Reservoir Computing: Echo State Network Study in Parallel

Austin Egri
egria@rpi.edu

Henry Grover
groveh@rpi.edu

Abstract

Image classification is a problem as old as computer vision. One of the most popular image classification datasets is the MNIST [1] digit dataset. Since 1998, countless machine learning algorithms have tried their hand at attaining state-of-the-art accuracy. Echo state computing is a subset of reservoir computing [2], a paradigm of the large concept of recurrent neural networks. The goal is to get the benefits of a recurrent network, but avoid the related issues that follow, such as costly training or the vanishing gradient problem. In this paper, we introduce a parallelized implementation of an echo state network with the objective of correctly classifying the MNIST dataset.

1 Introduction

Neural networks are currently the hot new approach to many modern machine learning problems. They are more complex and far more computationally intensive than previous techniques, so why use them? With every approach to solving difficult and abstract computational problems, there will never be a perfect algorithm to obtain the correct answer for every input. This is known as the ”No Free Lunch” theorem, which conceptualizes the fact that no implementation will achieve perfect results. That being said, the appeal of neural networks stems from its performance when put up against other machine learning algorithms. In most settings, neural networks will outperform their counterparts in overall accuracy. This is especially the case in problems relating to image classification due to a neural networks capability to capture general features and collectively learn higher level abstractions. Deep learning [3] has become increasingly popular recently, and when paired with neural networks it can efficiently learn to classify by automatically finding the most effective representation of the input.

When working with complex tasks such as image recognition or time series, while recurrent networks are usually the most suited for the job, however they are also the most difficult to train. Backpropogation of a recurrent neural network unfolds multiple layers that mitigate the gradient as it is computed between layers. A reservoir computing network can ignore this strain because the recurrent portion of the network is not trained and therefore does not undergo backpropogation. The only layer in reservoir computing networks that is trained is the final layer, non-recurrent out layer. Typically the parameters of all other layers are randomly initialized based on some condition that prevents chaotic behavior.

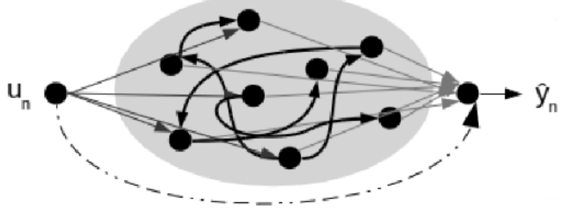


Figure 1: Architecture of a Reservoir Computing Network: inputs u_n are connected to a recurrent network, and $\hat{y}_n$ is the linear combination the network. [4]

An Echo State Network [2] introduces a new paradigm within recurrent neural network training. It is based on the concept that if a recurrent neural network is initialized with random weights, then training a linear output layer is sufficient to achieve outstanding results in most practical applications. While it may not outperform state-of-the-art convolutional neural networks [5], they present an interesting gateway into the field of reservoir computing as well as set a good base to build a highly parallelized distributed network.

Echo state networks are optimal for parallelization because of the fact that recurrent layers are not trained. They are not modified after initialization, and therefore they do not undergo backpropogation. This independence from gradients allows the *"reservoir"* nodes of the network to be independently computed. Echo state networks are also sparsely connected, meaning that a small number of connections are made between nodes. This is in contrast to densely connected networks where the number of links of each node is close the maximal number of other nodes. This is more similar to the output layer of echo state networks, where parameters are trained. In cases where all nodes in a layer are connected to all nodes in a consecutive layer, this is referred to as a fully connected layer. The sparsity of connections within the recurrent part of the network aid in parallelization because each distributed node needs a fewer number of input connections to other nodes, and consequently sends a fewer number of output signals to each output connection.

2 Literature Review

2.1 MNIST Dataset

The MNIST dataset[6] consists of over 60,000 images containing handwritten digits. It is a smaller version of the NIST dataset, except MNIST has the digits centered and formatted to touch the top and bottom of the frame. This dataset consists of 60,000 training and 10,000 testing images. This data is a popular choice among researchers to test out new algorithms, as it comes preprocessed and ready out of the box for image analysis. Additionally, it has the added bonus of applying itself to computer vision as the data consists of images as opposed to simple numerical values. The dataset was constructed by 500 writers, who were asked to write down a specific digit, thus the label was ensured. Many popular and successful algorithms were developed on this data and it is this reason that we used this for our recurrent neural network Echo State Machine.

