ABSTRACT

Recent trends in high-performance computing and deep learning lead to a proliferation of studies on large-scale deep neural network (DNN) training. However, the frequent communication requirements among computation nodes drastically slow down the overall training speed, which makes the bottleneck in distributed training, particularly in clusters with limited network bandwidth. To mitigate the drawbacks of distributed communication, researchers have proposed various optimization strategies. In this paper, we give a comprehensive survey of communication strategies from both algorithm and computer network perspectives. Algorithm optimizations focus on reducing the amount of communication in distributed training, while network optimizations focus on speeding up the communication between distributed devices. At the algorithm level, we describe how to reduce the number of communication rounds and transmitted bits per round, besides we shed light on how to overlap computation and communication. At the network level, we discuss the effect caused by network infrastructures, including communication schemes, network protocols, and topology. Finally, we extrapolate potential challenges and research directions for communication acceleration in distributed DNN training.

1 Introduction

Today, we are in the unprecedented era of deep learning research history, in which the waves of deep neural networks have swept through several application domains, ranging from autonomous driving[1], computer vision[2], natural language processing[3] to recommendation systems[4]. The popularity of deep learning promotes the development of DNN architectures, including fully-connected neural network (FCNN), convolutional neural network (CNN), recurrent neural network (RNN) and its variants (LSTM[5], GRU[3]). These neural networks have achieved state-of-the-art performance across various domain-specific tasks. Take computer vision as an example, well-designed DNNs, such as GoogLeNet[6] and ResNet-50[7] trained on ImageNet[8] dataset, have beaten humans on image classification tasks.

For the sake of better performance, DNNs are tended to be deeper and more sophisticated and are trained with the larger dataset. The rapid increase of data volume and model size results in a vast amount of computation, indicating that DNN
training will is time-consuming, even several days or weeks. High-performance hardware, such as graphics processing units (GPU) and tensor processing units (TPU), are applied to accelerate the training time.

Beyond using high-performance hardware, paralleling and deploying DNN training tasks on multiple nodes (consist of one or more machines) is another practical approach. Under this circumstance, each node only executes a part of the entire computation task. However, due to the iterative-ness property of DNN training, communication overhead is a critical bottleneck in distributed training because of the frequent communication requirements to exchange large amounts of data among different computation nodes. With the growth of the cluster scale, communication overhead increases explosively. Such a phenomenon greatly diminishes the advantage of parallel training as a majority of training time is spent on transferring data. When high-performance hardware accelerators are used, the proportion of time spent on communication increases further because they decrease the computation overhead, whereas keeping the communication overhead unchanged.

In this paper, we mainly focus on how to deal with communication overhead in distributed DNN training. Because distributed deep learning is a cross research field, both deep learning and distributed network communities have proposed communication optimization strategies from their own perspective. In this survey, we bring together, classify, and compare the huge body of work on communication optimization for distributed DNN training from the different communities that contribute to this area. The overview of optimization strategies is shown in Fig.1.

Previous studies have covered many research domains of distributed deep learning. Ben et al.[9] provided a detailed concurrence analysis of DNNs from different levels. Various training algorithms and modern state-of-the-art training frameworks are studied by Chahal et al.[10]. A recent review by Mayer et al.[11] discusses some challenges of managing large deep learning systems on distributed infrastructure. Mittal et al.[12] reported optimization techniques for deep learning applications of GPUs from architecture and system-level aspects. Some surveys have discussed the communication optimization issues in distributed deep learning, but we provide a broader investigation. In particular, we discuss optimization from algorithms at a high level to the low network level, which has not been done in previous surveys.

Figure 1: Overview of communication optimization strategies.
The rest of this paper is organized as follows. In section 2, we introduce the background of deep learning and provide an overview of distributed DNN training. In the next two sections, we discuss the optimization of algorithms and networks, respectively. Section 3 discusses communication rounds reduction, gradient compression and computation-communication overlap, whereas section 4 introduces communication schemes, network protocols and topologies optimizations. Finally, we conclude the entire paper and highlight potential challenges and research directions in section 5.

2 Background

In this section, we provide a brief introduction to large-scale distributed deep neural network training. It is known that the most popular method to train DNNs on single-machine remains mini-batch based stochastic gradient descent (SGD)[13] with error backpropagation[14]. When DNN training comes to parallelization, several problems need to be considered: (i) which part of the training task can be parallelized, (ii) architectures of computation nodes, and (iii) when to synchronize gradients.

2.1 Data and Model Parallelism

Both training datasets and the DNN model can be split onto multiple devices, corresponding to data parallelism and model parallelism, respectively. Data parallelism is the most popular method in distributed deep learning. In data parallelism, the entire training dataset is divided into \( N \) parts and dispatched on \( N \) nodes, see in Fig.2(a). Each node maintains a DNN replica and its parameters \( W_n \). The training process is as follows:

- Each node reads a minibatch of training data and executes forward and backward propagation to calculate its local gradients \( \nabla W_n \).
- Each node sends local gradients to a master node. After received gradients from all nodes, the master aggregates these gradients and updates the model by \( \tilde{W} = \tilde{W} - \frac{\eta}{N} \sum_{n=1}^{N} \nabla W_n \).
- The master broadcasts the latest model parameters to all other nodes.
- Repeat these three steps until the model converged.

Besides data parallelism, model parallelism is another approach to train DNNs, which splits neural networks onto multiple computation nodes. It is appropriate to use model parallelism when the DNNs are too huge to be fit in a single machine. In this paper, we mainly focus on data parallelism.

2.2 Centralized and Decentralized Architectures

The (logical) architecture of computation nodes can affect communication modes and network performance. Parameter server[15, 16] is the most popular centralized architecture in distributed deep learning. A parameter server usually includes a server node and several worker nodes. The server maintains global model parameters, whereas each worker stores a local model replica. If the server node has more than one machine, each machine maintains a partition of the entire model parameters. As for workers, each worker stores an entire model replica in data parallelism or a part of the model in model parallelism. Workers communicate with the server via push/pull operation, whereas there is
no communication between any workers. The drawback of the parameter server is the bandwidth bottleneck on the server-side with the increment of workers.

