SIMULATION OF PARALLELIZATION OF DEEP NEURAL NETWORKS BY DIVIDING DATA

ABSTRACT

This paper explores the methods for parallelizing DNN, including data splitting and K-fold Cross-Validation ($kfCV$) application across deep learning methodologies, focusing on the simulation instead of implementation. Recent studies show the successful parallelization of DNN thanks to the availability of massive data sets that can be used for training. However, to the best of our knowledge, there is no DNN simulator to examine the different methods proposed for such parallelization. In this work, a DNN simulator is proposed to examine DNN training efficiency and accuracy by using parallel nodes. The parallelization model simulated in this paper distributes data across n-nodes to leverage the strengths of distributed computing. This parallelization strategy aims to enhance the robustness and accuracy of models by conducting $10fCV$ in a distributed manner, enabling the simultaneous processing of multiple folds.

Keywords: Data Parallelization, 10-fold Cross-Validation, Scalable Deep Learning

1 INTRODUCTION

Our motivation was the work of Dean et al. [1], where they address the problem of accelerating the training of deep networks. In recent years, a surge in machine learning datasets has prompted researchers to explore scaling up machine learning algorithms through parallelization and distribution. Previous work has primarily focused on linear, convex models and sparse gradients. However, the interest lies in combining the advantages of both approaches without restricting the model's form. In deep learning, existing efforts have concentrated on training small models on single machines. Nevertheless, the focus is on scaling deep learning techniques to train very large models with billions of parameters without imposing limitations on the model's structure. A software framework called DistBelief has been introduced to facilitate distributed computation in neural networks and graphical models, allowing users to define computation at each node and manage communication between machines. Significant speedups in training large models are achieved by partitioning the model across multiple machines and leveraging all available CPU cores. Additionally, two distributed optimization algorithms, Downpour SGD, and Sandblaster L-BFGS, have been presented to parallelize computation across multiple model replicas, further enhancing the scalability and efficiency of training large models. Downpour SGD employs asynchronous stochastic gradient descent with parameter server shards, while Sandblaster L-BFGS distributes parameter storage and manipulation across multiple machines. These approaches enable considerably larger models to be trained than previously reported and significantly reduce overall training times by leveraging tens of thousands of CPU cores. The following are the parallelization techniques that the authors elude:

Proc. of the 2024 Annual Simulation Conference (ANNSIM'24), May 20-23, 2024, American University, DC, USA
P.J. Giabbanelli, I. David, C. Ruiz-Martin, B. Oakes and R. Cárdenas, eds.

Model parallelism is a technique used in distributed computing environments to train very large deep neural network models. In this approach, the model is divided or partitioned across multiple machines or processing units. Each machine is responsible for computing a portion of the model's computations. The framework automatically parallelizes computation within each machine using all available CPU cores and manages communication, synchronization, and data transfer between machines during training and inference. By distributing the model across multiple machines, model parallelism allows for efficient utilization of resources. It can significantly speed up the training process, particularly for large models with billions of parameters.

Data parallelism is another technique used in distributed computing environments to train deep neural network models. In data parallelism, multiple replicas of the model are created, and each replica processes a different subset of the training data. These replicas work simultaneously to optimize a single objective. The framework manages communication and synchronization between the replicas, allowing for efficient parallelization of the training process across multiple machines or processing units. Data parallelism enables the training of large models by distributing the workload across multiple replicas, leading to significant speed gains compared to training on a single machine.

In this work, we show data parallelization by splitting data in a small DNN (MLP), using a simulator, and performing experiments on training a Convolutional Neural network (CNN) in parallel using $kfCV$.

2 METHODOLOGY

We employ the DSMP simulator; for performing our experiments with MLP, we use text data with a 94,000 tweets dataset, and for image data, a CNN, using the MNIST dataset of handwritten numbers images containing 50,000 samples. For our CNN experiments, we use the Pytorch v2.1.0 deep learning framework [2], to program the training and perform the data distribution with weights synchronization.

Table 1: System configurations used to run the experiments.

