Journal of Machine Learning Research 1 (2020) 1-48 Submitted 11/15; Published 11/15

Distributed System for Deep Neural Network Training: A Survey

Abstract

Recent years have witnessed a growth in computation requirement to train modern deep neural networks with massive data volume and model size. Distributed systems are widely employed to accelerate the training process. In this article, we survey the principle and technology to construct such a system. Data parallelism and model parallelism are two fundamental strategies to parallelize the training process. Data parallelism separate training data to different nodes, while model parallelism partition the model. We summarize the architecture to utilize these strategies and how to minimize communication overhead and reach high scalability. Besides some compression techniques to accelerate data transmission are investigated in detail.

Keywords: distributed deep learning, deep neural network, distributed computing

1. Introduction

Deep neural network (DNN) has been proven a powerful method for tackling a wide range of artificial intelligence tasks in recent years. Together with their success is the rapid growth of their data volume and model size, which greatly increases the requirement of time and computing power to train. For example, CNN for image classification might take several weeks to train on a single GPU (Russakovsky et al., 2015). This is unacceptable for both production and research. To solve the problem, distributed systems are widely employed for DNN training. In a distributed training system, a large number of computation nodes (CPU, GPU, ASIC, FPGA, etc.) are linked together to complete the training task. It is challenging to design a system that can take full usage of these devices' power.

Though distributed large-scale computing system has been well studied, there are some characteristics that make distributed DNN distinguish itself from other distributed systems:

1. **Data throughput**: There is usually a huge amount of data transmission together with DNN. DNN can contain more than one billion parameters (Chilimbi et al., 2014) and millions of data samples (Deng et al., 2009). It is time-consuming to transmit so much data. Therefore, it is critical to minimize data transmission during the training process.

2. **Noise Insensitivity**: Different from most kinds of computing, DNN is insensitive to data noise and inaccuracy up to some limit. Such sensitivity can be exploited to accelerate computing and transmission.

3. **Scalability**: DNN training can be deployed to a wide range of computing clusters, from a few machines to thousands of GPUs. A well-designed architecture should keep high efficiency in both situations. It is sometimes also necessary to be able to easily add or remove nodes from the cluster, without redesigning the whole system.

4. **Fault Tolerance**: When running a cluster of servers, especially in cloud environment where machine and network are not reliable, node fault is a common concern. There should be enough protection to prevent single fault ruining the whole system.

Distributed training of can be approached via two fundamental strategy: data parallelism and model parallelism. For data parallelism, training data is divided into several parts and allotted to each node. On the other hand, model parallelism seeks splitting the network architecture. Each worker undertakes the computation of the its own subset of the network.

This survey is roughly divided into three parts. The first two parts focus on the algorithms and communication used in data parallelism and model parallelism respectively, and how we can optimize these strategies. The third part discusses how to compress model data to minimize transmission overhead with little loss of final results.

2. Data Parallelism

Data parallelism is a basic parallelism strategy. For deep learning, it means each worker node stores a replica of the model, with their local parameters, while the dataset is divided and stored in different nodes. When training, each node loads a minibatch of local training data and apply specific learning algorithm such as stochastic gradient descent (SGD) to compute how parameters should be updated. Then some synchronization mechanism is issued to aggregate all updating information and update parameters globally (Wei et al., 2015).

Data parallelism is easy to scale because no rearrangement of the model structure is involved. It is suitable to process a massive set of data, which is common in modern deep learning cases. Hence data parallelism is widely used in most of the distributed learning tasks.

2.1 Parameter Server

When training with data parallelism, the key operation is synchronizing among worker nodes, in order to make sure works for each node can be aggregated. **Parameter Server** (PS) is a basic architecture to achieve this goal. In PS architecture, there are some nodes called **parameter server nodes**. Each of the server nodes stores a part of model parameters. One parameter may be stored with a few replicas for fault recovery. There are multiple PS nodes used instead of only one, in order to balance the load and avoid bandwidth bottleneck of a single PS node. The number of PS nodes usually increases linearly with the number of total nodes. At the same time, training data are divided into some parts and allotted to each **worker nodes**. In addition, **task scheduler nodes** are sometimes needed to control the system and monitor abnormal nodes. It is noteworthy that one node may play multiple roles in this system. For example, a PS node is usually a worker node at the same time.

