An Examination of Neural Networks on Cluster Computers

Robert K.L. Kennedy
Florida Atlantic University
Boca Raton, Florida, USA
rkennedy@fau.edu

Taghi M. Khoshgoftaar
Florida Atlantic University
Boca Raton, Florida, USA
khoshgof@fau.edu

***Abstract*—Deep learning models often require large computational resources during the training process, and due to the size and complexity of the model and training data, usually have long training times. As a result, methods to accelerate the training process, specifically the training time, are areas of active research. Training on distributed cluster computers using data parallelism, using hardware acceleration, or the combination of the two are increasingly effective and accessible ways to reduce the large deep learning training times. In this paper, we train neural networks across different size clusters with and without hardware acceleration. We examine the effects of multi-worker multilayer perceptron training on highly class-imbalanced Medicare Part B data. Our results show the effects of distributed training, on training time and model performance, for multilayer perceptrons trained on class-imbalanced data are greater than the effects on convolutional neural networks trained on balanced image data.**

***Index Terms*—GPU acceleration, parallel computing, neural network, deep learning, distributed system, class imbalance, fraud detection**

I. Introduction

Deep learning refers to artificial neural networks (NN) that are characterized by a large number of layers and a large number of neurons. These networks typically perform best when trained on Big Data. As such, training a large number of neurons on vast amount of data results in a computationally expensive task. NN training, when the randomly initialized NN is fit to the training data, can take weeks, such as with "GoogLeNet" [1]. There exists a couple of different approaches to try and reduce the long training times. Training on cluster computers increases the total computational power that can be applied to a single problem by combining the processing power of multiple processors or machines. Another is to use graphics processing units (GPUs) to reduce training times. Both approaches aim to do the same thing, but also have their own challenges [1]–[5]. To the benefit of the researcher and research community, the type of GPUs required to realize significant performance increases are becoming more prevalent and more accessible, such as in cloud services like Microsoft's Azure and Amazon Web Services (AWS).

To use a GPU for NN training, hardware specific languages are typically required. For NVIDIA GPUs CUDA is the language used [6], [7]. These languages are supported in many of the popular NN libraries such as TensorFlow (and Keras), making it possible to utilize GPU acceleration without having to write device-specific code. Additionally, these libraries are open source. This allows for the researcher to train deep learning models using hardware acceleration on a variety of different systems. Specifically, this means GPUs can be used on standalone research computers or on a cluster of computers, each with one or more GPUs, managed by a cluster management software, such as the Slurm Workload Manager.

Slurm [8] is a cluster management and job scheduling system for clusters of Linux machines. Slurm does not have its own data management system or programming language. It is used to submit jobs of any supported language onto a set of compute nodes for execution. In our case, our Python-based NN training jobs were run on a shared university cluster that contained the GPU and CPUs we wanted to target. Every effort was made to ensure the resources that were allocated for our work were exclusive and comparable between our experiments to minimize any potential variation in performance that may occur on a shared resource. This is discussed further later in the paper.

In this paper, we examine the effects of parallelization on NN training time across different processors and a number of workers managed by Slurm. We train a convolutional neural network (CNN) on image data to use as a baseline and compare it to a multilayer perceptron (MLP) trained on highly class-imbalanced Medicare Part B fraud data. We performed random undersampling of the Medicare Part B Big Data to produce the highly class-imbalanced dataset for this work. The dataset's class imbalance is 99:1 (majority to minority class ratio). We also briefly examine any changes in model performance–measured in accuracy or AUC–when trained across multiple workers. To the best of the author's knowledge, this is the first work to compare training time performance of different NN architecture types when trained on multiple CPUs or GPUs, specifically on high class-imbalance Medicare data.

