A Performance Evaluation of Distributed TensorFlow

Tianlun WANG[1,2,a)] Yusuke TANIMURA[2,1] Hirotaka OGAWA[2]
Hidemoto NAKADA[2,1,b)]

Abstract:
TensorFlow is a deep learning framework which is developed by Google. The computations in TensorFlow are implemented and expressed as data flow graphs of multidimensional array data which is referred to as "Tensor." We know that, with TensorFlow we can implement parallel execution in a multi-GPU configuration and distributed execution in a multi-node configuration, but it is not clear how effective it is in the real environment. In this paper, we measured these performances for several mini batch sizes and network settings. From the experimental results, we confirmed that we can accelerate the execution in all the environment we have tested. We also found that the mini batch size has a big influence in the distributed environment of 1Gbps network. However, in the environment of 10Gbps network or the intra-node multipl-GPU configuration, we could achieve the linear speed up regardless of mini batch size.

Keywords: Distributed TensorFlow, Neural Networks, Recognition Benchmark

1. Introduction

As the remarkable success of the deep learning neural network technology, the applications of the technology widely spreading in all the conceivable areas. To support the application, there are so many deep learning frameworks are proposed, including the TensorFlow. One of the challenges of the deep learning technology is the fact that it is really computationally expensive. To mitigate the problem, parallelization is essentially required, and most of the deep learning frameworks supports parallelization method in some form or the other. TensorFlow, the deep learning framework from Google, also provides parallelization methods, as a natural extension of the data flow execution. It supports multi-GPU execution and multi-nodes execution. We evaluate the parallelization performance of them, in several mini batch size settings and two network configurations.

In section 2, we introduce the TensorFlow and its multi-GPU and multi-node implementation. In section 3, we provide experiment setup and the results. In section 4, we summarize our research and discuss future direction of the research.

2. Background

2.1 Introduction to TensorFlow

TensorFlow [5][4] is an open-source software library which is developed by Google, and it can do machine learning across a range of tasks. Google use TensorFlow to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning which humans use.

TensorFlow is developed as an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. A computation with TensorFlow can be executed with little or no change on a wide variety of heterogeneous systems, such as the mobile computing platforms including iOS and Android, Windows, OSX and Linux, ranging from mobile devices like smartphones up to large-scale distributed systems that have hundreds of machines or thousands of computational devices such as GPU cards. TensorFlow can run on multiple CPUs and GPUs, the system has great flexibility and can be used for expressing a wide variety of algorithms, including training and inference algorithms for making deep neural network models. And also it has been used for building machine learning systems into production across many different fields, such as computer vision, robotics, speech recognition, information retrieval, natural language processing and geographic information extraction.

TensorFlow computations are expressed as data flow graphs. The name TensorFlow is derived from the operations which such neural networks perform on multi-dimensional data arrays. These multi-dimensional arrays are referred to as "tensors". The data flow graphs can offer a large degree of flexibility in the structure and placement of operations. The nodes in the data flow graphs represent

1 筑波大学/ University of Tsukuba
2 産業技術総合研究所/ National Institute of Advanced Industrial Science and Technology
a) ou-tenrin@aist.go.jp
b) hide-nakada@aist.go.jp

