Practical Lessons of Distributed Deep Learning

Jun Yang [1] **Yan Chen** [1] **Siyu Wang** [1] **Lanbo Li** [1] **Chen Meng** [1] **Minghui Qiu** [1] **Wei Chu** [1]

Abstract

With the advent of big data and big model, there are increasing needs on training deep learning model in distributed mode. Although the open source deep learning software such as TensorFlow and MXNet do support training deep learning model in parallel, it is still a challenging task for data scientists to implement scalable and high performance distributed deep learning algorithms. In this paper, we share several practical lessons on optimizing distributed deep learning training process, including optimization strategies for typical model architecture such as DNN and CNN. For DNN, we exploit its computation-to-communication ratio to reduce the communication overhead. For CNN, we find hybrid-parallelism an effective way to squeeze the potential of strong-scaling. Experiments in off-the-shelf deep learning software show that, with our optimization strategies we are able to have 10x speed-up on AlexNet against the standard distributed implementation.

1. Introduction

In recent years, there is a trend towards solving problems in data-driven approach, such as in online advertising (Richardson et al., 2007), search engine (Yin et al., 2016), e-commerce recommendation (Linden et al., 2003), etc. Accumulated data play a more and more important role in those areas since they could help predict future pattern and discover implicit regulations. Also with the advance of communication technology and hardware industry, more and more devices are now connecting into Internet(Perera et al., 2015), which in turn generate huge amount of data(Tsai et al., 2014).

Along with those trends, in machine learning community, there is a track attracting more and more attention—– modeling problems with non-linear neural network, especially with multiple layers of neural networks. We call those models deep learning(LeCun et al., 2015). Deep learning actually is not a completely new stuff. In history, there are already several waves of using neural network for modeling task(Rosenblatt, 1958)(Rumelhart et al., 1988). However, at that time, there were some challenges limiting its spread(Bengio et al., 2015). Recently, with the advances of optimization techniques(Dahl et al., 2013)(Ioffe & Szegedy, 2015), the dedicated computation power brought by General Purpose Graphics Processing Unit (GPGPU)(Chetlur et al., 2014), and the breakthrough of new models(Krizhevsky et al., 2012)(He et al., 2016), deep learning brings significant improvement over traditional shallow models in several fields such as computer vision(Krizhevsky et al., 2012), speech recognition(Hinton et al., 2012), and NLP(Wu et al., 2016).

With the mixture of popularity of data-driven approach, IoTs generating more data, advances of modeling techniques and easy-to-get computation power via cheaper GPGPUs, there are quite a lot of requirements of training deep learning with big models and big data. To efficiently support these training tasks, distributed training system comes into our view due to that: i). Some models are too big to fit in a single GPGPU device(Shazeer et al., 2017); ii). Some models are so computation-intensive and take quite a long time to train in a single GPGPU device(Wu et al., 2016). Distributed training is a suitable way for solving the above challenges.

Recently, there are some open source deep learning software which already provide support for distributed training, such as MXNet(Chen et al., 2015) and TensorFlow(Abadi et al., 2016). However, these software just provide the primitive support for distributed training. Based on those distributed primitives, it is still not easy to write efficient distributed algorithm implementation, just as mentioned in (Sutter, 2005). To be even worse, some naive distributed implementation may be slower than the solo implementation.

In this paper, several practical lessons of optimizing distributed deep learning are discussed. The lessons shared here can be employed on two popular network architectures–Convolutional Neural Network (CNN) and

*Equal contribution [1]Alibaba. Correspondence to: Jun Yang <muzhuo.yj@alibaba-inc.com>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

Deep Neural Network (DNN) [1]. Due to the underlying differences of those network architectures, their optimization strategy are also somewhat different. For DNN, we exploit its computation-to-communication ratio to reduce the communication overhead. For CNN, we find hybrid-parallelism an effective way to squeeze the potential of strong-scaling. Experiments in off-the-shelf deep learning software show that, with our optimization strategies we are able to have 10x speed-up on AlexNet against the standard distributed implementation.

