correlation graph $G_t$ for training batch $t$
1: $G_t \leftarrow (D_t \cup P, \emptyset)$
2: for $(d, p)$ in $D_t \times P$ do
3:     $N_d \leftarrow$ find $r$-nearest neighbors of $d$ in LRU
4:     Data similarity vector $s$, where the vector entry $s_l \leftarrow$ reciprocal of distance between $d$ and $d_l \in N_d$, $1 \leq l \leq r$
5:     Parameter access vector $q$, where an entry $q_l \leftarrow 1$ if $p$ is accessed by $d_l \in N_d$, and 0 otherwise, $1 \leq l \leq r$
6:     $f(d, p) \leftarrow (\sum_{l=1}^{r} s_l q_l)/(r \sum_{l=1}^{r} s_l)$
7:     Add the edge $(d, p)$ into $G_t$ if $f(d, p) > c$, where $c$ is a pre-defined constant
8: return $G_t$ as the correlation graph for batch $t$.
```

should frequently update the parameter partition and allocation according to the parameter access locality. Even if the gradient vectors are sparse, it is still difficult to partition the parameter set dynamically, since data and parameter correlations are unknown before training. Another challenge is that the mitigating parameters among workers too frequently would cause additional communication overhead. We propose to estimate the communication cost f among workers during training and adjust the parameter allocation when necessary.

Unlike linear models, we cannot build the correlation graph G by scanning nonzero entries of training data samples. According to the chain rule, the gradient of parameters in an intermediate layer depends on the partial derivative information of DNN layers following it. Parameter updates relay on the backward propagation procedure. The complex connections between data samples and parameters aggregates during the backward propagation. Therefore, we can no longer simply use the distribution of nonzero entries to build the correlation graph. We propose to use the parameter access locality to estimate the data and parameter correlations.

We assume that similar data instances have the similar parameter access pattern in adjacent training iterations. For two data samples, their similarity can be measured by the distance between corresponding data vectors. In this paper, we use the Euclidean distance to quantify the data similarity. Although we cannot exactly know which set of parameters would be accessed before actually training with data, we can estimate it using the similar data samples that have been seen in previous iterations. Notably, we should look for

Algorithm 4 Parameter Partition for DNN Models

```
Input: Correlation graph $G_t$ for the following training batch
Output: Parameter partition for workers: $P_i, 1 \leq i \leq k$
 1: repeat
 2:     $P_{\max} \leftarrow \arg\max_j f(D_t, P_j)$
 3:     $P_{\min} \leftarrow \arg\min_j f(D_t, P_j)$
 4:     Find $p^* \in P_{\max}$ such that $p^* = \arg\min_{p \in P_{\max}} \text{abs}(f(D_t, P_{\max} \setminus \{p\}) - f(D_t, P_{\min} \cup \{p\}))$
 5:     $\Delta \leftarrow f(D_t, P_{\max}) - \max\{f(D_t, P_{\max} \setminus \{p\}), f(D_t, P_{\min} \cup \{p\}), \max_{P_j \neq P_{\max}, P_j \neq P_{\min}} f(D_t, P_j)\}$
 6:     if $\Delta <$ size of $p^*$ then
 7:         Break
 8:     Update $P_{\max} \leftarrow P_{\max} \setminus \{p^*\}$ and $P_{\min} \leftarrow P_{\min} \cup \{p^*\}$
 9: until $\Delta <$ size of $p^*$
10: return $P_i, 1 \leq i \leq k$ as the updated parameter partition
```

similar data from the most recent iterations since the parameter access pattern is also related with parameter staleness. Training the same data on completely different parameters would generate different gradients. Parameters in adjacent training iterations usually have similar values. During training, we can use a Least Recently Used (LRU) cache to record the set of parameters accessed by each used training data. The LRU cache not only costs a constant memory overhead, but also restricts the staleness of history data samples. With the LRU cache, we can frequently estimate and update the correlation graph G_t for training batch t.

