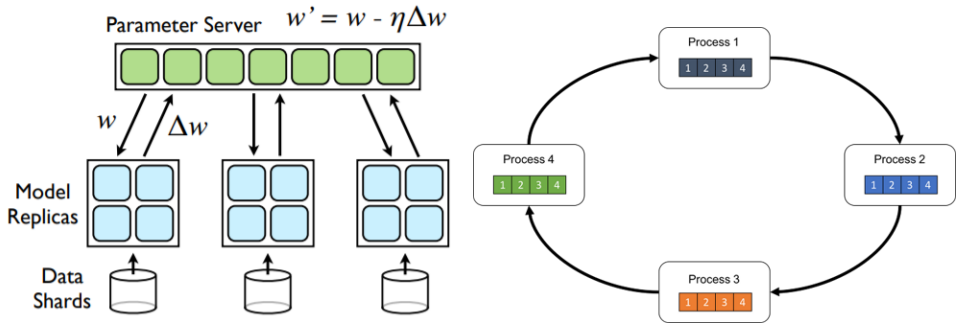
2021 CSE AI Framework Final Report
CaPS-Learning System: Convergence-aware Parameter Skipped Learning System in Distributed Environment

Hyunjoon Jeong (UNIST)
with1015@unist.ac.kr
https://github.com/with1015/CaPS-Learn

Abstract: In few recent years, the number of parameter size in machine learning field has increased highly. Although machine learning models have improved dramatically, hardware still straggles to train the large model. Although, the industry adopts data parallelism for the large model training, there are still heavy training time for many reasons. One of the well-known issues is communication problem between several nodes for sharing their updated parameter. In this project, we suggests *Converge-aware Parameter Skipped Learning System* to reduce communication time by discarding useless parameter updates. In suggested system, all parameters updates in model can be traced by optimizer and the distributed system regulates its transmission by given threshold. Also, the dynamic threshold scheduler can control its threshold from local minimum. By this system, this work reduced communication time for parameter server up to 9 percent with keeping convergence.

1. Introduction

Today, one problem of deep learning tasks is large size data. As data become larger, the training time in one device consume larger time. To minimize training time, recent deep learning (Pytorch [6], Tensorflow [7], etc) supports distributed training as data parallelism: centralized and decentralized. In centralized training, as known as using parameter server, each worker sends their updated parameter to master server. Then, the master server aggregates collected parameters and broadcasts them to workers. Figure 1(a) shows the centralized training with parameter server and workers. On the other hand, decentralized method does not have master server. Instead of it, each worker communicates in all-reduce manner like figure 1(b). In case of Pytorch [6] and Tensorflow [7], both frameworks provide decentralized method in default distributed learning API.



(a) Centralized training with parameter server (b) Decentralized training with All-Reduce

Fig. 1. Distributed deep learning topology

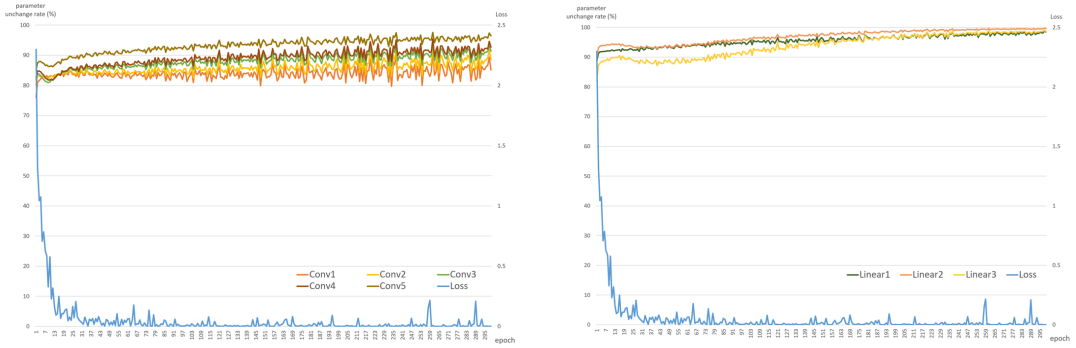
In both distributed training method, one of issues is communication between processes. Since both methods should send their updated parameters or gradients to other process, proper communication protocol is required. Especially, as tensor size become larger, communication bandwidth can have experience about bottleneck easily. Moreover, if the communication uses poor channel, such as WAN, more severe bottleneck can be caused. In GAIA [3], it criticized this high latency network problem with parameter server bottleneck, so that it suggested update avoiding method on parameters and it can reduce communication volume of each workers. Moreover, [1] suggests that the model can still converge, even if there are some skipping un-updated parameters. However, GAIA [3] only targets slow network, such as WAN, and it can be vulnerable to fall into local minimum due to update skipping. In case of [1], it does not guarantee in distributed training. As a result, goals of this project are,