2. Optimization Strategies

2.1. Late-multiply

Late-multiply is an optimization strategy suitable for DNN. DNN has densely-connected structure, so small number of DNN hidden units could result in big number of synapses. For distributed deep learning, this kind of structure will bring significant communication overhead. For example, for a fully-connected layer with N bottom hidden neurons and K top hidden ones, it has $K * N$ connecting synapses. Let N and K both be 10000, the total synapse number becomes 100 million, usually 4 bytes floating-point representation is used for storing one synapse, so $400MB$ data need to be communicated through network for this layer. Mainstream network device usually has 10Gbps bandwidth, in theory it takes at least $400ms$ to transmit synapses for this single layer, which is a non-negligible overhead.

Late-multiply aims at transmitting less data for a fully-connected layer with extra computation cost. Before discussing the core idea, let's review the backward propagation of fully-connected layer firstly.

Backward propagation of a fully-connected layer For a specific neuron y_j of a fully-connected layer, the feed forward process for calculating it is as following:

$$y_j = b_j + \sum_i x_i w_{ij}. \tag{1}$$

For backward propagation, the key point is to compute error derivatives with regard to all the weights. Let's take weight w_{ij} as an example:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial y_j}{\partial w_{ij}} \frac{\partial E}{\partial y_j} = x_i \frac{\partial E}{\partial y_j}, \tag{2}$$

where E represents error derivatives passed from upper layer.

Trade-off between computation and communication. For training neural networks, Stochastic Gradient Descent (SGD) algorithm is used in practice and the training data is split into multiple mini-batches. For each mini-batch with batch size M, number of bottom layer hidden units is N, number of top layer hidden units is K, bottom layer is a $M \times N$ matrix $\mathbf{X}$, error derivatives of the top layer is a $M \times K$ matrix $\mathbf{E}$, weight is a $K \times N$ matrix $\mathbf{W}$. In the back propagation stage, we update $\mathbf{W}$ using $\mathbf{W} = \mathbf{E}^T\mathbf{X}$ which is a matrix multiplication form of Eq. 2.

Algorithm 1 is a distributed version of backward propagation, where n is number of workers and i is worker index. Worker 0 needs to gather $\mathbf{W}$ from the other workers, sums them up and calculates the average $\mathbf{W}_{avg}$, then **Broadcast** $\mathbf{W}_{avg}$ back to other workers at last.

Algorithm 1 Distributed version of backward propagation.

Worker 0	**Other workers**
$\mathbf{W}_0 \leftarrow \mathbf{E}_0^T\mathbf{X}_0$	$\mathbf{W}_i \leftarrow \mathbf{E}_i^T\mathbf{X}_i$
Reduce $\mathbf{W}_i$ to worker 0	**Reduce** $\mathbf{W}_i$ to worker 0
$\mathbf{W}_{avg} \leftarrow \frac{\sum_i \mathbf{W}_i}{n}$	do nothing
Broadcast $\mathbf{W}_{avg}$ to other workers	**Broadcast** $\mathbf{W}_{avg}$ to other workers

During this process, the communication cost is one **Reduce** operation of matrix $\mathbf{W}$ plus one **Broadcast** operation. Usually N and K are large. For example, the last two fully-connected layers in Alexnet(Krizhevsky et al., 2012) both have 1024 neurons. The idea of trading computation for communication is leveraged here, as shown in Algorithm 2. The calculation of $\mathbf{W} = \mathbf{E}^T\mathbf{X}$ is postponed in each worker. Instead, matrices $\mathbf{X}$ and $\mathbf{E}$ of all workers are distributed to each other through **Allgather** operation. Then each worker executes $\mathbf{E}^T\mathbf{X}$ n times, sums up the result and calculates average.

Algorithm 2 Late-multiply on worker i

Allgather $\mathbf{X}_i, \mathbf{E}_i$

$W_{avg} \leftarrow \frac{\sum_i \mathbf{E}_i^T\mathbf{X}_i}{n}$

Complexity analysis. It is clear that Algorithm 2 needs extra $n-1$ times of multiplication of $\mathbf{E}^T\mathbf{X}$. However, the communication cost becomes one **Allgather** operation of matrix $\mathbf{X}$ and $\mathbf{E}$. As $\mathbf{X}$ is in the shape of $M \times N$ and $\mathbf{E}$ is in $M \times K$, where M, the mini-batch size, is usually smaller than the bottom layer neuron number N and the top layer neuron number K. Also with the high performance of matrix multiplication in GPGPU, the extra computation cost is trivial.