Due to the drawback of parameter server, decentralized architecture has attracted much attention, because it incurs fewer communication traffic[17]. Similar to the parameter server, each node in decentralized architecture holds an entire model replica, but they use collective communication operation Allreduce instead of push/pull to exchange gradients and parameters. Nevertheless, the Allreduce operation has a variety of implementations with definitely different performance, which means it may affect communication overhead. We will discuss related issues in section 4.

2.3 Synchronous and Asynchronous Updates

Owing to differences in network bandwidth and computing power, some nodes may calculate gradients faster while others may be slower. The main challenge comes with such circumstance is when to synchronize gradients among multiple computation nodes. There are three different methods to solve this problem reasonably: synchronous, asynchronous, and bounded delay updates.

2.3.1 Synchronous

In the synchronous update, the server does not update the model until it receives gradients from all workers at each iteration. In other words, faster workers will wait for slower workers. One well-known implementation of the synchronous update is bulk synchronous parallel (BSP)[18]. The characteristic of the synchronous mode is that the server will always receive the latest gradients of all nodes, which do not affect the model convergence. However, fast nodes perform nothing when waiting for slow nodes, leading to a waste of resources. Besides, it will also cause the straggler problem and slows down the overall training time.

2.3.2 Asynchronous

Asynchronous algorithms such as Hogwild[19] overcome the above problems. In asynchronous updates, fast workers do not wait for slow workers. One worker may be sending its local gradients to the server while others are calculating their gradients, as shown in Fig. 3.

Figure 3: Asynchronous updates, where green nodes represent servers while blue correspond to workers. Adapted from [20]

The main challenge raised by asynchronous updates is data staleness, because fast workers may always use stale parameters which jeopardize model convergence. In addition, the fastest worker is equivalent to update its local parameters by its sub-dataset, which makes the local model deviate from the global model. To overcome the drawbacks of asynchronous updates, researchers have tried to limit the staleness of parameters. In bounded stale updates, fast workers will use stale parameters but the staleness (same as delay in Fig.3) is limited[21, 22]. The limitation of staleness mitigates the straggler problem to some extent and increases training throughput. However, how to choose the limitation
of staleness is a question worth discussing, because too large value means completely asynchronous, while the small value is similar to synchronous.

3 Algorithm Optimization

In this section, we demonstrate how to reduce the communication overhead in distributed DNN training from the perspective of algorithms. Algorithm optimizations include reducing communication rounds and volume and increasing the computation-communication overlap ratio. These optimizations are independent of the underlying network, and most algorithms can run on top of various network infrastructures and protocols.

3.1 Communication Rounds

When using SGD to train deep neural networks, the entire training process usually consists of multiple epochs and iterations. When it comes to parallelization, nodes often exchange data at the end of every iteration. One intuitive way to cut down the communication overhead is reducing the number of data exchanges, or the communication rounds. Based on our analysis in section, the number of rounds for a node is related to batch size and communication period. Larger batch size and a more extended communication period can both reduce the number of data exchanging rounds.

3.1.1 Large Batch Training

Batch size is a significant hyper-parameter that controls the amount of data read by a node in each iteration. A large batch size usually better approximates the distribution of the input data and introduces fewer variance into gradient estimate than a small batch. Besides, a large batch of data also will take a longer time to be processed and incur fewer model parameters update. It is primarily due to the relationship between batch size, the number of iterations, and training data size. The following equation shows that using a large batch size leads to the reduction of the number of iterations, hence the parameter update is infrequently.

\[
\text{Batch Size} \times \# \text{ of iterations} = \# \text{ of training data size}
\]  

(1)

Under distributed training circumstance with a data parallelism scheme, the batch size is the sum of all local batch size of each node. Recall that in conventional distributed deep learning, nodes exchange gradients and model parameters at the end of each iteration. Since the shape and size of gradients and model parameters only depend on the DNN itself, the single iteration communicated message size remains constant, as changing batch size does not change their shape and size. Therefore, increasing batch size reduces the number of iterations, which in turn reduces communications rounds. Take ResNet-50 as an example, if we fix the number of epochs as 100 and set the batch size to 1024 on two
Table 1: Compare ResNet-50 Training Detail with Different Works

| Works          | Batch Size | Hardware               | Top-1 Accuracy | Training Time |
|----------------|------------|------------------------|----------------|---------------|
| He et al.[7]   | 256        | Tesla P100 × 8         | 75.3% (baseline) | 29h           |
| Goyal et al.[23]| 8k         | Tesla P100 × 256       | 76.3%          | 1h            |
| Cho et al.[25] | 8k         | Tesla P100 × 256       | 75.0%          | 50min         |
| Smith et al.[26]| 8k→16k     | TPU (256 tensorcores)  | 76.1%          | 45min         |
| Codreamu et al.[27]| 32k    | KNL × 1024             | 75.3%          | 42min         |
| You et al.[28] | 32k        | KNL × 2048             | 75.4%          | 20min         |
| Akiba et al.[29]| 32k        | Tesla P100 × 1024      | 74.9%          | 15min         |
| Jia et al.[30] | 64k        | Tesla P40 × 1024       | 76.2%          | 8.7min        |
| Ying et al.[31]| 32k        | TPU × 1024             | 76.3%          | 2.2min        |
| Mikami et al.[32]| 54k      | Tesla V100 × 3456      | 75.29%         | 2.0mins       |
| Yamazaki et al.[33]| 80k    | Tesla V100 × 2048      | 75.08%         | 1.2mins       |

machines, the entire training process requires 250000 iterations, in contrast, with the batch size is 8192 on 16 machines, only 15625 iterations are required[24]. Table 1 shows ResNet-50 training result with different configurations.