II. Related Work

When training deep learning models in parallel, either across multiple CPUs or GPUs on one or more machines, there are two main factors that contribute to the model's training time. The first is the total required computation time to fully train a given NN. The second is the additional communication cost between the set of processors. This can be between CPUs across a cluster, between multiple GPUs on a single machine, or between multiple GPUs across multiple machines in a cluster. For parallelized NN training to be effective a balance between these two factors is necessary. It is entirely possible to increase the overall training time if the communication costs of additional processing units (workers) outweigh the computational speed up of the additional workers. However, this balance is dependent on the underlying hardware since the interconnect speeds between nodes in a cluster or between GPUs can vary based on device type. Parallelization of gradient descent [9] and its derivatives is a challenging aspect of large-scale distributed machine learning. Work to parallelize stochastic gradient descent (SGD) has been researched [10]–[14]. Many of these works adhere to either a data parallel or a model parallel paradigm. In this paper, we adhere to a data parallel paradigm.

A. Parallelism Paradigms

In regards to deep learning, data parallelism is a paradigm where the training data (as opposed to the model itself) is parallelized across workers. The starting training data is divided and distributed across worker nodes in a system for training. Each worker can be either a CPU or GPUs across one or more machines. A copy of the initial model is sent to each worker for training. All workers concurrently train the same model on its own subset of the training data. Since each worker only has a subset of the data, it would train faster since it has less data to process through. At predetermined times, all models are aggregated into one model and then re-distributed to the workers. This results in all the workers training a single NN model, as opposed to having an ensemble of models. This parallel SGD approach is one of the oldest implementations of distributed NN training [15].

B. Graphics Processing Units

NVIDIA developed the CUDA language [6] to allow users to utilize GPUs for generalized computations, as opposed to just for rendering graphics (where the name comes from). GPUs were originally created to efficiently process the graphical shader programming model. This is used for rendering a user interface to a computer monitor. This required large amounts of parallel processing and resulted in hardware architectures that were highly parallel, had multi-core processors, and multithreading, as compared to traditional CPUs. The CUDA language allows researchers to take advantage of the GPU hardware and apply it to training NN models. Some NN models benefit from GPU acceleration more than others. GPUs need their own random-access memory, or video random-access memory (VRAM), to function. However, this poses a unique problem for training. Typically, VRAM sizes are much smaller than system RAM. Additionally, this memory is not user-replaceable which can lead to memory constraints for the processing tasks put onto a GPU. Clustering multiple GPUs across one or more computers can solve this memory problem. In this paper we use data parallelism to reduce required memory on each GPU device. Using one or more GPUs can significantly decrease training times as compared to using a CPU alone [2].

III. Methodology

Reducing overall training time for a neural network model is a primary reason for training in a parallel manner, using GPU acceleration, or a combination of both. However, reduced training times occur when the communication overhead of additional workers is outweighed by the increase in performance of the additional workers. Additionally, parallelization or hardware acceleration is only beneficial if there is not too significant of a change to the resulting, trained NN model, as compared to when using a single CPU worker. It is entirely possible that the runtime could spend more time in communication overhead than NN training and result in slower training times. It is also entirely possible that a NN model, trained in parallel or with hardware acceleration, ends up with practically worse model performance (e.g., classification performance).

In this paper, we run our experiments using a data parallel approach on CPUs and GPUs across various cluster sizes to examine training time performance for two different NN architecture types and two different dataset types and sizes. Our runtime is built using TensorFlow and Keras to abstract our NN models and to perform NN training. We use TensorFlow as the main interface with Nvidia CUDA [6], which provides us the GPU acceleration. The runtime is capable of training NNs on multiple GPUs across multiple machines in parallel. We use a combination of Slurm and TensorFlow's distributed functionality and only train in a data parallel manner. The training data is evenly divided across each worker along with a copy of the initialized NN model for training. In our case, a worker can be either a CPU or GPU worker. An epoch is defined when all workers have trained and processed through all of its data (a subset of the whole training data). After each epoch, all worker's NN models are then combined into a single coherent model. Then the next epoch starts. The result is a single trained NN model that was trained by numerous workers. If the model weights were not aggregated, it would result in several individually train models, which would be an ensemble as opposed to a single model.