2.2. Hybrid-parallelism

Another optimization strategy introduced is hybrid-parallelism, which is suitable for models consisting of both DNN and CNN layers. CNN has nice weight-sharing network structure, so usually CNN layer has high computation-to-communication ratio in distributed deep

[1] In this work, we simply use DNN to represent Fully Connected Neural Networks.

learning scenario. However, usually a deep learning model architecture consists of both CNN and DNN, e.g. AlexNet has 5 CNN layers followed by 3 DNN layers. And as presented in previous section, DNN's densely-connected structure is unfriendly for distributed execution. To fully release the parallel potential of CNN layers, hybrid-parallelism(Krizhevsky, 2014) is a suitable solution. Let's take AlexNet to illustrate the core idea in detail. AlexNet has 60 million parameters with just 650,000 neurons, consisting of five convolution layers, some of which are followed by max-pooling layers, and three fully-connected layers. Fully-connected layers occupy 96% of parameters, with just 6% of training time. Naive data-parallelism of AlexNet will meet scalability bottleneck very quickly.

With the hybrid-parallelism mode, the convolution layers are distributed in data-parallelism mode, while the fully-connection layers are placed into a single computation device. So the total strategy is a mixture of data-parallelism and model-parallelism. This strategy reduces exchanged data significantly compared against naive data-parallelism.

Complexity analysis. Let $|W|$ denote the weight number, $|N|$ denote the neuron number. In AlexNet, $\frac{|W|}{|N|} \simeq 90$. With naive data-parallelism, the data exchange complexity is $O(|W|)$. With hybrid-parallelism, it is $O(|N|)$. This illustrates hybrid-parallelism advantages over data-parallelism for AlexNet.

To ensure the correctness of hybrid-parallelism, several tricks need to be taken into consideration. In Figure 1, an abstract model is split into upper (part 2) and bottom parts (part 1). With hybrid-parallelism, data parallelism is used for bottom parts, and model parallelism is used for coordinating upper and bottom parts. Tricks are expressed based on this figure.

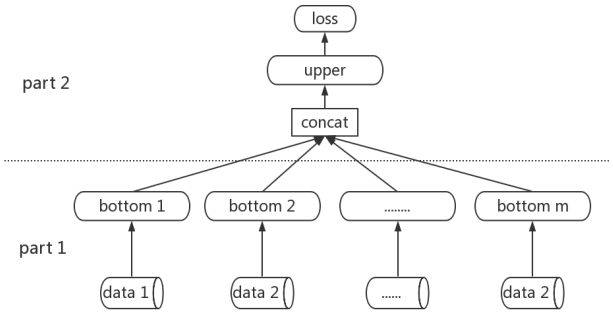

Figure 1. The hybrid-parallelism mode

- For distributed execution of deep learning model, gradient aggregation is necessary for coordinating the behaviors of different computation devices. In standard data-parallelism, gradient averaging is needed during aggregation phase, while in hybrid-parallelism, the aggregation phase (for bottom parts) just need to sum up gradients rather than averaging them.

- To avoid overfitting, weight decay is a popular mechanism which usually acts as an additional element of the loss function, for hybrid-parallelism, it is necessary to scale the regularization coefficient of bottom parts with an additional factor $\frac{1}{m}$. Here m is the replica count of bottom parts.

3. Benchmark Results

The optimization strategies mentioned in previous sections are experimented in several off-the-shelf deep learning software and convergence speed-up is observed and reported here. We did the benchmark studies on an in-house GPU cluster consisting of over 30 NVIDIA GPU K40M/M40 cards with 10Gbps Ethernet or 56Gbps InfiniBand connection support.