2.2 Parallel Deep Neural Networks on Blue Gene/Q

The largest drawback of deep neural networks has been the significant computational cost that it requires to train the parameters of the network. Parallelization is rarely a viable solution as a result of costly inter-process communication bottlenecks. In [7], researchers at IBM introduce a way of training deep neural networks utilizing second order optimization techniques. Traditional deep neural networks tend to increase training time due to various reasons: *i.* big data tasks tend to be associated with large training sets, on the order of billions, which often improves network performance [8] *ii.* large datasets trained on deep neural networks typically have a large number of parameters, especially in image detection and speech recognition [9] [10], and *iii.* the most popular methodology for deep neural network training is also one of the hardest to separate from serial techniques, the first order stochastic gradient descent. Second order optimization techniques have been explored for the sake of paralellization, however they are not guaranteed to run faster than their first order serial counterpart. In particular, [11] shows that parallelization of deep neural networks can be slower than serial SGD training. This is typically because of the communication while passing gradients between processes, coupled with the need to run more training iterations than SGD. The objective of [7] is to explore the speedup of a second order optimization deep neural network on the Blue Gene/Q (BG/Q) high performance computer system. The BG/Q system is designed to be one of the most efficiently scalable systems for computationally intensive tasks. Within the study, the authors abandoned socket communication in favor or MPI communication because it provides greater functionality when managing a large number of nodes (in some cases thousands) in a large scale application, and is also relatively portable. The BG/Q system communication library is also heavily optimized for multi-process communication using MPI. The newest version is supported on BG/Q, which has wide support for high performance computing. Another factor that comes in to play with parallelization is load balancing to ensure that the processors at hardware level are effectively overlapping computation and efficiently utilizing hardware. At the application level, the program remains synchronized by distributing data evenly across all compute nodes which helps the network reach better performance. If the run time variation between working nodes is minimized, the overall time is reduced with less wait between connections. In cases where not all inputs required the same amount of time to compute, the data would be initially pre-processed and sorted to assign an equal amount of work to each compute node. The results of the study claim that the second order hessian-free [12] optimization was 2-11× faster when compared to first order SGD on a GPU. On the BG/Q systems, the achieved speed-ups that scale linearly up to 4,096 processes. Beyond this, the speed-up is significant but is sub-linear. Utilizing the efficient communication capability of MPI paired with BG/Q, the authors show that deep neural networks can perform just as well as their state-of-the-art serial SDG competitors.

2.3 TensorFlow

The module TensorFlow[13] is one of the most widely used frameworks for machine learning. This module allows programmers to type both low-level and high level operations in a code segment, of which the module then creates an execution graph to streamline these mathematical operations. TensorFlow will also keep track of the gradients of variables in the execution pipeline, so that it is easily able to re-adjust them during the update step of machine learning algorithms. TensorFlow comes prepackaged with different gradient optimizers, allowing the user to customize the back-propagation optimization workflow. Back-propagation is one of the harder steps in machine learning to correctly program and debug, so having a module that will take care of both weight gradients and updating the weights is tremendously helpful. TensorFlow implements a cross-platform library, where the backend calculations are written in extremely fast executing c code. This code communicates with both a python client and a C++ client, so that the users avoid low level memory management and other c idiosyncrasies. TensorFlow also comes prepackaged with many pre-trained and pre-written famous networks, which are terrific for getting jump-started on a machine learning task.