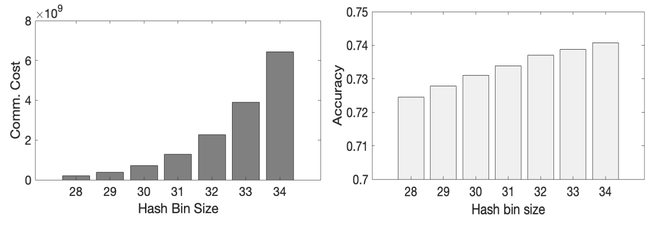
When training DNN models, training samples are usually split into multiple batches. We use $D_t \subseteq D$ to denote the set of data instances used in batch t. Except for several starting batches, we can estimate the correlation graph G_t for each training batch using the training trace history. The LRU cache would store recently used data samples and the list of parameters updated by each sample. The list can be obtained from the gradient information. A parameter is updated if the corresponding gradient is greater than a small threshold ϵ. Using the cache, we can build the graph G_t. Note that G_t is a weighted graph for DNN models. The weight of edge (d, p) represents the probability of which the parameter $p \in P$ is accessed and updated when training with the data $d \in D_t$.

The procedure to build the correlation graph G_t is shown in Alg. 3. We first initialize the bipartite graph G_t in line 1. Its vertex set contains the training data D_t for batch t and the parameter set P. The edge set is initialized to an empty set. In the following loop, we estimate the weight of each potential edge

$(d, p) \in D_t \times P$. We use r similar history traces in the LRU cache to estimate the weight of (d, p). Line 3 chooses r nearest neighbors of d from the cache based on the similarity or the reciprocal of distance between data vectors. We use N_d to denote the set of data samples similar to d. Lines 4 and 5 prepare the data similarity vector s and the parameter access vector q to calculate the edge weight $f(d, p)$. An element s_l of vector s shows the similarity between d and $d_l \in N_d$. The similarity is measured by the reciprocal of the distance between them. If the data $d_l \in N_d$ has accessed the parameter p, then the entry p_l of p is set to 1. Otherwise, it is set to 0. Line 6 calculates the weight of (d, p). $f(d, p) = (\sum_{l=1}^{r} s_l q_l)/(r \sum_{l=1}^{r} s_l)$. The term $\sum_{l=1}^{r} s_l q_l$ is the weighted sum of the parameter access history. Trace of the data with larger similarities contributes more to the sum. The other term $r \sum_{l=1}^{r} s_l$ normalizes the weighted sum. If p is accessed by all data samples in N_d, then $f(d, p) = 1$. If no one in N_d has visited p, then $f(d, p) = 0$. Line 7 updates the graph G_t. To reduce the density of graph G_t, the edge (d, p) is added into G_t only if the weight $f(d, p)$ is greater than a pre-defined threshold. When the weight is small, the parameter p is not likely to be updated by d during training. Finally, line 8 returns G_t after evaluating the weight of every possible edge.

With the estimated correlation graph, we can update the parameter partition and allocation. Notably, different from the fixed allocation introduced in Section 4, updating parameter partition has an additional cost of mitigating parameters among workers. Therefore, there is no need to update the parameter partition for every training batch. We use an unbalanced communication threshold to trigger the parameter reallocation. The parameter partition is recalculated if the difference between the largest and smallest communication workloads exceeds the threshold. Additionally, we only need to build graph G_t before reallocation instead of every batch.

The procedure of the parameter reallocation that considers the parameter mitigation cost is shown in Alg. 4. For reallocation, we repeatly move parameters from the partition with the largest communication cost to the partition with the smallest cost. Specifically, lines 2 and 3 find the partition $P_{\max}$ and $P_{\min}$ with the largest and smallest cost, accordingly. Note that G_t becomes a weighted function for DNN models. Therefore, the communication cost function f is overloaded. Instead of using cardinality (i.e., edge weights are uniform), it uses the sum of edge weights to evaluate the communication cost. Line 4 greedily chooses a parameter from $P_{\max}$ such that the communication cost difference between $P_{\max}$ and $P_{\min}$ is minimized after moving the parameter to $P_{\min}$. Line 5 calculates the potential decrease of the bottleneck traffic volume after moving p^* from $P_{\max}$ to $P_{\min}$. We use Δ to denote the network traffic decrease volume. Line 6 compares Δ with the cost of moving p^*. The first term $f(D_t, P_{\max})$ shows the bottleneck traffic before moving. The other term $\max\{f(D_t, P_{\max} \setminus \{p\}), f(D_t, P_{\min} \cup \{p\}), \max_{P_j \neq P_{\max}, P_j \neq P_{\min}} f(D_t, P_j)\}$ calculates the largest network traffic



(a) Communication cost (b) Accuracy

Figure 5. Partition with different hash bin size.