Slurm is used for job scheduling on a shared university Linux cluster and to allocate resources, either CPU or GPUs. Each of the experiments run with Slurm have between 1 and 8 worker nodes, where each worker is either a CPU or GPU depending on the experiment. In our case, we use Python and several Python libraries. Slurm does not have any functionality for abstracting worker nodes, or aggregating data, thus we use TensorFlow to slice the training data into subsets and into batches. Slurm is responsible for executing processes on individual hardware. We use TensorFlow itself for distributing the NN model, the training data, and aggregation of weights via their synchronous distribution strategy. It is a data parallel strategy that uses all-reduce for aggregation during training. TensorFlow is also used for dividing the training data into batches and distributing to each worker node. However, in our cluster each node has the same connection to the persistent storage, so each worker just accesses its assigned batch.

IV. Experiments

Our experiments were run on a Linux machine hosted on a university cluster. Our machine has 4 processors (each with 10 cores), 64 GBs of system memory and 8 logical K80 GPUs. We replicate each one 10 times to be able to statistically compare different experimental configurations. We compare the number of workers (1 through 8) and its effect on neural network training time and the resulting models' performance. For each experiment and each configuration of workers, we use Slurm to reserve the entire machine and allocate N/8 of the machine to our training routine, where N is the number of required workers per machine. For example, when using 2 workers Slurm reserves the entire machine and allocates 1/4 of the machine to our work, either 1/4 of the available CPUs or 1/4 of the available GPUs. This is done so that each experiment and configuration is comparable to the next, as well as to remove any chance some other portion of the machine is assigned to other work (the machine is in a shared cluster).

In this paper, we use training time, measured in seconds, as the main performance metric when evaluating our experiments. However, we also examine the effects of multiple workers on the model performance itself. We use accuracy for our 10-class classification model and AUC (area under the curve) for our binary classification model that is trained on highly imbalanced data.

A. Datasets

We use two datasets for our experiments. The first dataset is the MNIST image dataset and the second is a Medicare Part B fraud detection dataset. The image dataset is used to evaluate how our work scales when training on a well-known 10-class classification dataset. This is used to compare the results obtained training a smaller NN model on highly class-imbalanced Medicare data. The two datasets are described as follows:

1) The MNIST database of handwritten digits: MNIST [16], [17] is a widely used dataset for deep learning. This dataset consists of 70,000 28x28 grayscale images. Each image is of a handwritten digit and is labeled with one of ten classes, one for each of the ten Arabic numerals (0 through 9). This was originally adapted from the larger NIST Special Database 19 [18], which included letters as well. We chose this dataset because it is well-known, publicly available, and easy to use. We train a CNN on the MNIST dataset.

2) Medicare Part B Dataset: This dataset is a large fraud detection dataset (referred to as MedicareB) and consists of Medicare Part B claims. It can be used to detect fraudulent claims made to the Medicare system by a fraudulent medical provider. Medicare Part B is a U.S. government program that covers some medical procedure costs primarily for U.S. citizens 65 years and older. This data, which is provided by the federal government, is unlabeled. To make it a fraud detection dataset, fraud labels were added to each row by mapping from the List of Excluded Individuals and Entities (LEIE) database [19]. The result is a labeled dataset with no missing data points and is largely one-hot-encoded. The original dataset is provided by the Centers of Medicare and Medicaid Services [20]–[23]. The MedicareB dataset contains 8.4 million rows of data, each with 29 attributes. Fifteen of the attributes relate to the medical provider (e.g. type of physician), 14 attributes relate to the Medicare claim procedure (e.g. type medical procedure), and the rest are various metrics for the Medicare payments themselves (e.g. average charges a physician submits for a given medical services) [24]. The data spans from 2012 through 2018. This dataset is highly class-imbalanced. The effects of high class-imbalance are out of the scope of this paper. However, it has been shown that random under sampling (RUS) of the majority class can improve model performance [25]. RUS also reduces the size of the dataset. For our experiments, we randomly undersample the data to produce our working dataset with a 99:1 class imbalance ratio (majority to minority class ratio). This results in a dataset with 433,400 instances and roughly a 470 MB file size. Detailed analysis of the effects of class imbalance and multi worker runtimes are out of the scope of this paper. Thus, additional datasets, each with different ratios of RUS, were not considered.