- Reduce communication overhead by avoiding redundant updates in distributed environment.
- Keep convergence with only update valid parameters.
- Implement API to use easily in general-purpose deep learning models.

By achieving these goals, this project can expect faster communication from reduced volume, which discards redundant updates of training. Moreover, regardless of GAIA [3], it can keep robust training from local minimum problem.

2. Motivation

One of the unique characteristics in deep learning job is repetition of training step. When the training process continues, the model does not update all elements in trainable tensors. In this project, these un-trained parameters are called *stationary parameters* or *unchanged parameters*. To identify the ratio of stationary parameters, Figure2 shows the motivation experiment result. We used AlexNet [10] with CIFAR-10 [11] dataset. As training goes on, the loss graph shows convergence, but the ratio of stationary parameters increased. After 20 epochs, the loss graph shows more stationary than initial steps and layers also shows high ratio of stationary parameters. That is, some stationary parameters means that the optimizer does not use these parameters for model update. So, if the worker classifies these parameters and does not send them to parameter server, the communication volume will be reduced, while it does not affect to converge the model. From Figure 2, we can see that this tendency shows higher in fully-connected layer than convolution layer. From both motivation experiment result, we can see that there is opportunity to reduce communication overhead by restricting tensor transmission, who does not effect in significant updates. In the next session, to reduce communication volume of parameter server, *CaPS Learning System* is suggested.



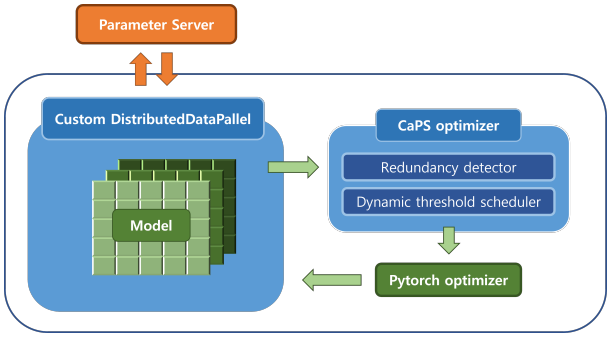
(a) Convolution layer unchanged parameter ratio

(b) Fully-connected layer unchanged parameter ratio

Fig. 2. Stationary parameter ratio in AlexNet [10] with CIFAR-10 dataset

3. CaPS Learning System

Figure 3 shows the whole overview of *CaPS Learning System* in worker side. The system implemented by Pytorch [6] to utilize dynamic graph and user-friendly API. The system consists of two parts: communication and optimizer. In communication part, it is based on Pytorch DistributedDataParall [4] and BytePS [9]. However, original version of DistributedDataParall API does not support NCCL communication protocol in gather operation, which is used in parameter server. So, this work had to use Gloo protocol instead of NCCL. When the model, which is wrapped by DistributedDataParall API, executes forward operation, and it delivers backward results to optimizer. *CaPS optimizer* checks parameters in redundancy detector function by threshold. Moreover, CaPS optimizer has scheduler to control parameter change threshold. This scheduler adjusts parameter change ratio threshold dynamically, by monitoring pre-defined metric. In this section, each components of the system will be introduced. Figure 3(b) shows the algorithm of *CaPS Learning System* usage in real deep learning code. Since the customized DistributedDataParallel [4] and *CaPS optimizer* is wrapper class, it is easy to port in the code (line 11-12). In training session, optimizer step starts to calculate unchange ratio between tensors (line 7) and optimizer gets metric for scheduling threshold in every epoch (line 9).