Config.	Description	OS
$C1$	Intel Core i7-4790S CPU @ 3.20GHz, 3201 Mhz, 4 Core(s), 8 Logical Processor(s)	Windows 10 Pro
$C2$	Intel Xeon Gold 5218 CPU @ 2.30GHz with 1.1TB of RAM, 64 logical cores, and 8 GPUs NVIDIA A100.	Linux Ubuntu 20.04.4 LTS

2.1 Data splitting

In data parallel training, each node trains the entire network for one epoch (one iteration) using its data. Afterward, the gradients are averaged, and the local weights are updated before moving on to the next epoch. The exchange of parameters is done through a parameter server or decentralized communication mechanisms such as all-reduce [3]. For a simplified example of using data parallelization in Deep Neural networks (DNNs), see Figure 1.

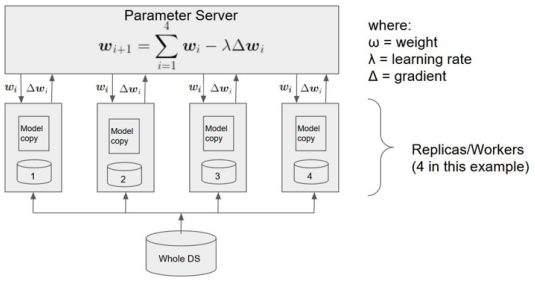

Figure 1: Data Parallelization schematic.

The data-splitting method involves independent data folds for parallel processing. For example, with a data set split in two, we utilize $10fCV$ on two separate nodes. The remaining test part's size is arbitrary, and this part will be tested similarly by a final model achieved when scaling up to more nodes. For each multiple-node configuration, we report the computational speed-up and the model's accuracy for image data.

3 SIMULATION OF SMALL DNN

We present the results of running the Twitter simulator developed in the DSMP laboratory [4]. We will focus on MLP with more than two hidden layers as the sample of DL algorithm for running our experiments. This simulator can parallelize data using $1,2,4$ and 8 nodes.

We use this formula to calculate the performance improvement (PI) for the simulator and image data: $PI = \frac{\text{Timing of 1 node} - \text{Timing of } x \text{ nodes}}{\text{Timing of 1 node}} \times 100$. It is calculated by taking the ratio of the timing result as the number of x nodes increases compared to 1 node.

In this experiment (using configuration C1, see Table 1), first we used the Multilayer Perceptron (MLP) with one hidden layer to train the mentioned Tweets network. This system recommends that a follower follow the top M followers. First, we run the simulator in a simple 80-20% data split for training and testing. In this experiment, we don't perform $10fCV$, and, as Table 2 shows, depending on the number of nodes, data is divided among each node. After training, by averaging the weights of each model, the final model is tested with the 20% of original data. Two separate conclusions can be made by observing the results of Table 2 as follows:

1. The original accuracy of building the model on one node decreases by splitting the data, which is expected because of training with less data on each node.
2. After increasing nodes to 4 and 8 nodes, the communication time between nodes for averaging the weights of their models dominates the parallelization speedup because of using CPU has a slow computation performance.

To solve the first problem, we added the $10fCV$ method on each node to keep the accuracy close to 33% of one node (see Table 2). In this experiment, configuration C1 was also used. For examining the second observation, since our simulator is limited to CPU, we removed the weight averaging part to see by using fast computing processors such as GPUs how much of a speedup our method can produce. Conversely, performing the same experiment to run the MLP process using $10fCV$ for $N = 1,2,4,8$, we observed a significant improvement of 64.85% for $n = 8$, vs. $n = 1$, shown in Table 3. For this experiment, we used configuration C1; see Table 1.

Table 2: MLP with an 80-20% data split

N	**PT (SECS.)**	**Accuracy (%)**	**Improvement vs. 1 Proc. (%)**
1	2,135.30	33.33	N/A
2	1,362.47	23.61	36.19
4	2,505.54	23.75	-17.34
8	4,818.00	0.15	-125.64

Table 3: MLP with 10-fold CV% data split

N	**PT (SECS.)**	**Improvement vs. 1 Proc. (%)**
1	9,511.66	N/A
2	9,123.58	4.08
4	8,277.06	9.28
8	2,909.78	64.85

Since we would like to show a simulation of parallelization over a DNN, we have converted the MLP into one by adding two hidden layers to the existing one using configuration C1 (see Table 1) to do the experiments faster. Accuracy was kept in the range of 35% for 4 nodes that were completed by this time. The current results that we have are portrayed in the following (Table 4). We can note an increase in the processing time as our DNN architecture has 3 hidden layers.