There are two basic methods to use SGD with parameter server – synchronous SGD and asynchronous SGD, as is shown in algorithm 1. In synchronous SGD, each of server nodes needs to wait for all worker nodes to complete their computation, then update the parameters. Extra synchronization overhead is needed in this mode. While in asynchronous

SGD, once the server received gradients from worker nodes, it immediately updates current parameters.

Algorithm 1 Distributed SGD

Synchronous SGD

Worker $r = 1, \ldots, m$:

load training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
pull the parameters $w^{(0)}$ from servers
repeat
    choose a minibatch $B_r^{(t)} \subseteq [n_k]$
    $g_r^{(t)} \leftarrow \sum_{k \in B_r^{(t)}} \frac{\partial}{\partial w^{(t)}} \mathcal{L}(y_{i_k}, x_{i_k}, w_r^{(t)})$
    push $g_r^{(t)}$ to servers
    wait for $w_r^{(t+1)}$ from servers
until end

Servers:

initialize parameters $w^{(0)}$
repeat
    wait for all $g_r^{(t)}$
    $w^{(t+1)} \leftarrow w^{(t)} - \eta \sum_{r=1}^{m} g_r^{(t)}$
until end

Asynchronous SGD

Worker $r = 1, \ldots, m$:

load training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
pull the parameters w from servers
repeat
    choose a minibatch $B_r \subseteq [n_k]$
    $g_r \leftarrow \sum_{k \in B_r} \frac{\partial}{\partial w} \mathcal{L}(y_{i_k}, x_{i_k}, w_r)$
    push g_r to servers
    wait for w_r from servers
until end

Servers:

initialize parameters w
repeat
    receive g_r from some worker
    $w \leftarrow w - \eta g_r$
until end

2.2 Synchronous SGD

Synchronous SGD is simple to implement and analyze. Because it works nearly the same as ordinary SGD. But it also faces some limitations.

Efficiency is the most prominent limitation. It is noteworthy that, the server must receive all gradients from worker nodes to employ gradient descent. Therefore the time of iteration is determined by the slowest node (slow in computation or network). Since network transmission is not stable and the speed may fluctuate wildly, and the network distance from different nodes may differ a lot, there is usually a large gap between the speed of fasted nodes and slowest nodes. Experiments show that it is much more time-consuming to wait for the last a few gradients than other ones Chen et al. (2017) (see Figure 1). This is called **straggler effects**.

Straggler effects cause faster worker nodes stay idle for a long time, which is a waste of computation resources. One remedy is to simply drop the slowest few nodes, which can significantly speed up synchronization. However, if too many nodes are dropped, equivalent batchsize will be reduced then convergence will slow down. According to Chen et al. (2017), dropping 4 nodes is the optimal tradeoff for 100 working nodes. However, the idle phenomenon still wastes a large proportion of computation time.

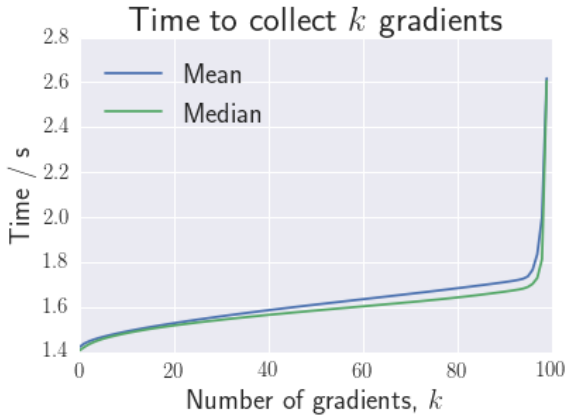

Figure 1: Waiting time for gradients

2.3 Asynchronous SGD

In asynchronous SGD, PS nodes need no waiting to synchronize between nodes. Hence asynchronous SGD is much faster than synchronous SGD because of no synchronization overhead is involved. In addition, asynchronous SGD is more invulnerable to machine failures than synchronous SGD, because if one worker fails, other workers can continue work and process training data (Dean et al., 2012).