(a) System overview

Algorithm 1 CaPS Learning System model training

```
Input: Model, DataLoader, Criterion, Optimizer
function TRAIN(Model, DataLoader, Criterion, Optimizer)
    for batch in DataLoader do
        output = Model.forward(batch)
        loss = Criterion(output, label)
        loss.backward()
        Optimizer.step()
    end for
    optimizer.get-validation(loss)
end function
Model = DistributedDataParallel(Model)
Optimizer = CapsOpimizer(Optimizer, parameters)
Train(Model, DataLoader, Criterion, Optimizer)
```

(b) CaPS Learning System usage algorithm

Fig. 3. CaPS Learning System overview and its usage in real training code

3.1. CaPS optimizer

To regulate tensor transmission, *CaPS Learning* provides its own optimizer to be enable to wrap existing optimizers in Pytorch [6]. In *CaPS optimizer*, the step wrapper function executes comparing process between current updated parameters and previous parameters. To classify redundant parameters for convergence of model, *CaPS Learning* defines that the changed parameter from loss backward can effects to deep convergence, and the other does not. So, the optimizer checks the number of unchanged parameters in each layer and compare them with given threshold. Figure 4 shows the naive solution to compare two tensors with how to define unchanged rate of layer. In figure 4, the example layer has 25 parameters both current update and previous state. The optimizer compare both tensors in element-wise manner, and find that there are 5 unchanged parameters. So, the unchanged rate become 20 percent. This is quite simple method to detect element change in the layer, but it has large overhead due to element-wise compare.

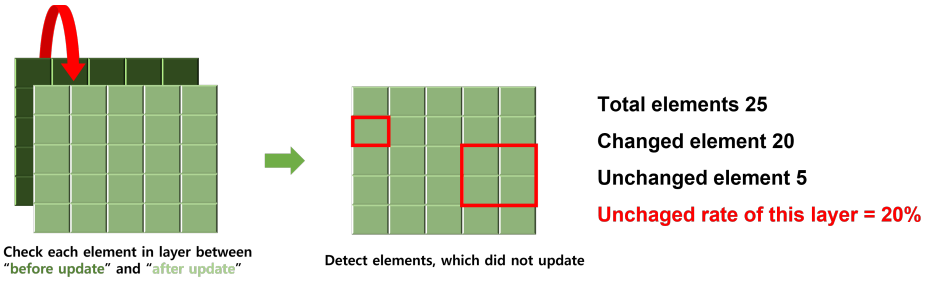
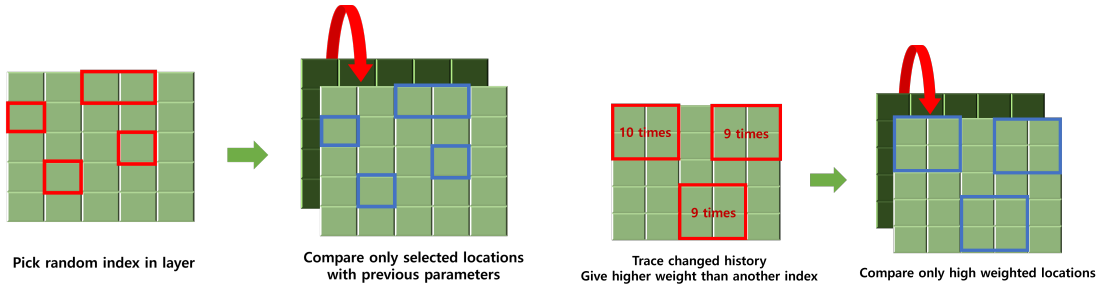