volume among workers after moving p^*. If the benefits Δ is less than the mitigation cost, then we break the loop. Otherwise, we move p^* and update the partition in line 8. The loop terminates when there is no additional benefit of reallocation, as indicated by line 9. Finally, line 10 returns the updated parameter sets.

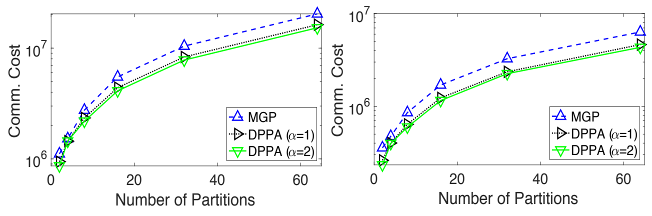
6. Experiment

6.1. Experiment Setup

In our experiment, we use Amazon EC2 to set up the machine cluster for training. The EC2 cluster we used has 16 t2.xlarge instances with 16GB RAM. On the EC2 cluster, we use PS-lite[1] to set up the parameter server. We install OpenMPI on each instance as its message passing interface. We evaluated our data and parameter partition algorithms with linear models. For dynamic parameter partition, we simulate the parameter pruning process using training snapshots. Specifically, we record parameter snapshots during training and drop parameters with small weights. For each snapshot, we use the parameter and data partition in the previous snapshots as the initial assignment and execute our reallocation algorithms based on the assignment. The network communication volume and parameter migration costs are simulated with the size of the parameters to be retrieved or moved.

We use both real-world and synthetic datasets to evaluate our algorithms. For real-world datasets, we use two LIBSVM [40] datasets: rcv1.binary and news20. Data and parameter correlation graphs are built upon those datasets. The correlation graph can be built by traversing input data attributes. By counting the non-zero attributes, we can find the correlation between data samples and parameters. Fig. 2 illustrates the statistics of node degrees of rcv1.binary and news20 datasets. From the figure, we can find that most nodes

[1]`https://github.com/dmlc/ps-lite`



(a) Bin size = 30 (b) Bin size = 32

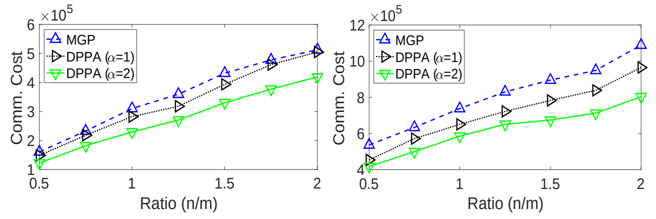
Figure 6. Impact of number of partitions.

have relatively small degrees. The frequency of large-degree nodes is small. It reveals that the connections in correlation graphs of those two datasets are sparse. It is more likely to reduce the communication cost across machines using data and parameter partitions on the sparse dataset. For synthetic datasets, we randomly generate correlation graphs with different $n : m$ ratios, where n is the number of data nodes and m is the number of parameter nodes. The sparsity of synthetic datasets can be evaluated by the ratio of the number of generated edges to the number of edges in the corresponding fully connected graph. We control the number of edges in the correlation graph to adjust the sparsity.

In our experiment, we compare our data and parameter partition algorithm (denoted as DPPA) with the multilevel graph partition[41] (denoted as MGP). The random partition is used as a baseline. The communication cost is evaluated by the accumulation of cross-machine push/pull counts. It reveals the inter-machine communication volume.