Nevertheless, directly deploying parallel SGD with a huge batch size in practice usually suffers generalization ability degradation compared with small batch training[34, 23]. Fig4. illustrates this situation, in which when the batch size exceeds 8k (we use 1k to denote 1024 samples), the error on the validation set increases dramatically as the batch size increases. The reason is that a large batch method tends to converge to sharp minima of the training function[34]. These minima are characterized by a significant number of large positive eigenvalues in the Hessian matrix, indicating that the curvature around these minima is large. As shown in Fig.5, sharp minima often generalize less well. Conversely, a small batch method converges to flat minima characterized by having many small eigenvalues of the Hessian matrix and small curvature. It is observed that the loss function landscape of DNN is such that large-batch methods are attracted to regions with sharp minima and unable to escape from it[34].

![Figure 5: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters). Adapted from [34]](image-url)

Many methods have been proposed to prevent model converging into sharp minima, including learning rate scaling rules[35, 23], various warmup schemes[23, 36], and layerwise adaptive rate scaling (LARS)[28, 37, 24]. The scaling rules describe that the learning rate should increase to prevent losing accuracy with the increase of batch size. There are two rules of increasing learning rate: Sqrt Scaling Rule[35] and Linear Scaling Rule[23]. When the batch size is multiplied by $k$, the first rule multiplies the learning rate by $\sqrt{k}$ to keep the variance of the gradient estimator constant[35], while the second multiplies learning rate by $k$ based on assumption introduced by [23]. When using linear scaling rule, the learning rate usually is too large at beginning for the model to converge. Hence, at the very beginning of training (i.e., 5 epochs), we slowly and smoothly increase the learning rate to the suggested value of the linear scaling rule. This scheme is called gradual warmup[23]. To deploy warmup on RNN (i.e., LSTM and GRU) training process, You et al.[36] proposed linear-epoch gradual warmup (LEGW) scheme, in which the warmup epochs is multiplied by $k$.
when increase the batch size $k$ times. LARS applies different learning rates for different layers in DNN, due to the distribution of gradients and parameters are various from different layers[28, 37, 24]. However, LARS performs poorly for attention models like BERT[38], indicating that its performance gains are not consistent across tasks[39]. Therefore, You et al.[39] proposed a general layerwise adaptive large batch optimization technique called LAMB, which performs well across various tasks such as BERT and ResNet-50 training with minimal hyperparameter tuning. As for RNN training, Chen et al.[40] proposed blockwise model update filtering (BMUF) to train LSTM on 16 GPUs and achieved near-linear speedup.

3.1.2 Periodic Communication

Recall that training DNNs using traditional distributed SGD, each worker at each iteration first performs a single SGD step and calculates local gradients, then exchanges gradients and parameters with nodes. Communication occurs at the end of each iteration. Suppose that we parallelize a DNN training task on $K$ nodes with $T$ iterations, the complexity of rounds of vanilla parallel SGD is $O(T)$. Due to such high complexity of communication rounds, many works suggest reducing the frequency of exchanging gradients and parameters.

![Diagram of communication frequency]

(a) Parallel SGD

(b) Local SGD

(c) One-Shot

Figure 6: Comparison of vanilla parallel SGD, local SGD and one-shot learning, where a green block indicates computation and a yellow block corresponds to communication.

In model averaging, individual model parameters trained on local nodes are averaged periodically. We consider averaging operation occurs at most $\tau$ iterations, where $\tau$ is a factor of the period. Averaging occurs in every iteration, i.e., $\tau = 1$, which is identical to vanilla parallel SGD. Conversely, averaging occurs only at the end of the training, i.e., $\tau = T$, which is equal to one-shot averaging. The case of $1 < \tau < T$ is another common setting. Fig. 6 depicts three cases. Experimental works[41, 42, 43, 44] have verified that model averaging can reduce communication overhead of the training time as long as the period is suitable. In addition, some theoretical studies[45, 46, 47, 48] have given analysis why model averaging can achieve good convergence rate.

The one-shot averaging, which only requires communication at the end of the training, is an extreme case of model averaging. It is shown that one-shot averaging has satisfying performance on both convex[41] and some non-convex[42] optimization problems. However, Zhang et al.[43] reported some certain non-convex optimization problems that cannot be solved by one-shot averaging. As a remedy, they suggest that more frequent averaging can improve performance.

Between one-shot averaging and vanilla parallel SGD, tremendous works have focused on periodical averaging, in which communication occurs every $\tau$ iterations. There is another aspect of research which aims to reduce communication cost by adding redundancy into training dataset. For instance, Haddadpour et al.[53] show that by adding a proper amount of redundancy through coding theoretic means, linear regression can be solved through one round of communication. Additionally, Haddadpour et al.[50] demonstrate that by properly infusing redundancy to the training data with model averaging, it is conceivable to significantly reduce the order of communication rounds as PR-SGD[45] with less accuracy degradation for general non-convex optimization. The advantages of model averaging have been examined from a practical point of view in[44]. Specifically, they show that model averaging performs well empirically in terms of reducing communication costs for a given accuracy. Arjevani et al.[47] studied theoretical lower bounds of the
Table 2: Comparison of Periodic Averaging SGD

| Works               | Communication Rounds \((T/\tau)\) | Linear Speedup |
|---------------------|-----------------------------------|----------------|
| vanilla Parallel SGD[49] | \(O(T)\)                        | ✓              |
| One-Shot[41, 42]    | \(O(1)\)                         | ✓              |
| K-AVG[48]           | \(O(T/\tau)\)                    | ✓              |
| PR-SGD[45]          | \(O(K^{3/4}T^{3/4})\)            | ✓ when \(T > N^3\) |
| Local SGD[46]       | \(O(K^{1/2}T^{1/2})\)            | ✓              |
| RI-SGD[50]          | \(O(K^{1/2}T^{1/2})\)            | ✓              |
| LUPA-SGD[51]        | \(O(K^{1/3}T^{1/3})\)            | ✓              |
| COCOD-SGD[52]       | \(O(K^{3/4}T^{3/4})\)            | ✓              |

number of communication rounds required under different settings in order to solve distributed convex learning and optimization problems.