The primary drawback is, in asynchronous SGD, the gradients pushed from worker nodes may correspond to "stale" parameters instead of the latest ones, in lack of synchronization. For this reason, the descent progress may not head the gradient direction. Then asynchronous SGD is potentially inferior in stability and accuracy (Chahal et al., 2018). Figure 2 is an experiment result showing how model convergence decreases with staleness (Chen et al., 2017). If the network fluctuates too wildly, the server may update its parameters according to too stale data. Hence the convergence of asynchronous cannot be ensured.

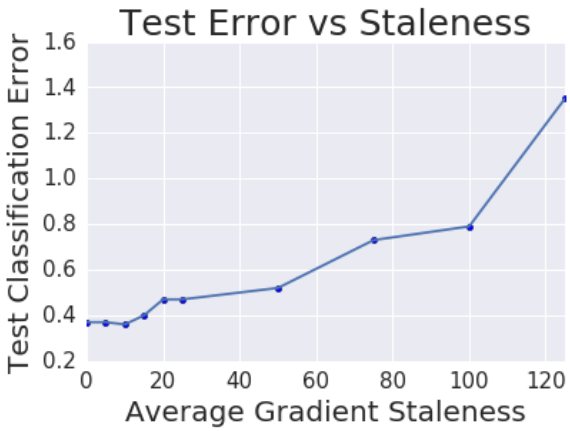

Figure 2: Training error increases with staleness s. It means that working node receives parameters s iterations ago.

Here are a few remedies for this drawback. **Bounded delay** is one of these methods. Bounded delay means that a maximal delay time τ is set. When worker requests parameters, PS inspect whether all tasks issued τ ago is completed. If not, PS will wait until all such tasks are completed then send out updated parameters (Zhang and Kwok, 2014). In this way, worker nodes will not use parameters too long time ago, thus improving convergence stability.

Li et al. (2013) proved that if loss function is Lipchitz continuous and the learning rate is small than a constant (determined by τ and Lipchitz constant of loss function), bounded delay can finally converge to a stationary point. This implies that bounded delay may keep most convergence property of normal SGD.

Zheng et al. (2019) proposed another promising method called Delay Compensated ASGD (DC-ASGD): PS need not only to remember current parameters $w^{(t)}$, but also parameters last pulled from each worker $w_{bak}(m)$. When updating the parameters according to gradient $g(m)$ from m-th worker, we set

$$w^{(t+1)} = w^{(t)} - \eta \left(g_m + \lambda_t g_m \odot g_m \odot (w^{(t)} - w_{bak}(m)) \right), \tag{1}$$

where $\odot$ denotes element-wise multiplication, λ_t is a constant depending on t. The intuition of this method is to use Taylor expansion and Hessian matrix to predict how the gradient changes since last pulling. Convergence for convex and non-convex optimization is theoretically proved.

Zhang et al. (2015b) provides a simpler method called **Elastic Averaging SGD** (EASGD) to reach similar goal. A quadratic penalty is added to optimization function:

$$\overline{\mathcal{L}}(w_1, w_2, \ldots, w_m, \tilde{w}) = \sum_{i=1}^{m} \left(\mathcal{L}(w_i, X_i) + \frac{\rho}{2} \left\| x_i^{(t)} - \tilde{x} \right\|^2 \right) \tag{2}$$

Then each worker applies SGD on this optimization function. Experiments show that this algorithm quickly achieves improvement compared to vanilla asynchronous SGD. **Gossiping SGD** is a decentralized version of EASGD (Jin et al., 2016). It uses the average of local weights connected with each worker instead for the center variable $\tilde{w}$, thus eliminating the need of a centralized PS. This method is used in decentralized architecture discussed in later parts.