6.2. Experiment Results

We first illustrate the impact of hashing on model size and accuracy. We use the hashing algorithm introduced in [8] to compress linear models. The model size is measured by the number of model parameters. Fig. 5 shows the communication cost and model accuracy when using different bin sizes in hashing. From the figure, we notice that the communication cost increase with the bin size, so as the model accuracy. It is because that using a smaller bin size would make the number of data attributes smaller, but also introduce more conflicts during hashing. Some data information is lost because of those conflicts. With smaller bin sizes, the model size decreases nearly exponentially, while the model accuracy decreases nearly linearly. The experiment result shows that hashing can significantly reduce the communication cost with a relatively small sacrifice on model accuracy when using smaller bin sizes. We can use bin sizes to balance the trade-off between inter-machine communication cost and model accuracy.

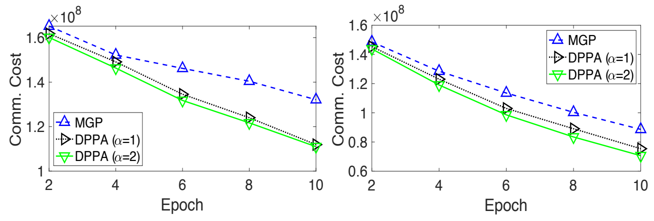


(a) Bin size = 30 (b) Bin size = 32

Figure 7. Impact of $n : m$ ratio.

Then, we investigate the impact of the number of partitions on partition algorithm performances. Also, we evaluate the influence of different bin sizes on the performances of data and parameter partition algorithms. Fig. 6 shows the communication cost of different partition approaches under different hashing bin sizes on the news20 dataset with random data and parameter allocation. We find that the overall inter-machine communication volume increases with the number of partitions. When data samples and parameters are distributed on a large number of machines, there will be frequent communication among those machines. DPPA can help to reduce the inter-machine communication volume. From the figure, we notice that the communication cost reduces if the bin size is smaller. Increasing the value of α of DPPA, i.e. using a higher sample rate, can improve the algorithm performance. It shows that hashing approach can both reduce the model size and the communication cost. In addition, our DPPA outperforms the MGP on different bin sizes. However, their performance gap decreases when the bin size decreases. It is because that the correlation graph becomes denser with a smaller hashing bin. In a relatively dense graph, the potential benefits of graph partition become smaller compared with a sparse graph.

Moreover, we investigate the influence of the relative size of the parameter set using synthetic datasets. The ratio $n : m$ represents the relative size. Recall that n represents the number of data samples and m is the number of parameters. If the ratio $n : m > 1$, then there are more data samples. In this case, the parameter set is relatively small. Fig. 7 shows the communication cost of different ratios with different hashing bin sizes. From each subfigure, we find that the inter-machine communication costs increase with the $n : m$ ratio. This shows that the communication among worker machines is more frequent when there are more data samples. In addition, comparing Fig. 7(a) with Fig. 7(b), we notice that the performance gap between MGP and DPPA becomes larger when the bin size is larger. This is also caused by the difference in the sparsity of their correlation graphs. Hashing with a smaller bin size can reduce the graph sparsity and cause more information loss. The improvement of data and parameter partition becomes more obvious when the



(a) Pruning rate = 0.05 (b) Pruning rate = 0.1

Figure 8. Dynamic partition for DNN models.

correlation graph or the training dataset is sparse.

Furthermore, we evaluate the dynamic partition for DNN models using our simulator. We record the inter-machine communication volume in each epoch. Fig. 8 shows simulation results. Notably, MGP is a graph partition algorithm that does not consider the dynamic changes on graphs. We rerun the MGP after each pruning event to update the partition. MGP does not consider the migration cost when recalculating the partition on a changed graph. From the simulation results, we can find that DPPA significantly outperforms MGP. The performance gap between DPPA and MGP becomes more obvious when the number of the epoch is larger. This shows that the migration cost of parameter reallocation is not negligible. The negative impact of omitting the migration cost may accumulate along training epochs. Comparing Fig. 8(a) and Fig. 8(b), we can also find that pruning can reduce the inter-machine communication cost during training, and using DPPA can efficiently find proper data and parameter partitions to avoid migration costs.