Fig. 4. Naive calculation method of unchange rate in model layer



(a) random search

(b) history-based search

Fig. 5. Optional calculation method of unchange rate in model layer

To remedy shortcomings of the naive solution, *CaPS optimizer* provides two optional methods: random search and history-based search. Figure 5 shows these two optional calculation for comparing with previous tensor update history. In Figure 5(a), it picks random element index and compares only these index. This method depends on

Python random sampling completely, so that the comparison ratio can be inaccurate than other method. Moreover, when the random sampling ratio, which means how many parameters can be picked, should be defined by user, so that there can exists missing unchanged parameters or misunderstood parameters. To revise this problem with low overhead, the other method is called *history-based search*. Figure 5(b) shows the history-based search method in the *CaPS optimizer*. In this case, the optimizer traces how many times each element has been updated. The optimizer keeps these history and tries comparison in history based region. For example, in Figure 5(b), the red zone has been updated many times than other elements, so that optimizer only compares this region.

3.2. CaPS dynamic threshold scheduler

One of the main concerns to skip some parameter update is the validation of training. In GAIA [3], as training goes on, it reduces permitted threshold of the number of unchanged parameters. However, if In *CaPS Learning*, the optimizer supports parameter change rate scheduler. When the user gives frequency parameter, which affects to check current training goes to convergence or custom metric, the optimizer tries to determine whether training falls into local minimum for every given frequency epoch. In each worker, when the epoch is finished, the optimizer stores training process metric, such as loss, training accuracy, validation accuracy or custom metric, in the queue. This queue can have limited capacity, which is called *history length* and given by user. Figure 6 shows the default

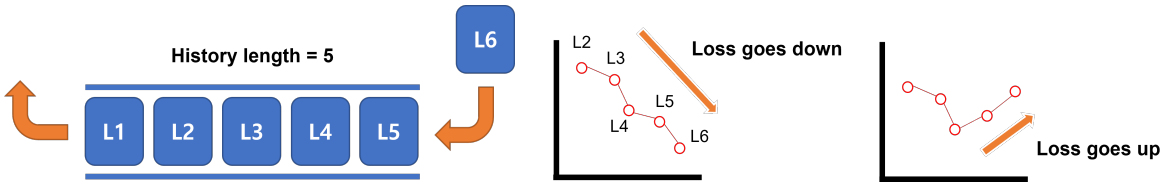


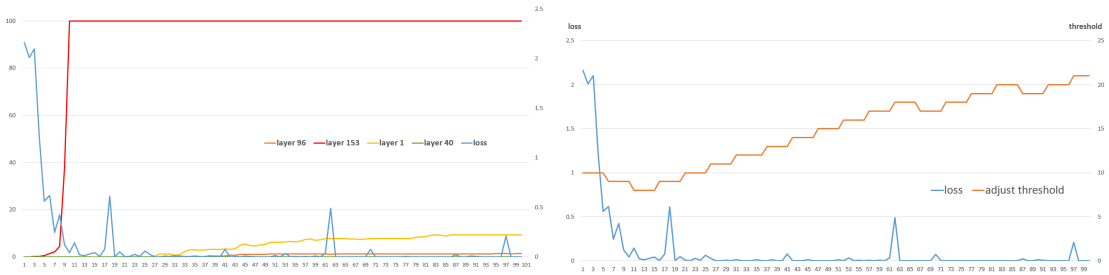
Fig. 6. History queue and default metric to adjust parameter threshold

operation of *history queue* with loss metric. When every epoch is finished, *history queue* push back loss value. In this figure, due to *history length*, it can posses 5 recent losses. When the training process reaches scheduling frequency epoch, *CaPS optimizer* investigates *history queue* and checks tendency of loss metric. If it detects recent losses shows strictly decrease, the optimizer keeps or decreases parameter update threshold. That is, it tries to reduce more communication volume. On the other hand, if the loss goes up partially or strictly, the optimizer adjusts threshold to higher value. In this case, each layer have experience about loose detection of unchanged parameters.