Besides model averaging, asymmetrical push and pull can also increase the communication period. Dean et al.[54] proposed a feasible solution called asymmetrical push/pull, in which workers request update parameters from servers every \(n_{fetch}\) step, and send gradients to server every \(n_{push}\) step (\(n_{fetch}\) may not equal to \(n_{push}\)). However, they only provide an experimental result for asymmetrical push/pull, while the convergence analysis is not provided. Chen et al.[55] developed Lazily Aggregated Gradient (LAG) methods, in which workers and servers synchronize gradients and parameters only when some conditions are violated. These conditions are generated and exported by the estimation of objective value descent for one iteration. LAG has convergence guarantees both experimental and theoretical. When use nine workers to train a linear regression model, LAG can decrease parameter server communication complexity from 5283 to 1756 compared with vanilla parallel SGD.

3.2 Gradients Compression

Each computation node requires communication operations to exchange gradients and model parameters during distributed training. Using 32 bits variables to represent each element in gradients and model parameters is the most commonly used configuration. When the size of gradients and model parameters is large, the communication bottleneck caused by exchanging a large amount of 32 bits single-precision variables impairs advantages of parallelization. For example, consider training the BERT architecture for neural language processing tasks with about 340 million parameters, implying that each full precision exchange between nodes is over 1.2GB[38]. Hence, we discuss how to reduce the size of the message in this section.

There are mainly three bandwidth-efficient approaches in distributed deep learning: quantization, sparsification and decomposition. The first substitutes 32 bits variables for low-precision variables (i.e., 8 bits, 4 bits, or even 1 bit), whereas the second only transports important variables to avoid unnecessary overhead. Since these two approaches are orthogonal, they can be combined to compress the communication further. The third is not as popular as the first two, it transmits small matrices instead of large by matrix decomposition. Fig. 7 compares various compression methods.

3.2.1 Quantization

As a low-precision data representation method, quantization first discretizes continuous values and maps them to different integers in a range. A standard method among quantization is to use the sign of each element to represent the gradient. As far as we know, one-bit SGD[56] has pioneered the research of communication quantization for distributed deep learning. Recently, a similar algorithm called signSGD[57] was proposed for non-convex optimization.

Because the difference between gradients processed by sign-based quantization and original gradients is too large, the quantized gradient is often a biased estimate of the original gradient[58], which makes the model converge slowly and significant accuracy loss[59]. To address this issue, we use the error feedback technique to correct the deviation of direction accumulated in previous iterations. Error feedback maintains a vector \(e\) to store the accumulated difference
As far as we know, Strom et al. first clips gradients via a static threshold, the element will not be transmitted if its absolute value is below this static threshold. However, it is not very easy to select a reasonable threshold for various DNNs. To remedy this issue, subsequent works applied local selection and dynamic threshold instead of a static threshold. A variant of threshold sparsification is Top-K method, in which each node only transmits \( k \) largest (absolute) values. Theoretical work gave a convergence analysis of Top-K sparsification under some assumptions, and pointed out that sparsification is equivalent to stale update.

To ensure model convergence, we usually select values from gradient residual rather than manipulate original gradients directly. The gradient residual is the sum of all previous gradients accumulated locally at each node. Stich et al. proved that with accumulated gradients, sparsified SGD has the same convergence rate as vanilla parallel SGD. DGC introduced momentum correction, local gradient clipping, momentum factor masking, and warm-up training to achieve better performance. By these methods, DGC compresses the gradient size of ResNet-50 from 97MB to 9,200KB.
Figure 7: Different compression schemes, the first is one layer’s original gradient as a matrix, while others are the output of various compression schemes taking the first as input. Red means negative values and green means positive. Adapted from [67].

Wangni et al. [76] proposed random sparsification, which drops out indices of gradients randomly. In order to guarantee the sparsified vector is unbiased, the remaining part is appropriately amplified. Experiments show that random dropping only causes little accuracy loss of 3-layer CNN on CIFAR-10 dataset.

Since quantization and sparsification are two orthogonal methods, they can be integrated for deep compression. The integration is straightforward based on a centralized architecture (Parameter Server). However, there is a challenge that needs to be solved for sparsification in a decentralized setting. Recall that sparsification only transmits “important” values in the gradient, which means that each node may receive different non-zero indices (dimensions) for the same gradient [9]. Fortunately, there are a series of implementation [77, 78, 79, 74] for sparse communication. Using such sparse collective communication protocols, we can implement sparse gradient exchange in a decentralized architecture, and combine it with quantization.

3.2.3 Matrix Decomposition

In the field of compression, an emerging method is to decompose the large gradient matrix into several small matrices before transmission, and reconstruct it after receiving. Transmitting two small matrices has less communication overhead than one huge matrix. This method is feasible because of the correlation between gradients. GradiVeQ[80] exploited such linear correlations between CNN gradients and applied principal component analysis (PCA) [81] to reduce gradients dimensions. Besides, the proposed method GradiVeQ also enables direct aggregation of compressed gradients. ATOMO[82] is a general compression method built on top of singular value decomposition (SVD) [83]. For a given gradient, ATOMO will produce a random unbiased gradient with minimal variance [82]. PowerSGD [67] performs low rank decomposition with error feedback, and avoids the computational expensive SVD step in ATOMO [82] to achieve better scalability.

3.3 Computation-Communication Overlap

Since the gradient is generated in the order from the last layer to the first layer during back propagation, there is no need to wait for the calculation of the previous layer to complete before sending the gradient of the later layer. In other words, former layers’ computation is independent of the latter layers’ communication, and the latter layers’ parameters update is independent of the former layers [84]. Hence, we can transmit the gradient of the $L_{th}$ layer while computing the $(L-1)_{th}$ layer’s gradient. Although overlap technique can efficiently reduce total communication time, it does not change communication time itself.

Poseidon [84] provided a wait-free backward propagation (WFBP) scheduling algorithm, see in Fig. 8(b). WFBP makes each layer to start its communication once its gradients are calculated after backward propagation. However, different layers may have different computation and communication times, which means Poseidon may not outperform than FIFO scheduling on some specific network models.