Nodes	MNIST-CPU	MNIST-GPU	MedB-CPU	MedB-GPU
1	2443.3	123.5	74.9	111.2
2	1244.3	98.2	52.3	73.7
3	901.5	72.6	90.1	141.7
4	693.0	65.9	85.0	136.2
5	586.7	61.1	87.3	134.1
6	493.4	57.0	83.0	126.1
7	430.8	53.3	87.2	133.8
8	385.8	50.1	83.4	126.1

TABLE I
Average training time, in seconds, for each Slurm cluster configuration.

B. Experimental Results

We ran two experiments to measure the effect of the number of workers on NN training time and model performance. We examine the effects on a CNN model trained on a 10-class image dataset and the effects on an MLP trained on Medicare Part B data. They consisted of training the same model, on the same data, across clusters with 1, 2, 3, 4, 5, 6, 7, and 8 workers. One set used CPUs as the workers and the other set used GPUs as the workers. We replicated our results 10 times.

1) A CNN on MNIST Data: The first set of experiments we ran on our Slurm setup was a CNN on the MNIST dataset. The first layer has 784 neurons (for the 28x28 images) followed by a series of convolutional layers directly followed by pooling layers. The first convolution layer has a size of 256, the second is 128, and the third and final convolutional layer has a size of 32. The function of the next layer is to flatten the layers to prepare it for the fully connected 10 neuron output layer and we use softmax for our activation function, the Adam optimizer, and categorical cross entropy as our loss function. We use accuracy as our model's performance metric (percent the model correctly classifies). We train this model across multiple workers using a data parallel approach and for 10 epochs each. This results in a CNN with 336,618 trainable parameters. We use the same model for all CNN experiments on the Slurm cluster.

Provided the overhead of adding additional workers does not outweigh the additional performance of the additional workers, training times are expected to decrease as additional workers are used. Figure 1 illustrates this effect. As can be seen, the training time consistently decreases as the number of workers (in this case GPUs) are increased. It is evident by the curve of the data that there are diminishing returns in going from 1 to 8 GPUs. However, even at 8 GPUs, training time speedups still occur. Table I shows that the training time is reduced with each additional GPU. A similar pattern occurs when using CPUs as the processor type. These results mirror that of the GPU results except that the training times are significantly slower when trained on a CPU. Table I shows that even using 8 CPU workers is significantly slower than using 1 GPU worker when training this type of model on this type of data (385.8 seconds vs 123.5 seconds). This shows that regardless of processor type, the cost of adding additional workers (mainly communication costs) is outweighed by the continued decrease in training times. It would be an area of future work to find the upper limit of how many workers this could be scaled to.

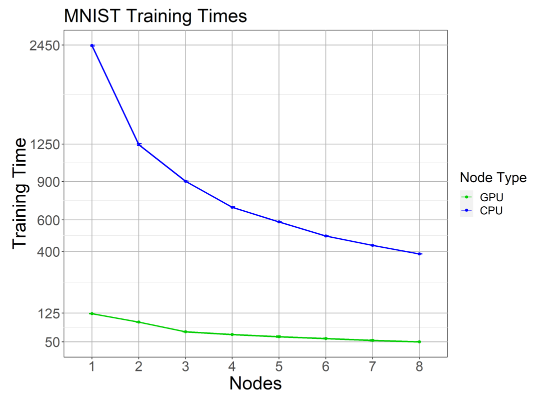

Fig. 1. Total training time vs. number of worker nodes (CPUs and GPUs) for the CNN trained on the MNIST dataset.