Table 4: DNN with an 80-20% data split

N	**PT (SECS.)**	**Improvement vs. 1 Proc. (%)**
1	2815.08	N/A
2	2647.11	5.97
4	4168.46	-48.08

We want to note that since our simulator runs only on CPU, which is slower than GPU, and weight adjustment is a computer-demanding task, we cannot precisely predict the speedup improvement of our method when performing GPU parallelization. However, we expect by much faster processors; this method can keep a similar accuracy and speedup of around 60% for multiple nodes that can be different for the type of data and NN variant.

4 EXPERIMENTS WITH CNN

In our study, we utilize 10-fold cross-validation ($10fCV$) to enhance the training and validation of deep neural networks (DNNs) using image data, specifically the MNIST dataset. The dataset is split into 80% for training and 20% for testing/validation. These experiments used C2 configuration (See Table 1). In Figure 2 we show our method for parallelizing $kfCV$.

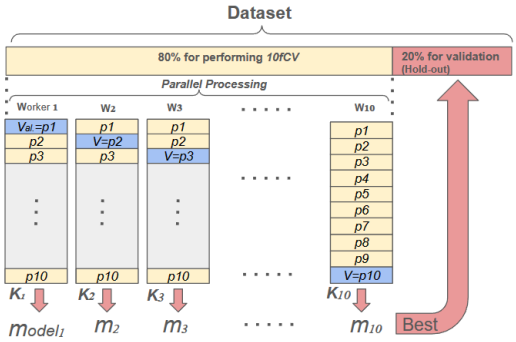

Figure 2: K-fold cross-validation method ($k = 10$).

These hyperparameters were used during training: a learning rate of 0.099, batch size of 1024, and 20 epochs. The methodology demonstrates improved accuracy and processing efficiency by distributing the training data across processors operating in parallel. This setup reduced training time and showed potential

for increased training accuracy with higher folds in cross-validation and varied processor counts, highlighting the significance of parallel computation.

We hypothesize that increasing the folds in cross-validation could lead to higher training accuracy. To examine the effect of computational resources on the network's performance, we varied the number of processors used in training, explicitly using $n = 1, 2, 4, 6$ and 8 processors. Note that we continue to split the dataset processing using data parallelization, and the $kfCV$ is also parallelized for $n > 2$. We recorded the training accuracy for each combination of k value and processor count detailed in tables (5 and 6).

Table 5: Training accuracies of image data in CPU using $kfCV$ method.

N	**k=2 (%)**	**k=4 (%)**	**k=6 (%)**	**k=8 (%)**	**k=10 (%)** $10fCV$
1	98.39	99.28	98.71	96.83	99.57
2	97.40	97.99	97.62	95.12	98.10
4	93.57	97.76	97.39	87.59	98.19
6	91.61	97.90	97.39	95.58	98.14
8	97.11	97.63	92.31	96.12	98.04

Table 6: Training accuracies of image data in GPU using $dkfCV$ method.

N	**k=2 (%)**	**k=4 (%)**	**k=6 (%)**	**k=8 (%)**	**k=10 (%)** $10fCV$
1	99.09	98.52	99.70	99.26	99.73
2	95.68	97.97	97.41	95.71	98.49
4	94.90	97.91	97.70	96.66	98.42
6	96.03	98.12	97.78	92.67	98.45
8	96.43	98.12	97.99	94.86	98.36

The results showed that $k = 10$ consistently achieved the highest training accuracies across all processor numbers. Therefore, $10fCV$ provided the best training accuracy for the CNN on the MNIST dataset, suggesting that a higher number of cross-validation folds might be more effective for this task.

We have performed two more experiments. First, we run the training of the CNN for $k = 10$ without data splitting (see Table 7). Afterward, we do the same, splitting the data (see Table 8); We can observe that when we split the data, there is a small decrease in accuracy compared to not splitting.

The simulation results confirmed by this experiment that shows parallelization by data splitting can increase DNN speedup up to 60%.

Table 7: CNN training using $10fCV$ without data splitting.