3.1. Late-multiply

For late-multiply strategy, we compared the time cost of communication directly. We run Alexnet(Krizhevsky et al., 2012) with $batchsize = 256$ both on 10Gbps Ethernet and 56Gbps InfiniBand. **Table 1** illustrates the experiment results. It's apparent that late-multiply reduces communication cost significantly It is a practical trick for neural networks with huge fully-connected layer and small batch size. From the result it is shown that late-multiply has better improvement on Ethernet over InfiniBand. It is due to that InfiniBand has RDMA support which eliminates additional data movements between GPU memory and main memory through PCIe. Thus the reduction of exchanged data size could have bigger impact upon Ethernet environment.

Table 1. Comparison of communication time cost. T refers to time cost without late-multiply and T_{opt} refers to that with late-multiply optimization.

	10 Gbps Ethernet		
	1 GPU	2 GPUs	4 GPUs
T (ms)	0	894	1287
T_{opt} (ms)	0	164	399
	56 Gbps InfiniBand		
	1 GPU	2 GPUs	4 GPUs
T (ms)	0	90	185
T_{opt} (ms)	0	51	136

3.2. Hybrid-parallelism

For hybrid-parallelism, we run experiments on AlexNet with the following considerations:

- We benchmarked on both K40M and M40 GPUs [2]. Also FFT acceleration switch was turned on. With these settings we made the distributed optimization a more challenging task.
- We chose single GPU implementation and naive data-parallel implementation as two baselines to highlight the improvement brought by hybrid-parallelism.
- We benchmarked with two kinds of input data: fake data and real training data. Using fake data helped eliminate the impact of data IO performance. With real training data, we could know the realistic performance.

M40 benchmark results are shown in **Table 2**, for data-parallelism and hybrid-parallelism correspondingly. And K40 benchmark results are provided in **Table 3**.

Table 2. M40 results. T_{fake} and T_{real} refer to the iteration time with fake and real data respectively.

	Data-parallelism			
	1 GPU	2 GPUs	4 GPUs	8 GPUs
Data size(MiB)	0	930	1861	3722
T_{fake} (s)	2.297	2.906	3.672	5.466
Speed-up	1X	0.79X	0.63X	0.42X
T_{real} (s)	2.615	2.861	3.574	5.882
Speed-up	1X	0.91X	0.73X	0.44X
	Hybrid-parallelism			
	1 GPU	2 GPUs	4 GPUs	8 GPUs
Data size(MiB)	0	54	89	161
T_{fake} (s)	2.297	1.238	0.662	0.443
Speed-up	1X	1.86X	3.47X	5.19X
T_{real} (s)	2.615	1.500	0.831	0.552
Speed-up	1X	1.74X	3.15X	4.74X

From the benchmark results, it is clear that hybrid-parallelism gains significant speed-up over naive data-parallelism implementation. For M40, the gap is at least **1.9X**, for K40, the gap is at least **2.17X**. The best improvement is **10X**, for M40 in 8 GPUs setting. It is due to that AlexNet contains over 60 million parameter, in naive data-parallelism implementation, for each iteration, there are $2 * 232$ MiB model weights/gradients to be spread over network. For 2 GPUs, the communication data size is 930 MiB (2 gradient pull + 2 weight push). For 4 GPUs, the communication data size just grow linearly. This significantly impedes scalability. With our hybrid-parallelism implementation, since the fully connected layers are placed into a single device. During the entire training process, the fully connected layers' weights don't need to be spread over the network, thus significantly reduces the network communication overhead, which in turn improves scalability.

[2] theoretical speaking, M40 has higher TFlops than that of K40m

Table 3. K40 results. T_{fake} and T_{real} refer to the iteration time with fake and real data respectively.

	Data-parallelism			
	1 GPU	**2 GPUs**	**4 GPUs**	**8 GPUs**
T_{fake} (s)	2.95	3.517	5.737	7.169
Speed-up	1X	0.84X	0.51X	0.41X
T_{real} (s)	3.487	8.956	9.906	7.407
Speed-up	1X	0.39X	0.35X	0.47X
	Hybrid-parallelism			
	1 GPU	**2 GPUs**	**4 GPUs**	**8 GPUs**
T_{fake} (s)	2.95	1.615	0.957	0.681
Speed-up	1X	1.83X	3.08X	4.33X
T_{real} (s)	3.487	2.304	1.329	0.776
Speed-up	1X	1.51X	2.62X	4.49X

4. Conclusions

In this paper, we introduce several practical optimization methods regarding to distributed deep learning. Late-multiply is suited to DNN to reduce communication overhead. For models mixed with CNN and DNN, hybrid-parallelism could fully exploit models' computation-to-communication ratio. We have also experimented those optimization strategy with existing deep learning software and provide benchmark results demonstrating their superiority over standard distributed implementation.