2.4 Allreduce architecture

In PS architecture, there is a prominent imbalance between PS nodes and non-PS nodes. If PS nodes are not worker nodes at the same time, their computational resource will be wasted; otherwise, if PS nodes are also worker nodes, their bandwidth requirement is much larger than normal worker nodes, because they need to transmit data both to worker nodes, but also other PS nodes. In both situations, there is some resource wasted. Some decentralized architectures are designed to alleviate this problem.

The task of PS is to collect gradient of workers, add them up and distribute back the result to workers. The key process is the summation. Inspired by technique common in HPC (High-Performance Computing), Andrew Gibiansky (2017) proposed using **Ring Allreduce** algorithm to complete this summation. In Ring Allreduce algorithm, workers

are number as $0, 1, \ldots, N-1$, and parameters are divided into N chucks $0, 1, \ldots, N-1$. Each chuck is in the charge of the corresponding worker. The algorithm is divided into two stages. After calculating the gradient of each parameter, in the first stage called Scatter-Reduce, every worker sends all gradient not in its charge to corresponding workers. In the second stage called Allgather, every worker sums up all gradients received and the gradient calculated by itself, then send the summation to all other workers. After the two stages, every node have the summation of all gradient sum of every parameter. (See algorithm 2 for details, or figure 3 for illustration)

Algorithm 2 Ring Allreduce of worker i

```
for j = 0, 1, ..., m − 1 do                                   ▷ Initialization
    gradient-of-chunk[j] ← gradient computed for each chunk
    gradient-sum-of-chunk[j] ← 0
end for
for j = 0, 1, ..., m − 1 do                                   ▷ Scatter-Reduce
    SEND-TO-MACHINE(i - j + 1, gradient-of-chuck[i])
    gradient-sum-of-chuck[i] += RECEIVE-FROM-MACHINE(i - j - 1)
end for
for j = 0, 1, ..., m − 1 do                                   ▷ Allgather
    SEND-TO-MACHINE(i - j + 1, gradient-sum-of-chuck[i])
    gradient-sum-of-chuck[i - j - 1] ← RECEIVE-FROM-MACHINE(i - j - 1)
end for
```

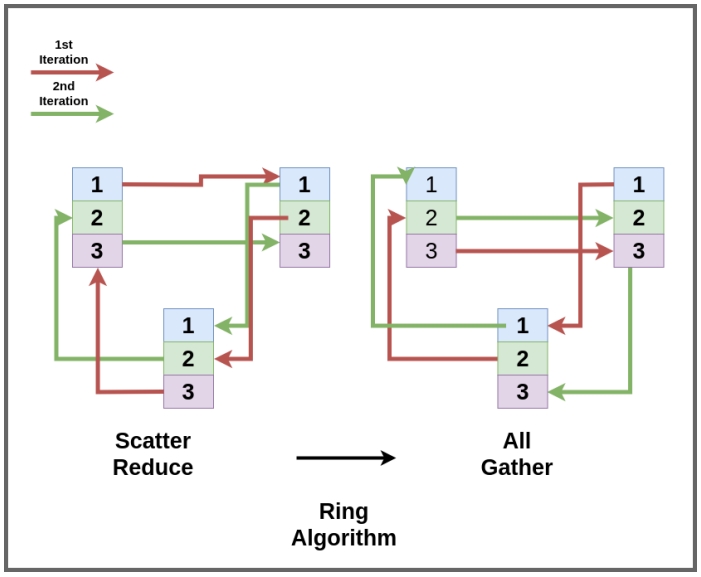

Figure 3: A graph credited to Chahal et al. (2018), which illustrates Ring Allreduce when $N = 3$.

