Geo-distributed Machine Learning using Parameter Servers

Mihir Shete and Felipe Gutierrez-Barragan

Abstract— **Large organizations operate in datacenters across the globe to offer low latency services to users located in its vicinity. The data generated by users which interact with these services is present in the respective datacenters distributed across the globe. It is a common practice for organizations to run simple algorithms like logistic regression or even complex machine learning algorithms to get insights from the data. One way to run these algorithms is to transfer the data to single centrally located datacenter and use one from many of the available distributed machine learning methods to obtain insights. But this approach can be costly in terms of time required and the money spent to transfer huge amount of data to a central location, also regulatory restriction might prevent transfer of data in certain cases. In this work we develop and evaluate techniques to run logistic regression and neural network based supervised machine learning algorithms on geo-distributed data. Our techniques are built on top of the Parameter Server based learning approach, and we evaluate the modified versions of synchronous and asynchronous stochastic gradient descent (SGD) considering tradeoffs between data exchanged across datacenters and the rate of convergence.**

I. INTRODUCTION

Large organizations that offer services to customers all over the world have geographically distributed data centers. These are conveniently located in the vicinity of populace they serve to offer better quality of service. Each data center offering these services collects important data from the customers it serves. Processing the data collected by these geo-distributed data centers can provide important insights into the service usage patterns, among other things, helping the organizations to develop future roadmap.

Following the rise of these geographically distributed services which produce data *in situ*, geo-distributed data analysis has become an increasingly important problem. Systems such as Geode by Vulimiri et al.[13] and Clarinet by Viswanathan et al.[14] have made progress in handling geo-distributed batch analysis. Geodes main goal was to reduce the inter-data center traffic and they show a 250x reduction in data transfer over inter-data center links as compared to the standard approach of moving the data from edge data centers to a centralized data center for batch analysis. Clarinet on other hand optimizes query completion time by 2x by leveraging recent advances in WAN aware data placement and task scheduling.

Distributed machine learning algorithms and systems are very popular areas of research [8], [9], [10], [11]. Even then, distributed Distributed Machine Learning has not been explored in the geo-distributed context as much as other core applications such as batch analytics, streaming, and graph analytics. The work done by Cano et al. (2016) [7] is one of the only accessible research papers towards geo-distributed machine learning. They implement a system that is able to derive a single model from the learning tasks processing different data sets in separate data centers. To overcome the high cross data center communication costs they employ and extend communication-sparse algorithm [9]. Such algorithm, however, makes various assumptions about the machine learning model in terms of sparsity and dimensionality of the data, loss functions; that may or may not hold true in the general geo-distributed case.

As a result of the minimal amount of previous work in geo-distributed machine learning there have not been any frameworks developed, specially geared towards facilitating these tasks. In the single datacenter context, however, various of such frameworks have been developed. A particular framework that has been extensively used to train some of the largest machine learning models since it introduction is the parameter server [3], [4]. An example of a parameter server success story is Dean et al. (2012) [1]. They introduced Downpour SGD which was able to achieve state of the art scalability and performance on the ImageNet dataset at the time, and one of the core components was the parameter server. The algorithm runs on a data parallel scheme in which multiple model replicas are trained on subsets of the dataset. parameter server also played an important role in the first massively parallel architecture for deep reinforcement learning [11]. The algorithm used in [1] was the Deep Q-Network algorithm which was used to learn how to play 49 different Atari games, and it delivered an improved performance in 41 of them

and a reduced train wall-time. Project Adam [12] is another case where the parameter server played a role in delivering state-of-the-art performance, scaling and task accuracy.

Overall, the parameter server framework has served as a tool for pushing the state of the art large-scale Machine Learning in a single data center. Our work aims to extend the research on parameter servers and use them in a geo-distributed setting. We will leverage the learning from this earlier research to develop a system which performs better in terms of network bandwidth usage than a naive approach where the workers and parameter servers are not WAN aware. We will quantify the network bandwidth gains using our approach and contrast it with tradeoffs like - decrease in convergence rate of the algorithm.