As future work, we would like to abstract distributed deep learning as a graph optimization problem since the forward and backward execution phase construct a typical computation graph. We can formulate it in the following way:

Problem formulation Given a certain amount of computation devices $\mathbf{R}$ and a specific model architecture $\mathbf{\Omega}$, we need to find the best placement strategy $\mathbf{S}$, with which the training graph could be split and placed into those computation devices with the shortest training time.

A few notations are defined as follows:

- $\mathbf{R}$ consists of N computation devices, notated as $r_i \in R; i \in [1, N]$;
- $\mathbf{\Omega}$ can be viewed as a composition of K sub-graphs or sub-models, notated as $\Omega_j \in \Omega; j \in [1, K]$;
- Placement strategy $\mathbf{S}$ can be viewed as paring computation devices with sub-graphs, notated as

$\{r_i, \vec{\Omega}_i\}; i \in [1, N], \vec{\Omega}_i \ni \{\Omega_j, \Omega_{j+1}, ...\}$ with $\Omega_j \in \Omega$. Note, Ω_j may be placed into multiple devices simultaneously.

It can be shown that this placement problem is an Integer Linear Programming[3] problem which is NP-hard. It's non-trivial to find the optimal placement strategy. In reality, it is not always expected to find the optimal strategy, often sub-optimal one is good enough for speeding up the training process. Usually, heuristic methods are used for looking for a good or sub-optimal placement strategy. Hybrid-parallelism actually can be viewed as a specific case of graph optimization problem. In the future, more effort will be put on this general optimization problem.

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

Bengio, Yoshua, Goodfellow, Ian J, and Courville, Aaron. Deep learning. *Nature*, 521:436–444, 2015.

Chen, Tianqi, Li, Mu, Li, Yutian, Lin, Min, Wang, Naiyan, Wang, Minjie, Xiao, Tianjun, Xu, Bing, Zhang, Chiyuan, and Zhang, Zheng. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chetlur, Sharan, Woolley, Cliff, Vandermersch, Philippe, Cohen, Jonathan, Tran, John, Catanzaro, Bryan, and Shelhamer, Evan. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Dahl, George E, Sainath, Tara N, and Hinton, Geoffrey E. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8609–8613. IEEE, 2013.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E, Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29 (6):82–97, 2012.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. Deep learning. *Nature*, 521(7553):436–444, 2015.

Linden, Greg, Smith, Brent, and York, Jeremy. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

Perera, Charith, Liu, Chi Harold, and Jayawardena, Srimal. The emerging internet of things marketplace from an industrial perspective: A survey. *IEEE Transactions on Emerging Topics in Computing*, 3(4):585–598, 2015.

Richardson, Matthew, Dominowska, Ewa, and Ragno, Robert. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pp. 521–530. ACM, 2007.

Rosenblatt, Frank. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Shazeer, Noam, Mirhoseini, Azalia, Maziarz, Krzysztof, Davis, Andy, Le, Quoc, Hinton, Geoffrey, and Dean, Jeff. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Sutter, Herb. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30 (3):202–210, 2005.

Tsai, Chun-Wei, Lai, Chin-Feng, Chiang, Ming-Chao, Yang, Laurence T, et al. Data mining for internet of things: A survey. *IEEE Communications Surveys and Tutorials*, 16(1):77–97, 2014.

[3] https://en.wikipedia.org/wiki/Integer_programming

Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V, Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yin, Dawei, Hu, Yuening, Tang, Jiliang, Daly, Tim, Zhou, Mianwei, Ouyang, Hua, Chen, Jianhui, Kang, Changsung, Deng, Hongbo, Nobata, Chikashi, et al. Ranking relevance in yahoo search. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 323–332. ACM, 2016.