7. Conclusion

We investigate the communication overhead of the model training process in parameter server frameworks. We propose to reduce the network bottleneck traffic by jointly considering model compression and model partition. For linear machine learning models, we build the fixed correlation graph of data samples and model parameters. We utilize the submodular property of the communication cost function to optimize the graph partition. Different partition heuristics are investigated. Our scheme can adaptively choose the partition approach to avoid large partition overhead. Additionally, we investigate the partition problem for DNN models whose correlation graph changes during training. A dynamic partition method is proposed for DNN models. In the dynamic partition, we use the training history to estimate the data-parameter correlation and the communication cost. Our partition algorithm not only dynamically partitions the correlation graph but also considers the parameter mitigation cost. Real-world dataset is used in the evaluation. Experiment and simulation results show that our approach can efficiently reduce the communication overhead for both linear and DNN models.

References

[1] Q. Pi, G. Zhou, Y. Zhang, Z. Wang, L. Ren, Y. Fan, X. Zhu, K. Gai, Search-based user interest modeling with lifelong sequential behavior data for click-through rate prediction, in: International Conference on Information & Knowledge Management, 2020, pp. 2685–2692.

[2] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding (2019). `arXiv:1810.04805`.

[3] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, A. Smola, Parameter server for distributed machine learning, in: NeurIPS Big Learning Workshop, 2013, p. 2.

[4] S. Wang, A. Pi, X. Zhou, J. Wang, C.-Z. Xu, Overlapping communication with computation in parameter server for scalable dl training, IEEE Transactions on Parallel and Distributed Systems 32 (9) (2021) 2144–2159.

[5] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, P. Li, Distributed hierarchical gpu parameter server for massive scale deep learning ads systems, arXiv preprint arXiv:2003.05622.

[6] M. Li, D. G. Andersen, A. J. Smola, K. Yu, Communication efficient distributed machine learning with the parameter server, in: NeurIPS, 2014, pp. 19–27.

[7] M. Li, D. G. Andersen, A. J. Smola, Graph partitioning via parallel submodular approximation to accelerate distributed machine learning, arXiv preprint arXiv:1505.04636.

[8] P. Li, X. Li, C. H. Zhang, Re-randomized densification for one permutation hashing and bin-wise consistent weighted sampling, Advances in Neural Information Processing Systems 32.

[9] Y. Duan, N. Wang, J. Wu, Minimizing training time of distributed machine learning by reducing data communication, IEEE Transactions on Network Science and Engineering.

[10] E. Variani, X. Lei, E. McDermott, I. L. Moreno, J. Gonzalez-Dominguez, Deep neural networks for small footprint text-dependent speaker verification, in: IEEE ICASSP, 2014, pp. 4052–4056.

[11] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: Efficient convolutional neural networks for mobile vision applications, arXiv preprint arXiv:1704.04861.

[12] R. Huang, J. Pedoeem, C. Chen, Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers, in: IEEE Big Data, IEEE, 2018, pp. 2503–2510.

[13] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, A. Krishnamurthy, Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints, in: ACM MobiSys, 2016, pp. 123–136.

[14] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, Z. Wang, Adaptive selection of deep learning models on embedded systems, arXiv preprint arXiv:1805.04252.

[15] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, M. Xiao, Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics, in: IEEE INFOCOM, 2020, pp. 1–10.

[16] K. Wu, Y. Jin, W. Miao, Z. Zeng, Z. Qian, J. Wang, M. Zhou, T. Cao, Soudain: Online adaptive profile configuration for real-time video analytics, in: IEEE/ACM 29th IWQOS, 2021, pp. 1–10. doi:10.1109/IWQOS52092.2021.9521328.

[17] S. Han, J. Pool, J. Tran, W. J. Dally, Learning both weights and connections for efficient neural network, in: NeurIPS, 2015.

[18] G. Li, C. Qian, C. Jiang, X. Lu, K. Tang, Optimization based layer-wise magnitude-based pruning for dnn compression., in: IJCAI, 2018, pp. 2383–2389.

[19] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, J. Ye, Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates, in: AAAI Conference on Artificial Intelligence, Vol. 34, 2020, pp. 4876–4883.

[20] Y. Huang, X. Qiao, J. Tang, P. Ren, L. Liu, C. Pu, J. Chen, Deepadapter: A collaborative deep learning framework for the mobile web using context-aware network pruning, in: IEEE INFOCOM, IEEE, 2020, pp. 834–843.