2.4 Horovod

Horovod[14] was introduced as a method of speeding up of making distributed deep learning easier. It was chosen to augment the open source deep learning framework of TensorFlow[13] . Initially, Uber found that the training time for its models would work on a single GPU, which is advantageous because there is not communication overhead among GPU machines. As the dataset size increased, this training was taking significantly longer (occasionally more than a week). Because of this, Uber was incentivized to begin distributed training. After testing standard distributed TensorFlow programming [15] , they found that there was significant room for improvement. Uber found that standard distributed TensorFlow was not scaling well. About 50% of its resources were being lost due to GPU scaling issues. Facebook showed that using parallelization techniques can train the ResNet-50 in 1 hour using 256 GPUs.[16] This paper used an approach where the model trains on different machines, and each gradient is then averaged across each machine's gradient using a parameter server. This server might or might not average all the gradients depending on the setup, but this server that is used to compute the gradient averages creates unnecessary overhead. Uber chose to use the ring-allReduce method for .[17] In this algorithm, a node communicates with its neighbors to perform message passing of the gradients. This algorithm is bandwidth optimal. Uber implemented this ring-allReduce algorithm and called named the python package Horovod. This package works as an out of the box library as an extension on TensorFlow, using this ring message passing algorithm for improved distributed speedup. Uber observed a 88% speedup from regular distributed TensorFlow.

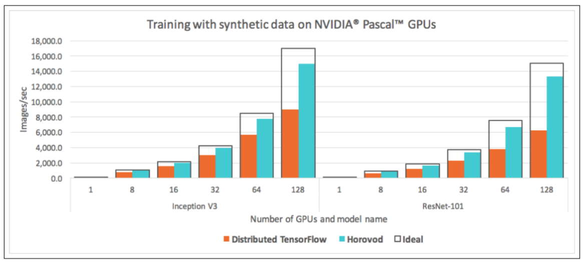


Figure 2: Uber's Horovod speedup relative to theoretical optimum and Distributed TensorFlow. [14]

In the Fig. 2, shown above, distributed TensorFlow performance is shown in orange with the metric of images per second and Uber's Horovod extension is shown in teal. It is clear that neither option achieves the optimal result, however the Horovod optimization comes far closer (88%) over both models examined.

3 The Model - Echo State Network (ESN)

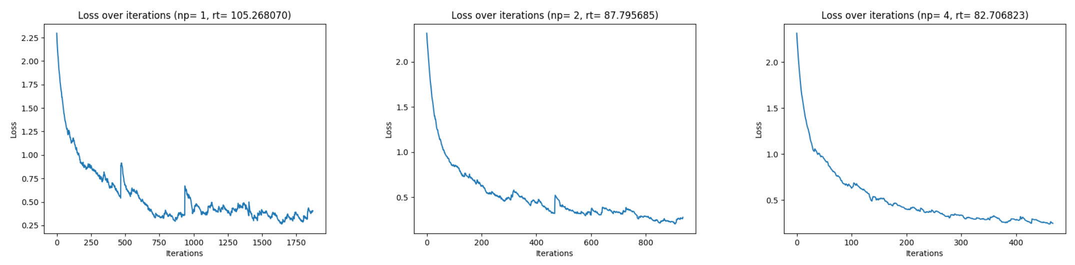
Since an Echo State Network is technically a recurrent neural network, it is not specifically designed for image classification. The approach we took was based off the model specified in "Echo

State Networks-Based Reservoir Computing for MNIST Handwritten Digits Recognition" [18]. This model treated each image input as a time series data point, so that way it could be inputted to the echo state machine. To do this, we treated each time step as a representation of a column of in image data. That is for time $t = 0$, the model took 28 length vector corresponding to the t column in the image. The model in the paper and for this project used 100 hidden neurons, that were concatenated together for each of the 28 steps in the time series. The module reported results for two hidden node sizes. The size of 1200 reported improved accuracy, however this would significantly increase the runtime of training. This report tested on the 100 sized hidden layers because of this issue. Additionally, a hyperparameter of .001 was used as the learning rate. All training runs were performed for 4 epochs over the data. Many training runs were performed on various batch sizes, but eventually it was found that having too small of a batchsize forced the algorithm to overfit to each batch, only allowing Because Horovard allowed for batch training, test cases were trained on the entirety of the MNIST dataset on each thread. A simple regression was performed from the hidden concatenated neurons to the output prediction values and softmax cross entropy loss was used as the objective was to predict an image classification value. To get the project started, the team used the github Echo State Network cell and example code from francesco-mannella [19]. This framework was extremely helpful to have an implementation of the echo state cell. The author provided an example on a time series using the Echo state cell.