In Allreduce algorithm, each worker node is equal – it needs to send and receive $(N-1)/N$ of total gradients. It can be easily proved that this is a lower bound of through-

put. Because this lower bound does not depend on N, this architecture can easily scale without a prominent increase in communication cost. Sometimes, a chuck is too big to transmit in one time, which cannot make full use of bandwidth, thus subdivide the chuck is needed (Patarasuk and Yuan, 2007). There are some other Allreduce algorithms, including **Recursive Halfing and Doubling** algorithm based on a binary tree instead of a circle, and its improvement **Butterfly-like Allreduce** algorithm, **Binary Blocks** algorithm (Thakur et al., 2005; Patarasuk and Yuan, 2009). These algorithms are simpler than Ring Allreduce but slower when N is large, thus appropriate for smaller systems.

In a scenario where the system consists of network or devices of low reliability, Raft consensus algorithm can be employed to attain fault tolerance (Ongaro and Ousterhout, 2014).

2.5 Transmission Optimization

Since network transmission and gradient computation can be done simultaneously, we can use pipelining to overlapping transmission time and computing time to accelerate synchronization. In Poseidon framework developed by Zhang et al. (2017), a pipeline called **Wait-free Backpropagation** (WFBP) is designed. Because backpropagation is executed layer by layer, when it propagates to i-th layer, the former $i-1$ layers are ready to be pushed to PS. Apart from the pipelining push process, when PS nodes are updating gradient, it can pull back a layer immediately after updating this layer. A coincidence is that many CNN model uses small-size, computational intensive convolutional layer in former layers, and large-size, computational friendly FC layers for later layers. So WFBP can overlap I/O intensive and computationally intensive process, to maximize resource utilization for these models.

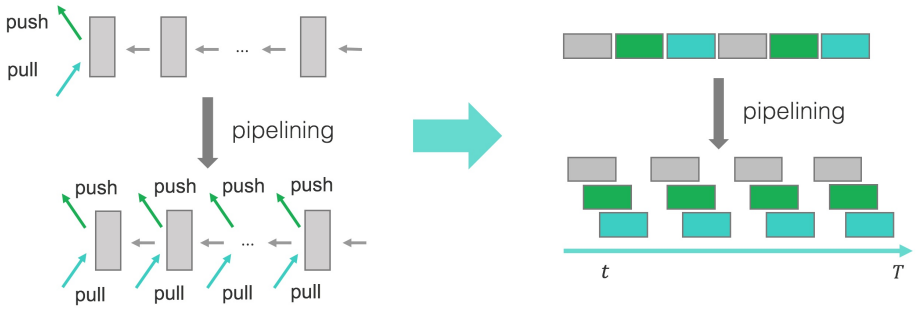

Figure 4: WFBP pipelines pull/push process. Green block represent push, while blue is pulling. (Zhang et al., 2017)

FC layers in DNN often take up the largest amount of parameters. A key observation to optimize FC layer transmission is to notice that, the gradient matrix of FC layers is a low-rank matrix. If W is a weight matrix between two FC layers l_i and l_{i+1}, let s_j be a sample in minibatch B. During the forward pass, s_j is fed into the network and produces a_j in layer l_i. During BP, E_j is produced when loss propagated to l_{i+1}. It can be verified that for gradient of W relative to s_j is $E_j a_j^\top$. Hence total gradient for this minibatch is $\sum_j E_j a_j^\top$, whose rank is no more than $|B|$. When batchsize is not large, worker nodes can

just push vectors E_j, a_j (called **Sufficient Factors**) to other nodes. For example, when transmitting AlexNet on 4 GPU nodes with batchsize 256, Sufficient Factors optimization can reduce the transmitting parameters from 134.2M to 18.9M (Krizhevsky et al., 2012). Other nodes need to execute some matrix multiplication to restore the gradient, which is a relatively small overhead compared with transmitting a large matrix. Zhang et al. (2015a) designed **Structure-Aware Communication Protocol** (SACP) to automatically choose whether the above optimization should be used in each layer.

3. Model Parallelism

Another parallelism strategy is model parallelism. In model parallelism, a DNN model is partitioned into M parts. Each worker node undertakes the computation task of one part. Model parallelism is suitable for large-scale networks containing a huge number of parameters. Because no parameter synchronization is required.