The effect of multiple workers on model performance is an important consideration when using clusters to train neural networks. Any significant reduction in training time would be in practice negated by too significant a drop in model performance. Figure 2 shows the model performance for each of the configurations across the 10 replications. This figure show that the model performance is slightly affected, and performance generally drops, slightly, as the number of workers is increased. Our experiments trained each model for a fixed set of epochs. One explanation to why there is a difference in model performance is that model convergence speed may be affected by the number of workers. Since we train for a fixed set of epochs, this would result in models that are at slightly different points in training, and it would be reflected in the model performance metric. Examining the effects of early stopping (early stopping is when NN training

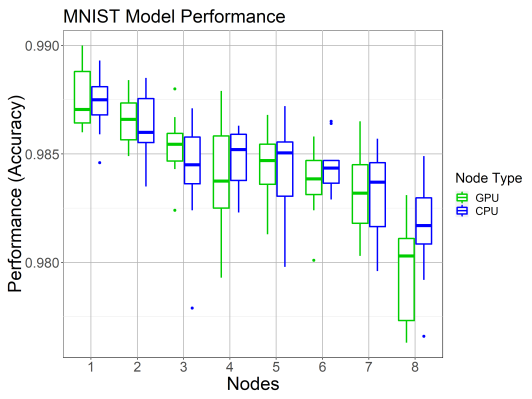

Fig. 2. Model performance vs. number of worker nodes (CPUs and GPUs) for the CNN trained on the MNIST dataset. Model performance is accuracy.

duration is dynamically determined, as opposed to training for a fixed set of epochs) may address this and was not considered in this paper.

It is interesting to see that there appears to be a larger variation in model performance when training on multiple GPUs compared to CPUs. This is potentially caused by a difference in random number generation between the two processor types. Given the same seed, a different implementation would result in two different models that are initialized with enough difference, that their convergence speed may differ significantly or that one model falls into a different local minima than the other. Table II shows that there is a statistically significant difference between some cluster configurations. It is important to note that this paper aims to compare performance differences of a single model and trying to find the overall best performing CNN model outside the scope of this paper. The cause of model performance differences between number of workers is an area of future work.

GPUs	Performance (groups)	CPUs	Performance (groups)
1	0.98765 (a)	1	0.98735 (a)
2	0.9866 (ab)	2	0.98626 (ab)
3	0.98536 (abc)	4	0.98471 (bc)
5	0.98451 (bc)	6	0.98447 (bc)
4	0.98398 (c)	5	0.98427 (bc)
6	0.98367 (c)	3	0.98424 (bc)
7	0.98313 (c)	7	0.98321 (cd)
8	0.97975 (d)	8	0.9818 (d)

TABLE II
TUKEY HSD TEST WITH GROUPINGS. MNIST MODEL PERFORMANCE, NUMBER OF GPUS AND CPUS

The Tukey HSD (honestly significant difference) tests in the paper use letter groupings to show which results are statistically different from each other and sorts them from highest to lowest performance. When using a GPU, Table II shows that a single GPU clearly has the highest performance with 98.765% accuracy, as denoted by it being in group “a” by itself. Using 8 workers is clearly the lowest performer with 97.975% accuracy. However, the results for 2 through

7 workers is not as clear. Results that share a letter in their groupings or are the same letter are not statistically significantly different from each other. For example, there is no statistically significant difference in model performance when training the CNN model on either 4, 6 or 7 GPUs (they are all in group “c”). Since 3 and 5 GPUs, have a “c” in the grouping, it means they are not different from those in group “c”. However, the Tukey test groupings show that with respect to using 3 GPUs there is no significant difference than using 1 or 5 GPUs. Importantly though, there is a statistical difference between using 1 GPU and 5 GPUs when comparing to each other. In short, what these groupings show is most of the configurations produced models with similar performance, but 1 and 8 GPUs are clear minimum and maximums. The Tukey HSD test for the CPU results, shown in Table II, indicates similar trends to that of the GPU results. Training with one CPU produces the most accurate model with 98.735% and training with 8 CPUs produces the lowest performing model with 98.180% accuracy. The groupings of the results of the other cluster configurations show there is little to no difference between them.

2) An MLP on Medicare Data: For our second set of experiments on our Slurm setup we train an MLP [26], [27] on the high class-imbalanced Medicare Part B fraud detection dataset. As mentioned previously, we perform RUS to sample from the original dataset to produce the dataset used in these experiments. It has a majority-to-minority class imbalance ratio of 99:1. The MLP has two fully connected layers, each with 100 neurons, producing a model with 24,001 trainable parameters. This model was trained for 10 epochs, similar to the previous one. This paper examines the effects of multiple workers on class imbalanced data. We present, in the previous section, results of a CNN trained on image data to compare to the MLP trained on imbalanced data. As the results show, using distributed training can have a large effect on training time and NN model performance, as compared to a CNN on image data. Additionally, we do not present our MLP model performance–as measured in AUC–itself as a novel result but rather we present the relative change in model performance of an MLP on a binary classification dataset between cluster sizes (i.e. our MLP AUC should be compared in a relative fashion to each other, not taken individually out of context).