4 Results

4.1 Performance Results

In terms of overall loss of the network, parallelization had some effect on the progression as the network trained. While the final loss is relatively the same among the tested networks regardless of the quantity of processors used, the graphs are different in their smoothness. It appears that as parallelization increases, the loss function is more constrained and less erratic as iteration increases. In Fig. 3a, we can see that as the loss function loses momentum, it is subject to higher degrees of fluctuation rather than leveling out at a specific level. In Fig. 3b the fluctuations have a similar location, however they are much less severe. Finally in Fig. 3c we can see that any similarity to 3a has been smoothed and the graph is rather constrained to a small value range. This makes sense because parallelization can be considered a form of batching. When one process is being executed, a $batchsize$ number of inputs are being computed at any given time. As more processes are introduced and parallelization increases, there are $batchsize \times nproc$ number of inputs being computed at once. Therefore, using multiple processes acts similarly to increasing batch size and therefore smooths out the loss function graph because there is less fluctuation from the true mean loss if a larger sample size is taken. Processing more inputs at once is more representative of what the true loss is for all inputs. This test was also run on 8, 16 and 24 processes in order to further compare the loss progression. In Fig. 4, we measure the loss over these higher number of processes. While the same phenomenon as figure 3 is occurring where the graphs are smoothed, we now choose to look at the behavior of the graphs. First, in Fig. 8a we can see that the loss initially underestimates the true loss, continues to overestimate, then finally plateaus at the true loss of the dataset. In Fig. 8b, this does not happen and there is a much more linear feel as the loss function decreases towards the true value. Finally, in Fig. 8c, it appears that the increases number of processes has caused the larger fluctuations in the graph to increase in size. Instead of initially underestimating the loss, the graph overestimates the loss during later iterations. We also note that the time for each graph increases as the number of processes increase. This is possibly a result of batch size change, because as the number or processes increases, a greater number of inputs are being computed at the same time and the theoretical batch size increases. If we compare the extreme cases, where batch size is either 1 input or the entire dataset, we can gain a better picture of what might be happening. If we have a batch size of 1, the computed loss will result in the exact loss for each input, and fluctuate drastically. If the batch size is the entire input dataset, then the computed loss is the mean loss for the dataset at some specific iteration. While both fall to an error minimum eventually, the larger batch size has a larger tendency to fall into a local error minimum because of the lack of fluctuation due to computing the total mean loss



(a) Loss calculated using 1 process (b) Loss calculated using 2 processes (c) Loss calculated using 4 processes

Figure 3: Above are 3 plots comparing Loss against iteration of the training algorithm. Each consecutive graph has twice the number of processes as the previous. While the loss may not be decreasing with parallelization, its calculation is smoother with more processes.

of the dataset in one go. The opposite extreme, batch size of 1, rarely falls into a local minimum because of its tendency to jump around. The small batch sizes oscillate around until they finally settle in the lowest error minimum. This is a drawback to a larger batch size and, therefore, a higher degree of parallelization. This is, however, comparing extremes which is why it is common practice to have mid-sized batch sizes. While the algorithm may not always fall into the global error minimum, the minimum that it does fall into is usually satisfactory for the task at hand, and well worth the reduced training time gained from utilizing vector and matrix operations that come with batch training.

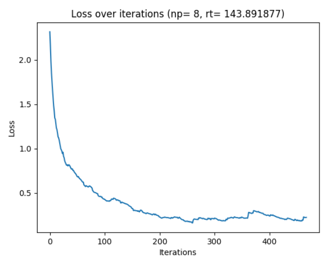
The overall accuracy of the program is also affected by the level of parallelization within the program. In Fig. 5 (shown below), we notice that as the number of processes increases, the level of accuracy also increases. As mentioned during the description of loss versus number of processes, as parallelization increases the theoretical batch size increases due to each process computing its own $batchsize$ number of inputs simultaneously.

This concept comes into play with overall accuracy as well, where the larger batch size settles on better minimum as a direct cause of the increase in the number of inputs being computed at a given time. This does appear to have a fundamental limit, as the accuracy improvement begins to plateau at 16 processes and even shows a slight negative consequence after 32 processes.