AlexNet is one typical example of model parallelism (Krizhevsky et al., 2012). Since one GPU (GTX 580, 3GB memory) is not enough to contain 1.2 million training examples, two GPUs are deployed to train this model. Each GPU contains roughly half of the parameters (shown in Figure 5). When training via SGD, training minibatch is copied to each GPU and computed individually.

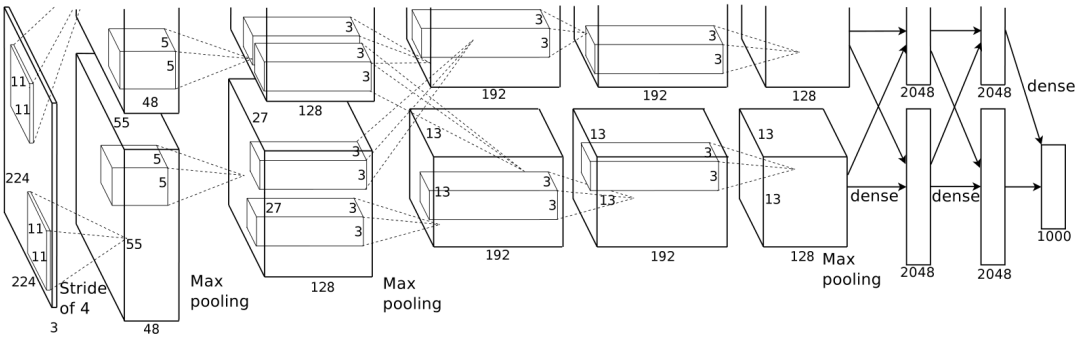

Figure 5: The architecture of AlexNet. The upper and lower part are stored in two GPUs respectively.

Deploying a model on multiple machines means frequent transmissions of parameters and intermediate results. So it is important to minimize transmission between nodes. By designing network hyperparameters, original AlexNet avoided data transmission between GPUs in its 1st, 3rd, 4th convolutional layer. However, intense data transmission in other layers is still inevitable. The author mentioned that training on two GPUs spends only slightly less time than on a single GPU, instead of 2× acceleration. To improve scalability, Krizhevsky (2014) later noticed that:

- Convolutional layers contains 90-95% of computation and 5% of parameters.
- FC layers contains 5-10% of computation and about 95% of parameters.

Hence the author proposed to parallelize different layers in different ways. Specifically, data parallelism is employed for convolutional layers, and model parallelism for FC layers. Experiments show favorable speedup compared with other methods – 3.75× speedup with 4 GPUs and 6.25× speedup with no accuracy loss.

Though partitioning in AlexNet is quite straightforward, when the number of nodes M is large, the designing of partitioning is getting complicated. If considering a DNN model as a graph, to minimize inter-node communication, the goal is to partition the graph with the least inter-part edge weights. This is an NP-hard problem in general cases. It will get more computationally intractable if computation time is taken into account. Mirhoseini et al. (2017) use execution time as cost function, then apply reinforcement learning method to find an optimal partitioning strategy.

Pipelining is an alternative partitioning strategy, which means each node undertakes the computation of one or several layers (Kim et al., 2016). Because forward pass and backpropagation works in opposite directions, when BP tells a node the gradient, the parameters on this node have already been updated several times since the forward pass. In other words, pipelining leads to the same "staleness" problem like it in asynchronous SGD. Since the "staleness" is just equal to the pipeline depth, as is shown in the above section, stable convergence can still be reached.

4. Compression Techniques

The transmission bandwidth is a primary bottleneck of distributed systems. Transmission between different GPUs via PCIe, NVLink, or between different machines via the network is much slower than that via GPU memory. More precisely, a typical deep model has parameters size in the order of 10 to 100 MB (sometimes much larger). With 10 Gbps network bandwidth, sending and receiving a set of parameters needs latency at least 10 ms – even under the assumption that there is only one active node and transmission is perfectly reliable. This latency is at a similar order of magnitude as the time GPU completes an iteration. Therefore if one can compress the parameter data, there will be significant training acceleration. Besides the compression rate, it is also important to reduce the compression cost and keep original accuracy.