II. BACKGROUND & METHODS

A. Background

At the core of every machine learning algorithm there is an optimization problem whose goal is to minimize the classification error of a given machine learning model. The training algorithm attempts to achieve this goal by processing data, followed by updating the model in the direction of a gradient that will (hopefully) make it better describe the processed data. This iterative process continues and multiple passes (training epochs) are done through the full dataset until an optimal (or good enough) solution is found, or until the model is considered to have converged. For many practical applications the training data, size of the model (number of model parameters), or both, may be extremely large. Therefore these types of algorithms will have enormous computing, bandwidth, and storage requirements that can only be achieved in a distributed computing environment.

The parameter server is one of the most commonly used frameworks/approaches to efficiently train and evaluate these machine learning models in a distributed setup. The parameter server architecture is depicted in Figure 1. The machine learning model is distributed over a set of servers (server group). Each worker node reads a different batch of the training data, pulls the model or a subset of the model, calculates the gradients, and finally sends the gradients to the parameter server where the model is updated.

B. Methods

We propose a hierarchical parameter server architecture for geo-distributed machine learning. The architecture is depicted in Figure 2. In this architecture

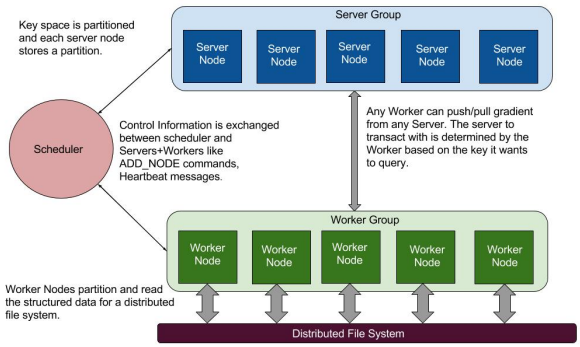

Fig. 1: **Parameter Server Architecture.** A distributed machine learning setup based on the parameter server architecture consists of a pool of workers and servers. The key space is distributed across all the servers using consistent hashing. Workers calculate the gradients and send them over to the servers. Servers have a user-defined *Optimizer* class which is used to calculate new weights from the gradients. The weights are stored on the servers and workers can query for the new weights whenever they are making a forward pass across the model to derive new gradients or classifying incoming or validation data.

every datacenter which contains a different subset of the training data will run a set of workers and local parameter servers called leaf parameter servers. The leaf servers run distributed SGD optimizer to update model parameters based on the gradients pushed by co-located workers. The SGD optimizer is slightly modified and it pushes the entire model parameters over to a root parameter server every user configured iteration. The root parameter server runs a global optimizer which just averages the model parameters received from all the leaf servers. The leaf servers pull the updated model from the root server and run further iterations of SGD on this model. The global optimizer running on the global parameter server is user specified, we are currently using a simple parameter averaging optimizer. In future we plan to develop and evaluate other types of optimizers based on weighted moving averages.

1) Hierarchical parameter server implementation in MXNet: MXNet[2] is a framework to ease the development of machine learning algorithms, especially