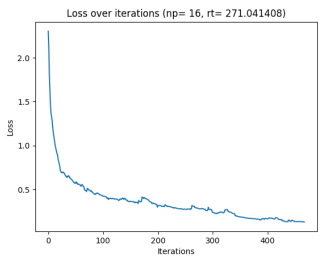
4.2 Analysis of Performance Results

The main goal of our study was to analyze the performance of our parallel neural network as we increases parallelization. Previously, we have described the effect that parallelization has on loss and accuracy of the neural network. This reassures the base assumption that parallelization does not hinder the credibility of the parallelized network. It is important to show that our accuracy results remain high, because if they were to suffer as a result of parallelization then the benefits of running a network in parallel such as training time and efficiency would not have any importance as the network itself cannot maintain reliable results. Now that we have supported the notion that our network can remain competitive, we can analyze the training time of the network when run at different parallel configurations.

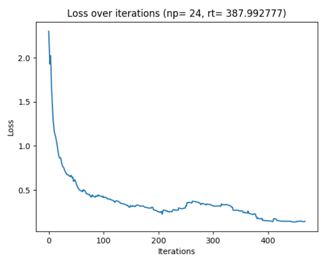
In a parallelized neural network, especially one that is practically designed for parallelization, the expectation is that training time decreases linearly with number of processes used. The recurrent portion of an echo state network does not change after initialization, and therefore can easily be computed in parallel on independent compute nodes. This theory is tested in [7], and shown that this is possible with the ideal optimization for parallelization. The authors used MPI and BG/Q optimization to achieve linear speedup up to a certain node limit. Our study uses Tensorflow with Horovod which is supposed to minimize communication between processes. In Fig. 6, we measure the overall run-time of our training process as a function of the number of processes used. Our implementation is running in parallel, however once the level of parallelization surpasses 4 processes our network takes a longer time to train. This is happening for a few reasons: *i.* the overhead generated from parallelization and communication during training time increases quadratically mostly due TensorFlow thresholding



(a) Loss calculated using 8 processes
Runtime: 144 (s)



(b) Loss calculated using 16 processes
Runtime: 271 (s)



(c) Loss calculated using 24 processes
Runtime: 388 (s)

Figure 4: Above are 3 plots comparing Loss against iteration of the training algorithm. As the number of processes becomes far higher than the 3 initial loss versus iteration graphs, we now look at the behavior of the graph rather than its smoothness.

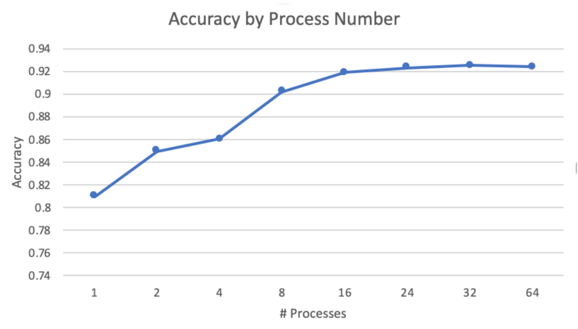


Figure 5: The final accuracy of the network increases compared to an increasing number of processes.

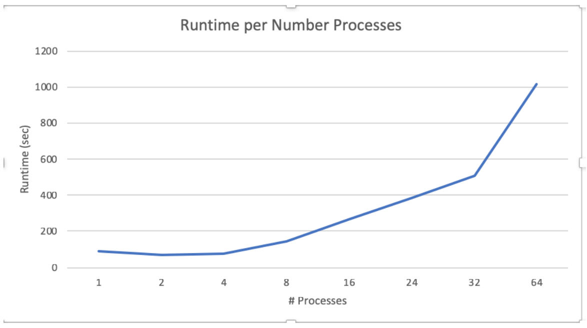


Figure 6

itself on the resources, *ii.* each process is computing its own full training set in these examples, therefore the speed that process is completes does not decreases, but rather more batches are computed during a single epoch. *iii.* Finally the processes are butting heads, fighting for similar resources so often times they get hung up, not allowing for the full benefit of parallelization.