4. Experiment

For *CaPS Learning System* evaluation, 8 GPU nodes used for this experiment. Each node uses Tesla P100 GPU with Intel Xeon CPU E5-2640 2.60GHz CPU and 32GB main memory. To measure effects of *CaPS optimizer* with loss convergence, CIFAR-10 [11] dataset was used and ResNet-50 [8] was used as model. All result logs was measured in first rank worker and all training result was trained up to 100 epoch and used 32 batch size. In case of ResNet-50 training, the experiment follows linear scaling rule [5].

4.1. Effect of optimizer and dynamic scheduler in convergence



(a) Parameter unchange ratio of each layer in ResNet-50

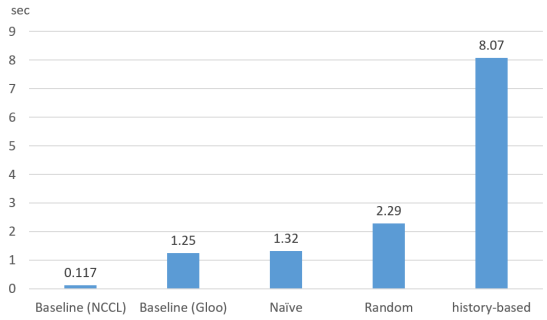
(b) Dynamic adjust threshold in scheduler with loss graph

Fig. 7. ResNet-50 loss convergence experiment with naive solution

Figure 7(a) shows the relationship between loss convergence and threshold ratio in ResNet-50 [8] by using naive solution. In this graph, the *CaPS opimizer* detected 4 types of tendency. As loss goes to converge, the latest layer (layer 153) shows up to 99 percent unchange ratio. On the other hand, middle layers (layer 40, 96) shows under 1 percent unchange ratio. In case of layer 1, it reached 11 percent at the end of training. Unlike AlexNet [10], ResNet-50 is deeper, so that middle layer, which consists of convolution layers and residual layers, shows larger change of parameters than initial and last layers of the model. Moreover, the role of layers can affect to unchange ratio. In case of convolution layer, its role is to extract feature of images, unlike fully-connected layer is to classify the extracted features, which has only 10 classes.

Figure 7(b) shows *CaPS optimizer* threshold change during training. In this experiment, history length was 3, checking frequency was 5 epochs and threshold rate is 1 percent per adjusting, so that scheduler only traced training process within 3 past epochs and check history queue on every 5 epochs. The initial threshold was 10 percent, but at the end of the training, the threshold has increased up to 21 percent. In this case, the size of history queue was only three, so that only recent loss could be applied as convergence history. Moreover, CIFAR-10 [11] is sensitive, so that scheduler increase threshold value repeatedly up to 21 percent.

4.2. *Overhead for comparison between layer histories*



(a) Iteration time after 20 epoch

Naive	23 ms
Random	741 ms
History-based	2187 ms

(b) Layer comparison overhead

Fig. 8. ResNet-50 [8] training iteration test with *CaPS optimizer* and baseline.

Figure8(a) shows iteration time after 20 epochs with baselines and *CaPS optimizer*. In case of naive solution, it shows 5 percent higher than Gloo protocol baseline. On the other hand, random search solution shows 83 percent increasing with Gloo baseline. Also, history-based search shows the highest iteration time. The first reason is, the *CaPS optimizer* uses parameter server, so that it uses *gather* and *broadcast* operation frequently. However, NCCL does not support *gather* operation, so the system should have to use Gloo protocol. To use Gloo, the system had to move tensors from device to host and this become memory copy overhead. Futhermore, baseline DistributedDataParallel [4] uses optimized all-reduce operation with register hook and buckets, so it can overlap communication with computation. In this experiment, both random search and history-based search used small portion of tensor (only 1 percent of each layer). However, in figure 8(b) naive solution, which uses element-wise comparison, shows only 23 ms for checking parameter updates in all layers. The reason is, in case of random search, the code uses Python random sampling API, so that it does not use CUDA operations. Moreover, history-based solution has to keep tensors history and this operates in CPU processor, so that CUDA host to device copy is launched frequently.