ⓒ 2017 Information Processing Society of Japan

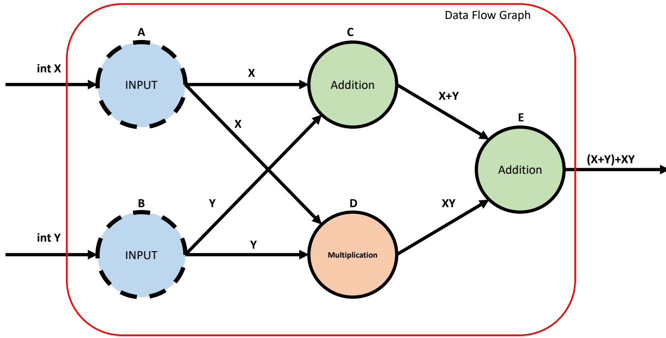

Fig. 1 TensorFlow computation in a single data flow graph

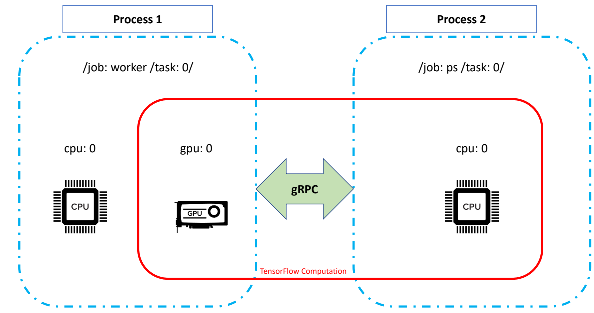

Fig. 3 A simple implementation of Distributed TensorFlow

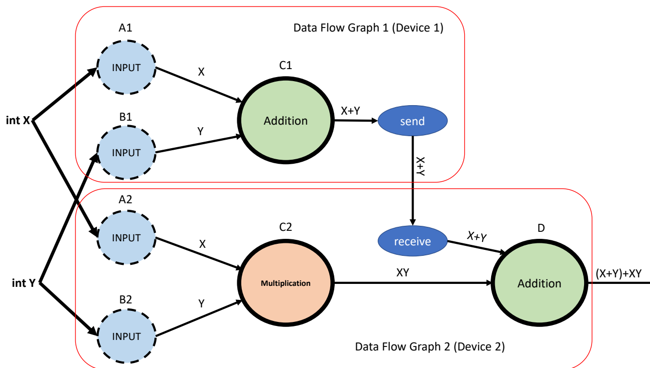

Fig. 2 TensorFlow computation between different devices

for some mathematical operations, such as addition, subtraction, and multiplication, etc. Figure 1 shows the TensorFlow computation in a single data flow graph. The edges in data flow graphs represent the multidimensional data arrays communicated between them. This architecture is very flexibile; it allows us to deploy computation to one or more CPUs or GPUs in many different devices with a single API. The TensorFlow computation using heterogenous devices is shown in Figure 2.

2.2 Introduction to Distributed TensorFlow

As mentioned above, the computation of TensorFlow are expressed as data flow graphs. As a result of the data flow graphs offer a large degree of flexibility in the structure and placement of operations, it allows for parallelizing computation across multiple workers. And it is often beneficial in the training of neural networks, given a large number of training data have to be processed. Besides, such parallelization is useful when the size of the model becomes extremely huge. Because some computations in the TensorFlow can be distributed, comparing to single-process TensorFlow, the distributed TensorFlow will be a better way when training large model.

Figure 3 shows a simple implementation of Distributed TensorFlow. In this example, we have multiple machines and are trying to put some variables on the CPU device of a different process. Distributed TensorFlow treats the remote devices in the same way as the local ones, we only need to add a little bit of information to these device names, and the runtime will put the variables in a different process, splitting up the graph between the devices in the different processes and adding the necessary communication. In this case, it will be using gRPC to transfer tensors between the process instead of DMA from the GPU device.

The Distributed TensorFlow can give us the flexibility to scale up to hundreds of GPUs; we can train outrageously large models with tens of billions of parameters, and customize every last detail of the execution or defining the whole training process in just a few lines of code. There are some core concepts in the distributed TensorFlow: Distributed device placement, In-graph replication, Between-graph replication, Sessions and Servers.

2.2.1 Distributed device placement

Before releasing TensorFlow, Google developed a system which was called DistBelief[6]. And in DistBelief, there are two distinct kinds of processes. It has parameter servers and worker replicas. Parameter Servers (PS) [7] are responsible for holding onto all of the model states. The worker replicas would do all the intensive part of the computation. They would do the input processing and calculate the loss for your network. And they do the backpropagation to calculate those gradients. In TensorFlow, Google mimics this architecture by designating some of the processes to be what we call PS tasks.

Figure 4 shows the distributed device placement in TensorFlow. We put the variable nodes and the operations that update them on those tasks. Then we put the rest of the graph, the nodes that do the pre-processing, the forward, and the backpropagation on the worker task, which tend to be more powerful and have multi-course CPUs and GPUs. Be different from DistBelief, the PS and the worker tasks run the same code in TensorFlow. They are just servers that you can send little bits of TensorFlow graph to, and it will react to those bits of the graph and execute them very quickly. This will give us more flexibility, we can use a GPU to accelerate some of the parameter update computations via a PS task, and we could have the workers store some state locally to avoid network traffic, to cache some of the reads. If the network was fast enough to deal with communication between the workers, we could even cut out the PS altogether and use direct connections between the workers.