One major benefit of introducing parallel programming is incorporating the ability to multi-process images all at once, increasing the overall productivity during training. As we can see in

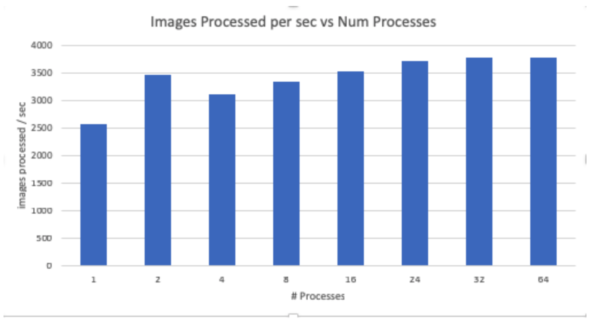


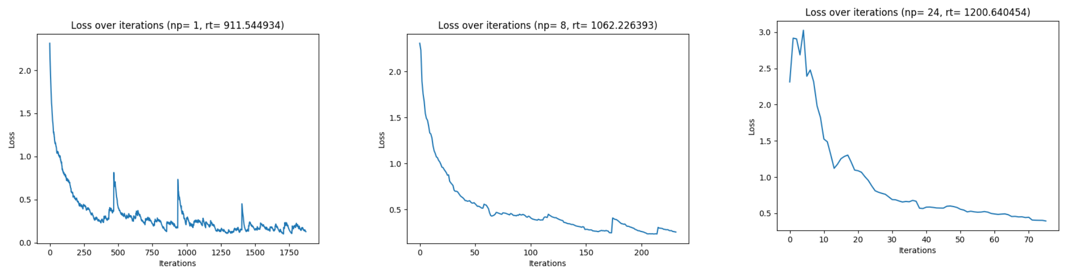
Figure 7

Figure 7, this was only partly true. Yes, the productivity increased as the process count increased, however only marginally so. This is not as we expected, as improved inter-process communication is supposed to increase this productivity even more so. Once again, this throttling is more likely due to TensorFlow hogging up computational resources, and not allowing each process to roam and fully compute without restriction.

The team also performed the above process count tests using a hidden layer size of 256 nodes.

For these tests, the subprocesses did not train on the entirety of the MNIST dataset during each epoch, but rather a subset of the MNIST dataset, depending on how many processes were running. It is clear that in the runtimes of this, it is reflected that the total runtime no longer scales in the same manner.

In Figure 10, we can see that it is clear that there occurs a drop of accuracy by using 256 hidden nodes on 24 processes. From the loss plot in figure 8.c, note how the loss gradient has not yet zeroed out. It is clear that this 24 process event did not finish training and could have done well to train



(a) Loss calculated using 1 process
Runtime: 911 (s)

(b) Loss calculated using 8 processes
Runtime: 1062 (s)

(c) Loss calculated using 24 processes
Runtime: 388 (s)

Figure 8: Above are 3 plots comparing Loss against iteration of the training algorithm. As the number of processes becomes far higher than the 3 initial loss versus iteration graphs, we now look at the behavior of the graph rather than its smoothness.

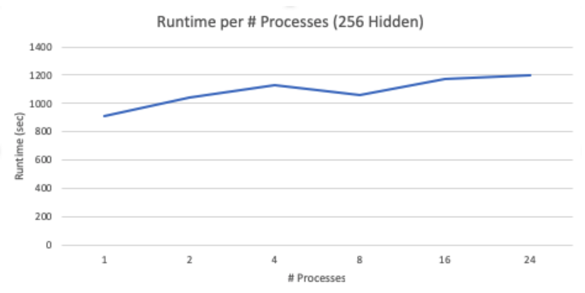


Figure 9: Runtime of 256 Hidden node processes runs

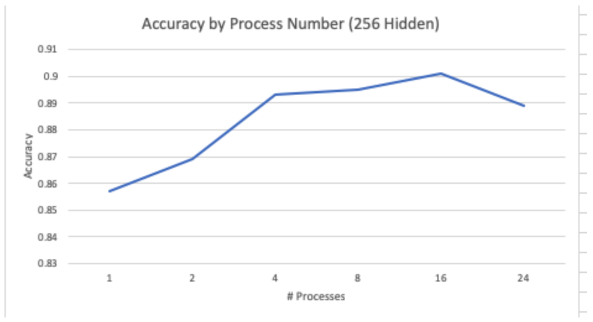


Figure 10: Accuracy of 256 Hidden node runs

for longer. It is also interesting to note how the smoothness of the losses changes as more concurrent processes are added, much like the 100 hidden node cases. This is likely because in the multiprocessed cases, each iteration is concurrently computing gradients, so the update step makes each successive loss far less volatile. It was interesting that the loss for 16 processes minimized far faster than the loss for 24 processes. It is not entirely clear why that occurred. In general it is expected that these losses are smoother, because it is kind of like stacking multiple differently trained models together to get what ends up being similar to a regularized model.