4.1 Lower Arithmetic Precision

One technique of compression is to convert the parameters to lower arithmetic precision during network transmission. Because compression and decompression are much faster than transmitting in most of the time, such compression can reduce transmission delay significantly.

Standard implementations of DNN typically use 32-bit floating-point precision representation. However, some works have shown that lower precision is usually sufficient. Courbariaux et al. (2015) experiment single-precision floating-point, half-precision floating-point, fixed-point on MNIST and CIFAR-10, showing that loss rate only increases slightly.

Here are also some tricks that can further improve performance. Gupta et al. (2015) noticed that rounding-to-nearest rounding strategy incurs bias during the conversion, so **stochastic rounding** (round a number to its upper or lower end in lower precision with the possibility proportional to the distance to these ends) is more recommended. Cour-

bariaux et al. (2015) proposed **dynamic fixed-point** to dynamically adjust the scaling factor of fixed-point representation according to overall parameter magnitude. Since fixed-point can express numbers within a certain range more efficiently, dynamic fixed-point can even further compress the precision (12-bit e.g.) with little accuracy loss.

It is surprising that we can even use 1-bit precision for extreme compression, which is called **quantization** with **error feedback** (Seide et al., 2014). A quantization function $\mathcal{Q}$ is chosen to transform gradient to 1-bit value, while the inverse $\mathcal{Q}^{-1}$ is used to restore the gradient from 1-bit value. Each worker i records the error induced by quantization $\Delta_i^{(t-1)}$. Each time t when a worker i computes a gradient $g^{(t)}$, it sends out the quantized value $G_i^{(t)} = \mathcal{Q}(g_i^{(t)} + \Delta^{(t-1)})$, and update $\Delta_i^{(t)} = g_i^{(t)} - \mathcal{Q}^{-1}(G_i^t)$. $\mathcal{Q}$ is usually chosen as `x -> x > 0`, and $\mathcal{Q}^{-1}$ varies according to current parameter mean and variance. Experiments have shown 10× speedup while accuracy varies little after quantization, and convergence is slightly slowed down.

4.2 Sparsification

Sparsification is a technique that when transmitting gradient, instead of sending a full set of gradients, only gradient with a large absolute value is sent. In this way, the gradient is represented by a sparse matrix.

The simplest strategy is to choose a static threshold and filter out all gradient smaller than this threshold (Ström, 1997). This strategy suffers from tuning problems when choosing the threshold. Another simple strategy is to choose a drop ratio then filter out the smallest $R\%$ gradients. To avoid error accumulation, an error feedback strategy similar to that used in quantization is used. Experiments show that 49% speed up on MNIST and 22% on NMT can be reached without damaging the final accuracy (Aji and Heafield, 2017). Sparsification can be also utilized combined with quantization. After sparcify the gradient, quantization is used to further compress the data size (Strom, 2015).

To further increase the compression rate, Lin et al. (2018) proposed a compression technique called **Deep Gradient Compression** based on Nestrov Momentum SGD. Since Nestrov Momentum SGD cannot be directly applied to sparcified situation, the authors proposed accumulating velocity to correct the bias. In addition to this issue, when sparsity is extremely high, for parameters with small gradients, their update suffers from a long delay. Thus when updates occur, they are based on outdated or stale parameters. This will mislead gradient descent to the wrong direction, especially for momentum-based SGD. To alleviate this issue, the authors applied **momentum factor masking** to stop the momentum for delayed gradients, which prevents the stale momentum from leading the parameters to the wrong direction. In this way, 600× compression can be reached without accuracy loss (see figure 6).