ⓒ 2017 Information Processing Society of Japan

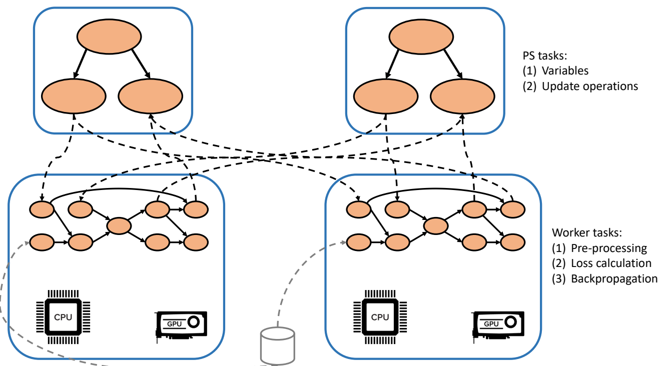

Fig. 4 The distributed device placement in TensorFlow

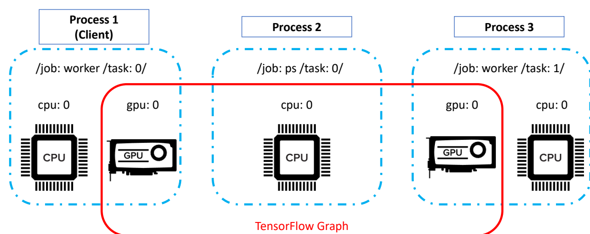

Fig. 5 In-graph replication in Distributed TensorFlow

2.2.2 In-graph replication

The replication works pretty well for distributed training, especially when a single model will fit in a single machine. Therefore, it has a strategy which is called In-graph replication in Distributed TensorFlow as shown in Figure 5.

By using this strategy, a single graph will be created on the distributed master, and it will include all of the replicas residing on its worker devices, and each task works on a different subset of the data. Firstly, we start by putting the variables on a PS task, and they would be in a central location where they can be accessed by all of the workers. Then split up a batch of input data into equal-sized chunks, loop over the worker tasks, and put a subgraph on each worker to compute a partial result. Finally, we combine all of the partial results into a single loss value that we optimize by using a standard TensorFlow optimizer.

TensorFlow will split up the graph across the workers when we tell it to compute the loss, and it will run across these worker tasks and the PS all in parallel. For small systems, the In-graph replication is an easy way to achieve, and it will not make a big modification to our existing programs.

2.2.3 Between-graph replication

The In-graph replication should be a good choice if we want to replicate across all the GPUs in a single machine. But when we are training a large model, the graph will get very big if we materialize all the replicas in it. And the client gets bogged down trying to coordinate the computation and to build this whole graph. To solve this problem, we should use another strategy which is called Between-graph replication as shown in Figure 6.

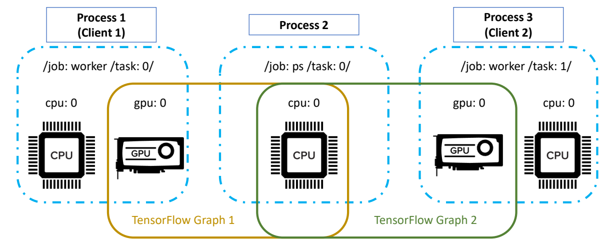

Fig. 6 Between-graph replication in Distributed TensorFlow

Instead of running one all-powerful client program that knows about all of the worker replicas, this strategy runs a smaller client program on each task. The client program just builds up the graph for a single replica of the model. And this client program is essentially doing the same thing with one key difference in the device placement. It takes kind of the non-parameter part of the graph, and it puts it on the local devices or the devices that are local to that worker replica. Each program is running its smaller graph independently when we run it, and they get mapped to different subsets of the devices that intersect on the PS task in the middle. And each replica places its variables on the same PS task. When we are running in distributed mode, any two clients that create a variable with the same name on the same device will share the same backing storage for that variable by default. Exactly this is what we want to see when we are doing replicated training.