Shi et al. [85] pointed out that there are three cases of WFBP. Fig. 9 depicts that both case 1 and case 2 are ideal cases in which we can hide the communication time easily. Case 3 could be more often happened, especially in the high latency or low bandwidth network environment. They find that two or more small messages can be merged into one large message before being sent, and the merged gradients can be communicated with a smaller cost that can be hidden by computation easily. The newly proposed method, named merged-gradients WFBP (MG-WFBP), achieves much better scaling efficiency than WFBP.

Priority-based parameter propagation (P3) [86] extended WFBP by priority-based scheduling, in which the first layer gets the highest priority, whereas the last layer gets the lowest priority. Tensor with the highest priority is processed first during the communication phase, no matter when it is generated. P3 uses the tensor partition technique to handle
Figure 8: Sequential vs. pipeline training scheme.

Figure 9: Three case of computation-communication overlap. Adapted from [85].

case 3, shown in Fig. 9. Tensor partition technique splits layers’ parameter matrix into small proper pieces and assigns priorities to every slice based on their parent layers’ processing order in the forward propagation. Hashemi et al.[87] exploited the execution order of computational graphs and proposed two heuristic scheduling algorithms: Timing-Independent Communication (TIC) and Timing-Aware Communication (TAC). Both algorithms are built on the properties of communication operations, such as communication dependency, communication time, and directly dependent compute load, etc. ByteScheduler[88] applies tensor partition and priority-based scheduling like P3, and designs a credit-based preemption approach to utilize network bandwidth fully. Credit-based preemption works like a sliding window, where credit is the window size. Small tensor pieces in the sliding window are sent simultaneously. ByteScheduler uses Bayesian optimization to find the ideal credit and partition size.

4 Network Infrastructures

In this section, we are mainly concentrated on optimizing low-level network infrastructures, including advanced centralized and decentralized architectures, messaging libraries, and network protocols. Such modifications have little effect on high-level algorithms.
4.1 Logical Architectures

Centralized and decentralized architectures have different communication patterns and performance. We will discuss modern and advanced architectures in this subsection.

4.1.1 Parameter Server

![Parameter Server Architectures](image)

Fig. 10: Parameter server architectures.

Fig. 10(a) depicts the traditional parameter server architecture, in which the server is responsible for storing and updating global model parameters. The server is prone to network bottlenecks, especially when there are many working nodes. Tree-based parameter servers [89, 90, 91] alleviate such bottlenecks to some extent.

Mai et al. [89] treat each server as the root and build a spanning tree connecting all workers. All workers are leaf nodes in the spanning tree, whereas other nodes in the tree are servers. Each worker pushes gradients to their parents. The parents aggregate all received gradients and push the results upstream towards the root where the last step of aggregation is performed. The global weights are multicasted from top root server to down leaf worker. Through such a spanning tree, communication overhead in both push and pull operations is reduced. Gupta et al. [90] proposed similar tree-based architectures like [89]. To further alleviate the network traffic, the root server broadcasts the global weights directly down a tree constructed within all workers, see in Fig. 10(b). Heish et al. [91] considered the physical distance of the parameter server and employed an intelligent communication mechanism over wide area networks (WAN) to efficiently utilize the bandwidth. In addition, some works such as Project Adam [92] and Geeps [93] improve the throughput of parameter server by caching and isolated communication.

4.1.2 Allreduce

The key issue with traditional communication strategies is that as the number of GPUs increases, communication cost increases linearly. A classical implementation of Allreduce is the combination of Reduce operation followed by Broadcast which sends the result from root to all processes. This implies a bottleneck on the root process. The optimized algorithms are based on a few principles: recursive vector halving, recursive vector doubling, recursive distance halving, recursive distance doubling, binary blocks, and ring [94].

To the best of our knowledge, Baidu first introduced ring based Allreduce into distributed deep learning [96]. Ring Allreduce is made of two phases: Reduce-Scatter and Allgather; each phase includes \( p - 1 \) communication steps when we use \( p \) GPUs, see in Fig.11. Each GPU maintains its local gradients, which is equally divided into \( p \) chunks. In the reduce-scatter phase, each node sends and receives different chunks of a stored tensor. For the received chunk, each node adds it to the corresponding position in the buffer. After \( p - 1 \) steps, each node holds a different part of the global result. In the all-gather phase, each node sends the part of the global result maintained by itself and receives other parts of the global result from other nodes. Each node holds a complete global result after \( p - 1 \) steps. Hence, Ring Allreduce needs a total of \( 2(p - 1) \) communication steps. The complete communication process is described in Fig. As early as 2009, Patarasuk et al. [97] proved that ring-based Allreduce is the bandwidth optimal Allreduce algorithms.

Mikami et al. [32] proposed a 2D-Torus Allreduce topology in which GPUs are arranged in a 2D grid. Each row contains \( ph \) GPUs while each column contains \( pv \) GPUs. There are three phases in 2D-Torus Allreduce: Reduce-Scatter, vertical Allreduce, and Allgather. Although 2D-Torus Allreduce is one phase more than Ring Allreduce, its overall communication overhead is still smaller because \( ph \) and \( pv \) are less than \( p \) [32]. A similar study by Ying et al. [31] aggregates gradients in two phases with a 2D-Mesh topology, which utilizes two parallel ring-based reductions, each...
Figure 11: The Ring Allreduce algorithm. Adapted from [95].

Figure 12: 2D-Mesh Allreduce across a hypothetical $3 \times 3$ torus. Adapted from [31].

Jia et al.[30] proposed hierarchical Allreduce to solve the problem for small tensor communication. They split all $p$ GPUs into several groups and conduct a three-phase reduction. Fig. 13 shows the three-phase operations. The first phase is a separate ring Allreduce operation in groups independently, each of which consists of $k$ GPUs. Secondly, a master node from each group operates ring Allreduce to get a global result. Finally, the master node in each group broadcasts the global result to every GPU in its group. Compared with Ring Allreduce, this three-phase hierarchical Allreduce decreases the running steps from $2(p - 1)$ to $4(k - 1) + 2(p/k - 1)$.