N	**Accuracy (%)**	**PT (secs)**	**Scalability Improvement vs. 1 Processor (%)**
1	98.63	80.90	N/A
2	97.56	82.36	-1.80
4	98.50	52.76	34.78

Table 8: CNN training using $10fCV$ with data splitting.

N	**Accuracy (%)**	**PT (secs)**	**Scalability Improvement vs. 1 Processor (%)**
1	96.98	93.06	N/A
2	95.40	57.52	38.19
4	96.51	43.15	53.63

5 CONCLUSIONS

In our research, we have implemented a novel DNN simulator to examine the efficacy of parallelization by data splitting and parallel 10-fold Cross-Validation ($10fCV$) in optimizing the training of deep learning (DL) models. The proposed simulator is developed by considering an MLP with three hidden layers replicated on each node, and then by communication between nodes, the simulator averages the weights and creates identical models on each node. The simulation results indicate that for the training of small DNNs when combining parallel processing with $10fCV$ and splitting data on nodes, the improvements in training speedup while keeping similar accuracy for using DNNs on multiple nodes can speed up the training up to 60% faster. $10fCV$ is used to make each replica model more general, which has led to significant improvements in both the efficiency of model training and the accuracy of the resultant models. The simulation results were confirmed by the parallel running of CNN with the same strategy.

Looking ahead, we aim to broaden the scope of our investigation to include other methods for simulating parallelizations of various DNNs or machine learning methods. By doing so, we hope to uncover new insights into the scalability and efficiency of advanced deep learning models, further contributing to the evolution of machine learning methodologies and their application in solving real-world challenges.

REFERENCES

[1] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1232–1240, 2012.

[2] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[3] J. Keuper and F.-J. Preundt, "Distributed training of dnns," in *2016 2nd Workshop on ML in HPC Environments*. IEEE, 2016, pp. 19–26.

[4] X. (2019) DSMP Lab a Multi-agent System Simulator for Distributed Recommender System. http://scs.ryerson.ca/ X/MAS-SIMULATOR.mp4.

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–43, Aug. 2019.

[6] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "PyTorch distributed: Experiences on accelerating data parallel training," Jun. 2020.

[7] O. Schuessler and D. Loyola, "Parallel training of artificial neural networks using multithreaded and multicore CPUs," in *Adaptive and Natural Computing Algorithms*. Springer Berlin Heidelberg, 2011, pp. 70–79.

[8] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2010, pp. 938–948.

[9] M. Ali, A. Anjum, M. U. Yaseen, A. R. Zamani, D. Balouek-Thomert, O. Rana, and M. Parashar, "Edge enhanced deep learning system for Large-Scale video stream analytics," in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*. ieeexplore.ieee.org, May 2018, pp. 1–10.

[10] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," Apr. 2016.

[11] Z. S. Kadhim, H. S. Abdullah, and K. I. Ghathwan, "Artificial neural network hyperparameters optimization: A survey." *International Journal of Online & Biomedical Engineering*, vol. 18, no. 15, 2022.

[12] E. Buber and D. Banu, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. IEEE, 2018, pp. 1–6.

[13] C. Shallue and G. Dahl, "Measuring the limits of data parallel training for neural networks," 2019. [Online]. Available: https://blog.research.google/2019/03/measuring-limits-of-data-parallel.html

[14] S. Pal *et al.*, "Optimizing multi-gpu parallelization strategies for deep learning training," *DeepAI*, 2019. [Online]. Available: https://deepai.org/publication/optimizing-multi-gpu-parallelization-strategies-for-deep-learning-training

[15] A. N. Kahira *et al.*, "An oracle for guiding large-scale model/hybrid parallel training of convolutional neural networks," *DeepAI*, 2021. [Online]. Available: https://deepai.org/publication/an-oracle-for-guiding-large-scale-model-hybrid-parallel-training-of-convolutional-neural-networks

[16] Y. LeCun, C. Cortes, and C. J. Burges, "Mnist handwritten digit database," http://yann.lecun.com/exdb/mnist/, 2010.

[17] D. Datta, D. Mittal, N. P. Mathew, and J. Sairabanu, "Comparison of performance of parallel computation of cpu cores on cnn model," in *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. IEEE, 2020, pp. 1–8.