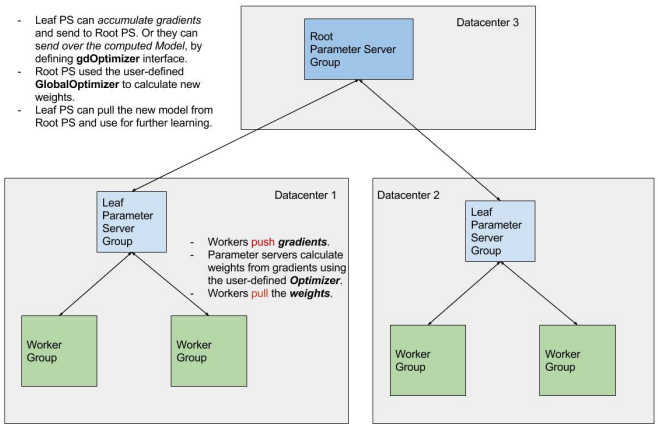

Fig. 2: **Hierarchical Geo-Distributed Parameter Server Architecture.** In a hierarchical distributed machine learning setup based we define root and leaf parameter server. Leaf parameter server are present in each datacenter which hosts the training data. Leaf parameter server run a modified version of distributed SGD optimizer, which can either store all the gradients received from workers and send them to the root parameter server after a fixed number of iterations or just send over the model parameters to the root parameter server after some iterations. The root parameter server, based on the data sent over by the leaf parameter server can have an optimizer function which can calculate the new model parameters. The parameters can be pulled by leaf parameter server and it can continue learning using these new parameters.

for deep neural networks. MXNet supports using a parameter server for distributed learning by exposing a distributed key-value store API to interact with the parameter server. The core of MXNet and it's parameter server implementation which is called parameter serverLite is written in C++. Figure 3 shows the class hierarchy of MXNet core implementation. To support hierarchical parameter server in MXNet we modified the Optimizer class. A method from the Optimizer class is executed on a parameter server when the distributed key-value store handles a push request from every worker. The modified Optimizer acts as a ZeroMQ endpoint which can communicate with a local worker thread. This worker thread is local to every parameter server instance and it establishes a communication channel with the root parameter server. In root parameter servers we sub-classed the Optimizer to define a GlobalOptimizer class which does operations on the parameters or weights of the model instead of the gradients.

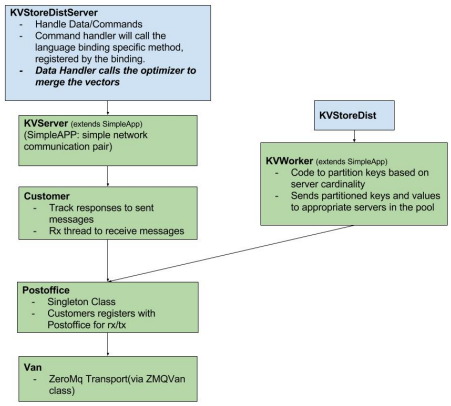

Fig. 3: **MXNet Architecture.** Major Classes in MXNet, parameter serverLite and their dependencies. Modules in Blue are implemented in MXNet and modules in Green are implemented in parameter serverLite.

2) Hierarchical parameter server implementation in TensorFlow: TensorFlow is a very generic machine learning library wherein we can express computations as data flow graphs. Tensorflow does not have out of the box parameter server support like MXNet but it supports executing portions of these data flow graphs on different machines. Using this flexibile programming interface we define a portion of the data flow graph to act as a parameter server which can exchange data in Tensorflow's native sparse data format. Dealing in sparse data format is important to gain insights on training with Criteo's dataset using a hierarchical parameter server as we will show in section 4.

III. EXPERIMENTAL SETUP

A. Datasets

For the purpose of evaluating our proposed methods and setup we chose to use 3 data sets - MNIST handwritten digits dataset, CIFAR-10 dataset, and the Criteo click prediction dataset.

MNIST is a dataset of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized

and centered in a fixed-size image of 28x28 pixels. Since the dataset is not too large and is pre-processed we believe it to be suitable for quickly evaluating our learning techniques, and guide us on what experiments can be fruitful on larger datasets.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, same number of classes as MNIST. With 6,000 images per class, there are 50,000 training images and 10000 test images. CIFAR-10 is 2 times larger than MNIST and since the algorithms have to consider RGB vectors it takes significantly longer to run one iteration of training.