Figure 3 illustrates the training time changes with respect to number of workers used for training. Unlike the results from the previous section, these results show that an MLP of this size and type of data does not necessarily scale well. For both GPU and CPU there is a clear speedup when going from 1 worker to 2 workers and then a clear increase in training time (greater than that of a single worker) when going from 2 workers to 3 workers. This shows that using 2 workers has significant performance benefits, with respect to training time, compared to using 3 or more. This is likely due to the increased overhead of using 3 or more workers in this type of Slurm cluster setup. It is interesting to see inefficiencies manifest so quickly after 2 workers for both GPU and CPU workers.

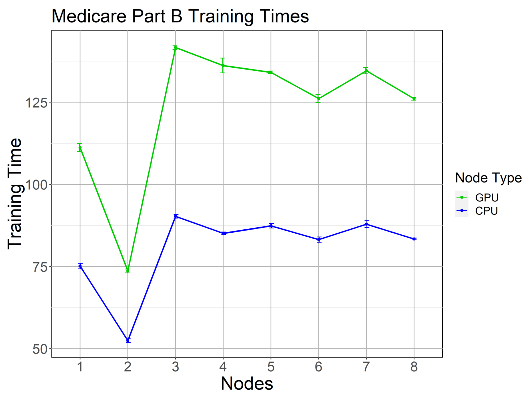

Fig. 3. Total training time vs. number of worker nodes (CPUs and GPUs) for the MLP trained on the class-imbalanced Medicare Part B dataset.

These results also show that using a single GPU vs a single CPU does not guarantee faster training times. Using Slurm, a single CPU took on average 74.9 seconds to train and a single GPU took on average 111.2 seconds to train, see Table I. This suggests that GPUs do not necessarily guarantee faster training times, for this type of model and on this type and size of data.

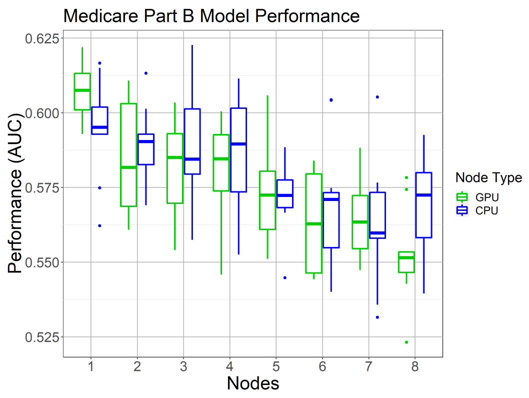

Fig. 4. Model performance vs. number of worker nodes (CPUs and GPUs) for the MLP trained on the class-imbalanced Medicare Part B dataset. Model performance is measured as AUC.

Figure 4 shows that the model performance is slightly affected by the number of workers when using GPUs and CPUs for an MLP on Medicare data. We use AUC as the performance metric for these models. Similar to the CNN experiments, using a single worker generally produces higher performing models and it reduces as the number of workers is increased. Table III presents a Tukey HSD test for the GPU and CPU results, respectively. Also like the CNN results, the HSD groupings show that the single workers produce statistically better models while the rest of the configurations produce lower performing models but are not necessarily better than the other. For example, using 2, 3, or 4 CPUs produce statistically similar performing models and using 5, 6, or 7 GPUs produce statistically similar performing models. This suggests that the difference in model performance is attributed to the distributed nature of the training routine and may be independent of model type and processor type.