4.3. *Communication overhead*

Figure 9 shows the communication overhead time in one iteration with Gloo baseline, and both two tensor comparison methods. The communication overhead was measured by using tensorboard with pytorch profiler. In baseline, which uses Gloo communication on same condition with previous experiments, the initial epoch and 20 epoch later did not have any differences in communication time. On the other hand, in case of naive and random search method, there were 6 to 9 percent reduction in communication time. Although there were slight improvement in communication time, this is not dramatic reduction, due to tensor comparison layer. For same reason, history-based tensor comparison method did not participate in this experiment. One of the other defects is, layers, which have high number of parameters such as convolution layer, did not participate in tensor skipping. Therefore, the reduction time would depend on only initial and final layers in the model.

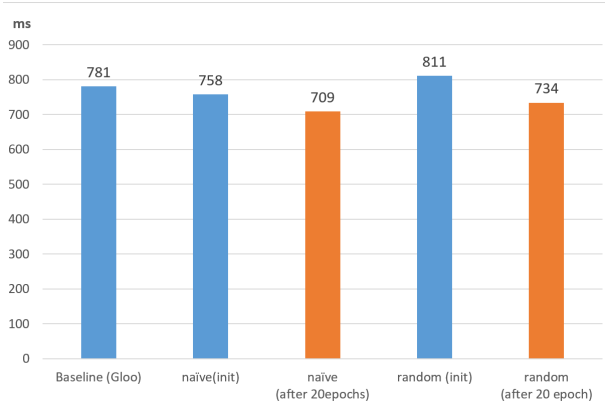


Fig. 9. Communication time in one iteration

5. Conclusion

In this project, the main goal was to reduce communication time by restricting layers submission, which do not update its values, to parameter server. There were three solution to detect tensor update: naive (element-wise), random and history-based. Only naive solution shows only 5 percent overhead with Gloo baseline, and others shows inferior iteration time. To relieve convergence issue from skipping layer submission, dynamic scheduler can control threshold of layer skipping. As our future works, given three solution should be optimized in CUDA level, not Pytorch API level. Moreover, it requires to find proper parameters for adjusting threshold: history queue length, threshold adjust value and adjusting frequency. The other problem is that NCCL protocol is not appropriate for parameter server, because *gather* operation is not supported. As a result, we also need to use RDMA instead of Gloo. Finally, parameter server should support load balance. In this system, each worker does not send all tensors, so that parameter servers can receive imbalanced number of layers. Since this can lead to communication bottleneck, load balance will be required.

6. Reference

References

1. *Convergence-aware Neural Network Training*, Hyungjun Oh; Yongseung Yu; Giha Ryu; Gunjoo Ahn; Yuri Jeong; Yongjun Park; Jiwon Seo, DAC 2020
2. *Elastic Parameter Server Load Distribution in Deep Learning Clusters, SoCC 2020*, Yangrui Chen; Yanghua Peng; Yixin Bao; Chuan Wu; Yibo Zhu; Chuanxiong Guo, SoCC 2020
3. *Gaia: Geo-distributed machine learning approaching LAN speeds*, Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, and Phillip B. Gibbons, Carnegie Mellon University; Onur Mutlu, ETH Zurich and Carnegie Mellon University, NSDI 2017
4. *PyTorch Distributed: Experiences on Accelerating Data Parallel Training*, Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, Soumith Chintala, arXiv:2006.15704v1
5. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*, Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He, arXiv:1706.02677
6. *PyTorch*, https://pytorch.org/docs/stable/index.html
7. *Tensorflow*, https://www.tensorflow.org/
8. *Deep Residual Learning for Image Recognition*, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2016
9. *A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters*, Yimin Jiang, Tsinghua University and ByteDance; Yibo Zhu, ByteDance; Chang Lan, Google; Bairen Yi, ByteDance; Yong Cui, Tsinghua University; Chuanxiong Guo, ByteDance, OSDI 2020
10. *ImageNet Classification with Deep Convolutional Neural Networks*, Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, NeurIPS 2012
11. *CIFAR dataset*, https://www.cs.toronto.edu/ kriz/cifar.html