Criteo is a technology company that specializes in performance display advertising. They take an algorithmic approach to determine what user they show an ad to, when, and for what products. Criteo has a global presence with more than 7000 servers distributed across 6 data centers on 3 continents. The Criteo click prediction training set consists of a portion of Criteo's traffic over a period of 7 days. Each row corresponds to a display ad served by Criteo. Positive (user clicked) and negatives (user did not click) examples have both been sub-sampled at different rates in order to reduce the dataset size. The examples are chronologically ordered. There are 13 integer features and 26 categorical features in the dataset, which expand to 33 million features after one-hot encoding the data. Out of the 33 million features pero example there will only be a maximum of 39 non-zero features (13 int features + 26 categorical features). Therefore, in order to efficiently train an ML model on this data we have to work with a sparse representation of it.

B. System Setup

We use virtual machines (VMs) in Cloudlab and Google Cloud to run our experiments. In Cloudlab setup we have 5 VMs in the same datacenter. Each VM has 4 Intel Xeon Sandy Bridge cores, 20 GB memory with a 315GB Virtual Disk. The maximum possible throughput between VMs in this setup is 1 Gbps. We use this setup for training and testing on MNIST and CIFAR-10. To simulate a geo-distributed setup we create artificial boundaries across virtual machines.

The Google Cloud setup has 15 VMs distributed across 3 datacenters: us-east1-c, us-central1-c, us-west1-a. One virtual machine in each datacenter runs a parameter server, such a VM has 4 core Intel Haswell processor with 8GB memory running Ubuntu 14.04. All other machines, which act as worker nodes, have 2 core Intel Haswell processors with 1.8GB memory and Ubuntu 14.04. Each machine has a 50GB SSD attached which contains the subset of the dataset that will be used by that node to train the model. Note that we keep a copy of the subset of the dataset that will be used by a given worker VM in Cloudlab and Google Cloud. We do this to avoid the overheads of accessing and writing to a distributed file system which can skew our measurements, specially for network traffic. Each worker running the training algorithm is assigned a rank which determines the offset in the training data the worker will process. In future experiments we would like to use a more realistic setup with training data stored on a distributed file system like HDFS.

C. Model Measurements

To evaluate the resulting models and compare the across setups (Serial SGD, Distributed Synchronous SGD, Distributed Asynchronous, Geo-Distributed SGD) we used the following metrics:

1) *Accuracy vs. Iterations:* After every n training mini-batches are processed we would calculate the accuracy of the current model on a small testing dataset.
2) *Gradient Norm vs. Iterations:* To see the convergence of the optimization procedure we kept track of the average gradient norm after every mini-batch.
3) *ROC Curve:* Denotes the performance of a binary classifier at various classification thresholds. Logistic regression model will output a probability value for a given example. Different thresholds will produce different true positive rates (TPR) and false positive rates (FPR), and the ROC curve are these TPR vs. FPR as the discrimination threshold is varied.

IV. EXPERIMENTS & RESULTS

A. Learning from the MNIST dataset

MNIST dataset consists of pre-processed images of handwritten digits. The dataset is widely used for evaluating various learning algorithms. In the first experiment we evaluate the performance of LeNet [12] on MNIST when training is running on a single node.

From Figure 4 we can see that with LeNet the convergence is achieved in 468 iterations which constitute a single epoch. Using this as our baseline, we will evaluate how SGD compares to it. There are 2 flavors of distributed SGD we consider - Synchronous SGD and Asynchronous/Downpour SGD. When running synchronous SGD, the parameter server will allow

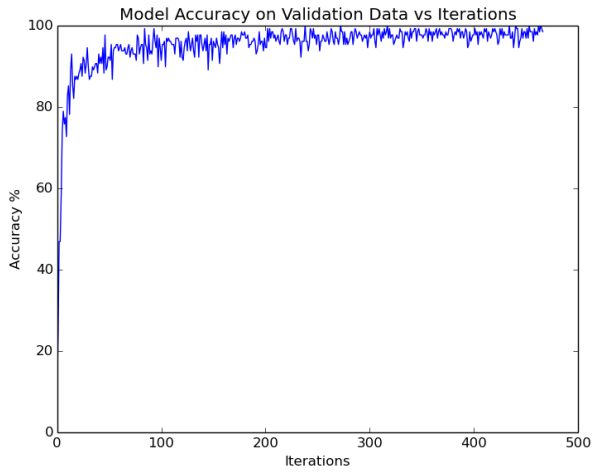

Fig. 4: **LeNet accuracy on MNIST testing dataset.** We train MNIST on a LeNet model in MXNet. The batch size is 128, learning rate is 0.1. The model converges to 98% accuracy in one epoch.