GPUs	Performance (groups)	CPUs	Performance (groups)
1	0.60669 (a)	1	0.59515 (a)
2	0.58504 (ab)	3	0.58942 (ab)
3	0.58182 (b)	2	0.58862 (ab)
4	0.58013 (b)	4	0.58617 (ab)
5	0.57411 (bc)	5	0.57193 (bc)
7	0.56412 (bc)	6	0.56893 (bc)
6	0.56321 (bc)	8	0.56889 (bc)
8	0.55234 (c)	7	0.56291 (c)

TABLE III
TUKEY HSD TEST WITH GROUPINGS. MEDICARE PART B MODEL PERFORMANCE, NUMBER OF GPUS AND CPUS

V. CONCLUSION

We present how parallelization affects NN training time when using a cluster of CPUs or GPUs, specifically, on highly class-imbalanced Medicare fraud data. Results show that depending on model type and dataset type, GPU acceleration can significantly reduce training time but is not guaranteed across all NN architecture types in this study. We show that distributed training has more of an effect on training time and model performance when training on high-class imbalanced data, as compared to image data that is not class imbalanced. With respect to our model type, size, and training data, we show that with this type of model and this type of dataset the parallelism does not scale as well beyond 2 nodes. A smaller MLP model trained on Medicare Part B data does not scale as well as a CNN on image data.

We briefly examined and presented results that measured the effects of multiple workers on model performance. We demonstrated that there is a statistically significant drop in model performance, when measured as accuracy or AUC, when using more than one worker during training. Once distributed training is used, our results show that the largest drop in performance occurs around 8 workers. Future work toward finding the limits of scalability, with respect to training times, model performance, model type, and dataset type, should be considered.

REFERENCES

[1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[2] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[6] J. Nickolls, I. Buck, and M. Garland, "Scalable parallel programming," in *2008 IEEE Hot Chips 20 Symposium (HCS)*, pp. 40–53, IEEE, 2008.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[8] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*, pp. 44–60, Springer, 2003.

[9] D. Saad, "Online algorithms and stochastic approximations," *Online Learning*, vol. 5, 1998.

[10] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[11] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, pp. 1223–1231, 2012.

[12] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, pp. 2595–2603, 2010.

[13] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," in *Advances in Neural Information Processing Systems*, pp. 685–693, 2015.

[14] T. Nordstrom and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," *Journal of parallel and distributed computing*, vol. 14, no. 3, pp. 260–285, 1992.

[15] X. Zhang, M. Mckenna, J. P. Mesirov, and D. L. Waltz, "An efficient implementation of the back-propagation algorithm on the connection machine cm-2," in *Advances in neural information processing systems*, pp. 801–809, 1990.

[16] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[17] "Dataset: The mnist database of handwritten digits." http://yann.lecun.com/exdb/mnist/.

[18] P. J. Grother, "Nist special database 19," *Handprinted forms and characters database, National Institute of Standards and Technology*, 1995.

[19] R. A. Bauder and T. M. Khoshgoftaar, "Medicare fraud detection using machine learning methods," in *Machine Learning and Applications (ICMLA), 2017 16th IEEE International Conference on*, pp. 858–865, IEEE, 2017.

[20] U.S. Government, U.S. Centers for Medicare Medicaid Services., "The official U.S. government site for medicare." https://www.medicare.gov.

[21] CMS., "Research, statistics, data, and systems." https://www.cms.gov/research-statistics-data-and-systems/research-statistics-data-and-systems.html.

[22] U.S. Government, U.S. Centers for Medicare Medicaid Services, "What's medicare." https://www.medicare.gov/sign-up-change-plans/decide-how-to-get-medicare/whats-medicare/what-is-medicare.html.

[23] CMS., "Center for medicare and medicaid services." https://www.cms.gov/.

[24] R. A. Bauder and T. M. Khoshgoftaar, "A survey of medicare data processing and integration for fraud detection," in *Information Reuse and Integration (IRI), 2018 IEEE 19th International Conference on*, pp. 9–14, IEEE, 2018.

[25] J. M. Johnson and T. M. Khoshgoftaar, "The effects of data sampling with deep learning and highly imbalanced big data," *Information Systems Frontiers*, vol. 22, no. 5, pp. 1113–1131, 2020.

[26] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[27] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.