5. Conclusion

In this survey, we summarize the primary architectures and techniques used in distributed deep learning. Modern large-scale distributed training systems usually apply a large variety of techniques to reach extreme performance. For example, Jia et al. (2018) combines mixed-

Task	Language Modeling on PTB			Speech Recognition on LibriSpeech			
Training Method	Perplexity	Gradient Size	Compression Ratio	Word Error Rate (WER) test-clean	Word Error Rate (WER) test-other	Gradient Size	Compression Ratio
Baseline	72.30	194.68 MB	1 ×	9.45%	27.07%	488.08 MB	1 ×
Deep Gradient Compression	**72.24 (-0.06)**	**0.42 MB**	**462 ×**	**9.06% (-0.39%)**	**27.04% (-0.03%)**	**0.74 MB**	**608 ×**

Figure 6: Performance of Deep Gradient Compression (Lin et al., 2018)

precision technique and allreduce architecture, trained ResNet-50 with 2048 Tesla P40 GPUs within 15 minutes.

The field of distributed training has seen great progress in recent years. Quite a few frameworks are developed for deploying high-performance distributed training, including DistBelief (Dean et al., 2012), Project Adam (Chilimbi et al., 2014), Poseidon (Zhang et al., 2015a), Horovod (Sergeev and Del Balso, 2018) and so on. These frameworks are usually a convenient encapsulation of former techniques, together with their own distinguished features and optimizations. With an active and innovative research direction, this field is primed to be a core component of future deep learning.

References

Alham Fikri Aji and Kenneth Heafield. Sparse Communication for Distributed Gradient Descent. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445, 2017. doi: 10.18653/v1/D17-1045.

Andrew Gibiansky. Bringing HPC Techniques to Deep Learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning/, 2017.

Karanbir Chahal, Manraj Singh Grover, and Kuntal Dey. A Hitchhiker's Guide On Distributed Training of Deep Neural Networks. *arXiv:1810.11787 [cs, stat]*, October 2018.

Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv:1604.00981 [cs]*, March 2017.

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. page 13, 2014.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv:1412.7024 [cs]*, September 2015.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and Andrew Y Ng. Large Scale Distributed Deep Networks. page 9, 2012.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and*

Pattern Recognition, pages 248–255, Miami, FL, June 2009. IEEE. ISBN 978-1-4244-3992-8. doi: 10.1109/CVPR.2009.5206848.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. page 10, 2015.

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, Xiaowen Chu, Tencent Inc, and Hong Kong Baptist University. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. page 9, 2018.

Peter H. Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv:1611.04581 [cs]*, November 2016.

Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. STRADS: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*, pages 1–16, London, United Kingdom, 2016. ACM Press. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901331.

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

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2012. ISSN 00010782. doi: 10.1145/3065386.

Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter Server for Distributed Machine Learning. page 10, 2013.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed training. page 13, 2018.

Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. *arXiv:1706.04972 [cs]*, June 2017.

Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. page 16, 2014.

Pitch Patarasuk and Xin Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, CA, USA, 2007. IEEE. ISBN 978-1-4244-0909-9. doi: 10.1109/IPDPS.2007.370405.

Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, February 2009. ISSN 07437315. doi: 10.1016/j.jpdc.2008.09.002.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int J Comput Vis*, 115(3):211–252, December 2015. ISSN 0920-5691, 1573-1405. doi: 10.1007/s11263-015-0816-y.

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. page 5, 2014.

Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv:1802.05799 [cs, stat]*, February 2018.

Nikko Ström. Sparse Connection and Pruning in Large Dynamic Artificial Neural Networks. page 4, 1997.

Nikko Strom. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. page 5, 2015.

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005. ISSN 1094-3420, 1741-2846. doi: 10.1177/1094342005051521.

Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, pages 381–394, Kohala Coast, Hawaii, 2015. ACM Press. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806778.

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. *arXiv:1512.06216 [cs]*, December 2015a.

Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Effcient Communication Architecture for Distributed Deep Learning on GPU Clusters. page 15, 2017.

Ruiliang Zhang and James T Kwok. Asynchronous Distributed ADMM for Consensus Optimization. page 9, 2014.

Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with Elastic Averaging SGD. page 9, 2015b.

Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous Stochastic Gradient Descent with Delay Compensation. *arXiv:1609.08326 [cs]*, August 2019.