workers to calculate gradients for *(n+1)th* batch only when it has received gradients for the *nth* batch from all workers. This model should give the same results as a serial SGD implementation with a relatively large mini-batch (if there are m workers sync SGD is combining the gradients of m mini-batches). In asynchronous mode, the parameter server does not impose any barrier on the workers and they can proceed independently with training. In its simplest form, asynchronous SGD has each worker pull the model, calculate the gradient, and send the gradient to the parameter server where the model is updated. To reduce the communication an asynchronous SGD implementation can choose to only fetch the model every n_{fetch} steps and push its updates every n_{push}. There is one more mode of operation known as bounded asynchronous SGD. The the pros and cons of each mode of operation is discussed in detail by Mu Li et al.[3]. The aspect of synchronous and asynchronous SGD we want to concentrate on is the rate of convergence and the network traffic generated to reach convergence.

We run the distributed SGD experiments on CloudLab cluster. One node (hostname: vm-28-1) is configured to run as the parameter server and the other 4 nodes as workers. Since there are 4 workers, each worker will only process 25% of the entire dataset, i.e they will run 25% of the total iterations compared to the serial case that was previously demonstrated in Figure 4. Figure 5 shows the accuracy of synchronous SGD. With 25% of the total number of iterations, we see that it achieves similar accuracy to the single node run shown in Figure 4.

The results for asynchronous SGD are not shown here because the training did not converge, even after 10 epochs. The accuracy converges to 10% which is the same as a random model. Refer to the project wiki for the asynchronous SGD results.

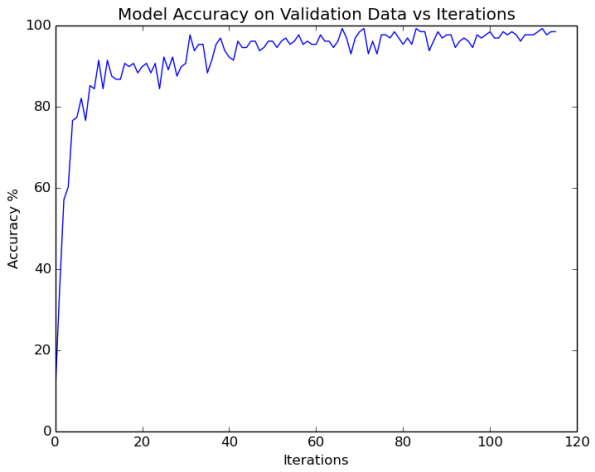

Fig. 5: **LeNet accuracy on MNIST testing dataset with synchronous SGD.** We train MNIST on a LeNet model in MXNet. The batch size is 128, learning rate is 0.1. The model converges to 98% accuracy in one epoch.

A naive way to design a geo-distributed machine learning system would be to have a setup similar to our downpour SGD experiment. The difference will be that the workers can be spread across different datacenters and the parameter server maybe on a completely different datacenter from the workers. To do a cost-benefit analysis of this approach we capture the network i/o done by the parameter server when we run the synchronous downpour SGD experiment. From Figure 6 we can see that the amount of data transferred over the network by the parameter server is 35x as compared to the training data size. So in a naive geo-distributed parameter server implementation we would have incurred a 35x network penalty for obtaining the same accuracy. We call this network penalty as *network usage amplification*. For a geo-distributed learning system to be usable the network usage amplification should be much less than 1 while