4.3 Future Work

There are numerous directions where we can take our study further. First of which, our code was not perfectly compatible with the inter-process communication available on our high performance computing system. While our project made use of MPI through the Horovad module [14], we still experience significant overhead at high levels of parallelization. We can use this as a direction to take our project, utilizing more communication optimizations available on the system. Some examples include hardware optimization such as transactional memory, thread level speculation, and potentially a list prefetch engine. Other software options could include further investigation of second order optimization functions such as the previously noted Hessian-Free [12] optimization, and use this as a potential base to compare other parallel friendly methodologies.

Another option for improving the algorithm would involve implementing early stopping and validation testing during training would help overcome this issue. Early stopping and validation was not included in this study, as the objective was to study performance, while training a ESN classifier that also worked. Since this classifier performed far better than guessing (1/10), this is assured. The ESN MNIST paper showed that they achieved optimal results using a hidden layer size of 1200 nodes [18]. This shows that it is likely that with some simple overfitting techniques such as early-stopping, regularization, and dropout, or even using a smaller learning rate, could have made this result achievable using even more hidden nodes than the 256 used in this example.

5 Conclusion

In our study, we have introduced a parallel implementation of the echo state network outlined in [18]. This approach tackles the image classification problem on the MNIST dataset, utilizing reservoir computing and recurrent neural network techniques to solve the problem. Our approach achieves a competitive accuracy and a more stable loss when run in parallel. A parallel approach has been shown in [7] to achieve good results while reducing training time dramatically. Parallel neural networks have numerous advantages over serially computed algorithms, such as the increase in batch size spread over each compute node, as well as the decrease in computing time and increase in images processed per second. While parallel network have been shown to equally perform against serial SGD networks, more work is needed in order to be able to efficiently and effectively parallelize proven networks that make use of lower order optimization functions and truly deep networks that backpropogate throughout the entire network.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and Haffner P. Gradient-based learning applied to document recognition. *dengfanxin.cn*, 1998.

[2] M. Lukosevicius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *sciencedirect*, 2009.

[3] Y. LeCun, Y. Bengio, and Hinton G. Deep learning. *nature*, 2015.

[4] N Schaetti, M. Salomon, and R. Couturier. Fig. 1 esn architecture, 2009.

[5] A Krizhevsky, I Sutskever, and G Hinton. Imagenet classification with deep convolutional neural networks. *nips.cc*, 2012.

[6] MNIST handwritten digit database, yann LeCun, corinna cortes and chris burges.

[7] I. Chung, T. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury. Parallel deep neural network training for big data on blue gene/q. *jmlr.org*, 2017.

[8] L. Deng, D. Yu, and G. Dahl. Roles of pre-training and fine-tuning in context-dependent dbn-hmms for real-world speech recognition. *microsoft*, 2010.

[9] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. *semanticscholar*, 2011.

[10] T. Sainath, B. Kingsbury, B. Ramabhadran, P. Fousek, P. Novak, and Mohamed A. Making deep belief networks effective for large vocabulary continuous speech recognition. *researchgate*, 2011.

[11] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Ng. On optimization methods for deep learning. *dl.acm.org*, 2011.

[12] J. Martens. Deep learning via hessian-free optimization. *toronto.edu*, 2010.

[13] Martn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. page 18.

[14] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow.

[15] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. page 19.

[16] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. page 12.

[17] Bringing HPC techniques to deep learning - andrew gibiansky.

[18] N. Schaetti, M. Salomon, and R. Couturier. Echo state networks-based reservoir computing for MNIST handwritten digits recognition. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pages 484–491.

[19] Francesco Mannella. Echo state networks with TensorFlow. contribute to francesco-mannella/echo-state-networks development by creating an account on GitHub. original-date: 2017-10-18T10:09:18Z.