2.2.4 Sessions and Servers

When we create a TensorFlow session, this session will only know about the devices on the local machine. Therefore, we need to create a thing called TensorFlow server on each machine. And we can configure those servers in a cluster; then they can communicate over the network as shown in Figure 7.

One TensorFlow cluster has lots of tasks that participate in the distributed execution of a TensorFlow graph, and each task is associated with a TensorFlow server, which includes a master server that can be used to create sessions. Also, a cluster can be divided into many jobs, and there will have one or more tasks on every job. At first, we need to provide a cluster specification to tell TensorFlow about the machines that we want to run on. A cluster specification is a directory of the type of the jobs: *ps* (parameter server) or *worker*. And after the name of the job, we also need a list of one or more network addresses that correspond to the task in each job. It is not necessary to type in all of these addresses by hand if we use some cluster manager software. (such as Borg, Kubernetes, and Mesos) The cluster manager will run an instance of our program on each machine in the cluster, giving it the same cluster specification, then it will start a TensorFlow server in each program, and pass it a particular

ⓒ 2017 Information Processing Society of Japan

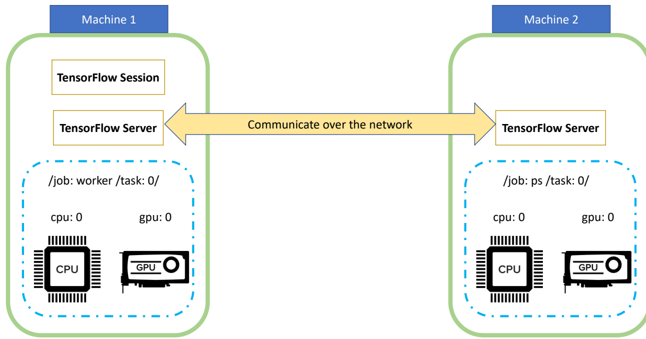

Fig. 7 Sessions and Servers in Distributed TensorFlow

Table 1 Cluster Setup

# nodes	17
CPU	Intel(R) Xeon(R) W5590 x 2
GPGPU	NVIDIA GeForce GTX 980 x 1 1126 MHz, 4GB
Memory/node	48GB
Operating System	Ubuntu 16.04
Network Interface	10G / 1G

Table 2 Multiple GPU node

CPU	Intel(R) Xeon(R) E5-2630
GPGPU	NVIDIA GeForce GTX TITAN X x 4 (Maxwell) 1000 MHz, 12GB
Memory	64GB
Operating System	Ubuntu 16.04
Network Interface	1G

job name and task index that matches the address of the local machine in that cluster. Finally, we create our session, and the session can run code on any device in the cluster. It will specify the local server's address as the target, which is what enables it to connect through the server to any of the machines mentioned in cluster specification.

The worker starts a training loop that just iterated over its partition of the data, running a training operation over and over again, it performs the heavy computation. And the behavior of the ps is not implemented at the low level in these servers or the execution engine, but instead it is built out of TensorFlow programming primitives, as these little bits of data flow graph that the worker ships to a server to say, managing some parameters for us and updating them in this way.

3. Experiments

We have performed two experiments; multi-node experiment and multi-GPU experiment. The former uses 'Between-graph replication', whereas the latter uses the 'In-graph replication'

3.1 Experimental Enviroment

We used a cluster shown in Table 1 for multi-node experiments, and other node shown in Table 2 for multi-GPU experiment.

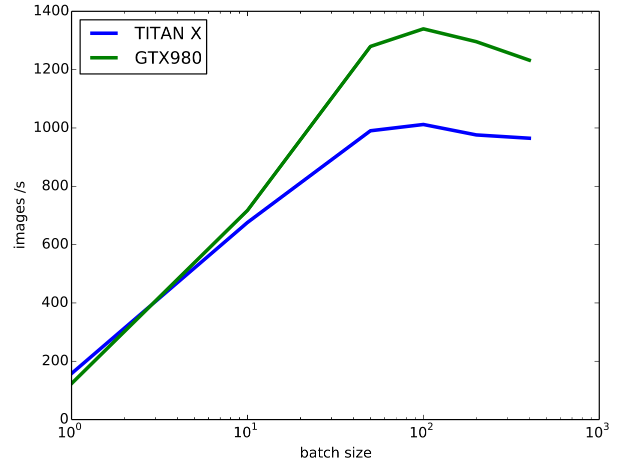

Fig. 8 The performance when implementing in 1GPU

3.2 Evaluation procedure and Dataset

We used the Cifar10[3] dataset to implement the evaluation. Cifar10 is one of the object recognition benchmark dataset, and it is widely used because the data size is small. The Cifar10 dataset consists of 50000 training images and 10000 test images in 10 classes, and each image is a 32x32 color image.