4.2 Message Level Libraries

The performance of different parameter server systems and various Allreduce algorithms partially depend on the implementation of message communication libraries. The parameter server is usually run on top of ZeroMQ[98] or gRPC[99]. ZeroMQ is a high performance and low latency asynchronous messaging library that supports multiple communication patterns, which gRPC is a high-performance remote procedure call (RPC) framework developed by Google. As far as we know, the most commonly used message communication libraries in the parameter server are still ZeroMQ and gRPC.
At present, there are many message level communication libraries that implement various Allreduce and other collective communication algorithms efficiently, including MPI[100], Gloo[101], NCCL[102], Baidu Allreduce[96], Aluminum[103] and BlueConnect[104]. Thanks to the high performance of MPI, there are many optimizations based on MPI_Allreduce, including Horovod[95], MXNet-MPI[105] and TensorFlow-MPI[106]. To reduce communication, Horovod uses tensor fusion that sends several small tensors simultaneously[95]. NCCL implements multi-GPU and multi-node collective communication primitives that are optimized for NVIDIA GPUs[102]. A series of studies by Awan et al.[107, 108] optimized Bcast operation based on NCCL and CUDA-Aware MPI, respectively. BlueConnect[104] decomposes a single Allreduce operation into a series of parallelizable Reduce-Scatter and Allgather operations to reduce the communication cost. Experiments show that the performance of BlueConnect incurs less communication overhead that Gloo and Baidu Allreduce with more GPUs[104].

4.3 Network Protocols

Traditional message level communication libraries are implemented based on TCP/IP protocol, which handles data sending and receiving by the socket. Each node must create a socket object and establish a connection to the receiver before sending data. Data is processed by the operating system and encapsulated with different protocol headers until to be copied into the network interface controller (NIC) buffer, as shown in Fig. 14(a). Such operations are wasteful in distributed training that needs low network latency. Therefore, high performance and low latency networks run on top of associated network hardware (i.e., InfiniBand) attract much attention; the two most common are remote direct memory access (RDMA)[109] and internet protocol over InfiniBand (IPoIB)[110]. As illustrated in Fig.14(c), RDMA allows one machine to directly read and write the memory of another machine without involving the operating system, which permits high performance and low latency networking. The network interface in RDMA is called verbs, which provide two types of communication paradigms: message and memory. IPoIB, as the name implies, encapsulates IP datagrams over an InfiniBand card and enables TCP/IP applications to run on top of InfiniBand without any code modification[110]. However, as shown in Fig.14(b), IPoIB cannot bypass the host operating system like RDMA in Fig. 14(c).

With the advent of RDMA and IPoIB, tremendous works have been done on improving the performance of distributed training systems, including MXNet[111, 112], TensorFlow[113, 114, 115], CNTK[116, 117], Caffe[118] and IBM deep learning platform[119], to leverage its high bandwidth and low latency. The memory communication paradigm is the most popular method due to its low memory demand[119, 113]. Some works[113, 117, 119] consider communications between GPUs and explore GPU direct RDMA (GDR), which allows an RDMA NIC to access GPU memory directly without going through host memory. Gradients are aggregated in GPUs by using the GDR technique. Furthermore, Biswas et al.[114] designed adaptive RDMA-based gRPC to dynamically adjust communication mechanisms for different message sizes in deep learning workload.

According to experimental result[113, 111, 115], using RDMA and IPoIB to replace TCP/IP protocol can significantly speed up training. In addition, RDMA performs better than IPoIB in distributed training[111, 112]. Experiments by Liu et al.[112] report that compared with IPoIB (53%), the RDMA-capable network achieves near-linear (96%) speed up when scaling Inception-v3 training on 100 GPUs. The result also fits the previous description of IPoIB and RDMA verbs.

Recently, Xia et al.[120] rethought the distributed training process of DNNs and designed a bounded loss tolerance transmission protocol, which ignores a fraction of packet loss during training. The new protocol achieves shorter job
completion time than traditional TCP/IP protocol. However, the random dropping rate has to be tuned when changing DNN architecture. It is challenging to choose a reasonable dropping rate for various DNNs.

5 Conclusion Remarks

It is shown that communication overhead is a significant obstacle to deal with in order to achieve desirable performance in distributed DNN training. In this paper, we provide a comprehensive survey on the recent research of communication and network optimization techniques for distributed DNN training. Below, we highlight potential research directions for distributed deep learning communication and network optimization.

At present, there are a considerable number of communication optimizations focus on the image classification task, especially CNNs trained on CIFAR or ImageNet dataset. For example, a large number of using large batch size studies focus on the ResNet-50 model. Only a few studies focus on RNNs. Future works can explore large batch training on more modern models. As for periodic communication, there remain some research fields for non-convex problems. Despite a lot of theoretical works for model averaging, the research on whether the linear speedup with $\tau > 1$ can be preserved for non-convex optimizations is still blank. Individually, the lower bounds on the number of communication rounds for non-convex optimizations to achieve linear speedup is an interesting research direction.

The core challenge need to be considered in gradient compression trade-off between convergence rate and used memory. Conventional approaches to prevent DNN from divergence include error feedback (for quantization) and local accumulation (for sparsification). More advanced methods such as squared error feedback[65] need to be explored in future works. The ratio of computation and communication is essential for deploying pipeline training. For the sake of a higher overlap rating, various algorithms give different strategies to arrange communication operations and shrink communication time. However, these algorithms are generally heuristic and achieve non-optimal solutions in such a schedule problem. Better optimization algorithms, such as dynamic programming, are necessary to solve this problem.

In recent years, tree-based parameter server and ring based Allreduce attract much attention. As for network protocols, RDMA is an excellent alternative to TCP protocol. Besides traditional lossless transport protocols, lossy protocols may be suitable for distributed deep learning. Such new protocols do not consider re-sending lost data packets but focus on reducing communication overhead, and the structure of the well-designed packets is necessary for more efficient communication. In addition, the physical topology of the data center network is also an interesting acquisition for distributed deep learning communication optimization. Recent work by Wang et al.[121] deployed parameter
server over BCube\cite{122} instead of Fat-Tree\cite{123} architecture, and achieved good performance on LeNet-5\cite{124} and VGG-19\cite{125} training.