[21] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, L. Tassiulas, Model pruning enables efficient federated learning on edge devices, IEEE Transactions on Neural Networks and Learning Systems.

[22] Z. Li, G. Yuan, W. Niu, P. Zhao, Y. Li, Y. Cai, X. Shen, Z. Zhan, Z. Kong, Q. Jin, et al., Npas: A compiler-aware framework of unified network pruning and architecture search for beyond real-time mobile acceleration, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 14255–14266.

[23] P. Wimmer, J. Mehnert, A. Condurache, Interspace pruning: Using adaptive filter representations to improve training of sparse cnns, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 12527–12537.

[24] X. Ma, M. Qin, F. Sun, Z. Hou, K. Yuan, Y. Xu, Y. Wang, Y.-K. Chen, R. Jin, Y. Xie, Effective model sparsification by scheduled grow-and-prune methods, in: International Conference on Learning Representations, 2022.

[25] X. Zhang, J. Liu, Z. Zhu, E. S. Bentley, Compressed distributed gradient descent: Communication-efficient consensus over networks, in: IEEE INFOCOM, IEEE, 2019, pp. 2431–2439.

[26] A. M Abdelmoniem, A. Elzanaty, M.-S. Alouini, M. Canini, An efficient statistical-based gradient compression technique for distributed training systems, Machine Learning and Systems 3.

[27] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, X. Zhao, Communication-efficient distributed deep learning with merged gradient sparsification on gpus, in: IEEE INFOCOM, IEEE, 2020, pp. 406–415.

[28] L. Abrahamyan, Y. Chen, G. Bekoulis, N. Deligiannis, Learned gradient compression for distributed deep learning, IEEE Transactions on Neural Networks and Learning Systems (2021) 1–1doi:10.1109/TNNLS.2021.3084806.

[29] E. Yu, D. Dong, Y. Xu, S. Ouyang, X. Liao, Cd-sgd: Distributed stochastic gradient descent with compression and delay compensation, in: 50th International Conference on Parallel Processing, Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3472456.3472508. URL https://doi.org/10.1145/3472456.3472508

[30] C. M. Bishop, Pattern recognition and machine learning, springer, 2006.

[31] X. Li, L. Yu, D. Chang, Z. Ma, J. Cao, Dual cross-entropy loss for small-sample fine-grained vehicle classification, IEEE Transactions on Vehicular Technology 68 (5) (2019) 4204–4212.

[32] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, K. Gai, Deep interest network for click-through rate prediction, in: ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018, pp. 1059–1068.

[33] Y. He, Y. Ding, P. Liu, L. Zhu, H. Zhang, Y. Yang, Learning filter pruning criteria for deep convolutional neural networks acceleration, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 2009–2018.

[34] H. Hu, D. Wang, C. Wu, Distributed machine learning through heterogeneous edge systems, in: AAAI Conference on Artificial Intelligence, Vol. 34, 2020, pp. 7179–7186.

[35] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, C. Guo, Elastic parameter server load distribution in deep learning clusters, in: ACM Symposium on Cloud Computing, 2020, pp. 507–521.

[36] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, A. Peste, Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, arXiv preprint arXiv:2102.00554.

[37] P. Li, A. Shrivastava, J. Moore, A. C. Konig, Hashing algorithms for large-scale learning, arXiv preprint arXiv:1106.0967.

[38] Z. Svitkina, L. Fleischer, Submodular approximation: Sampling-based algorithms and lower bounds, SIAM Journal on Computing 40 (6) (2011) 1715–1737.

[39] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, M. Erez, Prunetrain: fast neural network training by dynamic sparse model reconfiguration, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–13.

[40] C.-C. Chang, C.-J. Lin, Libsvm: a library for support vector machines, ACM transactions on intelligent systems and technology (TIST) 2 (3) (2011) 1–27.

[41] N. Jafari, O. Selvitopi, C. Aykanat, Fast shared-memory streaming multilevel graph partitioning, Journal of Parallel and Distributed Computing 147 (2021) 140–151.