The network used for evaluation is as same as the one described by the tutorial of TensorFlow. It is a standard network that two full bonding layers were provided behind the two layers composed of convolution and pooling.

3.3 The baseline performance with 1GPU

As the baseline performance we measured the performance with 1GPU, on one of the nodes of the cluster shown in Table 1 and the node shown in Table 2. In this measurement, we changed the mini batch size to 1, 10, 50, 100, 200, 400; and recorded how many images will be learned in one second. The result of this measurement is shown in Figure 8. Note that the X-axis is represented in logscale.

We can observe that the performance is improved as mini batch size increases. This is a well-known phenomenon; because the reuse ratio of data which has been loaded into memory on GPUs is improved. In our experiment, we got the maximum of the learning speed when mini batch size is 100; the performance degrade slightly with batch size 200 and 400. We are now investigating the result. We also realized TITAN X is slower than GTX 980, the reason is that the clock of TITAN X is somewhat lower.

3.4 The performance when implementing in multiple-GPUs

We used the nodes shown in Table 2 to implement the experiment of multiple GPUs. The mini batch sizes we used are 100, 200 and 400.

Figure 9 shows the result of the experiment. In all the cases, the performance has greatly improved along with the increasing of the number of GPUs. We got the fastest speed

ⓒ 2017 Information Processing Society of Japan

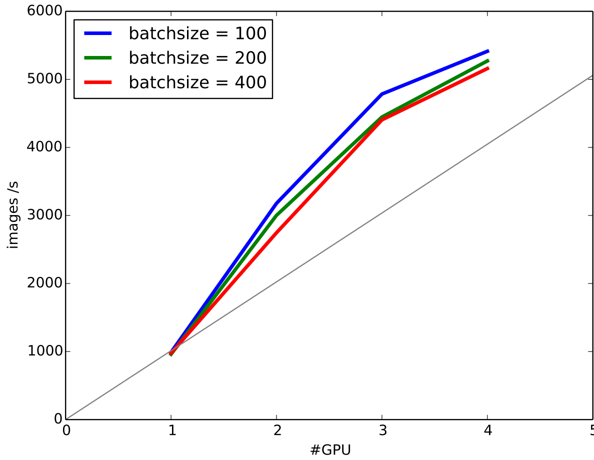

Fig. 9 The performance when implementing in multiple-GPUs

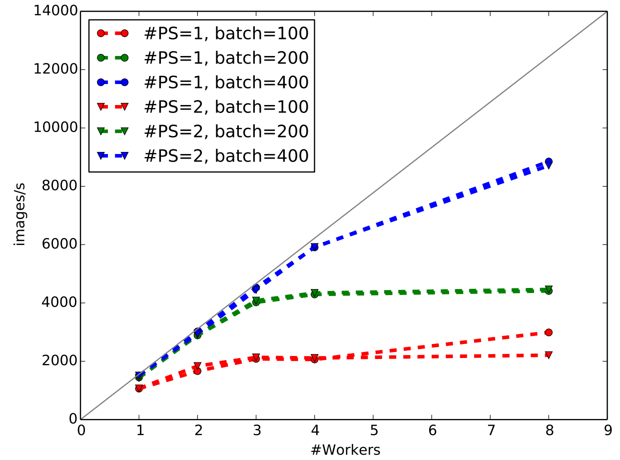

Fig. 10 The result of implementing in 1Gbps network

with the mini batch size 100. This is consistent with the results with single GPU execution shown in Figure 8.

The gray thin line in the figure shows the ideal speed up with parallelization, based on the single GPU performance. As can be seen, in all the cases, the performance is higher than in the ideal case, showing so called "super linear" performance improvement. This is quite unusual and we are investigating the reason.