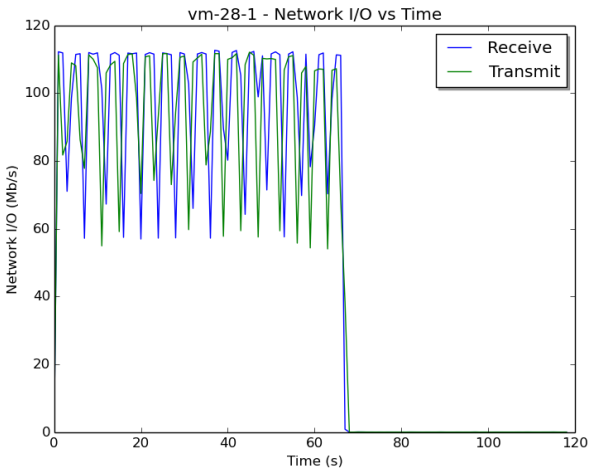

Fig. 6: **Network traffic profile for parameter server while running synchronous SGD on MNIST.** Total traffic transmitted by parameter server is 799.91MB, total traffic received is 813.47 MB. The size of MNIST training data is 45MB. There is ***35x network usage amplification*** as compared to using a simple approach of transferring the entire 45MB dataset to a single datacenter.

Sync Freq.	Test Acc. %	Net Ampl.
1	99.0	35
5	98.0	17.5
10	97.8	0.11

TABLE I: **Comparison of models obtained by changing the frequency of parameter exchange for MNIST.** We see that the test accuracy for the model is acceptable even as the network amplification is decreased to 0.11 when using hierarchical parameter-server architecture.

Sync Freq.	Test Acc. %	Net Ampl.
1	53.6	35
10	47.5	0.47

TABLE II: **Comparison of models obtained by changing the frequency of parameter exchange for CIFAR-10.** CIFAR-10 dataset is 110MB and the accuracy shown above is just for 1 epoch using BN-inception model for small images.

obtaining an acceptable accuracy for the learning task, unless there are regulatory restrictions preventing data movement then network usage amplification only represents the extra cost in terms of money spent to achieve the desired accuracy.

Table I compares our hierarchical geo-distributed model with classic distributed SGD for training on MNIST. In this experiment leaf parameter server was configured to push the entire model over to the root parameter server every *nth* interaction it did with the workers, root parameter server was running a simple Global Optimizer which just averaged the incoming model parameters with the current model parameters. The purpose of this experiment was to see how the choice of global optimizer and the frequency of updates affects the convergence. From Table I we can see that training on MNIST gave a satisfactory accuracy for testing data while achieving a lesser network usage amplification.

Table II shows the results of training over CIFAR-10 data using BN-inception model using a hierarchical parameter server. The experiments did not give a satisfactory accuracy even if we compare it to a single node run of the model which gave 63% accuracy on the testing data on running the training for one epoch. Thus, optimizing hierarchical architecture's parameter to successfully train on CIFAR-10 is still an open problem.

B. Learning on the Criteo Dataset

The lack of support for sparse data structures and operations on MXNet, led us to choose Tensorflow for the evaluations on the Criteo dataset. The need for sparse support in this particular dataset arises from two reasons. First, the size of the dense representation of the data makes the gradient calculation extremely slow and the size of the mini-batch that can be in memory at a time small. Second, even a simple ML model (e.g logistic regression) for this data will have at least as many model parameters/weights as the number of features (33 million), leading to extremely high network usage when replicating the model at the worker nodes. Therefore, sparse communication of the model and sparse operations on the data are required for an distributed and geo-distributed implementation of an ML model on this data.

We train a logistic regression model for binary classification (click/no-click). We have three working implementations of this task on Tensorflow.

1) A serial/single node mini-batch SGD implementation. This implementation reads n sparse train-

ing examples, collects the model parameters that will be updated given the sparse example indices (i.e a sparse model representation), calculates a gradient on the sparse data and model, averages the sparse gradients for the mini-batch, and finally updates the model.