Another critical research issue is the performance model and measurement tools of distributed training. Performance models allow us to analyze the various costs of distributed training theoretically\cite{126, 127}, and measurement tools help us find the bottleneck in real training tasks. Although deep learning frameworks provide profiler, more accurate network overhead tools such as SketchDLC\cite{128} are still required.

References

[1] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[2] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[3] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[4] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198. ACM, 2016.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

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

[8] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[9] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):65, 2019.

[10] Karanbir Chahal, Manraj Singh Grover, Kuntal Dey, and Rajiv Ratn Shah. A hitchhiker’s guide on distributed training of deep neural networks. *Journal of Parallel and Distributed Computing*, 2019.

[11] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[12] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, page 101635, 2019.

[13] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[15] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Ying Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

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

[17] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[18] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[19] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[20] DMLC. Ps-lite documents. https://ps-lite.readthedocs.io/.

[21] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[22] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[23] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wésolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[24] Yang You, Zhao Zhang, Cho-Jui Hsieh, Jim Demmel, and Kurt Keutzer. Fast deep neural network training on distributed systems and cloud tpus. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[25] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. Powerai ddl. *arXiv preprint arXiv:1708.02188*, 2017.

[26] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don’t decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[27] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. Scale out for large minibatch sgd: Residual network training on imagenet-1k with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291*, 2017.

[28] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

[29] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[30] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhuo Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[31] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

[32] Hiroaki Mikami, Hisahiro Suganuma, et al. Imagenet/resnet-50 training in 224 seconds. *arXiv preprint arXiv:1811.05233*, 2018.

[33] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

[34] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[35] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[36] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for lstm and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2019.

[37] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[39] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kuri Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2020.
[40] Kai Chen and Qiang Huo. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5880–5884. IEEE, 2016.

[41] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In Advances in Neural Information Processing Systems, pages 2595–2603, 2010.

[42] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pages 456–464. Association for Computational Linguistics, 2010.

[43] Jian Zhang, Christopher De Sa, Ioannis Mitliagkas, and Christopher Ré. Parallel SGD: When does averaging help? arXiv preprint arXiv:1606.07365, 2016.

[44] Hang Su and Haoyu Chen. Experiments on parallel training of deep neural network using model averaging. arXiv preprint arXiv:1507.01239, 2015.

[45] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 5693–5700, 2019.

[46] Sebastian Urban Stich. Local SGD converges fast and communicates little. In ICLR 2019 ICLR 2019 International Conference on Learning Representations, number CONF, 2019.

[47] Yossi Arjevani and Ohad Shamir. Communication complexity of distributed convex learning and optimization. In Advances in Neural Information Processing Systems, pages 1756–1764, 2015.

[48] Fan Zhou and Guojing Cong. On the convergence properties of a k-step averaging stochastic gradient descent algorithm for nonconvex optimization. In Proceedings of the 27th International Joint Conference on Artificial Intelligence, pages 3219–3227. AAAI Press, 2018.

[49] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. Journal of Machine Learning Research, 13(Jan):165–202, 2012.

[50] Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck Cadambe. Trading redundancy for communication: Speeding up distributed SGD for non-convex optimization. In International Conference on Machine Learning, pages 2545–2554, 2019.

[51] Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck Cadambe. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. In Advances in Neural Information Processing Systems, pages 11080–11092, 2019.

[52] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In Advances in neural information processing systems, pages 1223–1231, 2012.

[53] Tianyi Chen, Georgios Giannakis, Tao Sun, and Wotao Yin. Lag: Lazily aggregated gradient for communication-efficient distributed learning. In Advances in Neural Information Processing Systems, pages 5050–5060, 2018.

[54] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In Fifteenth Annual Conference of the International Speech Communication Association, 2014.

[55] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. SignSGD: Compressed optimisation for non-convex problems. In International Conference on Machine Learning, pages 559–568, 2018.

[56] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signSGD and other gradient compression schemes. In International Conference on Machine Learning, pages 3252–3261, 2019.

[57] Demjan Grubic, Leo Tam, Dan Alistarh, and Ce Zhang. Synchronous multi-GPU deep learning with low-precision communication: An experimental study. Proceedings of the EDBT 2018, 2018.
[60] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error compensated quantized sgd and its applications to large-scale distributed optimization. In International Conference on Machine Learning, pages 5321–5329, 2018.

[61] Sebastian Urban Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. In Advances in Neural Information Processing Systems, pages 4447–4458, 2018.

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

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

[64] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In Advances in Neural Information Processing Systems, pages 4440–4451, 2019.

[65] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. arXiv preprint arXiv:1802.07389, 2018.

[66] Peter Elias. Universal codeword sets and representations of the integers. IEEE transactions on information theory, 21(2):194–203, 1975.

[67] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. In Advances in Neural Information Processing Systems, pages 14236–14245, 2019.

[68] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In Sixteenth Annual Conference of the International Speech Communication Association, 2015.

[69] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In 2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC), pages 1–8. IEEE, 2016.

[70] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[71] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 440–445, 2017.

[72] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv preprint arXiv:1712.01887, 2017.

[73] Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Sparse binary compression: Towards distributed deep learning with minimal communication. In 2019 International Joint Conference on Neural Networks (IJCNN), pages 1–8. IEEE, 2019.

[74] Cèdric Renggli, Dan Alistarh, Torsten Hoefler, and Mehdi Aghagolzadeh. Sparcml: High-performance sparse communication for machine learning. arXiv preprint arXiv:1802.08021, 2018.

[75] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. In Advances in Neural Information Processing Systems, pages 5973–5983, 2018.

[76] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In Advances in Neural Information Processing Systems, pages 1299–1309, 2018.

[77] Huasha Zhao and John Canny. Kylix: A sparse allreduce for commodity clusters. In 2014 43rd International Conference on Parallel Processing, pages 273–282. IEEE, 2014.

[78] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. Topology-aware sparse allreduce for large-scale deep learning. In 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC), pages 1–8. IEEE, 2019.