3.5 The performance when implementing in multiple nodes

For the implementation of multiple-Nodes, we used the environment shown in Table 1. We changed the number of parameter servers to 1,2, and the number of workers was changed to 1,2,3,4,8; then measured the respective processing performance (number of processed sheets/second). In this case, the mini batch size was set to 100, 200, 400, and the network was set to 1 Gbps and 10 Gbps.

The result of experiments with 1 Gbps network is shown in Figure 10. The mini batch size of red line is 100, the green line is 200, and the blue line is 400. In either case, the performance improves as the number of workers increases; but it saturated with about 3 worker when mini batch size was 100, and saturated with about 4 workers when mini batch size was 200, while with mini batch size 400 the performance scales up to 8 workers. This is because the frequency of communication decreases with larger batch size.

The result of experiments with 10 Gbps network is shown in Figure 11. We can observe that the performance is improved by using the 10G network, which is quite reasonable. The performance with 10 Gbps network with different mini batch size is not as same as the result with 1 Gbps network; we cannot see a regular pattern.

The effect of having extra parameter servers are not obvious in our experiments. We suppose that the reason is the load (parameter) distribution imbalance among the parameter servers.

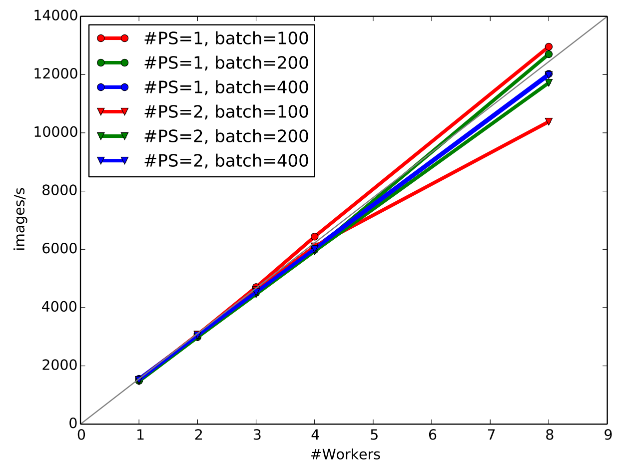

Fig. 11 The result of implementing in 10Gbps network

4. Conclusion

We evaluated the distributed TensorFlow in multi-GPU and multi-node settings. We employed Cifar10 as the dataset. The evaluation results showed that we could successfully speed up the execution in all the settings, while the speed up ratio depends the settings. The most influential parameter was the network link speed. With fast network link (10Gbps) we could gain linear speed regardless of other settings, whereas with slow network link (1Gbps) the mini-batch size dominates the speed up ratio.

Our future work includes the following research directions;

- Our evaluation settings are quite small in several aspects. We will 'scale-up' the evaluation in all the aspects; with larger datasets, larger neural networks, larger clusters. We will also use multiple multi-GPU nodes.
- In this paper, we focused on parameter server based method. Parameter server based methods are easy to implements, while it is slow. We will implement and evaluate the direct data exchange methods, like Chain-

ⓒ 2017 Information Processing Society of Japan

erMN[1] of Chainer[2][8].

- We could not achieve the speed up with multiple parameter servers in our experiments. This might be due to the unbalanced load between parameter servers. We will investigate this issue.

Acknowledgments This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). This work was supported by JSPS KAKENHI Grant Number JP16K00116.

References

[1] : ChainerMN : `https://github.com/chainer/chainermn`. Accessed: 2017-08-17.

[2] : chainer.org: `http://chainer.org`. Accessed: 2015-11-04.

[3] : The Cifar-10 dataset : `https://www.cs.toronto.edu/~kriz/cifar.html`. Accessed: 2017-08-17.

[4] : TensorFlow : `https://tensorflow.org/`. Accessed: 2017-08-17.

[5] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: A system for large-scale machine learning, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283 (online), available from ⟨https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf⟩ (2016).

[6] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K. and Ng, A. Y.: Large Scale Distributed Deep Networks, *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, USA, Curran Associates Inc., pp. 1223–1231 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2999134.2999271⟩ (2012).

[7] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J. and Su, B.-Y.: Scaling Distributed Machine Learning with the Parameter Server, *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, USENIX Association, pp. 583–598 (online), available from ⟨`https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu`⟩ (2014).

[8] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, (online), available from ⟨`http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`⟩ (2015).

ⓒ 2017 Information Processing Society of Japan