2) A distributed synchronous SGD implementation. The full model is maintained on a parameter server node. Each worker reads a sparse mini-batch of training data, and given the sparse indices it fetches the model parameters it will be calculating gradients for. Gradient calculation happens locally and each gradient per example in the mini-batch is aggregated into one single sparse gradient which is pushed to the parameter server. At the parameter server the incoming gradients are averaged and applied to the model. Once this is done each fetches the model again and repeats.
3) A naive geo-distributed synchronous SGD implementation. This follows exactly the same setup as the distributed synchronous SGD, however, the workers have to fetch the parameters from a parameter server located at another datacenter.

All three implementations were able to successfully train a model on a subset of the Criteo dataset. Results for accuracy, ROC curve, Precision-Recall curve can be found in the project wiki. A network amplification usage analysis has not been completed in this case for two reasons. First, a working hierarchical geo-distributed parameter server implementation is not implemented. The various challenges encountered to implement this on Tensorflow are discussed in the following section. Second, we found that even though we *think* we are sending a sparse representation of the model to each worker the network usage is extremely high (500 MB/s, and we would expect 1 MB/sec). Therefore, an accurate analysis for training on a geo-distributed setup on data of this nature (large and sparse) is not yet available. To see the current results see the project wiki.

V. DISCUSSION

A. Tensorflow Challenges:

There were two main challenges when working with Tensorflow. The first challenge was mainly due to the absolute requirement of sparse data structures and operations to train a model for the Criteo dataset. Since the sparse support in Tensorflow is still in its early days the available functions for training did not support sparse data structures as input, and therefore we had to implement everything from scratch. Second, there is not a lot of documentation on distributed Tensorflow that does not use the training API they offer, and our implementation did not use any of their training functions because of challenge 1.

B. Criteo Dataset Discussion:

The Criteo dataset is particularly interesting when comparing the network usage of the hierarchical geo-distributed PS with the naive geo-distributed PS. In the case of a logistic regression the a model with 33 million parameters is 250MB. In the hierarchical PS, transferring the full model every n iterations will still lead to an elevated network usage. On the other hand, in the naive geo-distributed PS the workers at other datacenters only fetch a small subset of the model which should lead to a lower network usage. Therefore, for a hierarchical parameter server to be able to compete network-usage-wise with the naive setup, it might need to only send the subset of the model that was updated during the n iterations.

REFERENCES

[1] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng. Large Scale Distributed Deep Networks. NIPS, 2012.

[2] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. NIPS Workshop on Machine Learning Systems, 2016.

[3] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. OSDI, 2014.

[4] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server.

[5] Ilan Lobel and Asuman Ozdaglar. Distributed Subgradient Methods for Convex Optimization Over Random Networks. IEEE Transactions on Automatic Control, 2011.

[6] Tim Dettmers. Deep Learning in a Nutshell: History and Training. NVidia Parallel for all blog, 2015.

[7] Ignacio Cano, Markus Weimer, Dhruv Mahajan, Carlo Curino, Giovanni Matteo Fumarola. Towards Geo-Distributed Machine Learning. Arxiv, 2016.

[8] Mcmahan, H. B., & Streeter, M. (2014). Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. Advances in Neural Information Processing Systems (Proceedings of NIPS), 19. Retrieved from http://papers.nips.cc/paper/5242-delay-tolerant-algorithms-for-asynchronous-distributed-online-learning.pdf

[9] Dhruv Mahajan, Nikunj Agrawal, S. Sathiya Keerthi, S. Sundararajan, Leon Bottou. An efficient distributed learning algorithm based on effective local functional approximations. Journal of Machine Learning Research 16 (2015) 1-32

[10] Google Research. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015.

[11] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen,Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, David Silver. Massively Parallel Methods for Deep Reinforcement Learning. International Conference on Machine Learning, 2015.

[12] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.