[79] Jiariu Fang, Haohuan Fu, Guangwen Yang, and Cho-Jui Hsieh. Redsync: Reducing synchronization traffic for distributed deep learning. arXiv preprint arXiv:1808.04357, 2018.

[80] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander Schwing, Murali Annavaram, and Salman Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training. In Advances in Neural Information Processing Systems, pages 5123–5133, 2018.
[81] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. Chemometrics and intelligent laboratory systems, 2(1-3):37–52, 1987.

[82] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In Advances in Neural Information Processing Systems, pages 9850–9861, 2018.

[83] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In Linear Algebra, pages 134–151. Springer, 1971.

[84] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jilinlai Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In 2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} '17), pages 181–193, 2017.

[85] Shaohuai Shi, Xiaowen Chu, and Bo Li. Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms. In {IEEE INFOCOM} 2019-{IEEE} Conference on Computer Communications, pages 172–180. {IEEE}, 2019.

[86] Anand Jayarajan, Jindiang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. arXiv preprint arXiv:1905.03960, 2019.

[87] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. arXiv preprint arXiv:1803.03288, 2018.

[88] Yanghua Peng, Yibo Zhu, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In Proceedings of the 27th {ACM} Symposium on Operating Systems Principles, pages 16–29, 2019.

[89] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In 7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15), 2015.

[90] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In 2016 {IEEE} 16th International Conference on Data Mining (ICDM), pages 171–180. {IEEE}, 2016.

[91] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} '17), pages 629–647, 2017.

[92] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} '14), pages 571–582, 2014.

[93] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems, pages 1–16, 2016.

[94] Rolf Rabenseifner. Optimization of collective reduction operations. In International Conference on Computational Science, pages 1–9. Springer, 2004.

[95] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799, 2018.

[96] Baidu. Baidu allreduce. https://github.com/baidu-research/baidu-allreduce, 2017.

[97] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. Journal of Parallel and Distributed Computing, 69(2):117–124, 2009.

[98] Pieter Hintjens. ZeroMQ: messaging for many applications. " O'Reilly Media, Inc.", 2013.

[99] Google. Grpc. https://grpc.io.

[100] William Gropp. Mpich2: A new start for mpi implementations. In European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pages 7–7. Springer, 2002.

[101] FaceBook. Gloo. https://github.com/facebookincubator/gloo.

[102] Sylvain Jeangey. Nccl 2.0. GTC, 2017.

[103] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Marc Snir, and Brian Van Essen. Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. 2018 {IEEE/ACM} Machine Learning in HPC Environments (MLHPC), pages 1–13, 2018.
[104] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tiered network for deep learning. In Proceedings of the Conference on Systems and Machine Learning (SysML), 2019.

[105] Amith R Mambdala, Georgios Kollias, Chris Ward, and Fausto Artico. Mxnet-mpi: Embedding mpi parallelism in parameter server task model for scaling deep learning. arXiv preprint arXiv:1801.03855, 2018.

[106] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed tensorflow with mpi. arXiv preprint arXiv:1603.02339, 2016.

[107] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl? In Proceedings of the 25th European MPI Users' Group Meeting, pages 1–9, 2018.

[108] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. Efficient large message broadcast using nccl and cuda-aware mpi for deep learning. In Proceedings of the 23rd European MPI Users' Group Meeting, pages 15–22, 2016.

[109] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. Technical report, RFC 5040, October, 2007.

[110] Jerry Chu and Vivek Kashyap. Transmission of ip over infiniband (ipoib). Technical report, RFC 4391, April, 2006.

[111] Mingfan Li, Ke Wen, Han Lin, Xu Jin, Zheng Wu, Hong An, and Mengxian Chi. Improving the performance of distributed mxnet with rdma. International Journal of Parallel Programming, 47(3):467–480, 2019.

[112] Jiulong Xue, Youshuan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–14, 2019.

[113]Ching-Hsiang Chu, Xiaoyi Lu, Ammar A Awan, Hari Subramoni, Jahanzeb Hashmi, Bracy Elton, and Dhabaleswar K Panda. Efficient and scalable multi-source streaming broadcast on gpu clusters for deep learning. In 2018 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS), pages 473–480. IEEE, 2018.

[114] Jilong Xue, Youshuan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–14, 2019.

[115] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. Accelerating tensorflow with adaptive rdma-based grpc. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pages 2–11. IEEE, 2018.

[116] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. Improving the performance of distributed tensorflow with rdma. International Journal of Parallel Programming, 46(4):674–685, 2018.

[117] Dip Sankar Banerjee, Khaled Hamidouche, and Dhabaleswar K Panda. Re-designing cntk deep learning framework on modern gpu enabled clusters. In 2016 IEEE international conference on cloud computing technology and science (CloudCom), pages 144–151. IEEE, 2016.

[118] Ching-Hsiang Chu, Xiaoyi Lu, Ammar A Awan, Hari Subramoni, Jahanzeb Hashmi, Bracy Elton, and Dhabaleswar K Panda. Efficient and scalable multi-source streaming broadcast on gpu clusters for deep learning. In 2017 46th International Conference on Parallel Processing (ICPP), pages 161–170. IEEE, 2017.

[119] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 193–205, 2017.

[120] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019, pages 22–28. ACM, 2019.

[121] Songtao Wang, Dan Li, Yang Cheng, Jinkun Geng, Yanhua Wang, Shuai Wang, Shu-Tao Xia, and Jianping Wu. Bml: A high-performance, low-cost gradient synchronization algorithm for dml training. In Advances in Neural Information Processing Systems, pages 4238–4248, 2018.

[122] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In Proceedings of the ACM SIGCOMM 2009 conference on Data communication, pages 63–74, 2009.

[123] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. ACM SIGCOMM computer communication review, 38(4):63–74, 2008.
[124] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[125] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[126] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1355–1364, 2015.

[127] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2017.

[128] Yemao Xu, Dezun Dong, Weixia Xu, and Xiangke Liao. Sketchdlc: A sketch on distributed deep learning communication via trace capturing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–26, 2019.